HighCode Certifications

# C#sharp2 (.NET)

Last Update - 2022.11.30

www.albdarb.com
www.highcode.am
**Albert S. Darbinyan**
"HighCode" LLC
French University in Armenia (UFAR)
Gavar State University (GSU)
Microsoft IT Academy, Polytechnic

.....

**Հարցաշար** (Քննությունը թեստ "մինուս բոնուս" համակարգով -> 25 հարց 50 միավոր; ժամանակը = 1ժ. 10ր.)

**Ղեկավարվող կոդ** (Managed code) [Runtime]

1. . NET հիմունքներ [Троелсен-48; Troelsen-3; Нейгел-31; Шилдт-32]
   C#, NET Core
2. NET Framework, CLR, JIT Compiler [Нейгел-43; Троелсен-50; Troelsen-4; Рихтер-37; Шилдт-37]
3. Ղեկավարվող կոդ [albdarb; Гриффитс –25; Троелсен-54; Нейгел-569]
   ILDasm, Protecting Intellectual Property, NGen [Troelsen - 10]

**Ավելցուկի հավաքիչ** (Garbage Collection) [Runtime]

4. Ավելցուկի հավաքիչ [Троелсен-484; Troelsen-343; Нейгел-375; Nagel-340; Рихтер-554]
5. Օբյեկտների սերունդներ [Рихтер-562; Троелсен-489; Troelsen-347]
6. System.GC [Троелсен-491; Troelsen-349]
7. Չղեկավարվող ռեսուրսներ [Нейгел-377; Nagel-344]
   Finalize() [Троелсен-494; Troelsen-354; Рихтер-576; Албахари-521; Нэш -433; Шилдт-153]
   Դեստրուկտոր [Троелсен-495; Troelsen-355; Нейгел-377; Nagel-344; Шилдт-153]
8. IDisposable Interface Троелсен-498; Troelsen-357; Нейгел-378]
   using { } [Troelsen-358]
9. Դեստրուկտոր և Դիսփոզ պատերն [Троелсен-501; Troelsen-362; Нейгел-380]
10. Թույլ հղումներ [Албахари – 533; Albahari -579; Гриффитс – 327; Nagel-342]

**Անապահով կոդ** (unsafe code)

11. Ցուցիչներ [Нейгел-384; Троелсен-444; Troelsen-442]
12. Անապահով կոդ unsafe [Троелсен-445; Troelsen-443; Нейгел-383; Шилдт-684]
13. fixed, stackalloc [Троелсен-449; Troelsen-448; Нейгел-394; Шилдт-685]

**Կոմպոնենտներ** (Components)

14. Կոմպոնենտների ստեղծում [albdarb; Шилдт.2004-671]
15. Հավաքագրման բլոկներ [Троелсен-516; Troelsen-603; Нейгел-549; Гриффитс- 632; Албахари-729]
16. Հավաքագրման բլոկի կառուցվածքը [Троелсен-518; Troelsen-604]
17. Փակ հավաքագրման բլոկ [Троелсен-532; Рихтер-88,110]
18. Տեղաբաշխված հավաքագրման բլոկ [Троелсен-537; Troelsen-625; Нейгел-562; Рихтер-96]
19. Ատրիբուտներ [Шилдт -562; Нейгел-433; Троелсен-578; Troelsen-648]
20. Ֆիքսված և անվանական պարամետրերով ատրիբուտներ [Шилдт -566]
21. Ներդրված ատրիբուտներ [Шилдт - 570]

22. Տվյալների վալիդացիա ատրիբուտներով [metanit.com/sharp(pr.24)]
23. extern [Шилдт -712; Гриффитс - 641]
24. Տիպերի դինամիկ ճանաչում [Шилдт-537; Рихтер-126]
25. Արտապատկերում [Шилдт-541; Троелсен-562; Troelsen-632; Нейгел-407]
26. Հավաքագրման բլոկի արտապատկերում [Шилдт-555; Троелсен-571,573; Troelsen-641,643 ]
27. Հավաքագրման բլոկի հետաձգված կապվածություն [Троелсен-575; Troelsen-645; metanit.com/sharp(parag.18)]
28. Պրոցես [Троелсен-620; Troelsen-529; metanit.com/sharp(pr.22)]
29. Դոմեն ծրագրային մաս [Троелсен-631; Troelsen-541; Нейгел-559; metanit.com/sharp(pr.22)]
    class System.AppDomain [Троелсен-632]
30. Օբյեկտի կոնտեքստ սահման [Троелсен-640; Troelsen-544]
    Հոսք, դոմեն, կոնտեքստ հարաբերություն [Troelsen-549]

**Հոսքեր** (Threads) [Runtime]

31. Բազմահոսքայնություն [ Шилдт-833; Нэш -362; Троелсен-621; Troelsen-530]
32. class Thread [Троелсен-698; Troelsen-552; Шилдт-836; Нэш -358; Нейгел-579]
    Հոսքերի ստեղծում և ղեկավարում [Троелсен-700; Troelsen-555]
    delegate ParameterizedThreadStart [Троелсен-704; Troelsen-557; Шилдт-844; Нэш-361]
33. Հոսքերի նախապատվություններ [Троелсен-701; Troelsen-554; Шилдт -800; Нейгел-582]
    Առաջնային և ֆոնային հոսքեր [Нейгел-581; Троелсен-702; Troelsen-559; Нэш -368]
    class AutoResetEvent [Troelsen-558]
34. Հոսքերի սինխրոնիզացիա [Троелсен-707; Troelsen-550,560; Нейгел-595; Нэш -373]
35. lock տիպի բլոկավորում [Шилдт -849; Троелсен-709; Troelsen-562; Нейгел-601]
36. class Monitor [Шилдт -855; Нэш -382; Троелсен-711; Troelsen-564]
37. Wait(), Pulse(), PulseAll() մեթոդներ [Шилдт -855; Нэш -387]
38. class Interlocked [Шилдт -843; Троелсен-712; Troelsen-565; Нэш -375]
39. class Mutex [Шилдт -863; Нейгел-609; Нэш -395]
40. class Semaphore [Шилдт-867; Нейгел-610; Нэш -396]
41. Event սինխրոնիզացիա [Шилдт-870; Нейгел-613; Нэш-398]
42. class Timer [Нэш -410; Нейгел-657; Троелсен-714; Troelsen-566]
    Stopwatch [Албахари-552]
43. volatile մոդիֆիկատոր [Kort – 40; Нэш-64, 378; Шилдт – 710]

**Ասինխրոն ծրագրավորում** (Asynchronous programming) [Runtime]

44. Դելեգատների ասինխրոն բնույթը [Троелсен-691; Нейгел-576; Нэш -403]
45. Ասինխրոն դելեգատների սինխոնիզացիա [Троелсен-692; Нейгел-576; Нэш -403]
46. delegate AsyncCallback [Троелсен-694; Нейгел-578; Нэш -403]
    class AsyncResult [Троелсен-696; Нейгел-578; Нэш -403]
    Արգումենտի փոխանցում ասինխրոն դելեգատով [Троелсен-697; Нэш-403]
47. class ThreadPool [Троелсен-716; Troelsen-568; Albahari -624; Нейгел-629; Нэш -402]
48. TPL, class Task [Шилдт-887; Нейгел-585; Троелсен-717,723; Troelsen-569, 575; Albahari -626]
49. class Parallel [Шилдт-906; Нэш -413; Троелсен-718,725; Troelsen-569, 577; Нейгел-589]
50. async, await [Троелсен-731; Troelsen-582; Албахари -587; Гриффитс-900]

**Հայցային ծրագրավորում** (Query Programming)

51. LINQ to object [Шилдт-637; Нэш -525; Троелсен-452; Troelsen-493]
52. LINQ to object - from, where, select, orderby [Шилдт-653,644,646,649; Нэш -530,534,535]
53. LINQ to object - հայցի խմբավորում group [Шилдт-655; Нэш -538]
54. LINQ to object – into, let [Шилдт-657659; Нэш -541,537]

55. LINQ to object - join [Шилдт-660; Нэш -531]
56. LINQ to objects – հայցային մեթոդներ [Шилдт-669]
57. PLINQ - Հայցի զուգահեռ իրականացում [Шилдт-917; Троелсен-728; Troelsen-580]
    class CancellationTokenSource [Шилдт-920; Троелсен-729; Troelsen-581]
58. LINQ to XML [Нейгел-1111; Covaci – 461]

**Մուտք/ ելք** (Input/Output)

59. Տվյալների հոսքեր [Нейгел-896; Шилдт-431]
    Ֆայլային ինֆորմացիա և դիրեկտորիաներ [Троелсен-740; Troelsen-724]
60. Stream աբստրակտ կլաս [Шилдт-432; Троелсен-752; Troelsen-737]
    Ֆայլային մուտք /ելք [Троелсен-753; Troelsen-738; Шилдт-441]
    Հոսք հիշողությամբ մուտք /ելք [Шилдт-463]
    Բուֆերացված մուտք /ելք [Нейгел-898]
61. Տեքստային մուտք / ելք [Троелсен-754; Troelsen-740; Шилдт-449, 465]
62. Բինար մուտք / ելք [Нейгел-897; Шилдт-454; Троелсен-758; Troelsen-744]
63. Ֆայլային մարկերի տեղաշարժ - Seek(), Position() [Шилдт-461]
64. Ասինխրոն մուտք / ելք [Liberty-454; Троелсен2008-712; albdarb]
65. Ցանցային մուտք / ելք, Server, Client [Liberty-458; albdarb]
    Ինտերնետ ցանցային կապ [Шилдт-1014]
66. Սերիալիզացիա,Դեսերիալիզացիա [Троелсен-762; Troelsen-748]
67. XmlSerializer [Троелсен-772; Troelsen-752; metanit.com/sharp(pg.10)]
68. JSON Serializer [Troelsen-756; Perkins-609; metanit.com/sharp(pg.10)]

**Կոլլեկցիաներ** (Collections)

69. Կոլլեկցիաներ Ցուցակ List<>[Троелсен-361; Troelsen-391; Нейгел-288; Nagel – 195; Шилдт-961]
    Կլաս ArrayList [Троелсен-345; Troelsen-374; Нейгел-288; Nagel – 195; Шилдт-932]
70. Կոլլեկցիա Ստեկ Stack<> [Троелсен-362; Troelsen-393; Нейгел-301; Nagel – 206; Шилдт-945,977]
71. Կոլլեկցիա Հերթ Queue<> [Троелсен-363; Troelsen-394; Нейгел-298; Nagel – 202; Шилдт-947,979]
72. Կոլլեկցիա Hashtable [Шилдт- 939]
    Բառարան Dictionary<K,V> [Троелсен-366; Troelsen-397; Нейгел-309; Nagel – 211; Шилдт-969]
73. Կոլլեկցիա SortedDictionary<K,V> [Троелсен-366; Шилдт-972; Нейгел-308]
    SortedList<K,V> [Троелсен-366; Шилдт-974; Нейгел-316; Nagel – 209]
74. Կոլլեկցիաներ HashSet<> [Нейгел-317; Шилдт-982]
    SortedSet<> [Троелсен-364; Troelsen-394; Нейгел-317; Шилдт-980]
75. Կոլլեկցիա LinkedList<> [Шилдт-965, Нейгел-303; Nagel – 208; Албахари-337; Albahari-366]
76. Զուգահեռ աշխատանքի կոլլեկցիաներ [Нейгел-325; Шилдт-983]
77. Կլաս կոլլեկցիաների արտադրողականությունը [Нейгел-333]

**Debug**

78. Պրեպրոցեսոր [Шилдт-528; Нейгел-105; Covaci – 498]
79. [Conditional] ատրիբուտ [Албахари-539]
80. Debug [Уотсон- 166; Нейгел-501 ]
81. Debug և Trace կլասներ [Уотсон- 166; Албахари-540; Covaci – 504]
82. TraceListener [Албахари-542; Covaci – 504]
83. Debugging և Release թարգմանություն [Нейгел-497]

**Անվտանգություն** (Security)

84. Անվտանգություն [Прайс-353; Albahari-851]
85. Տվյալների շիֆրացում և դեշիֆրացում [Прайс-356 ; Нейгел-674; Албахари – 839; Covaci – 528]

Սիմետրիկ կրիպտոգրաֆիա [Прайс-357; Албахари – 839; Albahari-855]
86. Ասիմետրիկ կրիպտոգրաֆիա [Албахари – 843; Albahari-861]
87. Տվյալների հեշավորում [Прайс-361; Covaci – 538; Албахари – 837; Albahari-853]
88. Թվային ստորագրություն [Прайс-365; Албахари – 845; Albahari-863]
89. Աուդենտիֆիկացիա և ավտորիզացիա [Прайс-371; Нейгел-665; Nagel-563]

**Ամփոփում** (Summary)
90. C#.NET ծրագրավորման **OOP**, **կոմպոնենտ** և **Runtime** էտապները [albdarb]

albdarb.com
user: books psw: “sharp” (ռուսերեն գրականություն)
user: books psw: “sharpeng” (անգլերեն գրականություն)

## ԳՐԱԿԱՆՈՒԹՅՈՒՆ


1. Шилдт Г. -“Полное руководство C#4.0”, 2010.
2. Троелсен Эндрю… - “Язык программирования C# 7 и платформы .NET и .NET Core”, 2018.
3. Andrew Troelsen …– “Pro C# 9 with .NET 5”, 2021.
4. Andrew Troelsen, Phil Japikse – “Pro C# 10 with .NET 6”, 2022.
5. Нейгел К. и др…. - ”C# 4 и платформа .NET 4 для профессиоралов”, 2010.
6. Нейгел К…- “C# 5.0 и платформа .NET 4.5 для профессионалов”, 2013.
7. Christian Nagel - Professional C# And . NET, 2021.
8. Гриффитс Йен – “Программирование на C#8.0”, 2021.
9. Нэш Трей – “C# 2010. Ускоренный курс для профессионалов”, 2010.
10. Рихтер Дж. -“CLR via C#. Программирование на платформе .NET Framework 4.5 на языке C#”,2013.
11. Албахари Джозеф…- C# 7.0. Справочник. Полное описание языка, 2018.
12. Joseph Albahari – “C# 10 in a Nutshell”, 2022.
13. Марк Дж. Прайс – “C#7.1 и .NET Core2.0”, 2018.
14. Mark J. Price - C#11 and .NET 7, Modern Cross-Platform Development Fundamentals, 2022.
15. Tiberiu Covaci…”MCSD Certification Toolkit (Exam 70-483): Programming in C#”, 2013.
16. Уотсон К., Нейгел К…. – “Visual C# 2010: полный курс. ”, 2010.
17. Jesse Liberty – “Programming C#”, 2001.
18. Jason Alls - High Performance Pprogramming C# and .NET, 2022.
19. https://metanit.com/sharp/

# Ղեկավարվող կոդ (Managed code)

## 1. dot NET հիմունքներ, C#, .NET Core

❖ .NET համակարգին նախորդող ծրագրային միջոցներ

[Троелсен-48; Troelsen-3; Нейгел-31; Шилдт-32 ]

❖ **Պատմություն:**

C ..... կառուցվածքային լեզու (70-ական թվ.)

C++ ..... օբյեկտային կողմնորոշված լեզու (80-ական թվ.)

Java ..... լրիվ օբյեկտային,
**համակարգային անկախությամբ,**
**ղեկավարվող կոդով**,
ինտերնետ կողմնորոշված լեզու
( 90-ական թվ.)

C# .....լրիվ օբյեկտային,
կոմպոնենտային,
համակարգային անկախությամբ,
և ղեկավարվող և չղեկավարվող կոդով (unsafe),
միջլեզվային **ինտեգրացիայի**,
տեղաբաշխված ծրագրավորում,
ինտերնետ կողմնորոշվածություն
( 2001 թ.)

**Win32/C**

- Ծրագրավորման այս էտապում մեծ աշխատանք էր կատարվում հիշողության ղեկավարման համար:
- Ցուցիչների առատ և պարտադրված օգտագործում:
- Մեծ քանակով գլոբալ ֆունկցիաների օգտագործման բարդություն:

**C++/MFC**

- C++ լեզուն - դա օբյեկտային կողմնորոշված հարմարեցում է C լեզվի վրա:
- MFC – ուղղակի լրացուցիչ մակարդակ է Win32 API համակարգի համար:

**Visual Basic**

- Ավելի պարզ և հեշտ կիրառելի լեզու է ավելի սակավ հնարավորություններով:
- Նա աշխատում է օբյեկտների հետ, սակայն օբյեկտային կողմնորոշված լեզու չէ:
- Չի ապահովում ժառանգականություն, չունի բազմահոսքայնություն և այլն:

**Java**

- Սինտաքսիսի պարզություն և տրամաբանական համակարգվածություն:
- Միջպլատֆորմային անկախություն:
- Միջլեզվային դժվարություններ:

**COM- component object model**

- Եթե ստեղծվել է կլաս, որը բավարարում է COM- ի պահանջներին, ապա ստացվում է բազմակի օգտագործման երկուական ծրագրային կոդ:
- COM- սերվերին կարելի է դիմել ցանկացած այլ լեզվից, բայց կա սահմանափակում - չկա ժառանգականություն լեզուների միջև:
- COM- ի կոմպոնենտների ներքին կառուցվածքը շատ բարդ է:
- “DLL hell” կապված “ինստալի” հետ:
- COM կարող է ապահովել միայն տարբեր լեզուների փոխհարաբերություն:

## ❖ .NET համակարգի առաջացման հիմքերը

[albdarb]

- Ծրագրի միջպլատֆորմային տեղափոխելիության անհրաժեշտություն: Այլ խոսքով **անկախություն** օպերացիոն համակարգից և տեխնիկական միջավայրից:
- Ծրագրի տեղադրման պրոցեսի պարզեցում: Ծրագրի **վերսիաների** կոնֆլիկտի լուծում:
- Ծրագրի կատարման միջավայրի անհրաժեշտություն, որը հսկում է ինչպես **պոտենցիալ վտանգավոր** ծրագրերը, այնպես էլ օպերացիոն համակարգի և ռեսուրսների **օպտիմալ** օգտագործումը:
- Ծրագրային լուծումների տեխնոլոգիայի բարձրացում, որը բերում է տարբեր **լեզուների կոդի ինտեգրացիայի**:

//……

## ❖ . NET տեխնոլոգիական լուծում

- Գոյություն ունեցող երկուական կոդի հեշտ օգտագործում, այլ խոսքով, պատրաստի բինար COM օբյեկտները հեշտությամբ միավորվում է նոր բինար .NET կոդի հետ:
- Pinvok (Platform Invocation) - Հնարավորություն է տալիս կանչել C լեզվով գրված գրադարանից, ներառելով նաև WIN32 API, որը պատկանում է Windows համակարգին:
- Լրիվ միջլեզվային համագործակցություն- բավարարվում է միջլեզվային ժառանգականությունը, արտակարգ իրավիճակները և debug:
- dotNET Framework - ավելի զարգացած համակարգ է: Այն ապահովում է տարբեր լեզուների **ինտեգրացիա:** օրինակ- լեզուն կարող է օգտագործել տիպեր, որոնք ստեղծվել են մի այլ լեզվում:
- Լեզվից անկախ ընդհանուր կատարման միջավայր, ունենալով նույն ներդրված տիպերը:
- Հեշտացվում է "ինստալը" և չկա անհրաժեշտություն ռեեստրում գրանցվելը:
- Ծրագրի հին և նոր վերսիաների խաղաղ գոյատևում:
- Ծրագրավորումը կենտրոնանում է **խնդրի էության մեջ,** ոչ թե ծրագրավորման էլեմենտների:
- Ծրագրային ապահովում ծառայությունների ձևով, որը հանդիսանում է ժամանակակից պահանջ ինտերնետի ներգործման շնորհիվ օգտագործելով XML (extensible markup language) կապի լեզուն և SOAP (simple object access protokol) կապի պրոտոկոլը:

//……

## ❖ C#sharp

[Троелсен-51; Шилдт-34; Гриффитс- 23]

- Եթե ամփոփենք C# լեզվի հատկությունները, ապա կարող ենք ասել.

✓ **Ստեղծվել է զրոյից** և սինտաքսիսը նման է C++ և Java լեզուներին:
✓ Աշխատում է ղեկավարվող կոդով:
✓ Լրիվ օբյեկտային և կոմպոնենտային:
✓ Համակարգային անկախությամբ:
✓ Կարող է աշխատել չղեկավարվող կոդով օգտագործելով ցուցիչների հնարավորությունը:
✓ Հիշողության ավտոմատ ղեկավարումով:
✓ Ընդհանուր լեզուների ինտեգրացիայով:
✓ Ունի ինտերնետ կողմնորոշվածություն:

## ❖ .NET Core [Троелсен-79; Прайс-47]

- .NET Core համակարգը հնարավորություն է տալիս ընդլայնել ծրագրավորման հնարավորությունները տարբեր ոչ Microsoft միջավայրերում, կիրառելով ինչպես տարբեր լեզուներով (Android, iOS) ծրագրավորում .NET ում, այնպես էլ նրանց իրականացումը տարբեր օպերացիոն համակարգերում (Linux, MacOS):

……..……

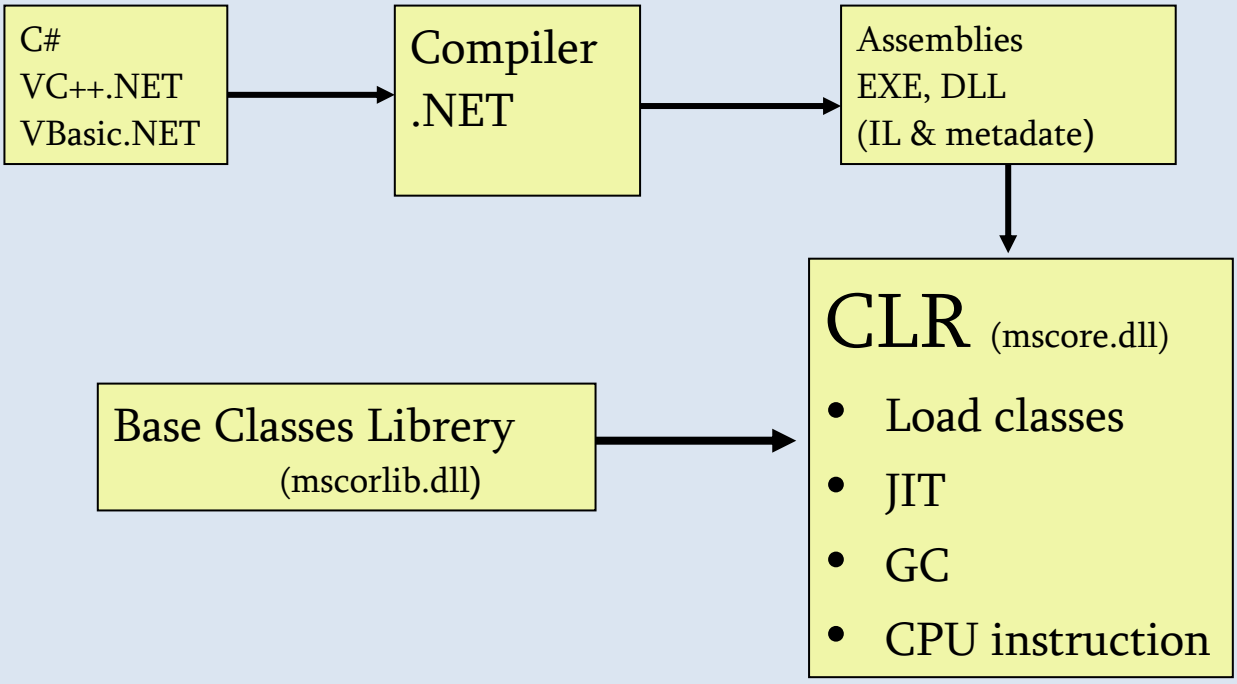
## 2. CLR - Ընդհանուր ծրագրային միջավայր (Ադապտացիա)

[Нейгел-43; Троелсен-50; Troelsen-4; Рихтер-37; Шилдт-37]

- **Framework** - դա նոր միջավայր է, որը կազմակերպում է ղեկավարվող կոդով ծրագրերի իրականացումը:
- **IL – Intermediate Language** – Ղեկավարվող կոդով աշխատելու համար պարտադիր է ունենալ միջանկյալ լեզու: Նա հանդիսանում է վիրտուալ մեքենայի ասեմբլերը և թարգմանված ծրագիրը տեղափոխվում է IL տեսքով: Ֆայլը .exe է, սակայն պրոցեսորի հրամաններ չէ: Նրա առավելություններից է զբաղեցրած հիշողության փոքր ծավալը: (նման է Java - բայթ կոդին): IL – ը համարվում է օբյեկտ կողմնորոշված մեքենայական լեզու: (Троелсен-57)
- **CLR** (Common Language Runtime) – Ընդհանուր ծրագրային միջավայր: Վիրտուալ միջավայր է, որը կատարում է IL տեսքով (ղեկավարվող կոդ) ծրագրի իրականացումը ռեալ համակարգչի վրա: Բարձրացնում է արտադրողականությունը – կարող է հարմարվել համակարգչի կառուցվածքին, օրինակ ստուգել պրոցեսորների քանակը: Կամ օրինակ կարող է հետևել ծրագրի կանչի հաճախականությանը և անհրաժեշտության դեպքում, եթե ծրագրի հատվածը երկար ժամանկ չի օգտագործվում այն վերաթարգմանվում է ավելի օպտիմալ լինելու համար:
  CLR – ը իրականացվում է **ընդհանուր ծրագրային մոդելով:** Ի տարբերություն նախկին մոդելների, որոնցից են օպերացիոն համակարգերի ֆունկցիաները հասանելիությունը DLL - նեի միջոցով, կամ աշխատանք COM օբյեկտներով, CLR - ը ամբողջ ծրագրային մոդելը ներկայացնում է **օբյեկտ կողմնորոշված համակարգով**: (Троелсен-65)
- **JIT – Just in time** - CLR – ի մաս է կազմում: Նա կոմպիլիտոր թարգմանիչ է: Ֆունկցիաները կամ մեթոդները թարգմանվում են **միայն առաջին կանչի ժամանկ**: Հաջորդ օգտագործման դեպքում չի կրկնում թարգմանությունը: Կամ ծրագրի հատվածը չի թարգմանվում, քանի դեռ այն չի օգտագործվում: **Քեշի առկայությունը բարձրացնում** է արտադրողականությունը – տարբերվում է ավելի օպերատիվությամբ: Օրինակ, եթե ծրագիրը փակելուց հետո նորից է կանչվում, ապա այն կատարվում է արագ, քանի որ այն կանչվում է RAM – ում գտնվող ծրագրից, այսինքն **քեշից:** Ջուգահեռ կատարվում է ծրագրի օպտիմիզացիա:
- **Mixed language programming** - միջլեզվային ինտեգրացիա և ծրագրավորում: (Троелсен-55)

✓ **CTS** (Common Type System) – ընդհանուր լեզվային տիպեր: (Троелсен-61)

✓ **CLS** (Common Language Specification) – ընդհանուր լեզվային օրենքներ: (Троелсен-65)




[Нейгел-66]

# 3. Ղեկավարվող կոդ

[albdarb; Гриффитс –25;  Троелсен-54; Troelsen-10]

- Ղեկավարվող կոդ – ծարագրավորման լեզվի տեխնոլոգիա է, ծարագրավորման զարգացման նոր էտապ և **համարվում է OOP հաջորդ զարգացում**:
- Ղեկավարվող կոդի ժամանակ ծրագիրը հապաղում է, տեխնիկական միջոցներին ավելի արտադրողական **հարմարվելու** համար:
- Ղեկավարվող կոդի  գաղափարը հիմնվում է **դեկոմպոզիցիայի** օրենքի կիրառման վրա: Նշենք աստիճանական անցումը:

✓ Ծրագրավորման նախնական էտապ.
"**Խմբավորվիր**, որ դիմանաս"- ծրագրային **մոդուլի** մակարդակ (սինթեզ):

✓ Բարդ խնդիրների ծրագրավորում.
"**Բաժանիր**, որ տիրես" - ծրագրային **մոդուլի** մակարդակ (անալիզ):

✓ Ծրագրավորման ժամանակակից էտապ (albdarb սահմանում).
"**Բաժանիր**, որ տիրես" - ծրագրի **պրոցեսի** մակարդակ (անալիզ):

Այս էտապը կոչվում է **ղեկավարվող կոդով** ծրագրավորում:  Կատարման ժամանակային բաժանումը որոշ արագության **կորուստի միջոցով** ստանում ավելի ինտելեկտուլ ծրագրային միջոց: Այլ խոսքով ծրագիրը դառնում է ավելի "**ադապտիվ**":  Իսկ ինչպես գիտենք բնությունը ունի **պատժող** օրենք – մի եղեք ուժեղ, մի եղեք խելացի, այլ եղեք ԱԴԱՊՏԻՎ: Դուք կդառնաք ամենահաջողակ տեսակը (FB):

//……

## ❖ ILDasm, Dotfuscators, NGen.exe ուտիլիտներ

[albdarb] [Рихтер-46; Троелсен-74; Нейгел-532]

- **ILDAsm**  ուտիլիտով կարելի է տեսնել IL միջանկյալ լեզվով ծրագրի տեսքը:
- Քանի որ IL -  ը ILDAsm "դեզասեմբլերով" տեսանելի է, ապա ինտելեկտուլ սեփականությունը ոտնահարվում է: Դա ճիշտ է և դրա դեմ կան **պաշպանողական** ուտիլիտեներ (**dotfuscators**) կամ կարելի է օգտագործել **սերվերի** վրա կատարվելու սկզբունքը:
- C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\ildasm.exe

**NGen.exe  ուտիլիտ** [Рихтер-46; Нейгел-532; 569]

- **NGen.exe** ուտիլիտը հնարավորություն է տալիս JIT թարգմանությունը պահել **արտաքին ֆայլում** և ծրագրի հաջորդ կանչի ժամանակ օգտվել նրանից: Այս դեպքում պարզ է, որ ծրագիրը միջին ստատիստիկ թարգմանություն է և զիջում է տեղում թարգմանման առավելությունից :
- Օրինակ կարող է "նատիվ" կոդը պահվել հետևյալ կատալոգում՝
  C:\Windows\assembly\NativeImages_v4.0.30319_64

……

# Ավելցուկի հավաքիչ (Garbage Collection)

## 4. Ավելցուկի հավաքիչ !

[Троелсен-483; Troelsen-343; Нейгел-375; Nagel-340; Рихтер-554]

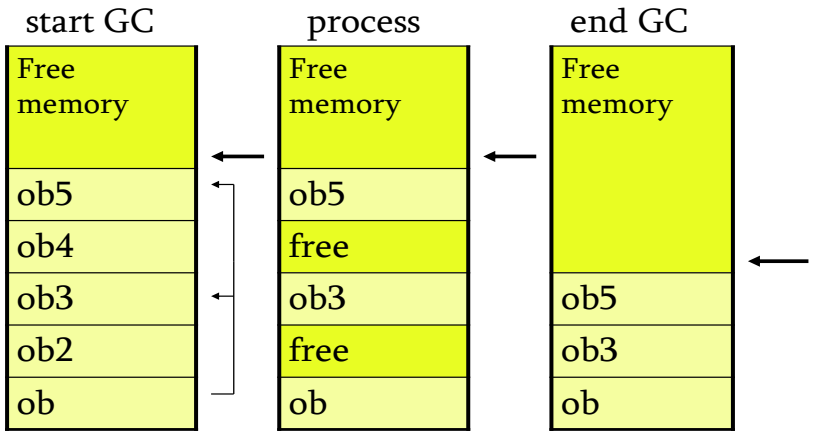
- Ինչպես գիտենք, չափային տիպերը, որոնք գտնվում են ստեկում, հեռացվում են ավտոմատ կերպով, երբ տիպը դուրս է գալիս ստեկի բլոկի տեսանելիության տիրույթից: Իսկ ինչ է կատարվում հղիչային տիպերի հետ, որոնց օբյեկտները ձևավորվում են կամ գտնվում են կույտում(heap): Ի տարբերություն շատ օբյեկտ կողմնորոշված լեզուների, որոնցից է օրինակ C++ -ը և որնոց դինամիկ հիշողությունում գտնվող օբյեկտները հեռացվում են ծրագրավորողի անմիջական հրամանների միջոցով, C# լեզվում, որտեղ դինամիկ հիշողությունը վերանվանվել է կույտ, հղիչային տիպերի հեռացումը կատավում է ավելցուկի հավաքիչի (GC) միջոցով:
- Ավելցուկի հավաքիչը ազատում է ծրագրավորողին օբյեկտների հղումնրի հետևման և չհղված օբյեկտների հեռացման անշնորհակալ և ժամանակատար աշխատանքից, այդ պրոցեսը իրականացնելով CLR համակարգի միջոցով ավտոմատ հեռացման ձևով:
- Երբ ստեղծվում է օբյեկտ (խոսքը հղիչային տիպերի մասին է), նրա հղումը, որը ցույց է տալիս "կույտ" հիշողության հասցե, ձևավորվում է ստեկ հիշողությունում, իսկ օբյեկտը ամբողջությամբ տեղավորվում է "կույտ" հիշողությունում: Շարունակելով ասենք, որ երբ օբյեկտը դուրս է մնում ծրագրային բլոկի սահմաններից, ստեկից ջնջվում է օբյեկտի հղումը( այսինքն հիշողության հասցեն) և այդ ժամանակ առաջանում է մի վիճակ, որը կարելի է անվանել անհասցե (անտեր) հիշողություն ցբաղեցված ծրագրային մաս, որը մեր օբյեկտն է: Իսկ ինչպես կամ ով պետք է հեռացնի այդ "անտեր" օբյեկտները: C# նույնիսկ չունի `delete` հրամանը(մյուս լեզուների նման) այդ չհղված օբյեկտներին հեռացնելու համար: Ահա առաջ է գալիս այն գաղափարը, որ .NET տխնոլոգիայի CLR -ում ներդրված է մի մեխանիզմ, որը կոչվում է ավելցուկի հավաքիչ և բանից պարզվում է նա ոչ միայն կարող է հեռացնել "անտեր" օբյեկտները, այլ նաև դա կատարում է ավոմատ կերպով: Եթե մեզ տանք այն հարցը, թէ երբ է աշխատում GC –ն, ապա կարող ենք ասել, որ նա կատարվում է ոչ դետերմինացված ձևով, այսինքն հստակ չգիտենք, թէ նա երբ կգա:
- Կարող ենք սահմանել ավելցուկի հավաքիչի աշխատանքի դեպքերը:
- ✓ առաջին, երբ ավարտվում է ծրագիրը, անպայման աշխատում է GC –ն:
- ✓ երկրորդ, երբ ծրագրի աշխատանքի ժամանակ հասնում ենք կրիտիկական հիշողության սահմանին:
- ✓ երրորդ, երբ ծրագրավորողն է կանչում GC –ն:
- Այսպիսով ինչ է տեղի ունենում, երբ ստեղծվում է հղիչային օբյեկտ: Օբյեկտի ստեղծման ժամանակ CLR – ը կատարում է հետևյալ գործողությունները.
- ✓ a).հաշվում է օբյեկտի ընդհանուր ծավալը:
- ✓ b).ստուգում է կույտ հիշողությունը, որպեսզի համոզված լինի անհրաժեշտ տեղի առկայությունը: Եթե կա բավարար հիշողություն, ապա ստեղծվում է օբյետը, որը ինչպես գիտենք նաև կանչում է կոնստրուկտոր, օբյեկտի ձևավորման նախնական աշխատանքների համար: Եթե բավարար հիշողություն չկա, ապա կանչվում է GC –ն, որը փորձում է հիշողություն ազատելու միջոցով բավարարի սեղծվող օբյեկտի պահանջները: Եթե չի ստացվում, ապա պարզ է, դադարեցվում է ծրագրի աշխատանքը և տալիս է համապատասխան հաղորդագրություններ:
- ✓ c).եթե ամեն ինչ բարեհաջող է, ապա փոխվում է հաջորդ օբյեկտի ստեղծման համար հիշողության հասցեի ցուցիչը:
- Ավելցուկի հավաքիչը ունի նաև առանձնահատկություն, երբ առանց հղման օբյեկտները հեռացվում են, ապա կատարվում պրոֆիլակտիկ գործ, խտացվում է հիշողության բաց տարածքները, կամ ավելի ճիշտ առկա օբյեկտները տեղաշարժվում են ծրագրի տվյալների մի ընդհանուր տարածք, ազատ տարածքները ավելի միասնական լինելու համար, որը իր հերթին **օպտիմալացնում** է ծրագրի հետագա աշխատանքը: Եթե օբյեկտի չափը ավել է 85KB –ից , ապա այն տեղադրվում է մեծ օբյեկտների համար հատկացված տարածքում և չի կատարվում օբյեկտների խտացուման պրոցեսը, քանի որ այն երկարատև է: (Нейгел-377)

# 5. Օբյեկտների սերունդներ

[Рихтер-562; Троелсен-489; Troelsen-347]

- Երբ ավելցուկի հավաքիչը ցանկանում է առանց հղման օբյեկներ փնտրել, նա պարտավորված չէ դիտարկել բոլորը: Ամեն օբյեկտ ստեղծման պահից ստանում է սերունդ կարգավիճակ 0,1,2 թվերի ձևով: Կիրառվում է հետևյալ մոտեցումը – որքան երկար է օբյեկտը գտնվում հիշողության մեջ, այնքան ավելի հավանական է, որ նա գտնվում է օգտագործման մեջ և նրան ավելորդ տեղը ստուգել իմաստ չունի:
- Սերունդները հիշողության մեջ ունեն իրենց համապատասխան հատկացումները:
- Երբ օբյեկտը է ստեղծվում նա ստանում է 0 արժեքով սերունդ:
- Օբյեկտը ստանում է 1 արժեք սերունդ, եթե այն չի հեռացվել ավելցուկի հավաքիչի կողմից:
- Օբյեկտը ստանում է 2 արժեք սերունդ, եթե այն դիմացել է ավելցուկի հավաքիչի մեկից ավելի օգտագործմանը:
- Այսպիսով կիրառելով սերունդների մեթոդը, ավելի նոր ստեղծված օբյեկտները, կհեռացվեն ավելի արագ, իսկ հին օբյեկտները կստուգվեն ավելի ուշ-ուշ:
- 0 և 1 սերունդները կոչվում են էֆեմեր և տարբերվում են համար 2-րդ սերնդից:
- Էֆեմեր սերունդների դիտարկումը կատարվում է առանձին հոսքով և դադարեցվում է նրանց օբյեկտների աշխատանքը: Միաժամանակ 2-րդ սերնդի օբյեկտների աշխատանքը շարունակվում է:
- NET 4.0 -ում մտցված է ֆոնային հոսքի գաղափարը, որը նշանակում է էֆեմեր սերունդների օբյեկտների աշխտանքը ֆոնային հոսքում, երբ կատարվում է համար 2-րդ սերնդի մաքրման աշխատանքները: Այլ խոսքով՝ NET 4.0 –ից սկսած զուգահեռ է աշխատում կամ 0,1 սերունդները, եթե մաքրվում է 2-րդ սերունդը կամ զուգահեռ է աշխատում 2-րդ սերունդը, եթե մաքրման տակ է 0,1 սերունդները:

OutOfMemoryException - արտակարգ իրավիճակ
Generation – 0, 1, 2 – սերունդներ



## 6. System.GC

[Троелсен-491; Troelsen-349]

- Collect() - աշխատում է ավելցուկի հավաքիչը:
- GetGeneration() – վերադարձնում է տվյալ օբյեկտի սերունդը:
- CollectionCount() – Վերադարձնում է թվային արժեք, որը ցույց է տալիս, թէ քանի անգամ է դիմացել օբյեկտը հավաքիչին:
- MaxGeneration – վերադարձնում է սիստեմայի max սերունդը:
- SuppressFinalize() – օբյեկտը չպետք է կանչի իր Finalize() մեթոդը:
- GetTotalMemory() - վերադարձնում է բայթերի քանակը, որը զբաղացնում են օբյեկտները, նույնիսկ նրանք, որոնք մոտակա ժամանակում պետք է հեռացվեն: Ունի bool պարամետր, որի true արժեքի դեպքում աշխատում է ավելցուկի հավաքիչը:
- WaitForPendingFinalizers() – Կանգնեցվում է ընթացիկ հոսքը քանի դեռ չի նայվել բոլոր **ֆինալիզացիայով** օբյեկտները: Հիմնականում կանչվում է Collect() մեթոդից հետո: GC.WaitForPendingFinalizers() մեթոդը ապահովում է, որպեսզի ֆինալիզացիաի բոլոր անհրաժեշտ քայլերը կատարվեն, ծրագրի հետագա աշխատանքները շարունակելուց առաջ: Նա հավաքիչի աշխատանքի ժամանակ **կանգնեցնում** է կանչվող հոսքի աշխատանքը:
- AddMemoryPressure() / RemoveMemoryPressure() - Allows you to specify a numerical value that represents the calling object's "urgency level" regarding the garbage collection process. Be aware that these methods should alter pressure *in tandem* and, thus, never remove more pressure than the total amount you have added.

```
using System;
        Console.WriteLine("Heap memory :" + GC.GetTotalMemory(false));
        Console.WriteLine("OS object generation :" + (GC.MaxGeneration));
        A obj = new A("one", 10);
        Console.WriteLine("obj is {0}", GC.GetGeneration(obj));
        //Console.WriteLine("Heap memory :" + GC.GetTotalMemory(true));
        GC.Collect();
        Console.WriteLine("obj is {0}", GC.GetGeneration(obj));
        object[] ob = new object[5000000];
        for (int i = 0; i < 1000000; i++)
        {
            ob[i] = new object();
        }
        GC.Collect(0);
        Console.WriteLine("obj is {0}", GC.GetGeneration(obj));
        GC.Collect();
        Console.WriteLine("\nGen 0 has been swept {0} times",GC.CollectionCount(0));
        Console.WriteLine("Gen 1 has been swept {0} times", GC.CollectionCount(1));
        Console.WriteLine("Gen 2 has been swept {0} times", GC.CollectionCount(2));
        Console.WriteLine("Heap memory :" + GC.GetTotalMemory(false));
class A
{
    int speed;
    string name;
    public A(string n, int s)
    {
        name = n;
        speed = s;
    }
}
```

.....

# 7. Չղեկավարվող ռեսուրսներ

[Нейгел-377; Nagel-344]

- Հիմնականում տիպերը իրենց աշխատանքի համար պահանջում են միայն հիշողություն: Այսպես built-in types, String, Attribute, Delegate և Exception տիպերը աշխատում են հիշողության բայթերի հետ: Բայց կան բարդ տիպեր, որոնք հիշողությանը զուգահեռ պահանջում են նաև **մեքենայական ռեսուրսներ** և տիպերի մաքրման հետ միաժամանակ նաև պետք է ազատել նրանց տրամադրված ռեսուրսները: (Рихтер)
- Չղեկավարվող ռեսուրսներ են – օրինակ ֆայլի դիսկրիպտորներ, ցանցային կապ(soket), կապ բազայի հետ, բիթային կարտա, մյուտեքս և այլն: Ավելցուկի հավաքիչը **չգիտի** ինչպես ազատել չղեկավարվող ռեսուրսները, կամ ավելի ճիշտ, նրա միջոցով ազատումը ունի ընդհանուր ձև, որը չի բավարարում ծրագրավորողին: Այդ դեպքում երբ ավելցուկի հավաքիչը սկսում է օբյեկտի հեռացումը, պետք է կատարել նաև հատուկ գործողություններ:
- Չղեկավարվող ռեսուրսների հետ գործ ունենք նաև երբ անմիջական օգտագործվում է օպերացիոն համակարգի API–ն: API – ի կանչը կատարվում է PInvoke –ի (Platform Invocation) միջոցով:
- Կլասի որոշման ժամանակ կարելի է օգտագործել երկու մեխանիզմ չղեկավարվող ռեսուրսի ազատման համար: Նրանք երբեմն միասին են օգտագործվում և քիչ են տարբերվում:
  1.Դեստրուկտորի հայտարարումով:
  2.System.IDisposable ինտերֆեյսի օգտագործում:

- Օրինակում ցուցադրվում է `Component` կլասի զբաղեցրած չղեկվարվող ռեսուրսների ազատումը:

```
using System.ComponentModel;
class A
{
    Component ob;
    public A()
    {
        ob = new Component();
    }
    ~ A()
    {
        ob.Dispose();
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
    }
}
```

## ❖ Finalize

[Троелсен-494; Troelsen-354; Рихтер-576; Албахари-521; Нэш -433; Шилдт-153]

- Օբյեկտի ֆինալիզացիան դա միջոց է, որտեղ կատարվում է չղեկավարվող ռեսուրսների մաքրման տրամաբանությունը: Երբեմն ասում են այնտեղ գրվում է պատժվողի **վերջին ցանկությունը**:
- Օբյեկտի ֆինալիզացիան ընդգրկվում է System.Object բազային կլասում և որոշում է Finalize() անունով վիրտուալ մեթոդով: Իր ստանդարտ իրականացումը ոչինչ չի կատարում և նույնիսկ նրա վերաորոշումը անթույլատրելի է:

```
Object
{
     protected virtual void Finalize() {          }
}
```

- Երբ վերաորոշվում է Finalize() մեթոդը, ապա որոշվում է տվյալ կլասի չղեկավարվող ռեսուրսների մաքրման լոգիկան: Քանի, որ նա `protected` է, ապա օբյեկտի միջոցով նրան կանչել հնարավոր չէ: Ինչպես նշել ենք, Finalize() հնարավոր չէ օգտագործել նույնիսկ `override` վերավորոշման միջոցով:
- `Finalize()` մեթոդը ավտոմատ աշխատում է ավելցուկի հավաքիչի աշխատանքի ժամանակ մինչև օբյեկտի հեռացնելը:
- Երբ աշխատում է ավելցուկի հավաքիչը, դադրեցվում է ծրագրի աշխատանքը: GC -ն առանձնացնում է բոլոր հեռացման ենթակա օբյեկտները և եթե կա **ֆինալիզացիայի** օբյեկտներ, ապա առանձնացնում է նրանց առանձին հերթում: Կատարելով **առանց ֆինալիզացիայի** "անտեր" օբյեկտների հեռացումը GC -ն վերջացնում է իր աշխատանքը և **թույլատրում** ծրագրի աշխատանքի շարունակությունը: Հետո սկսվում է ֆինալիզացիայի հոսքի **կատարումը զուգահեռ**: (ապացույցը օրինակում)
- Ֆինալիզացիայի օբյեկտների մաքրման աշխատանքի ավարտից հետո այդ օբյեկտները պահվում են "կախված" վիճակում մինչև ավելցուկի հավաքիչի **հաջորդ կանչը**: (դժվար ընկալելի գաղափար) [Албахари-521]
- eff-reachable [Troelsen-356]
- ✓ When the garbage collector determines it is time to free an object from memory, it examines each entry on the finalization queue and copies the object off the heap to yet another managed structure termed the finalization reachable table (often abbreviated as **freachable** and pronounced "eff-reachable"). At this point, a separate thread is spawned to invoke the Finalize() method for each object on the freachable table at the **next garbage collection**. Given this, it will take, at the least, **two garbage** collections to truly finalize an object.
- `Finalize()` մեթոդի աշխատանքը կազմակերպվում է **դեստրուկտորի** միջոցով:

### ❖ Դեստրուկտոր

[Троелсен-495; Troelsen-355; Нейгел-377; Nagel-344; Шилдт-153]

- Քանի որ, Finalize() մեթոդը հնարավոր չէ վերաորոշել,

```
class A
{
      protected override void Finalize() { } // Compile time error!
}
```

ապա ստեղծված է մի կառույց, որի անունն է **դեստրուկտոր** և այն վերաորոշում է `Finalize()` մեթոդը: Դա կատավում է նրա համար, որովհետև դեստրուկտոր կառույցին ուղղակի ընտելացած են ծրգրավորողները և հարգվում է նրանց սովորույթը:

- Դեստրուկտրը նման է C++ ում կիրառվող դեստրուկտորին, սակայն սկզբունքորեն տարբերվում է նրանից:

Հայտարարման ձևը

```
~A()
{                  // Unmanaged Resources            }
```

Համատասխանաբար կամպիլիատորը վերածում հետևյալ կոդի.

```
try
{                     // Unmanaged Resources     }
finally
{                  base.Finalize( );             }
```

- Դեստրուկտորը չունի պարամետրեր, չի վերադարձնում արժեք և չունի թույլատրության մոդիֆիկատոր:
- Դեստրուկտորը աշխատում է, երբ կանչվում է ավելցուկի հավաքիչը:
- Կլասների ստեղծման ժամանակ պետք է հնարավորին չափ քիչ օգտվել դեստրուկտորից, որովհետև այն ժամանակատար է:

- Երբ օբյեկտը տեղավորվում է դինամիկ հիշողությունում այն նշվում է, թե արդյոք դեստրուկտրով է թէ ոչ: Եթե դեստրուկտրով է, ապա օբյեկտի հղումը դրվում է հատուկ աղյուսակում, որը կոչվում է “ֆինալիզացիայի” հերթ:
- Դեստրուկտրով օբյեկտներ օգտագործում են ավելցուկի հավաքիչի մինիմում երկու անցում:
- Դեստրուկտրով նկարագրված օբյեկտները կարգավորում են օբյեկտի չղեկավարվող ռեսուրսների հեռացման մեխանիզմը, սակայն դա **ոչ դետերմինացված** պրոցես է (հստակ չի որոշված նրա կանչի ժամանակը) և խլում է բավական ժամանակ:

```
// Framework
using System;
class A
{
    long[] a = new long[10000000];  // ar=new long[10];  // Core **
    ~A()
    {         Console.Write("destruction ..... ");      }
}
class M
{
    static void Main()
    {
        for (int j = 0; j < 10; j++)
        {
            Console.Write(j + " ");
            A ob = new A();
        }
        // GC.Collect(); // աշխատում է զուգահեռ հոսքում, եթե առկա է ֆինալզիացիան
        // System.Threading.Thread.Sleep(100); //  ապացույց, որ աշխատում է զուգահեռ հոսքում
        // GC.WaitForPendingFinalizers();   // դադարեցվում է ընթացիկ հոսքը
        Console.WriteLine("END++++++++##############++++++++++");
    }
}
```

- Առանձնահատկություն: ( // Core **) Core –ի դեպքում ծրագրի ավարտից հետո դեստրուկտորի հրամանները չի ցուցադրվում:

❖ Ավելցուկի հավաքիչի բազում կանչ: Farmeworke –ում մեծ քանակով դեստրուկտորի կանչ է ցուցադրվում, իսկ Core –ի դեպքում ծրագրի ավարտից հետո կանչը չկա:

```
using System;
class M
{
    static void Main()
    {
        A ob = new A(1);
        Console.WriteLine("END........##############.........");
        Console.ReadKey();
}   }
class A
{
    int x;
    long[] a = new long[100];
    public A(int i)
    {         x = i;        }
    ~A()
    {
        A ob = new A(x+1);
        Console.WriteLine();
        Console.Write("destruction = " + x);
}   }
```

.....

## 8. IDisposable Interface, using { }

[Троелсен-498; [Troelsen-357] Нейгел-378; Рихтер-579]

### ❖ Dispose()

- Քանի, որ չղեկավարվող ռեսուրսները շատ թանկ են և պարզ է ցանկացած մեթոդում կարելի է իրականացնել չղեկավարվող ռեսուրսների մաքրման հրամանները: Սակայն, քանի որ C# ը պաշտպանում է կոնտրակտայի ծրագրավորման միջոցները, ապա հնարավորություն է տրվում օգտագործելով IDisposable ինտերֆեյսը, որը պարտականություն է օբյեկտի վրա, որպեսզի այն իրականացնի Dispose() մեթոդ:

```
public interface IDisposable
{        void Dispose();        }
```

- Այդ դեպքում, երբ օբյեկտը վերջացնում է իր գործը, ձեռքով կանչվում է "դիսփոս" մեթոդը, որտեղ նախատեսվում են ռեսուրսների ազատումը: Սա հնարավորություն է տալիս օբյեկտը չդասել ֆինալիզացիայի հերթի մեջ և չի կանչվում ավելցուկի հավաքիչ:
- Այլ խոսքով IDisposable ինտերֆեյսը հնարավորություն է տալիս ունենալ դետերմինացված մեխանիզմ չղեկավարվող ռեսուրսները ազատելու համար:

### ❖ using { }

[Троелсен-500; Troelsen-359; Нейгел-379; Шарп-376]

- IDisposable ինտերֆեյսի պարտականությունների տեխնոլոգիան բերում է իր առավելությունները: Եթե օբյեկտի օգտագործման տարածքը տեղադրենք `using {}` հատուկ հրամանի բլոկում, ապա բլոկի սահմանից դուրս գալու դեպքում Dispose() մեթոդը կանչվում է ավտոմատ:
- Օգտագործելով IDisposable ինտերֆեյսը կարելի է նաև կիրառել try...cach...finally համակարգը, որպեսզի համոզված լինել, որ նույնիսկ արտակարգ վիճակի դեպքում կկատարվի "դիսփոզը":

```
try
{       // use ob          }
finally
{       ob.Dispose();     }
```

Ոչ բոլոր ծրագավորողները կհամաձայնվեն անընդհատ արտակարգ վիճակ օգտագործել "դիսփոզ" կանչելու համար և կարող են նախընտրել `using {}` –ը:

- Օրինակում օգտգործվում է `using {}` –ը և երաշխավորվում է, որ երբ ծրագիրը բլոկից դուրս գա, ապա անմիջապես կկատարվի դիսփոզը:

```
using System;
class A : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("disposing");
    }
}
class M
{
    public static void Main()
    {   A ob = new A();
        A ob2 = new A();
        // ob.Dispose();
        // using (A ob=new A(), ob2=new A()) //ok!
        // using (A ob3=ob,ob4=ob2) //ok!
        using (ob)
        {
            Console.WriteLine("barev");
        }
}   }
```

# 9. Դեստրուկտոր և Դիսփոզ միասնական օգտագործում և պատերն

[Троелсен-501; Troelsen-362; Нейгел-380]

- Քննարկվել է չղեկավարվող ռեսուրսների ազատման երկու ձև - դեստրուկտրով և դիսփոզով: Դեստրուկտորի դեպքում մենք համոզված ենք, որ ռեսուրսները կազատվեն: Մյուս դեպքը, երբ օբյեկտի հետ աշխատանքից հետո կանչում ենք դիսփոզ: Բայց վերջինիս դեպքում, եթե ծրագրողը մոռանա կանչել դիսփոզ, ապա ռեսուրսները կարող են անհիմն երկար մնալ կամ հիշողության մեջ, կամ չանջատված ձևով:
- Կարելի է միևնույն կլասի սահմաններում համատեղել այս երկու ձևերը: Եթե ծրագրողը համոզված է, որ չի մոռանա դիսփոզը, ապա SuppressFinalize() մեթոդի միջոցով կարելի է հավաքիչին տեղեկացնել, որ նա հրաժարվի ֆինալիզացիայից: Միևնույն ժամանակ եթե ծրագրողը մոռանում է դիսպոզը, ապա վերջին հաշվով հավաքիչը կկատարի ֆինալիզացիա:

```
public class A : IDisposable
{
      ~ A(){          }
      public void Dispose()
      {     GC.SuppressFinalize(this);    }
}
//.....................................................................
using System;
class A: IDisposable
{
    ~A()
    {
        Console.Write("destructor "); // Framework
    }
    public void Dispose()
    {
        Console.Write("disposing ");
        GC.SuppressFinalize(this);
    }
}
class M
{
    public static void Main()
    {
        A ob;
        ob = new A();
        ob.Dispose();
    }
}
```

- **Ռեսուրսների ազատման պատերն**
- Նախորդ թեմայի թերություններից է դեստրուկտորի և դիսփոզի կոդի կրկնությունը: Ավելին - պետք չէ, որպեսզի ֆինալիզեն փորձի ազատել ռեսուրս, որը դրված է դիսփոզի վրա: Հաջորդը – օբյեկտի օգտագործողը պետք է հնարավորություն ունենա բազմակի օգտվի դիսփոզից:
- Ելնելով նշվածից, Microsoft – ը առաջարկում է ֆորմալ շաբլոն (պատերն) ռեսուրսներ ազատելու համար, որը զերծ է հավանական սխալներից, արտադրողական է և ունի պարզություն:

```
using System; //albdarb
class A : IDisposable
{
    public void Dispose()
    {
        f();
        Console.WriteLine("Call Dispose ");
        GC.SuppressFinalize(this);
```

```
    }
    void f()
    {
        Console.WriteLine("Clean up unmanaged resources ");
    }
    ~A()
    {
        f();
        Console.WriteLine("Call Destructor ");
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        // ob.Dispose();
        // ob.Dispose();
    }
}
```

// Մեկնաբանել ծրագիրը

```
using System; //Microsoft
public class A : IDisposable
{
    bool disposed = false;
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        f(true);
        Console.WriteLine(" Call dispose  ");
    }
    void f(bool dd)
    {
        if (!this.disposed)
        {
            if (dd)
            {
                Console.WriteLine("Dispose managed resources. GC.Collect()");
            }
            Console.WriteLine("Clean up unmanaged resources ");
        }
        disposed = true;
    }
    ~A()
    {
        f(false);
        Console.WriteLine("Call destructor ");
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        //ob.Dispose();
    }
}
```

.....

# 10. Թույլ հղումներ

[Албахари – 533; Гриффитс - 327; Nagel-342]

- Թույլ հղումների համար օգտագործվում է `WeakReference` կլասը: Այն ներկայացնում է իրենից հղում, որը նշում է օբյեկտ, սակայն թույլ է տալիս ավելցուկի հավաքիչին հեռացնել օբյեկտը:
- Ավելցուկի հավաքիչը չի կարող հեռացնել օբյեկտը, եթե այլ հղումներ նույնպես գոյություն ունեն տվյալ օբյեկտի համար և օգտագործվում են:

```
// Framework
using System.Text;
using System;
class Program
{
    static void Main(string[] args)
    {
        //StringBuilder ss = new StringBuilder("barev");
        // WeakReference weak = new WeakReference(ss); // հնարավոր չէ հեռացնել
        WeakReference weak = new WeakReference(new StringBuilder("barev"));
        Console.WriteLine(weak.Target);
        GC.Collect();
        Console.WriteLine(weak.Target);
    }
}
//.....
```

```
// Framework
using System;
class A
{
     int [] a = new int[100000];
}
class Program
{
    static void Main(string[] args)
    {
        // A ob = new A();
        // WeakReference w = new WeakReference(ob);
        WeakReference w = new WeakReference(new A());//
        System.Console.WriteLine(GC.GetTotalMemory(false));
        System.Console.WriteLine(GC.GetTotalMemory(false));
        GC.Collect();
        System.Console.WriteLine(GC.GetTotalMemory(false));
    }
}
.....
```

# Անապահով կոդ (Unsafe Code)

## 11. Ցուցիչներ

[Нейгел-384; Троелсен-444; Troelsen-442]

- Բացի հղիչային և չափային տիպերից dotNET - ը առաջարկում է աշխատանք **ցուցիչների** հետ: Այդ դեպքում ծրագրավորողը շրջանցում է CLR – ի հիշողության ղեկավարումը և կարող է այն ղեկավարել ինքնուրույն:
- Ցուցիչների օգտագործումը լինում է ոչ հաճախ և այդ դեպքում ծրագրի անվտանգ աշխատանքի պատասխանատվությունը մնում է ծրագրավորողին: Նշենք երկու ստանդարտ իրավիճակ, որի դեպքում կարելի է օգտագործել ցուցիչներ:

✓ Կա ցանկություն օպտիմալացնել ծրագրի հատվածներ.

✓ Անհրաժեշտ է օգտագործել C - գրադարան *.dll տեսքի կամ COM սերվերներ, որոնք պահանջում են ցուցիչով պարամետրեր:

- Կարելի է առանց ցուցիչի դիմել նշված գրադարաններին օգտվելով DllImport հնարավորությունից:
- Ցուցիչների թվաբանություն: Ցուցիչների անունները թույլատրվում է վերագրել, համեմատել և կատարել “increment, decrement”: Չի թույլատրվում կատարել թվաբանական գործողություններ: Գոյություն ունի նաև կրկնակի (**p) (եռակի ***p…..) ցուցիչ գաղափարը:
- void ցուցիչ (Нейгел-386; Troelsen-215): Եթե անհրաժեշտ է ունենալ ցուցիչ և միաժամանակ չնշել տիպը, ապա օգտվում ենք void ցուցիչից: Գլխավոր նշանակությունը կայանում է նրանում, որպեսզի կանչենք API ֆունկցիաներ, որոնք ունեն *void պարամետր:
- C# ում առանձնապես **անհրաժեշտություն չկա** void ցուցիչի օգտագործման:
- Function pointer (C# 9) [Albahari-248]

……

## 12. Անապահով կոդ - unsafe

[Троелсен-445; Troelsen-443; Нейгел-383; Шилдт-684]

- Չի ղեկավարվում պլատֆորմայի կողմից և համարվում է ռիսկի գոտի:
- Կարելի է օգտագործել **ցուցիչներ** – հետևյալ առավելություններով՝

✓ թույլատրվում է օգտագործել Windows API

✓ օգտագործվում է արտադրողականության բարձրացման համար

- Կոմպիլիատրում unsafe օգտագործման ձևը
- Project ->Propertis->Configuration propertis->Builds-> unsafe code=true
- unsafe կարող է դառնալ կլասը, ստրուկտուրան, մեթոդը, կլասի դաշտը, մեթոդի մեջ ցանկացած բլոկ: Մեթոդի լոկալ փոփոխականը, օբյեկտի ստեղծումը կամ ցանկացած հրաման չի կարող ստանալ unsafe:
- Ցուցիչը կարող է դիմել ստրուկտուրայի օբյեկտին, եթե այն չի ընդգրկում հղիչային տիպ:
- Ցուցիչը չի կարող դիմել հղիչային տիպերին, սակայն եթե տիպը կլաս է, ապա կարելի է դիմել կլասի չափային տիպին:

```
using System;
class Program
{
    unsafe static void Main(string[] args)
    {
        int a = 555;
        int* p;
        p = &a;
        Console.WriteLine((int)&a);
        Console.WriteLine((int)p);
        Console.WriteLine(*p);
```

```
        p++;
        Console.WriteLine((int)p);
        // int** p1;
        // p1 = &p;
        // int*** p2 = &p1;
        // Console.WriteLine(a);
        // Console.WriteLine(***p2);
        void* pv;
        pv = (void*)p;
        // Console.WriteLine(*pv); //error
        Console.WriteLine((int)pv); // ok
    }
}
//.....................................................................

using System;
struct S //class S // error !!
{
    public int a;
    // public string s = "barev"; // error !!
}
unsafe class M
{
    static void Main()
    {
        S* ps;
        S ob = new S();
        ps = &ob;
        ps->a = 16;
    }
}
```

❖ **Native-Sized Integers** (C# 9) nint, nuint [Albahari – 246]

.....

## 13. fixed, stackalloc

[Троелсен-449; Troelsen-448; Нейгел-394; Шилдт-685]

- fixed ներդրված հրամանը նշում է **ավելցուկի հավաքիչին,** որ ցուցիչը օգտվում է հղիչային օբյեկտների անդամներից և դրա համար օբյեկտը չպետք է տեղափոխվի մաքրման ժամանակ:

```
class A
{
    public int a;
}
unsafe class M
{
    static void Main()
    {
        A ob = new A();
        // int* p = &ob.a; //error
        fixed (int* pa = &ob.a)
        {
            *pa = 44;
        }
        System.Console.WriteLine(ob.a);
    }
}
```

- **stackalloc** ներդրված հրամանը ստեկում հատկացնում է հիշողություն, որը աշխատում է չղեկավարվող կոդի օրենքներով: Այդպիսի անհրաժեշտություն կարող է առաջանալ, եթե հղիչային տիպը, որպեսզի ավելի արագ աշխատի, տեղադրվի ոչ թե կույտում, այլ ստեկում: Օրինակ, ինչպես գիտենք զանգվածները աշխատում են կույտում և նրանց արագությունը ունի մեծ նշանակություն: Զանգվածը բերելով ստեկ, կարող ենք ապահովել ավել մեծ արագություն:

```
 unsafe
 {
       char* p = stackalloc char [256];
       for (int k = 0; k < 256; k++)
       p[k] = (char)k;
 }
//.....
```

```
// արագության համեմատում – պետք է լինի ավելի դանդաղ
using System;
class M
{
    static unsafe void Main()
    {
        Console.Write("size = "); // 100000 <= int limit
        int n = int.Parse(Console.ReadLine());
        DateTime dt = DateTime.Now;
        int[] A = new int[n];
        for (int j = 0; j < 1000; j++)
            for (int i = 1; i < n; i++)
                A[i] = A[i] + i;
        Console.WriteLine("barev");
        Console.WriteLine(DateTime.Now - dt);
    }
}
```

```
// արագության համեմատում – պետք է լինի ավելի արագ  - 25-30% -ով
using System;
class M
{
    static unsafe void Main()
    {
        Console.Write("size = "); // 100000 <= 255555 (ստեկի պատճառով փոքր տարածք)
        int n = int.Parse(Console.ReadLine());
        DateTime dt = DateTime.Now;
        int* p = stackalloc int[n];
        for (int j = 0; j < 1000; j++)
            for (int i = 1; i < n; i++)
                p[i] = p[i] + i;
        Console.WriteLine(DateTime.Now - dt);
    }
}
```

```
.....
```

# **Կոմպոնենտ** (Component)

## 14. Կոմպոնենտների ստեղծում

[albdarb; Шилдт.2004-671]

- **DLL** - Այս գաղափարը առաջացել է **միչև կոմպոնենտը** և իմաստը կայանում էր նրանում, որ **միատեղ** օգտագործվի ծրագրային կոդը: Իսկ մինչև դինամիկ կոմպանովկան (dll) եղել է ստատիկ կոմպանովկան (Link): Դա թարգմանված միջանկյալ կոդ էր, որը միացվում էր ծրագրին և աշխատում էր ծրագրի հետ: DLL – ի դեպքում ծրագիրը օգտվում էր պատրաստի երկուական աշխատող կոդից, որը միանում էր ծրագրին **աշխատանքի ժամանակ**, որից կարող են օգտվել այլ ծրագրեր: Կա DLL – ի մեկ կոպիա հիշողությունում: DLL – ում չկա **տիպերի հսկում** և դրա համար նա չէր կարող լրիվ հանդես գալ **որպես կոմպոնենտ** ծրագրերը տարբեր պատրաստի մասերից **հավաքելու** համար: **COM –** ը լուծում է այդ հարցը: Նրա թերությունը կայանում է նրանում, որ COM – ը աշխատում է օպերացիոն համակարգի ռեստրների հետ, որին **թույլատրություն** ունեն բոլոր ծրագրերը և հնարավոր է մի բան փոխել, որը մյուսների համար թույլատրելի չէ: Հաջորդը՝ հաջորդաբար պետք է **ռեսուրսները ազատել**, որը ծրագրավորողի վրա է դրված և մոռանալու դեպքում հնարավոր է լուրջ պրոբլեմներ: Ամենալուրջ թերությունը COM – ի, դա տիպերի համաձայնության թույլ լինելն է և պետք է կիրառել բարդ օրենքներ: Այս թերությունները վերացվում են **CLR համակարգում**:
- COM – ում նույնպես կարելի է փորձել տարբեր լեզուներով գրված ծրագրերը միացնել, սակայն այդ հարցը բազային մակարդակով լուծված է CLR համակարգում ստեզծելով CTS համակարգը: CLR համակարգը ստեղծում է ղեկավարվող կոդ, որը ունի հավաքիչ, հիշողության ղեկավարում, անվտանգություն:
- C# - ով կարելի է ստեղծել կոմպոնենտներ: Այս այնքան կարևոր է, որ երբեմն C# - ը անվանում են **կոմպոնենտային լեզու**: Այստեղ ավելի պարզեցված է օգտագործումը COM տեխնալոգիայի հետ համեմատած:
- **Կոմպոնենտը** դա ավելի բարձր կարգի ծրագրային կոդ է, որը ունի ինքնակառավարվող մաս և կցվելով ծրագրին այն կարող է ռացիոնալ օգտագործվել՝ օրինակ, ոչ անհրաժեշտության դեպքում ինքնաբացարկի ենթարվել, խնայելով ռեսուրսներ: Այն **անկախ** մոդուլ է, որը ստեղծված է ծրագրի բազմակի օգտագործման համար (**code reuse**) և ունի **երկուական ֆորմատ** (այսինքն ծրագարային տեսքը անհասանելի է): Նրա ներքին աշխատանքը **փակ է** արտաքին աշխարհի համար (պարփակվածություն): Այն ծրագիրը, որը օգտագործում է կոմպոնենտին կոչվում է **կլիենտ**: Կոմպոնենտը, որպես առանձին ծրագիր չի կարող հանդես գալ: Որպեսզի կլիենտը կարողանա օգտագործել կոմպոնենտը, պետք է նրանք ունենան նույն օրենքները՝ դա կոչվում է **կոմպոնենտային մոդել**: Ի պատիվ C# -ի հնարավորությունների, ցանկացած կլաս **լրիվ համընկնում** է կոմպոնենտի որոշմանը և թարգմանված երկուական տեսքով կարելի է օգտագործել տարբեր ծրագրային հատվածներում: Սակայն կլասը **դեռ** կոմպոնենտ չէ, որովհետև նա պետք է պահպանի կոմպոնենտային մոդելի **օրենքները:** Դրան կարելի է հասնել շատ հեշտ՝ կլասը պետք է **իրականացնի ինտերֆեյս** – System.ComponentModel.IComponent: Իրականցնելով ինտեֆեյսը, կլասը ստիպված կատարում է բոլոր պարտականությունները:
- IComponent – պարունակում է մեկ հատկանիշ և մեկ իրադարձություն:
- System. ComponentModel.`Component` կլասի օգնությամբ նույնպես կարելի է կլասը դարձնել կոմպոնենտ, քանի որ նա իր մեջ պարունակում է IComponent ինտեֆեյսի ստանդարտ իրականացումը:
- Component կլաս - ունի երկու բաց հատկանիշ - Container և Site
  public IContainer Container {get;}
  public virtual ISite Site { get; set; }
- Component կլասը ունի երկու բաց մեթոդներ ToString(), որը վերավորոշված է և **Dispose()**, որը օգտագործվում է երկու ֆորմայով:

- ✓ public void Dispose() - նախատեսված **ազատել** բոլոր ռեսուրսները, որոնք զբաղեցվում են կոմպոնենտի կանչի ժամանակ:
- ✓ virtual public void Dispose(bool how) - եթե how – ը true է, ապա այս ձևով ազատվում է և ղեկավարվող և չղեկավարվող ռեսուրսները: false – ի դեպքում ազատվում են միայն **ոչ ղեկավարվող ռեսուրսը**:
- ❖ Նշված ծրագիրը ստեղծվում է Class Library project – ի միջոցով և արդյունքում ստացվում է dll ծրագիր: Պարզ է, այդ դեպքում չպետք է ներառել օրինակում բերված Main() ծրագիրը:
- Եթե Component կլասից ժառանգում չլինի, ապա նույնպես կարելի է օրինակում նշված EncodeDecode կլասից օգտվել “Application- աջ քլիք- Add – Existing Item…” ձևով կլասը միացնելով project -ին, սակայն Dispose() **ռեսուրսների ազատումը** հնարավոր չի լինի և չի համապատասխանի **կոմպոնենտային չափանիշներին**:

```
using System;
using component_Encode;
using System.ComponentModel;
class M
{
    public static void Main()
    {
        EncodeDecode ob = new EncodeDecode();
        string text = "simple text";
        string css = ob.Encode(text);
        Console.WriteLine(css);
        string pss = ob.Decode(css);
        Console.WriteLine(pss);
        ob.Dispose();
        Console.ReadKey();
    }
}

namespace component_Encode;
{
    public class EncodeDecode: Component
    {

        public string Encode(string s)
        {
            string temp = "";
            for (int i = 0; i < s.Length; i++)
                temp += (char)(s[i] + 1);
            return temp;
        }
        public string Decode(string s)
        {
            string temp = "";
            for (int i = 0; i < s.Length; i++)
                temp += (char)(s[i] - 1);
            return temp;
        }
    }
}
```

.....

## 15. Հավաքագրման բլոկներ – Assembly (Անկախություն, “Պետականություն”)

[Троелсен-516; Troelsen-603; Нейгел-55, 509, 549; Гриффитс- 632; Албахари-729]

- C# -ը զարգացրել է կոմպոնենտայնության գաղափարը և OOP տեսանկյունից կատարվել է տիպի նոր որակավորում՝ այն դարձնելով **անկախ**: Անկախությունը դրսևորվում է ամբող ծրագրայի տիրույթում տիպի միօրինակության կամ յուրօրինակության ձևով (միակն է աշխարհում): Այսպիսով՝ ցանկացած **տիպ** բնորոշվում է ոչ միայն անունի տարածքով, այլ նաև այսպես կոչված **հավաքագրման** բլոկով: Հավաքագրումը իրականացվում է **ֆայլային համակարգով** տիպերի պահպանումով: Այլ խոսքով՝ dotNet համակարգով ծրագրերը կառուցվում են հավաքագրման բլոկներով: Հավաքագրումը **ղեկավարվող կոդ** է և աշխատում է CLR– ի միջոցով: Այն **տրամաբանական** կառույց է, ունի 2-ական **dll** կամ **exe** կատարման ֆայլային ընդլայնում և տեղափոխելի **Portable Executable** է:
- Հավաքագրումը կարելի է համարել **տիպերի խումբ**: Այն բաձրացնում է **կոդի կրկնակի** օգտագործման (**code reuse**) հնարավորությունը **կոմպոնենտային** մակարդակում: Ծրագրերը, որոնք կարող են ստեղծվել **տարբեր լեզուներով**, հավաքագրման բլոկների շնորհիվ հնարավորություն ունեն միասնական հանդես գալ ծրագրային պրոյեկտում՝ օգտագործելով CLR –ում ընդունված տիպային համապատասխանության օրենքները: Հավաքագրուման շնորհիվ նույն **անունով կլասները** կարող է դիտարկվել որպես **տարբեր տիպեր**:
- Հավաքագրումը **ինքնանկարագրվող** է (**self-describing**), որի շնորհիվ պարտադիր չէ ծրագրի գրանցումը օպերացիոն համակարգի **ռեեստրներում**: Ինքնանկարագրումը կատարվում է “մանիֆեստի” և “**մետատվյալների**” միջոցով, ապահովելով ինչպես այլ հավաքագրման բլոկների հետ կապը, այնպես էլ ներքին տվյալների նկարագրության իրականացումը:
- Հավաքագրումը կարող է տեղակայվել **“private”** կամ **“shared”:** Առաջինի դեպքում հավաքագրումը տեղադրվում է նույն կատալոգում, որտեղ գտնվում է նրան օգտագործող կլիենտային ծրագիրը: Երկրորդի դեպքում հավաքագրման բլոկը տեղակայվում հատուկ տեղում, որ կոչվում է GAC կատալոգ – **Global Assembly Cash**: GAC կատալոգից կարող են օգտվել նաև այլ ծրագրեր, որոնք իրականացնում են խնդրի համատեղ լուծման պրոյեկտում:
- Հավաքագրումը ավտոմատ կերպով դառնում է նաև **ծրագրային վերսիաների** միավոր: Վերսիան ներկայցվում է 4 թվանշանների միջոցով: <major>.<minor>.<build>.<revision >: Եթե AssemblyVersion հատկանիշով չենք վերագրում վերսիան, ապա լռելիությամբ ստացվում է (1,0,0,0):
- Առաջարկվող օրինակը ցույց է տալիս, որ ցանկացած տիպ հավաքագրման բլոկի շնորհիվ դառնում է յուրահատուկ, նույնիսկ, եթե արհեստական տիպ ենք ստեղծում, որը արդեն գոյություն ունի .NET գրադարանում: [Гриффитс- 640]

```
        System.String s = null;
        Show(s);     // our string
        string s2 = null;
        Show(s2);          // library string
        Show("barev");   // library string
        Show(Environment.OSVersion.VersionString); // library string
    static void Show<T>(T item)
    {
        Type t = typeof(T);
        Console.WriteLine(t.FullName);
        Console.WriteLine(t.Assembly.FullName);
    }

namespace System
{
    class String
    {       }
}
```

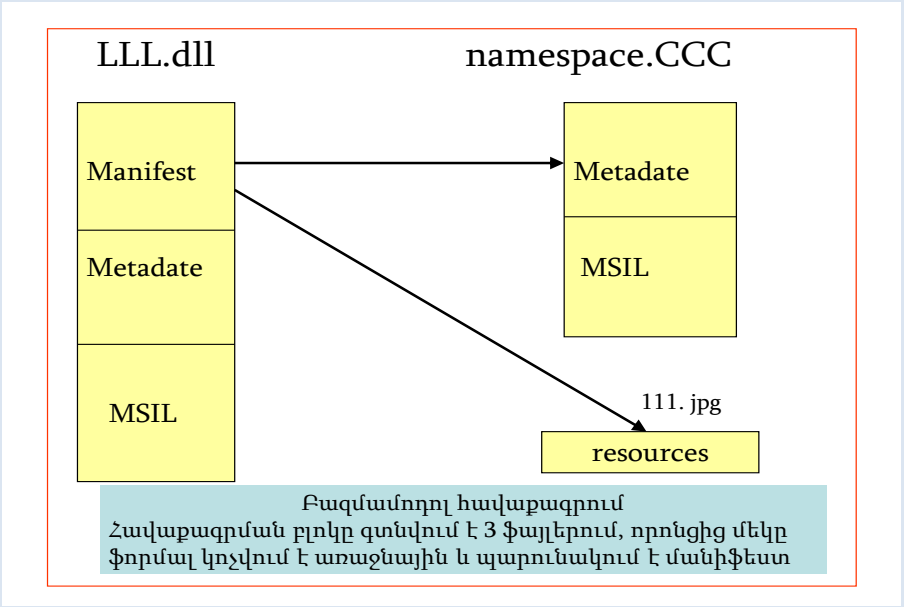
.....

# 16. Հավաքագրման բլոկի կառուցվածքը

[Троелсен-518; Troelsen-604; Нейгел-510 ]

- Հավաքագրման բլոկը ունի հետևյալ բաղկացուցիչ մասերը՝
- **Windws ֆայլի վերնագիրը**: Նշվում է, որ բլոկը կարող է աշխատել Windows օպերացիոն համակարգի հետ: Որոշվում է նաև ծրագրի տիպը: Կոնսոլային, գրաֆիկական կամ dll գրադարան: **dumpbin.com** ուտիլիտով կարելի է տեսնել պարունակությունը: (օրինակ. հավաքելով dumpbin/headers հրամանը և հավաքագրման բլոկի անունը): **Ներքին կառույց է** և անհրաժեշտ է միջամտել միայն կոմպիլիատրում **նոր լեզվի ավելացման** դեպքում:
- **CLR ֆայլի վերնագիր:** Պարունակում է տվյալներ, որոնք պետք է կիրառեն բոլոր հավաքագրման բլոկները: Կազմակերպվում է ֆլագների միջոցով: Օրինակ, նշվում է որտեղ է գտնվում մետատվյալները և ռեսուրսները, ինչպիսի տեսք ունի ծրագրային վերսիան: Բաց բանալու պարունակությունը: (օրինակ. հավաքելով dumpbin /clrheaders հրամանը և հավաքագրման բլոկի անունը): Նույնպես կարելի է դիտել **dumpbin.com** ուտիլիտով: Նույնպես հիմնականում թագցվում է ծրագրավորողից:
- **CIL**: Հավաքագրուման բլոկը պարունակում է Comon Intermediate Language լեզվով ծրագրային կոդ: (Троелсен-57)
- **Metadata** - պարունակում է բոլոր **տիպերի նկարագրությունները** տվյալ հավաքիչում: (Троелсен-59)

✓ Օրինակ, կլասը նկարագրելու համար նշվում է նրա բոլոր դաշտերը, մեթոդները, հատկանիշները, իրադարձությունները, բազային կլասները, ինտրեֆեյսները: Թարգմանիչը մետատվյալները ստեղծում է ավտոմատ:




- **Manifest** – նույնպես մետատվյալներ են **հավաքագրման բլոկնեի** մասին, բայց ավելի **ընդհանուր**, պարունակում է նկարագրություններ հավաքիչի մասին (Троелсен-60)

Հավքագրման մանիֆեստը պարունակում է.

✓ **վերսիայի** իդենտիֆիկատորը

✓ բոլոր **ներքին մոդուլների** ցուցակը

✓ **արտաքին մոդուլների** ցանկը, որոնք մասնակցում են ծրագրի աշխատանքի ժամանակ

✓ հավաքագրման բլոկի **խիստ անունը**

✓ անվտանգության ինֆորմացիա (ոչ պարտադիր)

✓ ռեսուրսների մասին ինֆորմացիա (ոչ պարտադիր)

- **Resources** – ոչ պարտադիր մաս, որը կարող է տվյալներ պարունակի, օրինակ, գրաֆիկական, ձայնային և այլն: Ռեսուրսներից կարելի է օգտվել, այն ավելացնելով նախագծում և կիրառել `GetManifestResourceStream()` մեթոդը: (Гриффитс- 635)

- Հավաքագրումը կարող է լինել **միամոդուլ** – մեկ ֆայլում, և **բազմամոդուլ** - բազմակի ֆայլերում: Միամոդուլի դեպքում առկա է **մեկ մանիֆեստ** և **մեկ մետատվյալ**: Բազմամոդուլի դեպքում գոյություն ունի **մեկ մանիֆեստ** իր մետատվյալով և **բազում** մետատվյալներով ֆայլեր: Բացմամոդուլ տարբերակը հիմնականում չի օգտագործվում: Նրա անհրաժեշտությունը կարող է առաջանալ, երբ **չենք ցանկանում CLR** -ը ձևավորի հավաքագրման ամբողջ տվյալները, այլ ցանկանում ենք ստանալ **միայն անհրաժեշտության** դեպքում: Սակայն կարելի է նշել, որ **նույն էֆեկտին** կարելի է հասնել խնդրի այդ մասը բաժանելով մի քանի հավաքագրման բլոկների, որին CLR ը արդեն կանչում է անհրաժեշտության դեպքում: Հիմնականում, բազմամոդուլի անհրաժեշտություն կլինի, եթե **տարբեր լեզուներով** է գրվում ծրագրային հատվածը և քանի որ, `.NET` նախկին լեզուների `Link` միացնող տեխնոլոգիան չունի:
- Նշենք հավաքագրման որոշ առավելություններ՝

✓ Առանձին ծրագրային միավոր է և անհրաժեշտություն չկա **առանձին ռեեստրներում** ինֆորմացիա գրել: (**Ռեեստրը**, դա “կոնֆիգ” ֆայլերի ընդհանուր տեղ է COM տեխնոլոգիաներում: Մինչև COM –ը կոնֆիգուրացիաները պահվել են .INI ֆայլերում):

✓ Մանիֆեստը հնարավորություն է տալիս ստանալ ծրագրային վերսիա և ազատում է **DLL Hell կոնֆլիկտային** վիճակից և նույնիսկ թույլ է տալիս **զուգահեռ** վերսիաների կատարում միևնույն պրոցեսի սահմաններում:

✓ Պրոցեսի բաժանումը **դոմենի** (կի)՝ դա մի նոր ծրագրային միավոր է, որտեղ բարձրացված է պաշտպանվածությունը: Պրոցեսի սահմաններում կարող են գործել մի քանի դոմեններ և մի դոմենից մյուսին դիմելու համար անհրաժեշտ է **proxy - օբյեկտ** անցում : Հավաքագրումը կանչվում է դոմենի մեջ և դոմենների շնորհիվ կարելի է մեկ պրոցեսում ունենալ տարբեր ծրագրային հատվածներ: (Нейгел-523)

✓ Ծրագրային բեռնավորումը կարելի է կատարել **հասարակ կոպիայի** միջոցով – xcopy:

❖ Գործում է internal և protected intenal, որը նշանակում է կլասի անդամները հասանելի են միայն **տվյալ հավաքագրման** բլոկում ինչպես նաև կարելի դիմել օբյեկտով:

❖ Գործում է private protected (new 7.2) [Troelsen-196]: Հնարավորություն է տալիս կլասի անդամին օգտագործել միայն տվյալ հավաքագրման բլոկում, պարզ է, եթե գործում է ժառանգականության կապը: Սակայն հնարավոր չէ օբյեկտով դիմելը՝ ի տարբերություն protected intenal -ի:

```
Console.ReadKey();
class A
{
    protected int a1;
    internal protected int a2;
    private protected int a3;
}
class B : A
{
    public void f()
    {
        a1 = 55; //ok
        a2 = 55; //ok
        a3 = 55; //ok
        A ob = new A();
        //ob.a1=77; // error
        ob.a2 = 77; // ok
        //ob.a3=77; //error
    }
}
class AA
{     }
public class BB
{      // public AA ob; //error internal ok   }
```

.....

# 17. Փակ հավաքագրման բլոկ

[Троелсен-532; Рихтер-88,110]

- Սովորական ծրագրավորումը .NET միջավայրում ստեղծում է հավաքագրման բլոկներ, որոնք կոչվում են **փակ հավաքագրում** (private) և նրա ֆայլերը տեղադրվում են **այն կատալոգում**, որտեղ գրված է ծրագիրը:
- Ծրագրերի բլոկների հեռացնելը կամ կոպիան կատարվում շատ առանց դժվարությունների՝ ուղղակի ձևով: Այն չի կարող **խանգարել** կամ **վնասել** այլ ծրագրերի աշխատանքներին: Գործում է internal թույլատրության մոդիֆիկատորը:
- Եթե ծրագիրը ունի հղում այլ հավաքագրման բլոկներ, ապա կատարման ժամանակ նրանց կոպիաները **բերվում է ծրագրի տարածք**:
- Փակ հավագագրումը ունի սովորական **անուն** և **վերսիայի համար**, որը գտնվում է **մանիֆեստում**: Օրինակ, եթե ունենք `App.dll` հավաքագրումը, ապա նրա մանիֆեստում `ildasm` -ով կարելի է հայտնաբերել հետևյալը.

```
.assembly App
{
   //.....................
   ver 1:0:0:0
}
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\ildasm.exe"
```

- Փակ բլոկները մեկուսացված են ընդհանուր ծրագրային աշխարհից, դրա համար CLR ը հաշվի չի առնում վերսիայի համարը բլոկի ֆիզիկական տեղը ընտրելիս:
- **Probing Process** – դա պրոցես է, երբ անհրաժեշտ է **որոշել հավաքագրման** երկուական ֆայլի **տեղը**: Տեղը որոշվում է ոչ բացահայտ կամ բացահայտ (implicit or ()explicit ) ձևով:

✓ **Ոչ բացահայտի** դեպքում CLR ը նայում է **մանիֆեստին**, որպեսզի իմանա, թե որտեղ է գտնվում անհրաժեշտ հավաքագրման բլոկը, որը կապված եղել է References –ի միջոցով: Մանիֆեստում այն ունի `extern` մոդիֆիկատորը և ունի հետևյալ տեսքը.

```
.assembly  extern App
 {            }
```

✓ **Բացահայտի** դեպքում իրագործումը կատարվում է Load() կամ LoadFrom() մեթոդների միջոցով, որոնք գտնվում են System.Reflection.Assembly կլասում: Դա կատարվում է հիմնականում **դինամիկ** կանչի համար, ծրագրի **կատարման** ժամանակ: [Гриффитс- 645]

```
Assembly asm = Assembly.Load("App");
Assembly asm = Assembly.LoadFrom("C:\\App.dll");
```

Եթե CLR ը ոչ մի ձևով չի հայտնաբերում հավաքագրման բլոկի անունը, ապա առաջանում **արտակարգ** իրավիճակ `FileNotFoundException`:

✓ CLR -ին օգնելու համար գոյություն ունի **`*.config`** ֆայլ (Օրինակ՝ ConsoleApp1.exe.config հավաքվում է որևէ խմբագրիչով), որտեղ նշվում է թե էլ որտեղ կարելի է գտնել հավաքագրման բլոկի անունը:

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
              <probing privatePath="folder1; folder2\alb; folder3\highcode"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

- Եթե CLR -ը **չի հայտնաբերում** DLL ֆայլերը, ապա **նորից է** սկսվում փնտրում արդեն EXE ֆայլերը և անհաջողության դեպքում առաջանում է `FileNotFoundException`:
- `*.config` "կոնֆիգ" ֆայլերը կարելի է ստեղծել ծրագրի նախագծման ժամանակ Project -> Add New Item ով:

.....

# 18. Տեղաբաշխված հավաքագրման բլոկ

[Троелсен-537; Troelsen-625; Нейгел-525,562; Рихтер-96]

- **Global Assembly Cache (GAC):** Ցանկացած տեղաբաշխված (shared) հավաքագրման բլոկ, դա տիպերի խումբ է, որին կարելի է օգտագործել **բազմակի** անգամ: Նրա գլխավոր առավելությունը և տարբերությունը փակից (private), որ մեկ օրինակով է հանդես գալիս և կարող են նույն կոմպոնենտից օգտվել **տարբեր** ծրագրեր: Նշենք, որ փակի հավաքագրման դեպքում Reference -ով կոմպոնենտը կոպիայի ձևով ընդգրկվում է պրոյեկտում:
- Հիմնականում ընդհանուր օգտագործվող ծրագրային բլոկները ժամանակ առ ժամանակ վերափոխվում են ավելի **նոր վերսիաների** և հարց է առաջանում ինչպես կազմակերպել, որպեսզի այն չվնասի այդ բլոկներից օգտվող ծրագրերին: Ծրագրի հավաքագրման բլոկը պետք է լինի փակ թե տեղաբաշխված, որոշվում է **նախագծման** էտապում:
- Ինչպես գիտենք GAC -ը դա հավաքագրման բլոկների պահպանման տեղ է, որին կարող են տեսնել և դիմել այլ ծրագրեր: Հին վերսիաներում այն գտնվում է `C:\Windows\assembly` կատալոգում: GAC –ում թույլատրվում է պատրաստի ծրագրային ֆայլերը պահել GAC կատալոգում, որոնք ունեն .dll ընդլայնում**: Չի թույլատրվում** .exe ընդլայնումով ֆայլեր պահել:
- Ծրագրային փաթեթ կարող են ներկայացնել առանձին ծրագրավորողները, կարող է լինել ծրագրավորման բաժին կամ նույնիսկ առանձին հիմնարկություն:

❖ **Strong Names** (խիստ անուն): Այն կազմակերպում է տեղաբաշխված հավաքագրման բլոկի իրականացումը: Քանի որ ընդհանուր օգտագործման միջավայրում ենք աշխատում, ապա անհրաժեշտ է բարձրացնել պաշտպանվածությունը և **միտումնավոր** և **պատահական** վնասի կանխարգելման համար: Երկու անկախ ծրագրային համակարգեր հնարավոր է ընդհանուր օգտակործման տեղում ներդնեն **նույն անունով** ծրագրեր, եթե օգտագործենք **խիստ անվան** հնարավորությունը: Քանի որ, GAC –ն նախատեսված է համատեղ ծրագրի բլոկ տեղակայելու համար, ապա անհրաժեշտ է ստեղծել ծրագրային փաթեթի երկուական .NET ֆայլի խիստ անունը:

- **Խիստ անունը** ձևավորելու համար օգտագործվում է **կրիպտոգրաֆիա RSA** ծածկագրումը: RSA -ն աշխատում է **զույք բանալիներով**՝ բաց և գաղտնի: **Գաղտնագրումը** կատարվում է բաց բանալով: **Վերծանումը** կատարվում է գաղտնի բանալով:
- Խիստ անունով հավաքագրման բլոկը կազմված է հետևյալ տվյալներից.

✓ հավաքագրման **բլոկի անունը**:

✓ հավաքագրման բլոկի **վերսիան,** որը նշվում AssemblyInfo.cs ֆայլի `[AssemblyVersion]` ատրիբուտում: Օրինակ. `[assembly: AssemblyVersion("10.2.1.0")]`

✓ **բաց բանալին**, որի ճանապարհը կարելի է նշել AssemblyInfo.cs ֆայլի `[AssemblyKeyFile]` ատրիբուտով: Օրինակ. `[assembly: AssemblyKeyFile(@"C:\file.snk")]`

Քանի որ, բաց բանալին շատ մեծ թիվ է իրենից ներկայացնում, ապա նրա փոխարեն ատրիբուտներում օգտագործվում են նրա **"հեշ" կոմպակտ թիվը**, որը կոչվում է 64 բիթանոց `public key token`:

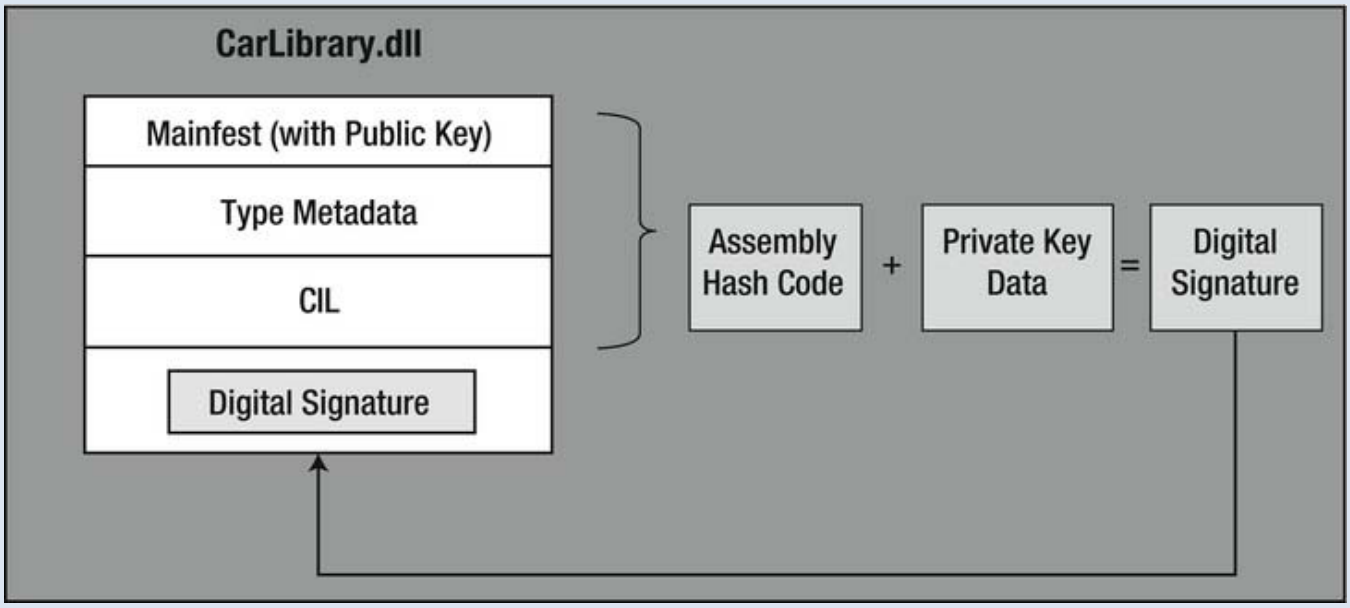
✓ լոկալացման համար այսպես կոչված "**կուլտուրան**", որը նշվում `[AssemblyCulture]` ատրիբուտում: (որոշվում է ֆոնտի **լեզուն** և ոչ պարտադիր է)

✓ ներդրված **թվային ստորագրություն,** որը օգտագործում է հավաքագրման բլոկի "**հեշ**" նշանակությունը և **գաղտնիության** բանալին:

- Բացատրենք ինչ է **հեշ կոդ :** Այն յուրահատուկ թիվ է և առաջանում է տվյալ հավաքագրման բլոկի IL հրամաններից և մետատվյալներից և կցվում է հավաքագրման բլոկին: Ցանկացած փոփոխություն ծրագրում բերում է նոր "հեշ" կոդի գեներացման:
- Խիստ անուն ստանալու համար **`sn.exe`** (Strong Name) ծրագրի օգնությամբ գեներացվում է **բաց և գաղտնի** բանալիները: Ծրագիրը ստեղծում է ֆայլ **`*.snk`** (Strong Name Key) անունով: Ստեղծվում է երկու տարբեր, սակայն մաթեմատիկորեն կապված բանալիներ: Թարգմանիչը գեներացնում է նաև հավաքագրման բլոկի "հեշ" կոդ: Հետո կատարվում միավորում "հեշ" կոդի և գաղտնի բանալիի հետ, որպեսզի ստացվի **թվային գաղտնի ստորագրությունը**: Երբ վեջացվում է խիստ անունի ձևավորումը

բաց բանալին ընդգրկվում է **մանիֆեստ:** Մանիֆեստում **գաղտնի բանալին չի գրանցվում**: Այնտեղ գրանցվում է միայն **թվային ստորագրությունը**:

- Signing




- **Խիստ անվան** ամբողջ **էությունը** կայանում է նրանում, որ **ոչ մի երկու** ծրագրավորող, բաժին կամ "կամպանիա" **չեն ունենա նույնականացում** ամբողջ .NET աշխարհում և ինչպես նշեցինք խիստ անունը գեներացվում է ծածկագրման **կրիպտոգրաֆիկ** եղանակով:

❖ "Խիստ անվան" կոդ կարելի է գեներացնել Windoaws Command Prompt -ում **`sn.exe`** ուտիլիտով՝

✓ C:\WINDOWS\system32> **cd** C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools

✓ `sn –k C:\1\albdarb.snk`

✓ `sn –tp C:\1\albdarb.snk` հրամանով կարելի է տեսնել բաց և `token` բանալիները:

✓ Կարող ենք `AssemblyInfo.cs` ֆայլին ավելացնենք `[AssemblyKeyFile]` ատրիբուտը, որը ցույց կտա `albdarb.snk` **ֆայլի տեղը**: Օրինակ.
`[assembly: AssemblyKeyFile(@"C:\albdarb.snk")]`

- Քանի որ խիստ անվան բաղկացուցիչ մասերից է նաև հավաքագրման **վերսիան**, ապա նրա ընտրությունը պարտադիր է: Նրա սկզբանկան արժեքն է
`[assembly: AssemblyVersion("1.0.0.0")]`

✓ <major>.<minor>.<build>.<revision >: վերսիայի գլխավոր համար, լրացուցիչի համար, թարգմանման համար և վերանայման համար: Հիմնականում վերսիայի թվերը ծրագրավորողի վրա է, սակայն կարելի է նաև ավտոմատ գեներացնել ասենք ամեն թարգմանման համար: Թվերի թույլատրելի միջակայքն է 0 - 65535:

- Եթե `[AssemblyCulture]` չի նշվում, ապա ժառանգվում է այն կուլտուրան որը դրված ընթացիք համակարգչում: Կուլտուրան հիմնականում նշում է **լեզուն** (en, ru,….):
- Ավարտելուց հետո `ildasm.exe` ծրագրով կարելի է դիտել մանիֆեստը: Կտեսնենք, որ **`.publickey`** դիսկրիպտորը պարունակում է բաց բանալին, իսկ **`.ver`** -ն պահում է վերսիայի համարը:
- **Առանձնահատկություն:** Խիստ անունով ծրագիրը չի կարող References ունենալ առանց խիստ անվան ծրագրերի հետ: [Албахари]

❖ "Խիստ անվան" կոդ կարելի է գեներացնել և տրամադրել ծրագրին **նաև** Visual Studio -ի միջոցով: Solution Explorer -> Properties -> Signing -> Sign the assembly -> check box-> նշել <New...> կամ արդեն `sn.exe –k` ով գեներացված կոդի անունը:

❖ Երբ վեջացվում է խիստ անունների ձևավորումը կատարվում հավաքագրման բլոկի **գրանցումը** GAC ում: GAC –ում "խիստ անունով" կոդը տեղակայելու համար չի թույլատրվում ձեռքով տեղադրումը, այլ օգտագործվում է `gacutil.exe` ուտիլիտը և –i բանալին:

`gacutil – i  AAA.dll` կամ `gacutil  /i  AAA.dll`
նշենք նաև, որ կա – u և - l բանալիներ, որոնք համապատասխանաբար հեռացնում են հավաքագրման բլոկը GAC ից  կամ ցուցադրում են  հավաքագրման բլոկների ցուցակը GAC ում:

- Խիստ անունով հավաքագրումը նույնպես ունի **Probing Process** և **`*.config`** ֆայլերի միջոցով կարող է հղում կատարել հավաքագրման բլոկներին, օգտագործելով  ինչպես <publisherPolicy> տեգը այնպես էլ կիրառել այլ տեգեր՝ օրինակ <codeBase>: [Троелсен-546; Нейгел-572]

❖ GAC –ի գրանցման օրինակ (VS 2022 (Core, Framework)): (Պրակտիկ)

- **Խիստ անունի գեներացիա** [Троелсен-542]

1. Framework: VS 2022 –ով: Սկզբից ստեղծում ենք խիստ անուն *.snk ֆայլի ընդլայնումով: Դրա համար վերցնենք որևէ ծրագիր և ստանանք նրա համար խիստ անուն՝ Solution Explorer-> Properties-> Signing->  Sign the assembly (նշենք new) և Key file name -ում գրենք խիստ անվան անուն: Ok -ից հետո պրոյեկտում կավելանա խիստ անունը:
2. Core –ի դեպքում sn.exe –k ուտիլիտով ստեղծում ենք և նոր խիստ անուն և նոր ստեղծվող ծրագրում՝ Properties->Build->Strong Name->Sign the assembly –ով նշում են արդեն գեներացված խիստ անունի ֆայլը և կապում նրա հետ:

- **Գրանցումը GAC –ում** [Троелсен-543]
- GAC  -ը կրող է տեղակայվել  C:\Windows\Microsoft.NET\assembly\GAC_MSIL բաժնում:
- Այնուհերև կատարենք `gacutil.exe` ծրագրի միջոցով GAC -ում գրանցում Windows ադմինիստրատորի ռեժիմում՝
  Windows->Search->Command Prompt->Admin->Run as Administrator (աջ մասում)->click
  Կիրառենք Command Prompt -ում **cd** (change directory) հրամանը `gacutil.exe` ծրագրի ֆոլդեր մտնելու համար՝

✓ C:\WINDOWS\system32> **cd** C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools

✓ Այնուհետև աշխատացնենք `gacutil –i` ուտիլիտը նշելով GAC ուղղարկվող կոմպոնենտի տեղը՝

C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools>gacutil -i C:\1\component_Encode.dll

✓ ստանում ենք պատասխան.  gacutil -i C:\1\ConsoleApp8.exe
All rights reserved.
Assembly successfully added to the cach

- Այնուհետև ստեղծենք նոր ծրագիր և GAC -ում գրանցված component_Encode հավաքագրման բլոկին դիմենք References –ով: GAC –ն գտնվում է հետևյալ կատալոգում՝
  C:\Windows\Microsoft.NET\assembly\GAC_MSIL\component_Encode և Refrence -ով կկապենք նոր ծրագրին:
- component_Encode ծրագրի կլասները պետք է լինեն `public`:

```
using System;
using component_Encode;
class Program
{
    static void Main(string[] args)
    {
        EncodeDecode ob = new EncodeDecode();
        Console.WriteLine(ob.Encode("albert"));
        Console.WriteLine(ob.Decode("bmcfsu"));
        Console.ReadKey();
    }
}
```

- **Առանձնահատկություն:** [Троелсен-544] Framework –ում մեր ստեղծած component_Encode կոմպոնենտը լռելիությամբ լոկալ կոպիայով է ընդգրկվում պրոյեկտ: Եթե անհրաժեշտ է GAC –ում գոյություն ունեցող կոպիայի հետ աշխատել, անհրաժեշտ է References-> component_Encode-> աջ քլիկ-> Properties->Copy Local = false: (2019 Core –ում այդպես չէ):
- Cood Lack!!!!!

# 19. Ատրիբուտներ

[Шилдт-562; Нейгел-400,433; Троелсен-578; Troelsen-648]

- Ինչպես հավաքագրման բլոկները կարող են ներկայացնել **մետատվյալներ**, այնպես էլ ատրիբուտները նույնպես հնարավորություն ունեն **օբյեկտների** համար ներառել **ուղեկցող տվյալներ**, որոնք **հավաքագրման բլոկում** հանդես են գալիս որպես **լրացուցիչ** տվյալաներ: Ատրիբուտները ծրագրում ինֆորմացիա են ավելացնում **նկարագրման** ձևով՝ փոխելով կամ ավելացնելով տիպին նոր **հատկություններ**: Այն ընդգրկում է **լրացուցիչ** տեղեկություններ կլասի, ստրուկտուրայի, մեթոդի, դաշտի մասին: Օգտագործվում է [ ] այս փակագծերը և դրվում է այն էլեմենտից առաջ, որին նա վերաբերվում է: Թույլատրվում է **մեկից ավելի** ատրիբուտներ միևնույն տիպի համար, որոնք կարելի է տարանջատվել կամ ստորակետով կամ [ ] փակագծերով: Նա **չի համարվում կլասի անդամ**:
- Ատրիբուտը համարվում է `System.Attribute` կլասից ժառանգված: Եթե ստեղծվել է ատրիբուտ որևէ անվամբ, ապա թարգմանիչը ներքին կարգով նրա վերջում ավելացնում է `Attribute` բառը: Կան **ստանդարտ** ատրիբուտներ և **օգտագործողի կողմից** ստեղծվող: Ստանդարտ ատրիբուտները գոյություն ունեն գրադարաններում: Նրանց օգտագործման օրինակներից են `[CLSCompilant]` ատրիբուտը, որից օգտվելու դեպքում թարգմանիչը ստուգում է նշված տիպերը համապատասխանում են արդյոք CLS համակարգին: Կամ, օրինակ `[DllImport]` ատրիբուտը թույլատրում է օգտվել ցանկացած չղեկավարվող C++ կամ Win Api ծրագրերից: Կամ `[Serializable]` ատրիբուտը, որը ապահովում օբյեկտի ամբողջական ֆայլային մուտք/ելքը:
- **Օգտագործողի կողմից** ստեղծելու դեպքում օգտվում ենք `AttributeUsage` ստանդարտ ատրիբուտից: Նա կարող է ցույց տալ, թէ ինչ նպատակի է ծառայում: Օրինակ – `All, Assembly, Class, Constructor, Delegate, Enum, Event, Field, Interface, Method, Parametr, Property, ReturnVelue, Struct:`
- Տվյալ կլասի ատրիբուտներին դիմելու համար օգտագրծվում է **Type** կլասը:

❖ Նոր ատրիբուտի **ստեղծումը:**

```
[AttributeUsage(AttributeTargets.All)]
public class albAttribute : Attribute
{     string priremark;
      public albAttribute(string comment)
      {
              ptiremark = comment;
      }
      public string remark
      {
              get  { return priremark; }
      }
}
```

❖ Ատրիբուտին **կցումը**

```
[albAttribute("This class uses an attribute.")]
class B
{       }
```

❖ Օբյեկտի ատրիբուտների ստացումը (նրա օգտագործումը): Այն բանից հետո, երբ ատրիբուտը միացված է էլեմենտին, ծրագրի այլ մասերը կարող են նրան կարդալ (կամ դիմել): Դրա համար կա երկու մեթոդ:

✓ **Առաջինը** `GetCustomAttributes()` մեթոդը, որը որոշված է `MemberInfo` կլասում և ժառանգված `Type` կլասի կողմից: Նա կարդում է բոլոր ատրիբուտների ցուցակը, որը կապված է էլեմենտի հետ այս ֆարմատով:

```
object[] GetCustomAttributes(bool searchBases);
```

`true` – ի դեպքում իր մեջ ընդգրկում է նաև ամբողջ ժառանգված կլասի հիերարխիան, հակառակ դեպքում միայն տվյալ տիպի սահմաններում:

- **Երկրորդը** `GetCustomAttributes()` մեթոդը, որը որոշված է `Attribut` կլասում և նրա կանչման ֆորմաներից մեկը:
  `static Attribute GetCustomAttributes(MemberInfo mi, Type attribtype);`

✓ mi – ն նկարագրում է էլեմենտը, **որի համար** հանվում է ատրիբուտը (B կլաս): Գործող ատրիբուտը նշվում է `attribtype` արգումենտով: Այս մեթոդը օգտագործվում է, երբ **հայտնի է ատրիբուտի անունը** (օրինակում `Alb`), որը պետք է ստանալ: Օրինակ, եթե կա `B` կլաս, որը ունի `albAttribute` ատրիբուտ, ապա ատրիբուտին կարելի է հղել օգտագործելով հետևյալ հաջորդականությունը՝

```
Type t = typeof(B);
Type taa = typeof(albAttribute);
albAttribute aa = (albAttribute)Attribute.GetCustomAttribute(t, taa);
```

- Ունենալով հղում ատրիբուտին, կարելի է կապնվել նրա էլեմենտների հետ: Այլ խոսքով ինֆորմացիան, որը կապված է ատրիբուտի հետ մատչելի է ծրագրին:
- Նայելով մետատվյալներին ILDasm - ով (եթե Core, ապա .dll ֆայլ) կարելի է համոզվել ատրիբուտների գոյության մասին:
- Ատրիբուտի պարամետրերից է՝

✓ `Inherited = true` նշանակում է ատրիբուտը կիրառվում տվյալ կլասի բոլոր ժառանգների վրա:

✓ `AllowMultiple = true` նշանակում է նույն էլեմենտի վրա կարելի է կիրառել տվյալ ատրիբուտը մեկից ավել անգամ:

```
using System;
[AttributeUsage(AttributeTargets.All, AllowMultiple = true, Inherited = false)]
class Alb : Attribute
{
    string s;
    public Alb(string comment)
    {        s = comment;    }
    public string nkaragrel
    {
        get {
            //if (s.Length == 6)
            //    s += "-2017";
            return s;
        }
    }
}
[Serializable]
[Alb("....albdarb.com.....")]
//[Alb("555")] // compile error if AllowMultiple = false
class B
{      }
class M
{
    public static void Main()
    {
        Type t = typeof(B);
        object[] attribs = t.GetCustomAttributes(false); // առաջին ձևի արդյունք
        foreach (object v in attribs)
            Console.WriteLine(v);
        Console.WriteLine("------------------------------");
        Type taa = typeof(Alb);
        Alb aa = (Alb)Attribute.GetCustomAttribute(t, taa); // երկրորդ ձևի արդյունք
        Console.WriteLine(aa.nkaragrel);
        Console.WriteLine(aa.TypeId);
    }
}
```

❖ Attributes on lambdas (C#10) N

# 20. Ֆիքսված և անվանական պարամետրերով ատրիբուտներ

[Шилдт -566]

- Ատրիբուտում կարող է նշված լինել պարամետրեր: Նրանք կարող եմ լինել ֆիքսված և անվանական: Սկզբից նշվում են ֆիքսված պարամետրերը, որից հետո թույլատրվում է անվանական պարամետրերի ներկայացում: Անվանական պարամետրերը նշվում են **վերագրման** օպերատրով: Ֆիքսված պարամետրերի հերթականություն ունի նշանակություն, իսկ անվանականը, ոչ:
- Ատրիբուտի պարամետրերը կարելի է արժեքավորել **միայն կոնստրուկտորում** կամ ատրիբուտի հետ վերագրման օպերատրով, եթե անվանական է:
- Անվանական պարմետրերը ներկայացվում են public դաշտերով կամ հատկանիշներով:

```
using System; //Schildt 571rus504eng
using System.Reflection;
[AttributeUsage(AttributeTargets.All)]
public class albAttribute : Attribute
{
    string s; // comment
    public string s_named; // named parameter
    public int p_named { get; set; } // named parameter
    public albAttribute(string comment)
    {
        s = comment;
        s_named = "aaa";
    }
    public string pp
    {
        get
        {
            //if (s.Length == 6)
            //    s+= "-wwwwwww";
            return s;
        }
    }
}
[albAttribute("classB", s_named = "additional info")]
class B
{
    // ...
}
class M
{
    static void Main()
    {
        Type t = typeof(B);
        Console.Write("Attributes in " + t.Name + ": ");
        object[] attribs = t.GetCustomAttributes(false);
        foreach (object v in attribs)
            Console.WriteLine(v);
        Type ta = typeof(albAttribute);
        albAttribute ra = (albAttribute)Attribute.GetCustomAttribute(t, ta);
        Console.WriteLine(ra.pp);
        Console.WriteLine(ra.s_named);
        Console.WriteLine(ra.p_named);
        Console.ReadKey();
    }
}
```

.....

# 21. Ներդրված ատրիբուտներ

[Шилдт - 570]

❖ Ներդրված ատրիբուտներից է՝ **[AttributeUsage]** կամ **[AttributeUsageAttribute]** ատրիբուտը, որը հնարավորություն է տալիս ստեղծել նոր ատրիբուտներ: Ունի հետևյալ կոնստրուկտորը **AttributeUsage(**`AttributeTargets validOn`**)**: Նրա պարամետրը **AttributeTargets** թվարկումն է, որի անդամներն են՝ `All; Assembly; Class; Constructor; Delegate; Enum; Event; Field; GenericParameter; Interface; Method; Module; Parameter; Property; ReturnValue; Struct.`

✓ Կարելի է երկու և ավելի անդամներ տրամաբանական կամով միացնել: `AttributeTargets.Field | AttributeTargets.Property`

✓ **AttributeUsage** ատրբուտը պարունակում է երկու անվանական պարամետրեր `AllowMultiple` և `Inherited`, որոնք ստանում են տրամաբանական արժեք: Եթե `AllowMultiple true` է, ապա ատրիբուտը կարելի է կիրառել միևնույն էլեմենտի վրա բազում անգամ: Եթե `Inherited true` է, ապա ատրիբուտը կիրառվում տվյալ կլասի բոլոր ժառանգների վրա: Լռելիությամբ `AllowMultiple = false Inherited = true`:

✓ **AttributeUsage** ատրիբուտը ունի նաև `ValidOn` հատկանիշ, որը վերադարձնում է `AttributeTargets` թվարկման էլեմենտ: Լռելիությամբ այն `AttributeTargets.All` է:

❖ **[System.Flags]** Թվարկման ֆորմատավորում [Шилдт -830]; Flags, Bitwise Operations [Troelsen-140]

- [Flags] ատրիբուտը հնարավորություն է տալիս անդամներին միավորել ֆլագով, իրականացնելով բիթային օպերացիաներ: Օրինակ՝ թվարկումից կարդալ մեկից ավել արժեքներ, եթե թվարկման անդամները **երկուսի աստիճաններ** են: Flags –ը դա տերմին է, երբ կողը տվյալ բիթում պարունակում է 1: Այլ խոսքով անդամը ֆլագ դառնալու դեպքում այն հանդիսանում է արդյունք:
- Առանց ատրիբուտի, եթե բիթային օպերացիայի արդյունքում համարժեք թվով անդամ կա, ապա վերադարձնում է նրա սիմվոլային արժեքը: Եթե չկա, ապա վերդարձնում է բիթայինի արդյունքի ամբողջ թիվը:

```
using System;
[Flags]
public enum Days
{
    // Փորձել փոխել  երկուսի աստիճաններ արժեքները: Արդյունքը տարբեր է:
    Mon = 1,
    Tue = 2, // 8
    Wed = 4,
    Thu = 8,
    R=9
}
class M
{
    static void Main()
    {
        Days x = Days.Mon|Days.Tue; // և 1 բիթում ա 1 և 2-րդ բիթում: երկու րդյունք: հանել ֆլագը
        Console.WriteLine(x);
}   }
```

❖ **[Obsolete]**

✓ Ատրիբուտը թույլ է տալիս ծրագրի էլեմենտը նշել որպես հին:

✓ `[Obsolete("instead.", false)]` ատրիբուտը դրված է մեթոդի վրա: Նրա առաջին պարամետրը հաղորդագրության տեքստն է, իսկ երկրորդ պարամետրը տրամաբնական է, որի `true` արժեքի դեպքում կատարվում է թարգմանության `error`:

✓ Օրինակում թարգմանիչը տալիս է զգուշացում և եթե `false` փոխարինենք `true`, ապա ծրագիրը կտա թարգմանման սխալ:

```
using System;
class M
```

```
{
    [Obsolete("instead.", false)]
    public static int MyMeth(int a, int b)
    {
        return a / b;
    }
    // Improved version of MyMeth.
    public static int MyMeth2(int a, int b)
    {
        return b == 0 ? 0 : a / b;
    }
    static void Main()
    {
        // Warning displayed for this.
        Console.WriteLine(MyMeth(4, 3));
        // No warning here.
        Console.WriteLine(MyMeth2(4, 3));
        Console.ReadKey();
}   }
```

❖ [Conditional] (կանցնենք նաև Debug բաժնում)

✓ Ատրիբուտը գտնվում է `System.Diagnostics` անունի տարածքում: Օգտագործվում է միայն մեթոդների համար:

✓ Հնարավորություն է տալիս ստեղծել պայմանական մեթոդներ, որոնք կանչվում են այն դեպքում, երբ `#define` դիրեկտիվայի միջոցով ստեղծվում է իդենտիֆիկատոր և կատարվում ծրագրի թարգմանության ճյուղավորում: Այս ձևը նման է `#if` դիրեկտիվայի աշխատանքին, որը կանցնենք հետո:

- Օրինակում մեթոդը կանչվում է միայն ըստ իդենտիֆիկատրի որոշմամբ և մեթոդը չի կանչվում, եթե համապատասխան իդենտիֆիկատոր չկա:

```
#define TRIAL //Schildt 569
using System;
using System.Diagnostics;
class Test
{
    [Conditional("TRIAL")]
    void Trial()
    {
        Console.WriteLine("Trial version, not for distribution.");
    }
    [Conditional("RELEASE")]
    void Release()
    {
        Console.WriteLine("Final release version.");
    }
    static void Main()
    {
        Test t = new Test();
        t.Trial(); // called only if TRIAL is defined
        t.Release(); // called only if RELEASE is defined
        Console.ReadKey();
}   }
```

- Գոյություն ունեն `Conditional` ատրիբուտի կիրառման որոշ սահմանափակումներ:

✓ Մեթոդները պետք է վերադարձնեն `void`:

✓ Մեթոդները կարող են լինել կլասի կամ ստրուկտուրայի անդամ և չեն կարող լինել ինտերֆեյսի անդամ:

✓ Մեթոդները չեն կարող լինել `override`:

.....

# 22. Տվյալների վալիդացիա ատրիբուտներով

[metanit.com/sharp(pr.24)]

- Եթե անհրաժեշտ է կլասի դաշտերը արժեքավորել թույլատրելի սահմաններում, ապա դա կարելի է **ստուգել** պարզ ծրագրային հրամաններով: Ատրիբուտներ նույնպես հնարավորություն են տալիս ստուգել և արժեքավորել այն համակարգի մակարդակով: Մենք այն անվանում ենք կլասի դաշտերի **վալիդացիա**:

```
using System;
        User ob = new User();
        Console.Write("Anun: ");
        string anun = Console.ReadLine();
        Console.Write("Tariq: ");
        int tariq = Int32.Parse(Console.ReadLine());
        if (!String.IsNullOrEmpty(anun) && anun.Length > 1)
            ob.Name = anun;
        else
            Console.WriteLine("sxal anun");
        if (tariq>= 1 && tariq <= 100)
            ob.Age = tariq;
        else
            Console.WriteLine("sxal tariq");
public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

### ❖ Ատրիբուտներով վալիդացիա

```
using System; // Framework: կատարել DataAnnotations -ի References կապ Assemblies բաժնում
using System.ComponentModel.DataAnnotations;
for ( ; ; )
{
        Console.Write("name:");
        string name = Console.ReadLine();
        Console.Write("age:");
        int age = Int32.Parse(Console.ReadLine());
        User ob = new User { Name = name, Age = age };
        var results = new List<ValidationResult>();
        var context = new ValidationContext(ob);
        if (!Validator.TryValidateObject(ob, context, results, true)) // false չստուգել
        {
            foreach (var error in results)
                Console.WriteLine(error.ErrorMessage);
        }
}
public class User
{
    // [Key][Required]
    public string Id { get; set; }
    [Required][StringLength(50, MinimumLength = 3)]
    public string Name { get; set; }
    [Required][Range(1, 100)]
    public int Age { get; set; }
}
```

.....

## 23. extern

[Шилдт -712; Гриффитс- 641]

- Գոյություն ունի երկու կիրառություն:
- **Առաջին: Արտաքին մեթոդների** հայտարարում: Նշվում է, որ մեթոդը չի գտնվում ղեկավարման կոդում, այսինքն դառնում արտաքին կոդ: Հայտարարման ձևը.

```
extern  ret-type meth-name(arg-list);
```

✓ Հաճախ օգտագործվում է `[DllImport]` ատրիբուտի հետ, որը նշում է գրադարանը, որտեղ գտնում է մեթոդը: `[DllImport]` ատրիբուտի անունի տարածքը հետևյալն է. `System.Runtime.InteropServices:`

✓ Արտաքին մեթոդը պետք է լինի C կամ C++ ծրագիր: Օրինակ.

```
#include <stdlib.h>
int __declspec(dllexport) AbsMax(int a, int b)
{
      return abs(a) < abs(b) ? abs(b) : abs(a);
}
```

✓ C -ում գրված `declspec(dllexport)` հրամանը նշում է `AbsMax(int a, int b)` ֆունկցիան կարելի է օգտագործել DLL գրադարանից: Ենթադրենք այն գրվել է ExtMeth.dll գրադարանում C –ի միջոցով:

✓ Այժմ ներկայացնենք C# ծրագիրը, որը օգտվում է `AbsMax(int a, int b)` մեթոդից.

```
using System;
using System.Runtime.InteropServices;
class M
{
    [DllImport("ExtMeth.dll")]
    public extern static int AbsMax(int a, int b);
    static void Main()
    {
        int max = AbsMax(-10, -20);
        Console.WriteLine(max);
    }
}
```

- **Երկրորդ կիրառություն:** Արտաքին հավաքագրման **պսեվդոանուն**, որը օգտակար է այն դեպքում, երբ երկու հավաքագրման բլոկներ ունեն նույն անունով էլեմենտներ:

✓ Օրինակ VS -ում: (Պրակտիկ)

`ClassLibrary1.cs` ֆայլում տեղադրենք ծրագիր.

```
using System;
namespace NS
{  public class A
   {
       public A()
       {
          Console.WriteLine("A1.dll");
       }
   }
}
```

✓ `ClassLibrary2.cs` ֆայլում տեղադրենք ծրագիր.

```
using System;
namespace NS
{   public class A
    {
        public A()
        {
            Console.WriteLine("A2.dll");
        }
    }
}
```

✓ Եթե ռեֆերենսով այս երկու ֆայլերը օգտագործվում են, ապա առանց `extern alias` հրամանի և պսեվդոանունի անհնար է նշված `A` -ից օգտվել, քանի որ անունի տարածքը նույնպես նույնն է:

✓ `a1b.cs` ֆայլը պետք է օգտվի նշված կլասներից: Սկզբից References – ով միացնում ենք ClassLibrary1.dll և ClassLibrary2.dll ֆայլերը: Այնուհետև ավելացված dll -ների վրա աջ քլիկով -> `properties-> aliases` -ում գրում ենք համապատասխանաբար ինչ որ `a1` և `a2` անուններ: Չմոռանալ `extern alias` հրամանը գրել `namespace` -ի մեջ կամ `using` -ներից վերև:

• Սկզբից նշենք error տարբերակը, եթե կլասը կցենք առանց `alias` –ի: Կրող է լինել առանց error, եթե ավելացնենք `// **` հրամանը միայն մեկ `alias` –ով: Այդ դեպքում `A ob = new A();` հրամանը կաշխատի, որովհետև մյուսը հղումը `alias` –ով է:

```
using NS;
namespace alb
{
      // extern alias a2; // ok  // **
      class M
      {
         static void Main()
         {
             A ob = new A(); // error // ok // **
         }
      }
}
```

❖ `extern` երկու `alias` տարբերակ՝

```
using System;
namespace alb
{
      extern alias a1;
      extern alias a2;
      class M
      {
         static void Main()
         {
             a1::NS.A t1 = new a1::NS.A();
             a2::NS.A t2 = new a2::NS.A();
         }
      }
}
```

Ծրագրի արդյունքում ստացվում է հետևյալ պատասխանները՝

```
A1.dll
A2.dll
```

✓ **Հրամանի տողով:** Պետք է ստեղծել սկզբից պսեվդոանուն /r բանալու միջոցով: /r -ը նշում է մետատվյալների ֆայլը:

```
/r:a1=test1.dll
/r:a2=test2.dll
```

Եվ հաջորդը քայլը օգտվել `extern` հրամանից:

```
extern alias a1;
extern alias a2;
```

✓ Որպեսզի ծրագիրը աշխատի պետք է նախորոք թարգմանել `test1.cs` և `test2.cs` ֆայլերը գրադարանային DLL տեսքի հետևյալ հարամանի տողի միջոցով.

```
csc /t:library test1.cs
csc /t:library test2.cs
```

✓ Այնուհետև պետք է թարգմանել `test3.cs` ֆայլը.

```
csc /r:a1=test1.dll /r:a2=test2.dll test3.cs
```

# 24. Տիպերի դինամիկ ճանաչում

[Шилдт-537; Рихтер -126]

- Տիպերի դինամիկ իդենտիֆիկացիա - օգտագործողը ճանաչում է օբյեկտի տիպը ծրագրի **կատարման** ժամանակ: Դա կատարվում է is , as , typeof միջոցով:
- is – ի միջոցով կարելի է որոշել տվյալ **օբյեկտը նշված տիպի է թէ ոչ**: Եթե համնկնում է, ապա վերադարձնում է true:
- as – ի միջոցով կատարվում է տիպերի վերափոխում և չի առաջացնում հատուկ իրավիճակ, նույնիսկ եթե դա չի թույլատրվում: Հաջողության դեպքում կատարվում է հղում տիպի վրա և հակառակ դեպքում վերադարձնում է null հղում:
- typeof – Տիպի մասին **ինֆորմացիա ստանալու** ձևերից մեկը, դա typeof - է: Նրա գործը, `System.Type` կլասից օգտվելով ինֆորմացիա ստանալ տիպի մասին: Ստանալով `Type` օբյեկտ կարելի է ստանալ ինֆորմացիա տվյալ տիպի մասին, օգտագործելով `Type` կլասի հատկանիշները, դաշտերը, մեթոդները:

```
using System;
using System.IO;
class A { }
class B : A{ }
class M
{
   public static void Main()
   {
     A a = new A();
     B b = new B();
     if (a is A) // is --------------------
         Console.WriteLine("object a have type A");
     if (b is A)
         Console.WriteLine("object b have type base A");
     if (a is B)
         Console.WriteLine("this text not view");
     if (a is object)
         Console.WriteLine("a is object");
     if (a as B == null) // as-------------------
     {
         Console.WriteLine("a not call B");
         Console.WriteLine(a.GetType());
     }
     else
         Console.WriteLine("null for a");
     if (b as A == null)
         Console.WriteLine("null for b");
     else
     {
         Console.WriteLine("b may call A");
         Console.WriteLine(b.GetType());
     }
     // typeof ----------------------------
     Type t = typeof(StreamReader);
     Console.WriteLine(t.FullName);
     if (t.IsClass)
         Console.WriteLine("class");
     if (t.IsAbstract)
         Console.WriteLine("abstract class");
     }
  }
```

.....

# 25. Արտապատկերում

[Шилдт-541; Троелсен-562; Troelsen-632; Нейгел-407]

❖ Զարգացնենք **տիպերի դինամիկ ճանաչումը:** `System.Reflaction` անունի տարածք:

- Արտապատկերում (Reflection) – ը հնարավորություն է տալիս ստանալ **տիպի** մասին ինֆորմացիա: Այդ ինֆորմացիան կարելի է օգտագործել ծրագրի **կատարման ժամանակ** մեծացնելով ծրագրի ճկունությունը: Արտապատկերումը տալիս է նույնատիպ ինֆորմացիա **մետատվյալների** մասին ինչպիսին տալիս է `ildasm.exe` ուտիլիտը:
- Տիպի մասին ինֆորմացիա ստանալու համար դիմում ենք `Type` կլասին, որը **արձագանքում** (**արտապատկերում** է) է մեր հայցին: `Type` կլասը որոշ անդամներ վերադարձնում են տիպ, որը պատկանում է `System.Reflaction` անունի տարածքին: Օրինակ. `Type.GetType()` մեթոդը կարող է վերադարձնել `MethodInfo` տիպի օբյեկտների մասիվ: `Type.GetField()` մեթոդը կարող է վերադարձնել `FieldInfo` տիպի օբյեկտների մասիվ:
- `Type` կլասի օբյեկտ հնարավոր չէ ստեղծել, քանի որ այն **աբստրակտ** է: Օգտվելու համար գոյություն ունի հետևյալ ձևերը:

✓ **Առաջինը**: օգտվում ենք **`Object`** կլասի `GetType()` մեթոդից ստեղծելով օբյեկտ.
```
A ob = new A();
Type t = ob.GetType();
```
✓ **Երկրորդ:** օգտվում ենք `typeof` հրամանից չունենալով օբյեկտի ստեղծման անհրաժեշտություն.
```
Type t2 = typeof(A); (նախորդ դաս)
```
✓ Երրորդ: օգտվում ենք `System.Type.GetType()` ստատիկ մեթոդից.
```
A ob = new A();
Type t3 = Type.GetType("A",false, true);
```
- որտեղ առաջի պարամետրը ցույց է տալիս **տիպը**, երկրոդ պարամետրը բուլիան է և ցույց է տալիս **արտակարգ իրավիճակ** կարելի է առաջացնել, եթե տիպը չի հայտնաբերվել: Երրորդ պարամետրը նույնպես բուլիան է և ցույց է տալիս անվան սիմվոլային **տողի ռեգիստրը** հաշվի առնել, թե ոչ: `true` հաշվի չի առնում ռեգիստրը:

❖ Օգտվելով **+ նշանից** կարելի է դիմել **նրդված կլասին**:

```
using System;
        A ob = new A();
        Type t = Type.GetType("A+B",false, true);
        Console.WriteLine(t);
        Console.ReadKey();

class A
{
    class B
    {
    }
}
```

❖ Նշենք **Type** կլասի մեթոդներ.

✓ `ConstructorInfo[] GetConstructors()` // Ստանում է **կոնստրուկտորների** ցուցակ:
✓ `EventInfo[] GetEvents()` // Ստանում է **իրադարձությունների** ցուցակ:
✓ `FieldInfo[] GetFields()` // Ստանում է տիպի **տվյալների** ցուցակ:
✓ `MemberInfo[] GetMembers()` // Ստանում է տիպի **անդամների** ցուցակ:
✓ `MetodInfo[] GetMetods()` // Ստանում է **մեթոդների** ցուցակ:
✓ `PpopertyInfo[] GetPropertties()` // Ստանում է **հատկանիշների** ցուցակ:

- Օրինակը հնարավորություն է տալիս ստանալ կլասի մեթոդների ցուցակը և նրանց պարամետրերը: Քանի որ բոլոր օբյեկտները պարունակում են նաև Object կլասի բոլոր բաց անդամները, ապա նրանք նույպես ցուցադրվում են: Ցուցադրվում է Void add_ee(dd value) Void remove_ee(dd value) event –ի մեթոդները: Նաև կցուցադրվի կլասի դաշտերը, կոնստրուկտորները և իրադարձությունները:

```
using System;
using System.Reflection;
        Type t = typeof(A);
        Console.WriteLine("Methods class " + t.Name);
        MethodInfo[] mi = t.GetMethods();
        foreach (MethodInfo m in mi)
        {
            ParameterInfo[] pi = m.GetParameters();
            Console.Write(m.ReturnType.Name + " " + m.Name + "(");
            for (int i = 0; i < pi.Length; i++)
            {
                Console.Write(pi[i].ParameterType.Name + " " + pi[i].Name);
                if (i + 1 < pi.Length)
                    Console.Write(",");
            }
            Console.WriteLine(")");
        }
      Console.WriteLine("Fields class " + t.Name);
      FieldInfo[] fi = t.GetFields();
      foreach (FieldInfo v in fi)
           Console.WriteLine(v);
      Console.WriteLine("Constructors class " + t.Name);
      ConstructorInfo[] c= t.GetConstructors();
      foreach (ConstructorInfo v in c)
           Console.WriteLine(v);
      Console.WriteLine("Events class " + t.Name);
      EventInfo[] e = t.GetEvents();
      foreach (EventInfo v in e)
           Console.WriteLine(v);
Console.ReadKey();
delegate void dd();
class A
{
    public event dd ee;
    public decimal x;
    public string s;
    public A(decimal x, string s)
    {
        this.x = x;
        this.s = s;
    }
    public bool fff(int x)
    {
            return false;
    }
    public void ff(int a, long b)
    {
    }
    int f() // չի ցուցադրվում, որովհետև private է
    {
        return 777;
    }
}
```

.....

# 26. Հավաքագրման բլոկի արտապատկերում

[Шилдт-555; Троелсен-571]

## Dynamically Loading Assemblies [Troelsen-641]

- Իրական արտապատկերման օգուտակարությունը կայանում է նրանում, երբ կատարման ժամանակ այլ **հավաքագրման բլոկներից** ստանում ենք անհրաժեշտ ինֆորմացիա և նույնիսկ **իրականացնում** նրա մեթոդները:
- `System.Reflaction` անունի տարածքում է գտնվում `Assembly` կլասը, որը հնարավորություն է տալիս դինամիկ կանչել հավաքագրման բլոկ և հետազոտել նրա անդամները: Կարելի է կանչել փակ և բաց հավաքագրումներ, որոնք գտնվում են **տարբեր** ծրագրային տարածքներում:
- `Load()` և `LoadFrom()` մեթոդներով թույլատրվում է ծրագրային ձևով ստանալ նույն ինֆորմացիան, որ հանդիպում է `*.config` ֆայլում:

✓ `Load()` մեթոդի օգտագործում.

```
// Load version 1.0.0.0 of CarLibrary using the default culture.
Assembly asem = Assembly.Load(@"ClassLibrary1, Version=1.0.0.0,
                              PublicKeyToken=null, Culture="""");
```

✓ `LoadFrom ()` մեթոդի օգտագործում.

```
// Load the ConsoleApp77.exe assembly.
Assembly asem = Assembly.LoadFrom("ConsoleApp77");
```

❖ `Assembly.LoadFrom` – ի օրինակ.

```
using System; //[Шилдт-555]
using System.Reflection;
class M
{
    public M(int a,char c)
    {
    }
    public static void Main()
    {
        Assembly asm = Assembly.LoadFrom("ClassLibrary1.dll");
        //Assembly asm = Assembly.Load(@"ConsoleApp77");
        Type[] alltypes = asm.GetTypes();
        foreach (Type temp in alltypes)
            Console.WriteLine("Found: " + temp.Name);
        Type t = alltypes[0]; // use first class found
        Console.WriteLine("Using: " + t.Name);
        // Obtain constructor info.
        ConstructorInfo[] ci = t.GetConstructors();
        Console.Write("Available constructors: ");
        foreach (ConstructorInfo c in ci)
        {
            // Display return type and name.
            Console.Write(" " + t.Name + "(");
            // Display parameters.
            ParameterInfo[] pi = c.GetParameters();
            for (int i = 0; i < pi.Length; i++)
            {
                Console.Write(pi[i].ParameterType.Name + " " + pi[i].Name);
                if (i + 1 < pi.Length)
                    Console.Write(", ");
            }
            Console.WriteLine(")");
        }
    }
}
```

❖ Տեղաբաշխված հավաքագրման բլոկի արտապատկերում [Троелсен-573]

❖ Reflecting on Framework Assemblies [Troelsen-643]

```
using System;
using System.Linq;
using System.Reflection;
class M
{
    private static void f(Assembly a)
    {
        Console.WriteLine("***** Info about Assembly *****");
        Console.WriteLine("Loaded from GAC? {0}", a.GlobalAssemblyCache);
        Console.WriteLine("Asm Name: {0}", a.GetName().Name);
        Console.WriteLine("Asm Version: {0}", a.GetName().Version);
        Console.WriteLine("Asm Culture: {0}", a.GetName().CultureInfo.DisplayName);
        Console.WriteLine("\nHere are the public enums:");
        // Use a LINQ query to find the public enums.
        Type[] t = a.GetTypes();
        var publicEnums = from pe in t
                          where pe.IsEnum && pe.IsPublic
                          select pe;
        foreach (var pe in publicEnums)
            Console.WriteLine(pe);
    }
    static void Main(string[] args)
    {
        Console.WriteLine("***** The Shared Asm Reflector App *****");
        // Load System.Windows.Forms.dll from GAC.
        string s = null;
        s = "System.Windows.Forms," +"Version=4.0.0.0," +
                      "PublicKeyToken=b77a5c561934e089," + @"Culture=""";
        Assembly asm = Assembly.Load(s);
        f(asm);
    }
}
```

.....

## 27. Հավաքագրման բլոկի հետաձգված կապվածություն (Framework, Core)

[Troelsen-645; metanit.com/sharp(parag.18)]

- **Հետաձգված կապվածությունը** հնարավորություն է տալիս **կատարման ժամանակ** ծրագրային իրականացումներ: Դա կարող է լինել մեթոդի ընտրություն կամ ասենք օբյեկտի ստեղծում և անդամների օգտագործում:
- Հետաձգված կապվածության դեպքում կա **սխալ տիպի ընտրման** ռիսկի հնարավորություն: Սակայն այն բարձրացնում է ծրագրի **ընդլայնման** հնարավորությունը, երբ ծրագրի լրացուցիչ ֆունկցիոնալ հնարավորությունները հայտնի չեն:
- Հետաձգված կապվածության դեպքում կարելի օգտվել `System.Activator` կլասից և նրա `CreateInstance()` ստատիկ մեթոդի օգնությամբ ստեղծել օբյեկտ և օգտվել նրա մեթոդներից:
- Փորձենք օգտվել `ConsoleApp77` ծրագրի f() մեթոդից և կատարման ժամանակ տալ պարամետրի այլ արժեք:

```
using System; // ConsoleApp77
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(f(55));
    }
    public static double f(int n)
    {
        return ++n;
    }
}
//.....................................................................
using System;
using System.Reflection;
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Assembly asm = Assembly.Load("ConsoleApp77"); //References load (if Core then .dll)
            Type t = asm.GetType("Program", true, true);
            object ob = Activator.CreateInstance(t);
            // ստանում ենք f() մեթոդը
            MethodInfo method = t.GetMethod("f");
            // կանչում ենք մեթոդը և տալիս արգումենտ
            object result = method.Invoke(ob, new object[] { 99 });
            Console.WriteLine(result);
            Console.WriteLine("Invoke Main if no public");
            method = t.GetMethod("Main", BindingFlags.NonPublic | BindingFlags.Static);
            method.Invoke(ob, new object[] { new string[] { } });
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
}   }
```

✓ մեթոդի կանչը կարելի է ապահովել `object result = method.Invoke(ob,null)` հրամանով, եթե պարամետրի փոխանցման անհրաժեշտություն չկա:

✓ Main-ի կանչի համար օգտագործվում է օգտագործման բիթային “մասկա” (եթե փակ է և ստաիկ): Ինչպես նաև Main-ը պահանջում է string վեկտոր, որը կարելի է բավարարել `new string[] { }`-ով:

.....

# 28. Պրոցես

[Троелсен-620; Troelsen-529; metanit.com/sharp(pr.22)]

- **Պրոցեսը – երբ ծրագրային հատվածը գտնվում է RAM –ում , ունի կատարման վիճակ և իրեն ուղեկցող ռեսուրսներ:** Նշենք `Process` կլասի որոշ անդամներ.
- ExitCode - Հատկանիշ, որը արժեքավորվում է պրոցեսի ավարտի ժամանակ մշակելով Exited իրադարձությունը (ասինխրոն) կամ WaitForExit() մեթոդը (սինխրոն):
- ExitTime – Հատկանիշ, որը պարունակում է պրոցեսի ավարտը DateTime տեսքով:
- Handle - Հատկանիշ, որը պարունակում է օպերացիոն համակարգի կողմից տրված դիսկրիպտորը:
- HandleCount – Հատկանիշ, որը պարունակում է դիսկրիպտրների քանակը տվյալ պրոցեսի համար:
- Id - Հատկանիշ, որը պարունակում է տվյալ պրցեսի PID – ն:
- MachineName - Հատկանիշ, որը պարունակում է համակարգչի անվանումը:
- MainWindowTitle - Հատկանիշը ստանում է գլխավոր պատուհանի անվանումը:
- Modules – Հատկանիշը ցույց է տալիս .exe և .dll մոդուլները:
- PriorityBoostEnabled – Հատկանիշը արագացուցիչ է, որը հնարավորություն է տալիս արագացնել պրոցեսը, երբ այն դառնում է գլխավոր պատուհան:
- PriorityClass - Հատկանիշ, որը հնարավորություն է տալիս փոփոխել պրոցեսի պրիորիտետները:
- ProcessName - Հատկանիշ, որը պարունակում է պրոցեսի անունը:
- Responding - Հատկանիշը ցույց է տալիս օգտագործողին կարելի է պատասխանել թէ ոչ:
- StartTime – Հականիշ, որը պարունակում է DateTime տիպի տվյալներ պրոցեսի սկսելու մասին:
- Threads - Հատկանիշ, որը ստանում է պրոցեսի հոսքերը ProcessThread տիպի զանգվածի ձևով:
- CloseMainWindow() –Մեթոդը փակում է գլխավոր պատուհանը վերջացնելով նաև պրոցեսը:
- GetCurrentProcess() – Ստատիկ մեթոդը վերադարձնում է ընթացիկ պրոցեսը:
- GetProcesses() – Ստատիկ մեթոդը վերադարձնում է պրոցեսների զանգված, որոնք աշխատում են տվյալ մեքենայում:
- Kill() - Մեթոդ, որը անմիջապես դադարեցնում է պրոցեսը:
- Start() – Մեթոդ, որը սկսում է պրոցեսի կատարումը:

```
using System;
using System.Diagnostics;
using System.Threading;
class Program
{
    static void Main()
    {
        Process[] ar = Process.GetProcesses(".");
        foreach (Process v in ar)
            Console.WriteLine(v.Id + " " + v.ProcessName);
        Console.WriteLine("----------------------");
        Process p;
        try
        {
            p = Process.GetProcessById(7064);
            Console.WriteLine(p.ProcessName);
            Console.WriteLine(p.MainWindowTitle+" "+p.HandleCount + " " + p.MachineName);
        }
        catch
        {             Console.WriteLine("-> Sorry...bad PID!");
        }
        // Process p2 = Process.Start("calc.exe");
        // Console.WriteLine(p2.ProcessName);
        // Thread.Sleep(2000);
        // // p2.Kill();
}   }
```

# 29. Դոմեն ծրագրային մաս

[Троелсен-631; Troelsen-541; Нейгел-52, 521, 559; metanit.com/sharp(pr.22)]

- Երբ թողարկվում է ծրագիրը, օպերացիոն համակարգը ստեղծում է պրոցես իր հիշողության հատկացումով, որտեղ կատարվում է տվյալ ծրագրի հրամանները: Տվյալ պրոցեսի սահմաններում CLR –ը իր հերթին կատարում է **տրամաբանական բաժինների** հատկացում, որը կոչվում է **դոմեն**:
- Ըստ .NET- ի, ծրագրի **հավաքագրման** մասերը տեղավորվում են տրամաբանորեն բաժանված մասերի - դոմենների մեջ, այլ ոչ թե պրոցեսների, որը կատարվում էր օրինակ Win32 – ի դեպքում: Դոմենը հնարավորություն է տալիս հավաքագրման բլոկները առանձնացնել պրոցեսի ներսում: Պրոցեսը կարող է պարունակել մեկից ավել դոմեններ: Այդպիսի լրացուցիչ բաժանումը տալիս է հետևյալ առավելությունները.

✓ Հնարավորություն է տրվում լինել **անկախ** օպերացիոն համակարգից:

✓ Ավելի **քիչ ռեսուրս** է պահանջվում ( պրոցեսորի ժամանակի, հիշողության): Այս դեպքում արագանում է դոմենից դոմեն անցումը:

✓ Ապահովվում է ծրագրի լավագույն ձևով **մեկուսացումը** և դոմենի **վնասման** դեպքում շարունակվում են պրոցեսի կազմի մյուս դոմենները:

- Այսպիսով պրոցեսը կարող է ունենալ ցանկացած քանակությամբ դոմեններ, որոնցից յուրաքանչյուրը **լրիվ մեկուսացված** է մյուսներից: Ասածից հետևում է, որ եթե ծրագիրը կատարվում է մի դոմենում և պետք է տվյալներ ստանա մյուս դոմենից (գլոբալ, ստատիկ), ապա նա պետք է անցնի դոմենի սահմանը միայն **հեռահաղորդման պրոտոկոլներով** (.NET remoting protocol) կամ WCF տեղաբաշխված համակարգով;
- Երբ թողարկվում է պրոցես, ավտոմատ ստեղծվում է **լռելիությամբ** դոմեն:
- Ծրագրի կատարման ընթացքում, անհրաժեշտության դեպքում ծրագիրը կարող է ստեղծել **նոր դոմեններ**: Եթե անհրաժեշտ է, ծրագրողը ինքը նույնպես կարող է ստեղծել դոմեններ, օգտագործելով System.AppDomain կլասի ստատիկ մեթոդներ:

❖ **class System.AppDomain** [Троелсен-632]

**Մեթոդներ**

- CreateDomain() - Ստատիկ մեթոդ, որը ստեղծում է դոմեն:
- GetCurrentThreadId() - Ստատիկ մեթոդ, որը վերադարձնում է ակտիվ հոսքի ID – ն տվյալ դոմենի սահմաններում:
- Unload() – Ստատիկ մեթոդ, որը հեռացնում է նշված դոմենը պրոցեսի սահմաններից:
- CreateInstance() - Մեթոդ, որը ստեղծում է օբյեկտ նշված հավաքագրման բլոկում:
- ExecuteAssembly() - Մեթոդ, որը կատարում է նշված ֆայլի անունով հավաքագրման բլոկը տվյալ դոմենի սահմաններում:
- GetAssemblies() - Մեթոդ, որը ներկայացնում է հավաքագրման բլոկների ցուցակը, որոնք աշխատում են տվյալ դոմենի սահմաններում:
- Load() - Մեթոդ, որը հնարավորություն է տալիս կանչել հավաքագրման բլոկը տվյալ դոմենի մեջ:

```
using System;
using System.Reflection;
class Program
{
    static void Main()
    {
        AppDomain d = AppDomain.CurrentDomain;
        ff(d);
        Console.WriteLine("----------------");
        AppDomain d2 = AppDomain.CreateDomain("SecondAppDomain");
        ff(d2);
    }
    public static void ff(AppDomain ad)
    {
```

```
        Assembly[] asm = ad.GetAssemblies();
        Console.WriteLine(ad.FriendlyName);
        foreach (Assembly a in asm)
        {
            Console.WriteLine(a.GetName().Name);
            Console.WriteLine(a.GetName().Version);
        }
    }
}
//.....................................................................
```

**Հատկանիշներ**

- BaseDirectory - Օգտագործողը ստանում է հավաքագրման բլոկի ճանապարհը:
- CurrentDomain – Ստանում ենք դոմենի ընթացիք հոսքը:
- FriendlyName - Ստանում ենք ընթացիք դոմենի Friendly անունը:

```
using System;
class Program
{
    static void Main()
    {
        AppDomain d = AppDomain.CurrentDomain;
        Console.WriteLine(d.FriendlyName);
        Console.WriteLine(d.Id);
        Console.WriteLine(d.IsDefaultAppDomain());
        Console.WriteLine(d.BaseDirectory);
    }
}
//.....................................................................
```

**Իրադարձություններ**

- AssemblyLoad – Իրադարձություն, որը առաջանում է հավքագրման բլոկի կանչի ժամանակ:
- AssemblyResolve - Իրադարձություն, որը առաջանում է, երբ հնարավոր չէ իդենտիֆիկացնել հավաքագրման բլոկը:
- DomainUnload - Իրադարձություն, որը առաջանում է երբ հեռացվում է դոմենը:
- ProcessExit - Իրադարձություն, որը առաջանում է երբ ավարտում է պրոցեսը:
- TypeResolve - Իրադարձություն, որը առաջանում է, երբ հնարավոր չէ իդենտիֆիկացնել տիպը:
- UnhendledExeption - Իրադարձություն, որը առաջանում է, երբ առաջացած արտակարգ իրավիճակը մնում է առանց մշակման:

```
using System;
class Program
{
    static void Main()
    {
        AppDomain d = AppDomain.CurrentDomain;
        d.ProcessExit += new EventHandler(d_ProcessExit);
        AppDomain d2 = AppDomain.CreateDomain("SecondAppDomain");
        d2.DomainUnload += new EventHandler(d2_DomainUnload);
        //AppDomain.Unload(d2);
        Console.ReadKey();
    }
    static void d_ProcessExit(object sender, EventArgs e)
    {
        Console.WriteLine("######   Exit      ######");
    }
    static void d2_DomainUnload(object sender, EventArgs e)
    {
        Console.WriteLine("######   unloaded second     ######");
}   }
```

❖ Եթե անհրաժեշտ է օգտագործել Ryference -ով .dll -ը դոմենում և թողարկել նրա անդամները, ապա դա կատարվում է CreateInstanceAndUnwrap կամ CreateInstance() + Unwrap() մեթոդների միջոցով:

- CreateInstanceAndUnwrap –ի օրինակ նույն ծրագրի սահմաններում՝

```
using System;
public class A : MarshalByRefObject
{
    public void f()
    {
        Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
    }
}
class Program
{
    public static void Main()
    {
        A localA = new A();
        localA.f();
        AppDomain dom = AppDomain.CreateDomain("New domain");
        A remoteA = (A)dom.CreateInstanceAndUnwrap("ConsoleApp??", "A");
        //A remoteA = (A)dom.CreateInstanceAndUnwrap(typeof(A).Assembly.FullName, "A");
        remoteA.f();
}   }
```

❖ Օրինակ Ryference –ով՝

✓ CreateInstanceAndUnwrap("ClassLibraryError", "ClassLibraryError.Class1");

- Ստեղծենք ClassLibraryError.dll

```
using System;
namespace ClassLibraryError
{
    public class Class1 : MarshalByRefObject
    {
        public void f()
        {
            Console.WriteLine("portsenq error domein");
            int a = 7;
            int b = 4; // b=0;
            Console.WriteLine(a/b);
        }
    }
}
```

- Ստեղծենք ConsoleApp.exe

```
using System;
using ClassLibraryError;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
        AppDomain dom = AppDomain.CreateDomain("SecondAppDomain");
       //ObjectHandle ob = dom.CreateInstance("ClassLibraryError",
                                            "ClassLibraryError.Class1");
       //Class1 ob1 = (Class1)ob.Unwrap();
        Class1 ob2 = (Class1)dom.CreateInstanceAndUnwrap("ClassLibraryError",
                                                         "ClassLibraryError.Class1");
        Console.WriteLine(dom.FriendlyName);
        ob2.f();
    }
}
```

❖ **Դոմեյնի օպտակարությունը** [metanit.com/sharp(pr.22)] (պրակտիկ) (only Framework)

- Օրինակ, որը նոր դոմեյն է կանչում ֆակտորիալ լուծող հավաքագրման բլոկը և աշխատանքից հետո այն **հեռացնում**, թողնելով պրոցեսը աշխատանքի մեջ: Այս ձևով կատարվում է ծրագրի հիշողության զբաղեցման **օպտիմալացում**: [(Core) PlatformNotSupportedException: Secondary AppDomains are not supported on this platform.]
- Սկզբից գրենք ֆակտորյալ լուծող `.exe` ծրագիրը: Այն ընդունում է թիվ `Main(args[])`-ի պարամետրով: Այնուհետև երկրորդ ծրագրում սետղծենք նոր դոմեյն և կանչենք ֆակտորյալ հավաքագրման բլոկը, փոխանցելով նրան արգումենտ:

```
using System;
namespace Factorial
{
    class Program
    {
        static void Main(string[] args)
        {
            // for(int i=0;i<200;i++) // no parallel
            //    Console.Write("++++++++++");
            int n = Convert.ToInt32(args[0]); // 1
            // Console.Write("from factorial programm tiv = ");  // 1
            // int n = Convert.ToInt32(Console.ReadLine());      // 1
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            Console.Write("factorial = "+ result);
        }
    }
}
```

- Կցենք `Factorial.exe` հավաքագրման բլոկը `References`-ով

```
using System;
namespace domainoptimal
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain d = AppDomain.CurrentDomain;
            Console.WriteLine("Main domain id= " + d.Id);
            AppDomain fD = AppDomain.CreateDomain("Factorial Domain");
            Console.WriteLine("new domain id= "+ fD.Id);
            fD.DomainUnload += f;
            Console.Write("tiv = ");
            int n = Convert.ToInt32(Console.ReadLine());
            // Main() argument for Factorial
            string[] arg = new string[] { n.ToString() };
            fD.ExecuteAssembly("Factorial.exe", arg);
            // for(int i=0;i<200;i++) // no  parallel
            //    Console.Write("           ##########");
            AppDomain.Unload(fD);
            Console.WriteLine("continue without factorial assembly");
        }
        static void f(object sender, EventArgs e)
        {           Console.WriteLine("Domen Factorial Unloaded");
        }
    }
}
```

# 30. Օբյեկտի կոնտեքստ սահման

[Троелсен-640; Troelsen-544]

- Ինչպես նշվել է, պրոցեսի **դոմենային** մասը նախատեսված է **հավաքագրման բլոկների** իրականացման համար: Ծրագրի դոմենային մասը նույնպես կարող է ներառել տարանջատում օբյեկտի մակարդակով, որը կոչվում է **կոնտեքստ**: Նրա էությունն է՝ **օբյեկտներին** առանձնացնել և տալ **հատկություն**, որը կկատարվի կոնտեքստային սահմանային բաժանումով:
- Կոնտեքստի օգտագործման ժամանակ CLR– ը հնարավորություն է ստանում ղեկավարել հատուկ **պահանջներով օբյեկտներ**, օգտագործելով կոնտեքստի սահմանի հսկման հնարավորությունը: Օրինակ, եթե կլասը պետք է ունենա հոսքերի ավտոմատ սինխրոնիզացիա, ապա օգտագործվում է [Synchronization] ատրիբուտը և որի շնորհիվ CLR–ը օբյեկտը տեղադրում է սինխրոնիզացված կոնտեքստում:
- Ինչպես ամեն պրոցես ստեղծում է լռելիությամբ դոմեն, այնպես էլ դոմենը ստեղծում է լռելիությամբ կոնտեքստ, որը նշվում է կոնտեքստ **0 համարով** և այն կոչվում է **ստանդարտ** կոնտեքստ: Նշված կոնտեքստում ընդգրկված օբյեկտները առանձնակի կոնտեքստային **պահանջներ** չունեն: Նրանց քանակը կարող է լինել մեկից ավելին:

**Կոնտեքստ անկախ և կոնտեքստ կապված օբյեկտներ**

- Տիպերը, որոնք չեն օգտվում կոնտեքստային սահմանափակումից կոչվում են **կոնտեքստ անկախ**: Նրանց օբյեկտները հասանելի են դոմենի ցանկացած մասից, չունենալով CLR –ից **առանձնակի պահանջներ**: Մյուս կողմից, օգտվելով System.ContextBoundObject կլասից կարելի է ունենալ **կոնտեքստ կախվածությամբ** օբյեկտներ իրենց պաշտպանվածությամբ և սահմանափակումներով:
- Բացի դրանից նշված օբյեկտները կարող են լինել կոնտեքստային ատրիբուտներով, որոնց բազային կլաս է հանդիսանում System.Runtime.Remoting.Contexts.ContextAttribute կլասը:
- Կարելի է ստեղծել սեփական կոնտեքստային ատրիբուտները, ժառանգելով ContextAttribute կլասը: Այդ դեպքում պետք է վերավորոշել բոլոր վիրտուալ մեթոդները:
- Օրինակ՝ կոնտեքստ կապվածությունը հնարավորություն է տալիս ունենալ հոսքային անվտանգություն, չօգտագործելով հոսքային սինխրոնիզացիային բարդ լոգիկան.

```
[Synchronization]
public class A : ContextBoundObject
{
}
```

- [Synchronization] ատրիբուտը օբյեկտը տեղավորում է սինխրոնիզացիայի կոնտեքստի մեջ, սակայն որպեսզի այլ հոսքեր չօգտվեն տվյալ կլասի հղումներից և տվյալները **չվնասեն**, անհրաժեշտ է ժառանգել ContextBoundObject կլասից, որը և բարձրացնում է **պաշտպանվածությունը**:
- Օրինակում `LeaseLifeTimeServiceProperty` համարվում է .NET հեռահար մշակման ներքին մակարդակի հատկանիշ և կարելի է **այն անտեսել**:

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Threading;
class A
{
    public A()
    {
        Context c = Thread.CurrentContext;
        Console.WriteLine($"{GetType()} context= {c.ContextID}");
        foreach (IContextProperty v in c.ContextProperties)
            Console.WriteLine("Proprty: " + v.Name);
    }
}
```

```
[Synchronization]
class A2 : ContextBoundObject
{
    public A2()
    {
        Context c = Thread.CurrentContext;
        Console.WriteLine($"{GetType()} context= {c.ContextID}");
        foreach (IContextProperty v in c.ContextProperties)
            Console.WriteLine("Proprty: " + v.Name);
    }
}
class Programm
{
    static void Main(string[] args)
    {
        A ob1 = new A();
        A ob2 = new A();
        A2 ob3 = new A2();
        A2 ob4 = new A2();
    }
}
```

❖ **Հոսք, դոմեն, կոնտեքստ հարաբերություն** [Troelsen-549]

- `System.Threading` անունի տարածքը ծառայում է բազմահոսքային աշխատանքի համար: Ծրագիրի հատվածը կարող է իմանալ տվյալ պահի հոսքը.
  `Thread t= Thread.CurrentThread;`
- Դոմենում կարող է լինել մեկից ավելի հոսքեր: Հոսքը նույնպես կարող է ծառայել տարբեր դոմենների, սակայն ոչ զուգահեռ - այսինքն հոսքը կարող է միայն մեկ դոմենում լինել: Ծրագիրը կարող է ստանալ տվյալ հոսքի դոմենը.
  `AppDomain ad = Thread.GetDomain();`
- [Recall that with .NET Core, there is only a single AppDomain. Even though extra AppDomain's cannot be created, an application's AppDomain can have numerous threads executing within it at any given time.] [Troelsen-550]
- Ցանկացած հոսք կարող է CLR – ի միջոցով տեղափոխվել կոնտեքստից կոնտեքստ կամ տեղավորվել նոր կոնտեքստում: Որպեսզի ստանանք ընթացիկ կոնտեքստը, որի սահմաններում կարող է հայտնվել հոսքը, ապա պետք է օգտվել հետևյալ հրամանից.

```
using System.Runtime.Remoting.Contexts;
Context ctx = Thread.CurrentContext;
ExecutionContext ctx2 =Thread.CurrentThread.ExecutionContext;
```

……

# Հոսքեր (Thread)

## 31. Բազմահոսքայնություն

[Шилдт-833; Троелсен-621; Troelsen-550; Нэш-362]

- Հոսքը դա ծրագրի հրամանների խումբ է, որի համար հատկացվում է կատարման ժամանակի քվանտ, համակարգչի պրոցեսորով (CPU -ով) ծրագիրը իրականացնելու համար: .NET -ում պրոցեսորը դառնում է **վիրտուալ** պրոցեսորը՝ CLR -ը:
- Գոյություն ունի զուգահեռություն **պրոցեսի մակարդակով** և զուգահեռություն **հոսքի** մակարդակով: Պրոցեսի զուգահեռության դեպքում հնարավորություն է տրվում կատարել զուգահեռ **ծրագրեր**: Հոսքի զուգահեռությունը բերում է հնարավորության, որով կարելի է **ծրագրի տարբեր մասեր** կատարել զուգահեռաբար:
- Բազմահոսքայնությանը հնարավորություն է տալիս ծրագիրը դարձնել **էֆեկտիվ:** Գոյություն ունի տարբեր արագության սարքեր և հաճախ պրոցեսորը գտնվում է պարապուրթի մեջ: Այս դեպքում հնարավորություն է տրվում կատարել ծրագրի այլ հատված:
- Բազմահոսքայնությունը իրենից ներկայացնում է **իլյուզիա**, եթե համակարգիչը ունի միայն մեկ պրոցեսոր:
- Ղեկավարվող հոսքը **վիրտուալ** հոսք է և պարտադիր չէ միանշանակ արտապատկերի օպերացիոն համակարգի հոսքին:

**Հոսքերի աշխատանքը պրոցեսում**

```
using System;
using System.Diagnostics;
class Program
{
    static void Main()
    {
        Process p;
        try
        {
            p = Process.GetProcessById(1972);
        }
        catch
        {
            Console.WriteLine("Sorry...bad PID!");
            return;
        }
        Console.WriteLine("threads used by: {0}", p.ProcessName);
        ProcessThreadCollection t = p.Threads;
        foreach (ProcessThread pt in t)
        {
            Console.Write(pt.Id+ " ");
            Console.Write(pt.ThreadState.ToString()+" ");
            Console.Write(pt.PriorityLevel+ " ");
            Console.WriteLine(pt.StartTime.ToShortTimeString());
            //pt.IdealProcessor = 4;
        }
    }
}
```

.....

## 32. Կլաս System.Threading.Thread

[Троелсен-698; Troelsen-552; Шилдт-836; Нейгел-579; Нэш -361]

- `CurrentContext` - հատկանիշ, որը վերադարձնում է կոնտեքստը որտեղ կատարվում է հոսքը:
- `CurrentThread` - ստատիկ հատկանիշ, որը վերադարձնում է կատարվող հոսքի հղումը
- `GetDomain(), GetDomainID()` - ստատիկ մեթոդ, որը վերադարձնում է հղում կատարվող դոմենին, կամ նրա իդենտիֆիկատրին:
- `Sleep()` – ստատիկ մեթոդ, որը դադարեցվում է հոսքը նշված ժամանակով:
- `IsAlive` - (get) վերադարձնում է տրամաբանական արժեք հոսքի թողարկման մասին:
- `IsBackground` - (get, set) կարդում է կամ ստեղծում ֆոնային հոսք:
- `Name` - հնարավորություն է տալիս տալ հոսքին հասկանալի անուն:
- `Priority` - (get, set) հնարավորություն է տալիս տալ հոսքին առավելություններ:
- `ThreadState` - կարդում է հոսքի վիճակի ինֆորմացիան:
- `Abort()` - տրվում է CLR - ին հրաման հոսքի ավարտի մասին:
- `Join()` - դրվում է բլոկ մնացած հոսքերին մինչև կավարտվի իր ընթացիկ հոսքը:
- `Resume()` - վերաթողարկվում է դադարեցված հոսքը:
- `Start()` - տրվում է CLR - ին հրաման հոսքը սկսելու մասին:
- `Suspend()` - դադարեցվում է հոսքը:

```
using System;
using System.Threading;
class M
{
    static void Main()
    {
        Thread t = Thread.CurrentThread;
        t.Name = "TTTTT";
        Console.WriteLine("Main():" + Thread.CurrentThread.ManagedThreadId);
        Console.WriteLine("Name of current AppDomain: {0}",Thread.GetDomain().FriendlyName);
        Console.WriteLine("ID of current Context: {0}",Thread.CurrentContext.ContextID);
        Console.WriteLine("Thread Name: {0}", t.Name);
        Console.WriteLine("Has thread started?: {0} ", t.IsAlive);
        Console.WriteLine("Priority Level: {0}", t.Priority);
        Console.WriteLine("Thread State: {0}", t.ThreadState);
    }
}
```

❖ **Հոսքերի ստեղծում և ղեկավարում** [Троелсен-700; Troelsen-555]

```
using System.Threading;
using System;
class M
{
    static void Main()
    {
        Thread t = new Thread(new ThreadStart(ff));
        t.Start();
        for (int i = 0; i < 10000; i++)
            Console.WriteLine("################");
        // Console.ReadKey();
    }
    static void ff()
    {
        while (true)
            Console.WriteLine("                    ......................");
    }
}
//.....................................................................
```

```
using System;
using System.Threading;
class M
{
    static Thread t1; static  Thread t2;
    static void Main()
    {
        t1 = new Thread(new ThreadStart(ff1));
        t2 = new Thread(new ThreadStart(ff2));
        t1.Start();
        //t1.Join();
        t2.Start();
        //  t1.IsBackground = true;
        //  t2.IsBackground = true;
        //  t1.Suspend();
        //  t1.Resume();
        //  t1.Abort();
    }
    static void ff1()
    {
        Thread.Sleep(200);
        for (int i = 0; i < 100; i++)
            Console.WriteLine("inc+++++++: " + i);
    }
    static void ff2()
    {
        Thread.Sleep(200);
        for (int i = 100; i >= 0; i--)
            Console.WriteLine("            -------dec: " + i);
    }
}
```

❖ **Դելագատ ParameterizedThreadStart**

[Троелсен-704; Troelsen-557; Шилдт-844; Нэш-361]

- `ThreadStart` դելեգատի միջոցով կարելի է օգտագործել մեթոդ, որը վերադարձնում է `void` և չի ստանում պարամետրեր: Որպեսզի կարողանանք երկրորդային հոսքին **տալ պարամետրեր** օգտագործվում է `ParameterizedThreadStart` դելեգատը:

```
using System;
using System.Threading;
class A
{
    public int a = 10, b = 10;
}
class M
{
    static void Add(object d)
    {
        Console.WriteLine("Add():" + Thread.CurrentThread.ManagedThreadId);
        A ap = (A)d;
        Console.WriteLine("{0} + {1} is {2}", ap.a, ap.b, ap.a + ap.b);
    }
    static void Main()
    {
        Console.WriteLine("Main():" + Thread.CurrentThread.ManagedThreadId);
        A ap = new A();
        Thread t = new Thread(new ParameterizedThreadStart(Add));
        t.Start(ap);
    }
}
```

# 33. Հոսքերի նախապատվություններ

[Троелсен-701; Troelsen-554; Шилдт-847; Нейгел-582]

- `Thread` կլասը ունի հատկանիշ Priority, որով կարելի է հոսքին տալ տարբեր կատարման առավելություններ: Լռելիությամբ այն ստանում է `Normal`: Հատկանիշին կարելի է տալ տարբեր արժեքներ օգտվելով `System.Threading.ThreadPriority` թվարկումից:

```
public enum ThreadPriority
{
        AboveNormal,
        BelowNormal,
        Highest,
        Lowest,
        Normal, // Default value.
}
```

- Հոսքերի մակարդակները պլանավորողի համար ավելի շատ հուշող են քան պարտադրող:

```
using System;
using System.Threading;
class M
{
    Thread t1, t2, t3;
    static void Main()
    {
        M ob = new M();
        ob.ff();
        Console.ReadKey();
    }
    void ff()
    {
        t1 = new Thread(new ThreadStart(ffi));
        t2 = new Thread(new ThreadStart(ffd));
        t3 = new Thread(new ThreadStart(ff3));
        t1.Priority = ThreadPriority.Normal;
        t2.Priority = ThreadPriority.Normal;
        // t3.Priority = ThreadPriority.Lowest;
        t3.Priority = ThreadPriority.Highest;
        t1.Start();
        t2.Start();
        t3.Start();
    }
    void ffi()
    {
        for (int i = 0; i < 100; i++)
            Console.WriteLine("inc: " + i);
    }
    void ffd()
    {
        for (int i = 100; i >= 0; i--)
            Console.WriteLine("           -------dec: " + i);
    }
    void ff3()
    {
        for (int i = 100; i >= 0; i--)
            Console.WriteLine("                        ###############");
    }
}
```

❖ **Առաջնային և ֆոնային հոսքեր**

[Нейгел-581; Троелсен-702; Troelsen-559; Հэш-367]

- Գոյություն ունի առաջնային (foreground) և ֆոնային(background) հոսքեր: Լռելիությամբ ստեղծվում է առաջնային հոսք: Ստեղծվող հոսքերի քանակը անսահմանափակ է: Կա հետևյալ առանձնահատկությունը՝ **առաջնային հոսքի ավարտի հետ միաժամանակ ավարտվում են բոլոր ֆոնային հոսքերը**:
- Պրոցեսը շարունակվում է այնքան ժամանակ քանի դեռ գոյություն ունեն առաջնային հոսքեր:
- Thread կլասի միջոցով ստեղծված բոլոր լռելիությամբ հոսքերը առաջնային են:
- Ֆոնային հոսք ստանալու համար համակարգի կողմից օգտագործվում է ThreadPool կլասը:
- Եթե անհրաժեշտ է առաջնային հոսքը դարձնել ֆոնային, ապա դա կարելի է կատարել Thread կլասի IsBackground հատկանիշով, նրան տալով true.

```
Thread t = new Thread(new ThreadStart(ff));
//t.IsBackground = true;
t.Start();
Console.WriteLine("main");
void ff()
{
    for (int i = 0; i < 30; i++)
        Console.WriteLine("increment: " + i);
}
```

❖ **class AutoResetEvent** [Troelsen-558]

- One simple and thread-safe way to force a thread to wait until another is completed is to use the AutoResetEvent class. In the thread that needs to wait, create an instance of this class and pass in false to the constructor to signify you have not yet been notified. Then, at the point at which you are willing to wait, call the WaitOne() method. When the other thread is completed with its workload, it will call the Set() method on the same instance of the AutoResetEvent type.

```
AutoResetEvent _waitHandle = new AutoResetEvent(false);
Thread t = new Thread(new ThreadStart(ff));
t.IsBackground = true;
t.Start();
Console.WriteLine("main");
_waitHandle.WaitOne();
void ff()
{
    for (int i = 0; i < 30; i++)
        Console.WriteLine("increment: " + i);
    _waitHandle.Set();
}
```

❖ Խնդիր * Join() –ի կիրառություն // error

```
Thread t = new Thread(new ThreadStart(ThreadMethod));
t.Start();
Console.Write("Mainthread ");
t.Join();
Console.WriteLine(t.IsAlive);
t.IsBackground = true;
public static void ThreadMethod()
{       Console.Write("ThreadMethod");      }
```

- Բացատրություն:

✓ Պարզվում է, որ եթե Join() մեթոդ է կիրառվել, ապա t հոսքը ավատրված է համարվում և աշխատանքը կարելի է շարունակել: Այդ դեպքում, t.IsBackground = true հրամանը չի աշխատի (կառաջացնի Exeption), որովհետև հոսք չկա: Սակայն այլ հատկանիշներ կարող են աշխատել, օրինակ Console.WriteLine(t.IsAlive); հրամանը:

# 34. Հոսքերի սինխրոնիզացիա

[Троелсен-707; Troelsen-560; Нейгел-595; Нэш -373]

- C# -ում հոսքերի հետ աշխատանքը կախված է CLR – ի առանձնահատկությունից: Օրինակ, հստակ հնարավոր չէ ասել միանգամից կկատարվի այն, թե ոչ:
- Քանի որ, հոսքերը CLR – ի պահանջով կարող են տեղափոխվել կոնտեքստի սահմաններից, պետք է հաշվի նստել նաև այն բանի հետ, որ ծրագրի որոշ էլեմենտներ գտնվում են տարբեր հոսքերի ազդեցության տակ և կարող են վնասել միմյանց: Միայն **ատոմար** հրամաններն են ապահովված այդպիսի վնասներից, իսկ բազային հրամաններում նրանց քանակը չնչին է: Նույնիսկ **վերագրման** հրամանը ատոմար չէ:

**Հոսքերի սինխրոնիզացիայի դերը**

- Բազմահոսքային դոմեններն նույնպես գտնվում են հոսքերի ազատ թույլատրության տակ և որպեզի պաշտպանվեն տարբեր չնախատեսված միջամտումներից օգտագործվում են տարբեր սինխրոնիզացիայի միջոցներ – օրինակ, բլոկ համակարգ, մոնիտոներ, [Synchronization] ատրիբուտ և այլն:
- Չի կարելի ասել, որ .NET համակարգը լրիվ լուծում է բազմահոսքային պրոբլեմները, սակայն կարելի է ասել , որ շատ ավելի հեշտացվել է այն: Այդ հնարավորությունները կարելի է գտնել System.Threading անունի տարածքում: Սակայն նշված անունի տարածքը միակ ձևը չէ բազմահոսքային խնդիրներ ստանալու համար: Այդպիսի հնարավորություն ունի նաև ասինխրոն դելեգատները որոնց դեպքում տարբեր բլոկավորման համակարգերի անհրաժեշտություն չկա և ավելի հեշտ է կազմակերպման տեսակետից:
- Սինխրոնացում – կոորդինացնում է հոսքերի կատարումը: Հիմնականում երբ **միևնույն** ռեսուրսին են դիմում: Կամ հոսքը սպասում է իրադարձության, որը կախված է այլ հոսքից: Սինխրոնիզացիայի հիմքում ընկած է **բլակիրովկան**, այսինքն ղեկավարում է որևէ օբյեկտի կոդի բլոկին և այդ դեպքում թույլ չի տալիս այլ հոսքերին օգտագործել այդ մասը: Բլակիրովկան ընկած է C# -ի հիմքում, դրա համար նա շատ հեշտ է օգտագործվում:

- **Խնդիրը առանց սինխրոնիզացիայի**

```
using System.Threading;
class M
{
    static int c = 0;
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {
        while (c < 50)
            System.Console.WriteLine($"!!!!!!! {c++}");
    }
    static void ff2()
    {
        while (c < 50)
            System.Console.WriteLine($"            +++++++ {c++}");
    }
}
```

.....

# 35. lock - բլոկավորում

[Шилдт-849; Троелсен-709; Troelsen-562; Нейгел-601]

- Եթե հոսքում օգտագործվող **մեթոդը ստատիկ** չէ, ապա բլոկավորումը օգտագործվում է կամ `this` պարամետրի միջոցով կամ `Object` տիպի օբյեկտով (`lock` -ի պարամետր):
- Եթե հոսքում օգտագործվող մեթոդը ստատիկ է, ապա բլոկավորումը օգտագործվում է `Object` տիպի օբյեկտով (`lock` -ի պարամետր):

```
using System.Threading;
class M
{
    static int c = 0;
    static object ob = new object();
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {   //lock (ob)
        while (c < 30)
            lock (ob)
                System.Console.WriteLine($"!!!!!!! {c++}");
    }
    static void ff2()
    {   //lock (ob)
        while (c < 30)
            lock (ob)
                System.Console.WriteLine($"            +++++++ {c++}");
    }
}
//.....................................................................

using System.Threading;
class Program
{
    static int x = 0;
    static object ob = new object();
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread t = new Thread(f);
            t.Name = "hosq-" + i;
            t.Start();
        }
    }
    public static void f()
    {
        lock (ob) //lock (ob)
        {
            for (int i = 0; i < 5; i++)
                System.Console.WriteLine($"{Thread.CurrentThread.Name}:{x++}" );
        }
    }
}
.....................................................................................................................
```

## 36. class Monitor

[Шилдт-855; Нэш-382; Троелсен-711; Troelsen-564]

- Մոնիտորը ավելի բարդ խնդիրների համար է: Այս կլասը գտնվում է `System.Threading` անունի տարածքում: Նրա օբյեկտով հնարավորություն է տրվում՝ երբ սկսել և երբ վերջացնել սինխրոնիզացիան: Նրան երբեմն անվանում են ինտելեկտուալ բլակավորում: `lock` հիմնային խոսք է և օգտվում `Monitor` կլասից - այսինքն `Monitor` կլասի պարզեցված օգտագործումն է:
- Մոնիտորը հարմար է այն խնդիրների համար, որտեղ ռեսուրսների սինխրոնիզացիան և **հերթի** սպասման տեխնոլոգիաները ավելի երկար են ու բարդ: Կրիտիկական սեկցիաներ կոչվող սինխրոնիզացիայի մասեր կարելի մտնել և դուրս գալ `Monitor.Enter()` և `Monitor.Exit()` մեթոդներով: Երբ սինխրոնիզացում կատարվում է lock հրամանի միջոցով, ապա կատարվում է հետևյալ կոնստրուկցիայի կիրառություն ՝ Monitor.Enter(this);
  try { ..... }
  finaly
  { Monitor.Exit(this); }
  Այն տեսանելի է `ILDasm.exe` ուտիլիտով:
- Enter() , Exit() մեթոդների օգնությամբ կատարում է նույնը ինչ lock հրամանը ( սակայն lock - ը ներդրված հրաման է) : Որպեսզի կարողանանք ստանալ հնարավորություն որևէ օբյեկտի բլոկավորման, ապա կանչվում է Enter(): Բլոկավորումը հանելու համար օգտագործվում է Exit():
  public static void Enter(object ob)
  public static void Exit(object ob)
- Եթե Enter() մեթոդի կանչելու դեպքում տվյալ օբյեկտը հասանելի չէ, ապա հոսքը սպասում է այնքան ժամանակ մինչև օբյեկտը չի ազատվում:

✓ public static bool TryEnter(object ob) - այս մեթոդը վերադարձվում է true, եթե կանչվող հոսքը ստանում է բլոկ տվյալ օբյեկտի համար: Ամեն դեպքում հոսքը սպասում է իր հերթին կամ կարելի է սինխրոնիզացիայի այլ տարբերակի դիմել:

```
using System;
using System.Threading;
class M
{
    int c = 0;
    static void Main()
    {
        M ob = new M();
        ob.ff();
    }
    void ff()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Name = "ThreadOne";
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Name = "ThreadTwo";
        t2.Start();
    }
    void ff1()
    {
        Console.WriteLine(Thread.CurrentThread.Name);
        while (c < 50)
        {
            Monitor.Enter(this);
            System.Console.WriteLine($"!!!!!!! {c++}");
            Monitor.Exit(this);
        }
    }
```

```
    void ff2()
    {
        Console.WriteLine(Thread.CurrentThread.Name);
        while (c < 50)
        {
            Monitor.Enter(this);
            System.Console.WriteLine($"            +++++++ {c++}");
            Monitor.Exit(this);
        }
    }
}
```

.....

## 37. Wait(), Pulse(), PulseAll() մեթոդներ

[Шилдт-855; Нэш-387]

- Monitor կլասը ունի ավելի մեծ հնարավորություններ: Հոսքերը միմյանց վրա կարող են ազդել `Wait(), Pulse(), PulseAll()` մեթոդների միջոցով: Օրինակ, եթե հոսքում ծրագրային հատվածին տալիս ենք `Monitor.Wait()`, ապա այն բաց է թողնում սինխրոնիզացվող մասը և այնտեղ կարող են մտնել այլ թույլատրվածներ, որոնք `Monitor.Pulse()`–ով հերթի մեջ են:
- Ենթադրենք հոսքը աշխատում է բլոկավորված ծրագրային մասում և այն դիմում է այլ ռեսուրսների, որոնք ստիպում են սպասել: Ուրեմն ավելի հարմար է հոսքին ժամանակավոր ազատել բլոկավորումից, որին ծառայում են հոսքի `Wait(), Pulse(), PulseAll()` մեթոդները: Նշված մեթոդները հոսքերի փոխհարաբերությունների ձև է: Եթե հոսքը այլ ռեսուրսների նկատմամբ բլոկավորված է, ապա խելացի կլինի կանչել Wait() հրամանը և հոսքը կանցնի սպասման վիճակի: Ծրագրի այդ հատվածը դուրս է գալիս բլոկից և այլ հոսքեր կարող են մտնել այդ հատվածը: `Pulse(), PulseAll()` միջոցով հետ է վերադարձվում իր (զիճած) `Wait()` -ով տրված հոսքը: `Pulse()` –ի դեպքում ազատվում է ռեսուրս միայն հերթում առաջին սպասողի համար: `PulseAll()` –ի դեպքում ռեսուրսը ազատվում բոլոր սպասողների համար:
- public static bool Wait(object waitOb)
- public static bool Wait (object waitOb, int միլիվարկյաններ)
- public static void Pulse(object waitOb)
- public static void PulseAll(object waitOb)
- Օրինակում օգտագործվում է `Wait(), Pulse()` մեթոդները տարբեր հոսքերով հաջորդական թվեր ստանալու համար, սպասել և թույլատրել սկզբունքով: Պարզ է `Wait(), Pulse()` պետք է մտնեն որևէ սինխրոնիցիայի բլոկի մեջ:

❖ **Օրինակ 1**

```
using System.Threading;
class M
{
    static int c = 0;
    static object ob = new object();
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {
        while (c < 40)
        {
            lock (ob) // կամ Monitor.Enter(ob); //որևէ բլոկավուրում պարտադիր է
            {
```

```
                Monitor.Pulse(ob); //Monitor.Pulse(ob);
                System.Console.WriteLine($"!!!!!!! {c++}");
                Monitor.Wait(ob);
            }
        }
    }
    static void ff2()
    {
        while (c < 40)
        {
            lock (ob) // կամ Monitor.Enter(ob); //որևէ բլոկավուրում պարտադիր է
            {
                Monitor.Pulse(ob); //Monitor.Pulse(ob);
                System.Console.WriteLine($"            +++++++ {c++}");
                Monitor.Wait(ob);
            }
        }
    }
}
```

❖ **Օրինակ 2**

- Ծրագիրի ff1() հատվածը ցուցադրում է 20 անգամ “!!!!!!!” և սպասում 3 վարկյան: Այնուհետև 51 –ից շարունակում է մինչև 100: ff2() –ում “+++++++” ցուցադրումը չի կարող լինել 50-ից ավելին, որովհետև 50 –ի վրա Monitor.Wait(ob) սպասում է հավերժ՝ առանց թայմերի կամ `Monitor.Pulse()` -ի:

```
using System.Threading;
class M
{
    static int c = 0;
    static object ob = new object();
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {
        lock (ob)
            while (c < 100)
            {
                //Monitor.Pulse(ob);
                if (c == 20)
                    Monitor.Wait(ob, 3000);
                System.Console.WriteLine($"!!!!!!! {++c}");
            }
    }
    static void ff2()
    {
        lock (ob)
            while (c < 100)
            {
                //Monitor.Pulse(ob);
                if (c == 50)
                    Monitor.Wait(ob);
                System.Console.WriteLine($"            +++++++ {++c}");
            }
    }
}
```

# 38. Interlocked

[Шилдт-873; Троелсен-712; Troelsen-565; Нэш-375]

- Երբեմն անհրաժեշտ է լինում բազմահոսքային աշխատանքը իրականացնել մեկ փոփոխականի ուղեկցությամբ: Օրինակ, ենթադրենք անհրաժեշտ է իմանալ հոսքերի քանակը, որոնք ակտիվ են տվյալ պահին: Դրա համար կարելի է ընտրել ստատիկ փոփոխական և համապատասխանաբար ավելացնել կամ փոքրացնել արժեքը: Սակայն, այդպիսի աշխատանքների համար գոյություն ունի նաև պարզեցված սինխրոնիզացիա, որը իրագործվում է `Interlocked` կլասի միջոցով: Դա կատարվում է Interlocked.Increment() և Interlocked.Decrement() մեթոդների միջոցով:

✓ `Decrement()` - Անվտանգ փոքրացվում է 1 – ով: (++)
✓ `Increment()` - Անվտանգ ավելացվում է 1 –ով: (--)

- `Interlocked` կլասը կարող է նաև՝

✓ `Exchange()` - Անվտանգ վերագրվում է արժեք: (=)
✓ `CompareExchange()` - Անվտանգ համեմատում է երկու մեծություններ և եթե նրանք հավասար են, ապա նրանցից մեկը փոխարինվում է երրորդ պարամետրի արժեքով:
✓ `Add()` – Գումարում է երկու փոփոխականների արժեքները, արդյունքը թողնելով առաջինում: (+)
✓ `Read()` – Կարդում է հիշողությունից 64 բիթանոց տվյալ ատոմար մակարդակով, եթե գտնվում է 32 բիթանոց համակարգում, քանի որ 64 բիթը պահվում է երկու հասցեներում: 64 բիթանոց համակարգում կարդացումը ատոմար է:

```
public void AddOne()
{
     //   անվտանգ մեկով ավելացնել և վերադարձնել արդյունքը
     int newVal = Interlocked.Increment(ref i);
}

public void SafeAssignment()
{
       Interlocked.Exchange(ref i, 7); //  i -ին  վերագրել 7
}

public void CompareAndExchange()
{
       // Եթե i = 77, փոխել նրան 99 ով
       Interlocked.CompareExchange(ref i, 99, 77);
}
```

- Օրինակում WriteLine() – ը բլոկի մեջ չէ և ստուգելը `lock` – ի նման հնարավոր չէ:

```
//Interlocked.Increment(ref c)
using System;
using System.Threading;
class M
{
    static int c = 0;
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {
        while (c < 50)
```

```
            Console.WriteLine($"!!!!!!!" + Interlocked.Increment(ref c));
    }
    static void ff2()
    {
        while (c < 50)
            Console.WriteLine("          +++++++" + Interlocked.Increment(ref c));
    }
}
//.............................................................

// Interlocked.Add(ref a, b);
using System;
using System.Threading;
class M
{
    int a = 0;
    int b = 1;
    static void Main()
    {
        M ob = new M();
        ob.ff();
    }
    void ff()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        Thread t2 = new Thread(new ThreadStart(ff2));
        t1.Name = "Thread One";
        t2.Name = "Thread Two";
        t1.Start();
        t2.Start();
        Thread.Sleep(100);
        Console.WriteLine(a); // պետք է լինի 200 000 + 200 000 = 400 000
    }
    void ff1()
    {
        for(int i=0;i<200000;i++)
           a = a + b; //
           //Interlocked.Add(ref a, b);
        Console.WriteLine(Thread.CurrentThread.Name+ "- a= " + a);
    }
    void ff2()
    {
        for (int i = 0; i < 200000; i++)
           a = a + b; //
          //Interlocked.Add(ref a, b);
        Console.WriteLine(Thread.CurrentThread.Name + "- a= " + a);
   }
}
.....................................................................................................................
```

# 39. Mutex

[Шилдт-863; Нейгел-609; Нэш-395]

- `System.Thraeding.Mutex` – կլաս (mutual exclusion - **իրարամերժ**):
- `Mutex` –ը սինխրոնիզացիան կատարում է **օպերացիոն համակարգի միջոցով**: Այն էֆեկտիվ է օգտագործել այն դեպքում, երբ անհրաժեշտ է սինխրոնիզացնել տարբեր **պրոցեսներ**: Նրա իրականացումը պահանջում է ավելի շատ ռեսուրսներ (**34 անգամ** ավելի շատ ժամանակ քան `Monitor` -ը) և օգտվում է օպերացիոն համակարգի միջուկից: Բլոկը **սկսելու** համար օգտագործվում է `WaitOne()` մեթոդը, որը վերադարձնում է `bool` տիպ՝ `true`, եթե ավարտը տեղի է ունենում իր ընթացքով և `false`, եթե վերջանում է օպերացիոն համակարգի հատկացված ժամանակի պատճառով: Բլոկը հանվում է `RelaseMutex()` հրամանով, որը ոչինչ չի վերադարձնում (`void`):
- Եթե `Mutex` –ը տարբեր պրոցեսներում է կանչվում, ապա նշվում է նաև իդենտիֆիկացիոն անուն: Այդ անունը գտնվում է գլոբալ տարածքում և չպետք է համընկնի այլ անունների հետ: Եթե `Mutex` –ը օգտագործվում է առանց անունի, ապա այն համարվում է **լոկալ** և չի գործում տարբեր պրոցեսների միջև:
- public Mutex();
- public Mutex(bool initiallyOwned);
- public Mutex(bool initiallyOwned, string name);
- public Mutex(bool initiallyOwned, string name, out bool createdNew);
  createdNew – ն `true` է, եթե մյուտեքսը բավարարվել է և `false`, եթե չի բավարավել:

```
using System;
using System.Threading;
class M
{
    static Mutex mt = new Mutex();
    static int c = 0;
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {
        while (c < 50)
        {
            //Thread.Sleep(1);
            bool b = mt.WaitOne();
            Console.WriteLine($"!!!!!!! {c++} {b}");
            mt.ReleaseMutex();
        }
    }
    static void ff2()
    {
        while (c < 50)
        {
            //Thread.Sleep(2);
            bool b = mt.WaitOne();
            Console.WriteLine($"            +++++++ {c++} {b}");
            mt.ReleaseMutex();
        }
}   }
```

- Փորձարկել WaitOne() և ReleaseMutex() տարբեր կոմբինացիաներ, որոնք կտան Runtime error:
- `Mutex` –ի դանդաղության պատճառով հնարավոր է երկու կամ ավելի արդյունքի ցուցադրում:

# 40. Semaphore

[Шилдт-867; Нейгел-610; Нэш-396 ]

- Semaphore սինխրոնիզացիան **նման է** Mutex -ին, այն տարբերությամբ, որ թույլատրվում ղեկավարել հոսքերի միաժամանակ դիմման քանակի չափը: Դիմման քանակը ղեկավարվում է Semaphore –ի հաշվիչով: Եթե հոսքին անհրաժեշտ է ընդհանուր ռեսուրս, ապա դիմում է թույլատրության Semaphore -ի օբյեկտին նրա հաշվիչի միջոցով: Հաշվիչի զրոյի դեպքում սպասում է, իսկ դրական թվի դեպքում ստանում է թույլատրություն և Semaphore -ի հաշվիչի պարունակությունը **փոքրացվում է մեկով**: Երբ հոսքը ավարտում է աշխատանքը Semaphore -ի հաշվիչի պրունակությունը ավելանում է մեկով:
- Semaphore սինխրոնիզացիան լայն կիրառություն ունի **ցանցային** կապերում, որտեղ թույլատրության քանակը մեկից ավել է և միևնույն ժամանակ սահմանափակ:
- Օրինակում (2,2) պարամետրի դեպքում տվյալ հոսքի բաժնում ավելացված թիվը հաջորդաբար չէ, որովհետև թույլատրվում է երկու հոսքերի միաժամանակ դիմում "int c" ռեսուրսին: (1,1) պարամետրի դեպքում, տվյալ հոսքի բաժնում ավելացված թիվը հաջորդաբար է ներկայացվում, որովհետև սինխրոնիզացիան սահմանափակվում է մեկ հոսքի սահմանում:
- Semaphore կլասի կոնստրուկտորներից է public Semaphore(int initialCount, int maximumCount): initialCount պարամետրը սկզբնական թույլատրության քանակն է: maximumCount պարամետրը այն քանակն է, որ կարող է տրամադրել Semaphore –ը:
- Semaphore –ը ինչպես Mutex –ը, օգտվում է WaitOne() մեթոդից, որը ժառանգվում է WaitHandle կլասից: WaitOne() մեթոդով ստանում ենք թույլատրություն ռեսուրսին, իսկ Release() մեթոդով ազատում ռեսուրսը:

```
using System;
using System.Threading;
class M
{
    //static Semaphore sm = new Semaphore(1, 1);
    static Semaphore sm = new Semaphore(2, 2);
    static int c = 0;
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static  void ff1()
    {
        while (c < 50)
        {
            sm.WaitOne();
            Console.WriteLine($"!!!!!!! {c++}");
            sm.Release();
        }
    }
    static  void ff2()
    {
        while (c < 50)
        {
            sm.WaitOne();
            Console.WriteLine($"            +++++++ {c++}");
            sm.Release();
        }
}   }
```

# 41. Event սինխրոնիզացիա

[Шилдт-870; Нейгел-613; Нэш-398]

- Իրադարձության մեթոդի կիրառությունը նույպես սինխրոնիզացիա է կազմակերպում ամբողջ համակարգի տարածքում: Այն իրականացվում է `ManualResetEvent` կամ `AutoResetEvent` կլասների միջոցով, որոնք գտնում են `System.Threading` անունի տարածքում և ժառանգվում են `EventWaitHandle` կլասից: Իրադարձություններով սինխրոնիզացիան .NET 4.0 -ում ավելացվեցին `ManualResetEventSlim` և `CountdownEvent` նոր կլասներով:
- Event սինխրոնիզացիան օգտագործվում է այն ժամանակ, երբ հոսքը **սպասում է** ինչ որ **իրադարձության**, որը տեղի պետք է ունենա մեկ այլ հոսքում: Հենց այդպիսի իրադարձություն տեղի է ունենում, այլ հոսքը տեղեկացնում է և սպասող հոսքը շարունակում է իր աշխատանքը:
- `ManualResetEvent` կամ `AutoResetEvent` կլասները ունեն `bool` պարամետով կոնստրուկտորներ, որի `true` -ն որոշում է հենց սկզբից իրադարձության տեղեկություն: `false` –ի դեպքում սկզբնական տեղեկությունը բացակայում է:
- `WaitOne()` մեթոդը, որը նույնպես ինչպես `Mutex` –ը ժառնագվում է `WaitHandle` կլասից, սպասում է այլ հոսքից իրադարձության ազդանշանին:
- `Set()` մեթոդը առաջացնում է իրադարձություն տվյալ հոսքում և պահում է ազդանշանային վիճակում:
- `Reset()` մեթոդը հոսքին բերում ոչ ազդանշանային վիճակի: `ManualResetEvent` կլասի օգտագործման դեպքում `Reset()` մեթոդի կիրառությունը պարտադիր է:

```
using System;
using System.Threading;
class M
{
    //static ManualResetEvent mre = new ManualResetEvent(false);
    static AutoResetEvent mre = new AutoResetEvent(false); // (true)
    static int c = 0;
    static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(ff1));
        t1.Start();
        Thread t2 = new Thread(new ThreadStart(ff2));
        t2.Start();
    }
    static void ff1()
    {
        while (c < 50)
        {   //Thread.Sleep(1);
            mre.WaitOne();//mre.Set(); //*
            Console.WriteLine($"!!!!!!! {c++}");
            mre.Set();//mre.WaitOne(); //*
            //mre.Reset();
        }
    }
    static void ff2()
    {
        while (c < 50)
        {
            mre.Set();
            Console.WriteLine($"            +++++++ {c++}");
            mre.WaitOne();
            //mre.Reset();
        }
    }
}
```

# 42. Timer

[Нэш-410; Нейгел-620; Троелсен-714; Troelsen-566]

- **`Timer`** կլասը գոյություն ունի հետևյալ անունների տարածքում.
- `System.Threading` – Ունի հզոր ֆունկցիոնալ հնարավորություններ: Փոխանցվում է դելեգատ, որը կանչվում է նշված ժամանակի միջակայքում: System.Threading.Timer - ի կոնստրուկտորի առաջին պարամետրը մեթոդ է, որը վերադարձնում է `void` և ունի `object` տիպի պարամետր: 2 – րդ պարամետրը ցանկացած օբյեկտ է, որը ստանում է կատարող ֆունցիան: 3- րդ պարամետրը ֆունկցիայի թողարկման սկզբնական ժամանակն է: 4-րդ պարամետրը ֆունցիայի կրկնման ժամանակը, որի - 0 կամ `Timeout.Infinite` արժեքը նշում է միայն մեկ թողարկում:
- `System.Timers` – վերցնում է Component կլասից և համարվում է ղեկավարման էլեմենտ: `System.Timers.Timer` աշխատում է հետևյալ կերպ - այն մեթոդը, որը պիտի կանչվի ժամանակի միջակայքի ավարտից հետո, կանչվում է `Elapsed` իրադարձության միջոցով: `Start() / Stop()` – ը ստեղծում է իրադարձություն, որը թողարկում է թայմերը: `AutoReset` – ը որոշում է կրկնությունը:
- `System.Windows.Forms` – Հետադարձ կապ կատարվում է միայն իր սահմաններում - ֆորմում:
- `System.Web.UI` – Վեբ ծրագրավորման համար:

```
using System; //Kort 220
using System.Threading;
public static class Program
{
    public static void Main()
    {
        Timer t = new Timer(ff, null, 0, 2000);
        Console. ReadKey();
    }
    private static void ff(Object ob)
    {
        Console.WriteLine(DateTime.Now);
        GC.Collect();
    }
}
//.....................................................................

using System; //Нэш 410 – namespase Threading // --
using System.Threading;
public class M
{
    private static void ff(object state)
    {
        Console.WriteLine("The current time is {0} on thread {1}",
        DateTime.Now, Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(1000);
    }
    static void Main()
    {
        Console.WriteLine("Press <enter> when finished\n");
        Timer myTimer = new Timer(ff, null, 0, 300);
        //Timer myTimer = new Timer(new TimerCallback(ff), null, 0, 300);
        Console.ReadKey();
        myTimer.Dispose();
    }
}
//.....................................................................
```

```
using System; // Нейгел 610  - namespace Timers
using System.Timers;
using System.Threading;
public class M
{
    private static void TimersTimer()
    {
        var t1 = new System.Timers.Timer(1000); // interval
        // var t1 = new Timer(1000); //error !! // why?
        t1.AutoReset = true;
        t1.Elapsed += TimeAction;
        t1.Start();
        Thread.Sleep(5000);
        t1.Stop();
        t1.Dispose();
    }
    static void TimeAction(object sender, ElapsedEventArgs e)
    {
        Console.WriteLine("System.Timers.Timer {0:T}", e.SignalTime);
    }
    static void Main()
    {
        TimersTimer();
    }
}
//.....................................................................
```

### ❖ Stopwatch

[Албахари-552]

- Նախատեսված է բարձր ճշտությամբ ժանամակի աշխատանքի և ցուցադրման համար:
- Անունի տարածք - System.Diagnostics

```
using System;
using System.Diagnostics;
using System.Text;
class Program
{
    const int iter = 20000;
    static void Main(string[] args)
    {
        Stopwatch sw = Stopwatch.StartNew();
        System.IO.File.WriteAllText("test.txt", new string('*', 30000000));
        Console.WriteLine(sw.Elapsed);
        //---------------
        string s = "";
        DateTime dt = DateTime.Now;
        for (int i = 0; i < iter; i++)
            s += "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n";
        Console.WriteLine(DateTime.Now - dt);
        //--------------
        string s2 = "";
        Stopwatch sw2 = Stopwatch.StartNew();
        for (int i = 0; i < iter; i++)
            s2 += "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n";
        Console.WriteLine(sw2.Elapsed);
        //---------------
        StringBuilder sb = new StringBuilder();
        dt = DateTime.Now;
        for (int i = 0; i < iter; i++)
            sb.Append("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n");
```

```
        Console.WriteLine(DateTime.Now - dt);
        //----------------
        StringBuilder sb2 = new StringBuilder();
        sw2 = Stopwatch.StartNew();
        for (int i = 0; i < iter; i++)
            sb2.Append("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n");
        Console.WriteLine(sw2.Elapsed);
    }
}
```

## 43. volatile մոդիֆիկատոր

[Kort – 40; Нэш-64, 378; Шилдт – 710]

- `System.Threading.Volatile` կլասը հնարավորություն է տալիս անջատել կոմպիլիատորի JIT օպտիմիզատորը: `Volatile` կլասը ունի `Write()` և `Read()` մեթոդները, որոնց օգտագործումը ուղղեկցվում է կոմպիլիատորի **օպտիմիզացիայի անջատումով:** Նշված հնարավորությունը կարելի է իրականացնել ավելի պարզ ձևով օգտվելով `volatile` մոդիֆիկատորից:
- Կլասի դաշտը կարող է ընդունել `volatile` մոդիֆիկատորը: Ֆորմալ այս մոդիֆիկատորը ղեկավարում է փոփոխականին վերագրումը և նրանից կարդալը: Հնարավորություն է տրվում ցանկացած ժամանակ գրել կարդալ *օպերացիոն համակարգի* կողմից և որ ավելի հավանական է, օգտագործել այն **տարբեր հոսքերում**, առանց օպտիմիզացիոն կորուստի վտանգի:
- Եթե կոդը աշխատում է **մեկ հոսքի** սահմաններում, ապա C# կոմպիլիատորի **օպտիմիզացնող հնարավորությունները** չեն վնասում ծրագրին, որը կարող է նույնիսկ **հեռացնել** ավելցուկ կոդի հատված: Սակայն, եթե ծրագրային կոդը գտնվում է բազմահոսքային օգտագործման տիրույթում, ապա հնարավոր է տվյալների **վնասում** այն առումով, որ կոդը կգտնվի տարբեր հոսքերի կողմից կոնկուրենտ իրավիճակում և հնարավոր է մի հոսքում կատարվի օպտիմիզացիոն կիրառություն: Նշված կոնֆլիկտից խուսափելու համար դաշտը կարող է նշվել `volatile` մոդիֆիկատրով: Դա նշանակում է **JIT կոմպիլիատորը** չպետք է կիրառի այդ դաշտի վրա **օպտիմիզացիոն** մեխանիզմներ:
- Վերցնենք հետևյալ ծրագրային կոդի հատվածը, որը բացարձակ տեսական է և պրակտիկ օրինակով դժվար ցուցադրելի.

```
static int _flag = 0;
// volatile static int _flag = 0;
static int _value = 0;
public static void Thread1() // դեռ չաշխատած, կարող է օպտիմալացվել թարմանման ժամանակ
{
    _flag = 1;
    _value = 5;
}
public static void Thread2()
{
    if (_flag == 0)
        Console.WriteLine(_value);
}
```

✓ Եթե աշխատենք առանց հոսքերի բաժանման և կանչենք `Thread1()` `Thread2()` մեթոդների հաջորդականությունը, ապա ծրագրի օպտիմիզատորը `_flag` - ին անմիջական կվերագրի մեկ և միշտ կունենաք `_value = 5` արդյունքը:

✓ Եթե աշխատենք հոսքերով, ապա հնարավոր է օպտիմիզատորի `_flag = 1` արժեքը արդեն խափանի տարբր արդյունքներ ստանալու հոսքային հնարավորությունից, քանի որ հոսքի կիրառման դեպքում արդյունքը հավանականային է կախված ծրագրային հապաղումներից:

✓ Նշվածից խուսափելու համար կարելի է օգտագործել `volatile` մոդիֆիկատորը:

```
volatile static int _flag = 0;
```

# Ասինխրոն ծրագրավորում (Asynchronous programming)

## 44. Դելեգատների ասինխրոն բնույթը

[Троелсен-691; Нейгел-576; Нэш-403]

❖ **Դելեգատների ներքին կառուցվածքը**

[Троелсен-689]

- Դելեգատները հնարավորություն են տալիս ունենալ տիպային անվտանգությամբ մեթոդի ցուցիչ: Երբ ստեղծվում է դելեգատ, .NET համակարգը ստեղծում է `seald` կլաս, որը ժառանգում է `System.MulticastDelegate` (որը իր հերթին ժառանգվում է `System.Delegate` կլասից). Այդ բազային կլասները հնարավորություն են տալիս ստանալ մեթոդների հասցեներ, որոնց կարելի է կանչել հետագայում:
- `public delegate int dd(int x, int y);` այս հայտարարումը թարգմանիչը վեր է ածում հետևյալ կլասի:

```
sealed class dd: System.MulticastDelegate
{
      public dd(uint methodAddress);
      public int Invoke(int x, int y);
      public IAsyncResult BeginInvoke(int x, int y, AsyncCallback cb,  object state);
      public int EndInvoke(IAsyncResult result);
}
```

- `Invoke()` մեթոդը աշխատում է **սինխրոն** և դա նշանակում է քանի դեռ չի ավարտվել կանչված մեթոդը ծրագրի տվյալ հատվածը չի կարող շարունակվել:
- `BeginInvoke()` և `EndInvoke()` մեթոդները ապահովում է **ասինխրոն** աշխատանք, որը նշանակում է ծրագիրը կարող է շարունակվել **ֆոնային** հոսքում զուգահեռ ձևով: Նույն արդյունքին կարելի է հասնել հոսքերի միջոցով, սակայն նշված ձևը օգտագործման տեսակետից ավելի պարզ է:
- Դիտարկենք դելեգատի օգտագործումը, որը աշխատում է սինխրոն:

```
using System.Threading;
using System;
public delegate int dd(int x, int y);
class M
{
    static void Main()
    {
        Console.WriteLine($"Main() thread = {Thread.CurrentThread.ManagedThreadId}");
        dd b = new dd(Add);
        int answer = b(10, 10);
        Console.WriteLine("Doing more work in Main()!");
        Console.WriteLine($"10 + 10 = {answer}");
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add() thread = {Thread.CurrentThread.ManagedThreadId}");
        Thread.Sleep(1000);
        return x + y;
    }
}
//.....................................................................
```

### ❖ Դելեգատների ասինխրոն բնույթը

- Ասինխրոն դելեգատները հնարավորություն են տալիս մեթոդները կանչել **առանձին հոսքով** և չունենալ հոսքի կազմակերպման բարդույթները: Դա կատարվում է `BeginInvoke()` և `EndInvoke()` մեթոդների միջոցով:

```
sealed class dd : System.MulticastDelegate
{   // Used to invoke a method asynchronously.
   public IAsyncResult BeginInvoke(int x, int y, AsyncCallback cb, object state);
   // Used to fetch the return value of the invoked method.
   public int EndInvoke(IAsyncResult result);
}
```

- int x int y հայտարարվում են դելեգատում: Հաջորդ երկու պարամետրերը չօգտագործելու դեպքում վերագրվում է null :

- `IAsyncResult` ինտերֆեյսը կապող օղակ է հանդիսանում, երբ անհրաժեշտ է `EndInvoke()` մեթոդի միջոցով **արդյունք վերադարձնել** ուղարկող մեթոդին: Ավելի ճիշտ `BeginInvoke` – ը արդյունքը գրում է քեշում, իսկ `EndInvoke` -ը քեշից վերադարձնում է ծրագրին: Պարզ է, եթե մեթոդը void է վերադարձնում, ապա վերադարձման մշակում չի գնում, այսինքն, կարելի է մեթոդը կանչել ու մոռանալ նրա մասին:

```
public interface IAsyncResult
{
   object AsyncState { get; } // պարամետրի փոխանցում
   WaitHandle  AsyncWaitHandle { get; } // ժամանակի թողարկում
   bool CompletedSynchronously { get; } //
   bool IsCompleted { get; }  // մեթոդի ավարտի ստուգում
}
//.....................................................................

using System.Threading;
using System;
public delegate int dd(int x, int y);
class M
{
    static void Main()
    {
        Console.WriteLine($"Main() thread = {Thread.CurrentThread.ManagedThreadId}");
        dd b = new dd(Add);
        IAsyncResult iob = b.BeginInvoke(10, 10, null, null);
        Console.WriteLine("Doing more work in Main()!");
        for (int i = 0; i < 30; i++)
            Console.WriteLine("#######" +i);
        int answer = b.EndInvoke(iob); //
        Console.WriteLine($"10 + 10 = {answer}"); //
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add()  thread = {Thread.CurrentThread.ManagedThreadId}");
        for (int i = 0; i < 30; i++ )
            Console.WriteLine("               ----------" +i);
        Thread.Sleep(1000); //
        return x + y;
    }
}
//.....................................................................
```

- Հաջորդ օրինակը ցուցադրում է դելեգատների միջոցով զուգահեռացման սահմանափակումներից մեկը: Դելեգատները օգտվում են **հոսքերի “պուլից”** (կհ) և դրա պատճառով իրեն տրամադրվում է **սահմանափակ** հոսքեր, կախված պրոցեսորի (CPU) **հոսքերի** քանակից: Օրինակում `ff5()` մեթոդը չի աշխատի, քանի դեռ որևէ հոսք չի ազատվել:

```
using System.Threading;
using System;
public delegate void dd();
class M
{
    static int N = 30;
    static void Main()
    {
        Console.WriteLine($"Main() thread = {Thread.CurrentThread.ManagedThreadId}");
        dd b1 = new dd(ff1);
        b1.BeginInvoke(null, null);
        dd b2 = new dd(ff2);
        b2.BeginInvoke(null, null);
        dd b3 = new dd(ff3);
        b3.BeginInvoke(null, null);
        dd b4 = new dd(ff4);
        b4.BeginInvoke(null, null);
        dd b5 = new dd(ff5);
        b5.BeginInvoke(null, null);
        Thread.Sleep(1000);
        Console.WriteLine($"Main() thread = {Thread.CurrentThread.ManagedThreadId}");
    }
    static void ff1()
    {
        Console.WriteLine("ff1() Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i < N; i++)
            Console.WriteLine("  ######");
    }
    static void ff2()
    {
        Console.WriteLine("ff2() Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i < N; i++)
            Console.WriteLine("          ++++++");
    }
    static void ff3()
    {
        Console.WriteLine("ff3() Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i < N; i++)
            Console.WriteLine("                  ======");
    }
    static void ff4()
    {
        Console.WriteLine("ff4() Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i < N; i++)
            Console.WriteLine("                              ------");
    }
    static void ff5()
    {
        Console.WriteLine("ff5() Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i < N; i++)
            Console.WriteLine("                                          ??????");
    }
}
```

# 45. Ասինխրոն դելեգատների *սինխրոնիզացիա*

[Троелсен-692; Нейгел-576; Нэш-403]

- Ասինխրոն դելեգատը կորցնում է իր իմաստը (նախորդ օրինակում Main() – ը սպասում արդյունքի), եթե որոշ պայմանների դեպքում նա ընկնում է **բլոկի տակ** (պարապուրթ): Որպեսզի կարողանանք Main() - ում աշխատանքը շարունակենք պետք է օգտվել `IAsyncResult` ինտերֆեյսի `IsCompleted` հատկանիշից, որը հնարավորություն է տալիս ֆոնային հոսքի աշխատանքի **ավարտը ստուգել**: `IsCompleted` հատկանիշի `true` –ն ցույց է տալիս հոսքի աշխատանքի ավարտը:
- Նշված ձևով սինխրոնիզացիան ավելի մոտ է անընդհատ ստուգող **sreial polling** համակարգերին:
- Նախորդ օրինակից տարբերվում է նրանով, որ Main() -ը կաշխատի այնքան, որքան կաշխատի **Add** մեթոդը: Այս ձևով կարելի է օգտագործել Main() ի պարապուրթը:

```
using System.Threading;
using System;
public delegate int dd(int x, int y);
class M
{
    static void Main()
    {
        Console.WriteLine($"Main() thread = {Thread.CurrentThread.ManagedThreadId}");
        dd b = new dd(Add);
        IAsyncResult iob = b.BeginInvoke(10, 10, null, null);
        while (!iob.IsCompleted) //
        {
            Console.WriteLine("Doing more work in Main()!");
        }
        // while (!iob.AsyncWaitHandle.WaitOne(2, true))
        // {
        //    Console.WriteLine("Doing more work in Main()!");
        // }
        int answer = b.EndInvoke(iob);
        Console.WriteLine($"10 + 10 = {answer}");
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add() thread = {Thread.CurrentThread.ManagedThreadId}");
        Thread.Sleep(5);
        return x + y;
    }
}
//.....................................................................
```

### ❖ Սպասման դիսկրիպտոր

[Нейгел-577; Троелсен-694]

- Եթե անհրաժեշտ է ղեկավարել **ժամանակային** թողարկման ցուցանիշը, ապա օգտագործվում է հետևյալ ձևը.

```
while (!iob.AsyncWaitHandle.WaitOne(2, true))
{
        Console.WriteLine("Doing more work in Main()!");
}
```

- `IAsyncResult` ինտերֆեյսը ներկայացնում է նաև `AsyncWaitHandle` հատկանիշը, որը ունի ավելի ճկուն աշխատանքի ձև: Նա վերադարձնում է `WaitHandal` տիպի օբյեկտ, որի մեթոդներից է `WaitOne()`-ը: Նշված մեթոդը կարող է ղեկավարել սպասելու ժամանակը:
- Օրինակում ժամանակային ցուցիչը որոշում է Main() -ի թողարկման ընդհատումը: Այլ խոսքով ամեն 2 միլիվայրկյանից հետո կատարվում է `while` ցիկլը:

……..……

# 46. Դելեգատ AsyncCallback, Կլաս AsyncResult, Արգումենտի փոխանցում

[Троелсен-694; Нейгел-578; Нэш-403]

- Եթե անհրաժեշտ է, որպեսզի **չստուգենք** ասինխրոն կանչի ավարտը, կարելի է օգտագործել `BeginInvoke` մեթոդի պարամետրում `AsyncCallback` դելեգատը (նա մինչև հիմա `null` էր), որը **ավտոմատ տեղեկացնում** է իր մասին, երբ վերջացվում է ասինխրոն կանչը:
- `AsyncCallback` դելեգատը **որոշում է** IAsnycResult պարամետրը և վերադարձնում է `void` :
- Քանի որ, `BeginInvoke` -ը աշխաում է ֆոնային հոսքով, ապա գլխավոր հոսքը ավարտվելու դեպքում հնարավոր է ասինխրոն աշխատանքի ժամանակից շուտ ավարտ (օրինակում հանելով `Sleep` հրամանը):
- Օրինակում, երբ Add – ը վերջացնում է իր աշխատանքը, հետադարձ կապը իրականացվում է `ffzang()` մեթոդով:

```
using System.Threading;
using System;
public delegate int dd(int x, int y);
class M
{
    static void Main()
    {
        Console.WriteLine($"Main()thread = {Thread.CurrentThread.ManagedThreadId}");
        dd b = new dd(Add);
        IAsyncResult iob = b.BeginInvoke(10, 10, new AsyncCallback(ffzang), null);
        Console.WriteLine("Doing more work in Main()!");
        for (int i = 0; i <=30; i++)
            Console.WriteLine("###### "+i);
        Thread.Sleep(1000);
    }
    static void ffzang(IAsyncResult iob)
    {
        Console.WriteLine($"ffzang()  thread = {Thread.CurrentThread.ManagedThreadId}");
        //Console.WriteLine($"ffzang()  thread = {Thread.CurrentThread.GetHashCode()}");
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add()  thread = {Thread.CurrentThread.ManagedThreadId}");
        for (int i = 0; i <= 30; i++)
            Console.WriteLine("                !!!!!! "+i);
        return x + y;
    }
}
//.....................................................................
```

- Ցուցադրենք առանց `IAsyncResult`-ի օգտագործմամբ արդյունքի մշակում.

```
using System.Threading;
using System;
public delegate int dd(int x, int y);
class M
{
    static  dd b;
    static void Main()
    {
        Console.WriteLine($"Main()thread = {Thread.CurrentThread.ManagedThreadId}");
        b = new dd(Add);
        IAsyncResult iob = b.BeginInvoke(10, 10, new AsyncCallback(ff), null);
        Console.WriteLine("Doing more work in Main()!");
        for (int i = 0; i <= 30; i++)
            Console.WriteLine("######");
```

```
        Thread.Sleep(1000);
    }
    static void ff(IAsyncResult iob)
    {
        Console.WriteLine($"ff()  thread = {Thread.CurrentThread.ManagedThreadId}");
        Console.WriteLine("sum = " + b.EndInvoke(iob));
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add()  thread = {Thread.CurrentThread.ManagedThreadId }");
        for (int i = 0; i <= 30; i++)
            Console.WriteLine("               !!!!!!");
        return x + y;
    }
}
//.....................................................................
```

❖ **Կլաս AsyncResult**

[Троелсен-696; Нэш-403]

- Եթե անհրաժեշտ է և օգտվել AsyncCallback դելեգատից և մշակել արդյունքը AsyncResult կլասի միջոցով, ապա անհրաժեշտ է օգտվել System.Runtime.Remoting.Messaging անունի տարածքից: Ավելացնենք նաև, որ տրվում է հնարավորություն AsyncCallback - ին ծառայող մեթոդում օգտվել գլխավոր դելեգատից: AsyncDelegate ստատիկ հատկանիշը վերադարձնում է հղում սկզբնական ասինխրոն դելեգատին:
- Արդյունքի մշակում AsyncResult կլասի միջոցով.

```
using System.Threading;
using System;
using System.Runtime.Remoting.Messaging;
public delegate int dd(int x, int y);
class M
{
    static void Main()
    {
        Console.WriteLine($"Main() thread {Thread.CurrentThread.ManagedThreadId}");
        dd b = new dd(Add);
        IAsyncResult iob = b.BeginInvoke(10, 10, new AsyncCallback(ff), null);
        Console.WriteLine("Doing more work in Main()!");
        Thread.Sleep(1000);
    }
    static void ff(IAsyncResult iob)
    {
        Console.WriteLine($"ff()  thread = {Thread.CurrentThread.ManagedThreadId}");
        AsyncResult ob = (AsyncResult)iob;
        dd b = (dd)ob.AsyncDelegate;
        Console.WriteLine("sum = " + b.EndInvoke(ob));
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add()  thread = {Thread.CurrentThread.ManagedThreadId}");
        return x + y;
    }
}
//.....................................................................
```

❖ **Արգումենտի փոխանցում ասինխրոն դելեգատով**

[Троелсен-697; Нэш-403]

- Այժմ նշենք `BeginInvoke(int x, int y, AsyncCallback cb, object state)` մեթոդի վերջի պարամետրի դերը: Այս պարամետրի միջոցով կարելի է փոխանցել հետադարձ կապի մեթոդին **լրացուցիչ ինֆորմացիա**: `object state` -ը հնարավորություն է տալիս փոխանցել ցանկացած տիպի փոփոխական: Այդ դեպքում հետադարձ կապի մեթոդը օգտվում է `AsyncState` **հատկանիշից**:

```
static void Main()
{
      IAsyncResult iob = b.BeginInvoke(10, 10, new AsyncCallback(ff),"barev #####");
}
static void ff(IAsyncResult iob)
{
      string s = (string)iob.AsyncState;
      Console.WriteLine(s);
}
//.....................................................................

using System.Threading;
using System;
public delegate int dd(int x, int y);
class M
{
    static dd b;
    static void Main()
    {
        Console.WriteLine($"Main() thread {Thread.CurrentThread.ManagedThreadId}");
        b = new dd(Add);
        IAsyncResult iob = b.BeginInvoke(10, 10, new AsyncCallback(ff),"sum = ");
        Console.WriteLine("Doing more work in Main()!");
        Thread.Sleep(1000);
    }
    static void ff(IAsyncResult iob)
    {
        Console.WriteLine($"ff()  thread = {Thread.CurrentThread.ManagedThreadId}");
        string s = (string)iob.AsyncState;
        Console.WriteLine( s+ b.EndInvoke(iob));
    }
    static int Add(int x, int y)
    {
        Console.WriteLine($"Add()  thread = {Thread.CurrentThread.ManagedThreadId }");
        return x + y;
    }
}
.....................................................................................................................
```

# 47. class ThreadPool

[Троелсен-716; Troelsen-568; Albahari -624; Нейгел-584, 629; Нэш-402]

- Հոսքերի “պուլը” **համակարգի կողմից** տրամադրվող **պատրաստի** հոսքերի ծառայություն է: Այն ազատում է նոր հոսքերի ստեղծման ժամանակ պահանջվող ռեսուրսներ **վատնող միջոցներից**:
- Հոսքերի “պուլը”, դա հնարավորություն է, երբ ծրագրավորողը ավելի է կենտրոնանում լուծվող խնդրի դժվարությունների մեջ, քան խորանում բարդ հոսքերի տեխնիկական հնարքներում:
- Համակարգի կողմից կատարվում է անհրաժեշտ քանակի հոսքերի տրամադրումը և ղեկավարումը կախված **համակարգչի հնարավորություններից** (պրոցեսորների քանակ) և գոյություն ունեն որոշ **սահմանապակումներ**, որոնք ստիպում են հրաժարվել հոսքերի “պուլից”: Օրինակ.

✓ Հոսքերի “պուլը” աշխատում է **ֆոնային** ռեժիմում և թույլ չի տալիս փոփոխել հոսքի **առավելությունը**: Այն միշտ ունի `ThreadPriority.Normal` վիճակ:

✓ Երբեմն անհրաժեշտ է լինում հոսքը **կանգնացնել և վերաթողարկել**, որը նույնպես չի թույլատրվում:

✓ Հոսքերի “պուլը” չի տրամադրի պրոցեսորի (CPU) **հոսքերից ավել զուգահեռություն:**

- Հոսքերի “պուլը” հարմար է օգտագործել, երբ գոյություն ունեն ծրագրի **ոչ մեծ միավորներ**, որոնք իրականացվում են ասինխրոն ռեժիմում:
- Հոսքերի “պուլը” կազմակերպվում է ստատիկ `ThreadPool` կլասով, որը գտնվում է `System.Threading` **անունի տարածքում**:
- Որպեսզի պատրաստի հոսքից օգտվենք օգտագործում ենք `ThreadPool.QueueUserWorkItem()` **մեթոդը**: Նշված մեթոդը գերբեռնված է:

```
public static bool QueueUserWorkItem(WaitCallback callBack);
public static bool QueueUserWorkItem(WaitCallback callBack, object state);
```

- `WaitCallback` դելեգատը, որը կփոխարինի մեր զուգահեռ աշխատող մեթոդին (օրինակում `ff()`), ոչինչ չի վրադարձնում և կարող է ստանալ `object` տիպի ոչ պարտադիր պարամետր: Եթե `QueueUserWorkItem()` մեթոդում երկրորդ պարամետրը չի նշվում, ապա ավտոմատ ստանում է `null`:

❖ Ծրագրում սկզբից ցուցադրվում է տվյալ համակարգչում տրամադրվող **մաքսիմում** աշխատանքային և մուտք/ելքի հոսքերի **քանակը**: Հետո կատարվում է `ff()` մեթոդի թողարկում հոսքերի “պուլում”, նրան տրամադրելով որևէ պատրաստի հոսք:

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        int nWorker;
        int nPort;
        ThreadPool.GetMaxThreads(out nWorker, out nPort);
        Console.WriteLine($"Max Threads : {nWorker}, I/O : {nPort}");
        //WaitCallback cb = new WaitCallback(ff);
        Console.WriteLine("Main thread : " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i < 18; i++)
        {
            // ThreadPool.QueueUserWorkItem(cb);
            ThreadPool.QueueUserWorkItem(ff); //
        }
        Thread.Sleep(1000);
    }
    static void ff(object state)
    {
            Console.WriteLine("Pool Thread: " + Thread.CurrentThread.ManagedThreadId);
    }
}
```

❖ Հոսքերի “պուլի” ծառայություններից է նաև **ասինխրոն դելեգատների** կազմակերպման համար պատրաստի հոսքի տրամադրումը: Հաջորդ օրինակը կարելի է համեմատել ասինխրոն դելեգատների սինխրոնիզացիայի օրինակի հետ:

- Օրինակը **արգելում է ավել** `ThreadPool` հոսք տրամադրել, քան կարող է տալ տվյալ համակագչի պրոցեսորը: Հինգ մեթոդները թողարկվում են 4 պուլերում, եթե համակարգիչի պրոցեսորը ունի 4 հոսք (`Thread`): Եվ այդ դեպքում `ff5()` մեթոդը կցվում է այն ժամանակ, երբ **ազատվում** է որևէ “պուլ” և ????? կցուցադրվի վերջում: Main() –ը գտնվում է առանձին հոսքում անկախ տրամադրված `ThreadPool` հոսքերի քանակից:

```
using System;
using System.Threading;
class Program
{
    static int N = 20;
    static void Main(string[] args)
    {
        Console.WriteLine("Main thread = " + Thread.CurrentThread.ManagedThreadId);
        ThreadPool.QueueUserWorkItem(ff1);
        ThreadPool.QueueUserWorkItem(ff2);
        ThreadPool.QueueUserWorkItem(ff3);
        ThreadPool.QueueUserWorkItem(ff4);
        ThreadPool.QueueUserWorkItem(ff5);
        Thread.Sleep(500);
    }
    static void ff1(object state)
    {
        Console.WriteLine("Pool Thread: " + Thread.CurrentThread.ManagedThreadId);
        for(int i=0; i <= N; i++)
           Console.WriteLine("  !!!!!!");
    }
    static void ff2(object state)
    {
        Console.WriteLine("Pool Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i <= N; i++)
            Console.WriteLine("         ++++++");
    }
    static void ff3(object state)
    {
        Console.WriteLine("Pool Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i <= N; i++)
            Console.WriteLine("                  ======");
    }
    static void ff4(object state)
    {
        Console.WriteLine("Pool Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i <= N; i++)
            Console.WriteLine("                            ------");
    }
    static void ff5(object state)
    {
        Console.WriteLine("Pool Thread: " + Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i <= N; i++)
            Console.WriteLine("                                      ??????");
    }
}
```

## TPL (Task Parallel Library) [Троелсен-717; Troelsen-569]

- **Անունի տարածքը** `System.Threading.Tasks`: Նշված անունի տարածքը ընդգրկում է գրադարան – TPL(Task Parallel Library): Այն զուգահեռ ծրագրավորման նորույթ էր `C# 4.0` – ում: TPL գրադարանը ավտոմատ հասկանում է ծրագրի բաշխումը տարբեր ռեսուրսներում: `Task` կլասով կատարում է **դինամիկ** բաշխում պրոցեսորների միջև օգտագործելով հոսքերի **ThreadPool** հնարավորությունը և սահմանափակումը: TPL գրադարանից օգտվելու դեպքում ավելի **պարզեցված է ծրագրավորումը:**

## 48. class Task

[Шилдт-887; Нейгел-585; Троелсен-723; Troelsen-575; Albahari -626]

- TPL – ի հիմքում դրված է **`Task`** կլասը: Այն տարբերվում է `Thread` կլասից նրանով, որ աշխատում է **ասինխրոն** ձևով և ասինխրոն դելեգատների նման օգտվում է հոսքերի "պուլից" ապահովելով զուգահեռության **ֆոնային** հոսքով աշխատանքը:
- Ի տարբերություն `Thread` կլասի, որը պահում է հոսքի անուն Name –ի տակ, **`Task`** -ը չունի Name, սակայն նա ունի `Id` հատկանիշ, որը պահում խնդրի իդենտիֆիկատորը:
- **`Task`** -ի կոնստրուկտորներից է `Task(Action ac)`, որտեղ `Action` -ը դելեգատ է, որը վրադրաձնում է void և չի ստանում պարամետր: **`Task`** –ը զուգեռությունը թողարկում է `Start()` հրամանով:
- Այսպիսով **`Task`** կլասը աշխատում է ֆոնային հոսքում, այսինքն, եթե ավարտվում գլխավոր `Main()` մեթոդի հոսքը, ապա ավտոմատ **ավարտվում է `Task`** -ի աշխատանքը: Որպեսզի `Main()` մեթոդում չօգտագործենք `Sleep()` հրամանը, կարելի է օգտագործել **`Task`** կլասի `Wait()` մեթոդը, որը `Main()` -ին կստիպի սպասել այնքան ժամանակ, մինչև չավարտվի ֆոնային ծրագրերը: Եթե սպավող առաջադրանքների քանակը շատ է, ապա կարելի է օգտագործել `WaitAll()` մեթոդը, որը ստատիկ է և ունի `params` պարամետր: Գոյություն ունի նաև `WaitAny()` մեթոդը, որը կարելի է կիրառել այն դեպքում, երբ անհրաժեշտ է սպասել միայն հոսքերից **ցանկացած մեկի** ավարտին:
- ❖ **Օրինակը** ստեղծում է զուգահեռություն **մեկ մեթոդի** կիրառումով և սինխրոնիզացիայի բացակայության պատճառով (**`f()`** մեթոդ ռեսուրսին 2 "թասկեր" են դիմում) **i** փոփոխականը ստանում է անկանոն արժեքներ և նույնիսկ ունի թվերի կրկնություն:

```
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
class Program
{
    //static object ob = new object(); // for synchronization
    static void f()
    {
        WriteLine("Task f() Id = " + Task.CurrentId);
        WriteLine("Thread f() Id = " + Thread.CurrentThread.ManagedThreadId);
        //lock (ob) // for synchronization
        //for (int i = 0; i <= 30; i++)
               // WriteLine("         Task" + Task.CurrentId + "......." + i);
    }
    static void Main(string[] args)
    {
        WriteLine("Main Thread: " + Thread.CurrentThread.ManagedThreadId);
        //f();
        Task.Run(()=>f());//
        //Task t1 = new Task(f);
        //Task t2 = new Task(f);
        //t1.Start();
        //t2.Start();
        //for (int i = 0; i <= 30; i++)
           //WriteLine("######" + i);
```

```
        //Task.WaitAll(t1, t2);
        // Task.WaitAny(t1, t2);
        // t1.Wait();
        // t2.Wait();
        Thread.Sleep(10);
    }
}
```

❖ Հաջորդ օրինակը զուգահեռացնում է մեթոդները և ապացուցում է, որ **Task** կլասը զուգահեռությունը **դինամիկ** բաշխում է պրոցեսորների միջև օպտիմալ ձևով, չանցնելով համակարգչի պրոցեսորների քանակը: Այլ խոսքով թողարկում է բոլոր զուգահեռ մեթոդները ThreadPool կլասի նման և ????? տողը չի ցուցադրվի ինչն չազատվի որևէ հոսքի "պուլ":

```
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
class Program
{
    static int N = 20;
    public static void f1()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("         !!!!!" + i);
    }
    public static void f2()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("                 +++++" + i);
    }
    public static void f3()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("                       =====" + i);
    }
    public static void f4()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("                               -----" + i);
    }
    public static void f5()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("                                       ?????" + i);
    }
    static void Main(string[] args)
    {   WriteLine("Main Thread: " + Thread.CurrentThread.ManagedThreadId);
        Task t1 = new Task(f1);   Task t2 = new Task(f2);    Task t3 = new Task(f3);
        Task t4 = new Task(f4);   Task t5 = new Task(f5);
        t1.Start(); t2.Start();  t3.Start();   t4.Start();  t5.Start();
        for (int i = 0; i <= N; i++)
            WriteLine("#####" + i);
        Task.WaitAll(t1, t2, t3, t4, t5);
   }
}
```

## 49. class Parallel

[Шилдт-906; Нэш-413; Троелсен-718,725; Troelsen-569; 577; Нейгел-589 ]

- `Parallel` կլասը նույնպես վերցնում է այնքան հոսք որքան պրոցեսոր գոյություն ունի (+1 Main – նինը, որը տրամադրվում է զուգահռ աշխատող ծրագրային մասերին) և սպասում է բոլոր հոսքերի ավարտին՝ առանց սպասման որևէ հրամանի օգտագործման: Ունի նաև զուգահեռացում **տվյալների մակարդակով** - օրինակ, զանգվածը մշակվում է իր տարբեր մասերում միաժամանակ:
- `Parallel` կլասը ունի հետևյալ մեթոդները
  `Parallel.Invoke()` – զուգահռացնում է **մեթոդների** աշխատանքը:
  `Parallel.For()`- զուգահռացնում է հաշվարկը **ցիկլում** և կարող է օգտագործել մի քանի հոսքեր:
  `Parallel.ForEach()` - զուգահռացնում է հաշվարկը `ForEach()` ցիկլում:

//.....

❖ **`Parallel.Invoke`** [Шилдт-906]

- `Invoke()` մեթոդը հնարավորություն է տալիս ծրագրի **զուգահեռ** կատարում, պարամետրում նշված մեթոդների միջոցով և Main –ի հոսքը տրամադվում է "դատարկ" թասկի և ստացվում է NxCPU + 1 զուգահեռություն: Օրինակում ավելացնել **նոր f6() մեթոդ և փորձել**:
- `Invoke()` -ի պարամետրը `Action()` դելեգատի զանգված է նշված `params` պարամետրով:
  `static void Invoke(params Action[] actions)`
- Այն մեթոդը, որը թողարկում է `Invoke()` -ին **բաց է թողնում** (ազատում է) իր հոսքը այլ մեթոդների համար և **սպասում է** նրանց ավարտին:
- Օրինակում չկա պրոցեսորների քանակի սահմանափակում հոսքերի համար և զուգահռությանը մասնակցում են բոլոր մեթոդները իրենց տրամադված հոսքերով:

```
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
class M
{
    static int N = 30;
    public static void f1()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("!!!!!" + i);
    }
    public static void f2()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("         +++++" + i);
    }
    public static void f3()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("                 =====" + i);
    }
    public static void f4()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
            WriteLine("                         -----" + i);
    }
    public static void f5()
    {
       WriteLine($"Task{Task.CurrentId} Thread{Thread.CurrentThread.ManagedThreadId}");
       for (int i = 0; i <= N; i++)
```

```
            WriteLine("                                    ?????" + i);
    }
    static void Main()
    {   WriteLine("Main Thread: " + Thread.CurrentThread.ManagedThreadId);
        Parallel.Invoke(f1, f2, f3, f4, f5);
        WriteLine("Main thread ending ##############################.");
}   }
//.....................................................................
```

❖ Parallel.For [Шилдт-910; Нейгел-589]

- For() մեթոդը հնարավորություն է տալիս զուգահեռ կատարում ցիկլի սահմաններում: Աշխատանքը կատարվում է **զանգվածների** կամ **կոլեկցիաների** հետ և օրինակում զուգահեռացումը իրականացվում է **պայմանական օպերատորի** միջոցով: Ծրագիրը կատարվում է **ֆոնային հոսքերում** և ինչպես արդեն նշել ենք, Main() **սպասում է** նրանց ավարտին: Հոսքերի քանակը **համապատասխանեցվում** են տվյալ համակարգչի **պրոցեսորի** հոսքերի քանակին և +1 Main ի հոսքը:

```
using System;
using System.Threading.Tasks;
using System.Threading;
class DemoParallelFor
{   static int[] ar;
    static void ff(int i)
    {
        if (ar[i] < 100)
        {
            ar[i] = 0;
            Console.WriteLine("   !!!!!!!");
        }
        if (ar[i] > 100 & ar[i] < 200)
        {
            ar[i] = 11;
            Console.WriteLine("           +++++++");
        }
        if (ar[i] > 200 & ar[i] < 300)
        {
            ar[i] = 22;
            Console.WriteLine("                     =======");
        }
        if (ar[i] > 300 & ar[i] < 400)
        {
            ar[i] = 33;
            Console.WriteLine("                                -------");
        }
        if (ar[i] > 400)
        {
            ar[i] = 44;
            Console.WriteLine("                                        ???????");
        }
        Console.WriteLine("Thread = " + Thread.CurrentThread.ManagedThreadId);
    }
    static void Main()
    {
        Console.WriteLine("Main starting.");
        ar = new int[500];
        for (int i = 0; i < ar.Length; i++)
            ar[i] = i;
        Parallel.For(0, ar.Length, ff);
        Console.WriteLine("Main  ending.");
}   }
```

.....

## 50. async, await (Framework 4.5)

[Троелсен-731; Troelsen-582; Албахари -587; Гриффитс-900]

- async,await ասինխրոն ծրագրավորումը նույնպես կատարվում է **ֆոնային ռեժիմում** և գլխավոր հոսքի ավարտի հետ միասին ավարտվում են async ֆոնային հոսքերը: async, await -ը հնարավարություն է տալիս **սինխրոն** կոդին **ազատել** զբաղեցրած հոսքը և այդ ձևով կազմակերպում ծրագրի կատարման **զուգահեռացում**:
- async նեդրված հրամանը հնարավորություն է տալիս մեթոդին, անանուն մեթոդին կամ լյամբդա արտահայտությանը աշխատել ասինխրոն ձևով: Միևնույն ժամանակ await հրամանի միջոցով ավտոմատ **կանգեցվում** է ընթացիք հոսքը:
- async, await միասնական աշխատանքը **պարտադիր** է: await հրամանը ներքին հրաման չէ և կապտում է միայն async նշված կառույցի ներսում: async հրամանը կարող է հանդես գալ ինքնուրույն, իսկ await –ը, ոչ:
- Մեթոդը, որը նշված է async, կարող է վերադարձնել երեք տիպի արժեքներ - void, Task, Task<T>, մեթոդը, որը նշված է await չի կարող void վերադարձնել, այլ միայն Task կամ Task<T>:

```
// Առանց զուգահեռության
using System;
using System.Threading;
class Program
{
    static async void ff()
    {
        Console.WriteLine("ff thread: "+Thread.CurrentThread.ManagedThreadId);
        for (int i = 0; i <= 10; i++)
             Console.WriteLine("ff()..............");
    }
    static void Main(string[] args)
    {   //int async = 99; // ok          int await = 77; //ok
        Console.WriteLine("Main thread: " + Thread.CurrentThread.ManagedThreadId);
        ff();
        Console.WriteLine("Main End");
    }
}
//.....................................................................
using System; // async,await
using System.Threading;
using System.Threading.Tasks;
class Program
{
    //static async Task ff() // եթե անհրաժեշտ է կապ դեպի Main()
    static async void ff()
    {
        Console.WriteLine("ff thread: "+Thread.CurrentThread.ManagedThreadId);
        Console.ReadKey();
        string s= await ff2();
        Console.WriteLine("ff thread after: " + Thread.CurrentThread.ManagedThreadId);
        Console.WriteLine(s);
        for (int i=0;i<=10;i++)
            Console.WriteLine("                end async" +i);
    }
    static  Task<string> ff2() // եթե անհրաժեշտ է string  արդյունք
    {   //Console.ReadKey();
        Console.WriteLine("ff2 thread: " + Thread.CurrentThread.ManagedThreadId);
        return Task.Run
               (() =>
               {
                  Console.WriteLine("ff2 Task.Run " + Thread.CurrentThread.ManagedThreadId);
```

```
                for (int i = 0; i <= 30; i++)
                        Console.WriteLine("        !!!!!" +i);
                   return "barev";
               }
               );
    }
    static void Main(string[] args)
    {
        Console.WriteLine("Main thread: " + Thread.CurrentThread.ManagedThreadId);
        ff();
        for (int i = 0; i <= 30; i++)
                Console.WriteLine("#####"+i);
        Console.WriteLine("End Main");
        Thread.Sleep(1000); //
        //ff().GetAwaiter().GetResult(); // Կպպասի ff() –ի ավարտին Task –ի դեպքում
        //Task t= ff();
        //t.Wait();// Կպպասի ff() –ի ավարտին Task –ի դեպքում
    }
}
//.....................................................................
```

- Կարելի է `async` -ի մեջ **մի քանի** `await` օգտագործել: Այդ դեպքում օգտվում ենք `await Task.WhenAll()` **մեթոդից**, որի պարամետրը `Task` տիպի վեկտոր է: Նրանք կաշխատեն զուգահեռ և ThreadPool կլասի նման միաժամանակյա զուգահեռության **սահմանափակում ունի** կախված ֆիզիկական **պրոցեսորների** քանակից:

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Program
{   static async Task ff()
    {
        Console.WriteLine("ff start : " + Thread.CurrentThread.ManagedThreadId);
        Task t1 = Task.Run
             (() =>
             {
                 Console.WriteLine("t1 : " + Thread.CurrentThread.ManagedThreadId);
                 for (int i = 0; i < 50; i++)
                     Console.WriteLine("     !!!!!");
             }
             );
        Task t2 = Task.Run
             (() =>
             {
                 Console.WriteLine("t2 : " + Thread.CurrentThread.ManagedThreadId);
                 for (int i = 0; i < 50; i++)
                     Console.WriteLine("          *****");
             }
             );
         await Task.WhenAll(new[] { t1, t2 });
         Console.WriteLine("ff end : " + Thread.CurrentThread.ManagedThreadId);
    }
    static void Main(string[] args)
    {   Console.WriteLine("Main : " + Thread.CurrentThread.ManagedThreadId);
        Task t = ff();
        for (int i = 0; i < 50; i++)
            Console.WriteLine("#####");
        t.Wait(); //
    }
}
```

❖ “Վիզուլ” ծրագրավորման ժամանակ պատուհանը մեկ հոսքի տակ է գտնվում և եթե որևէ ղեկավարման էլեմենտ է օգտագործվում, մնացած էլեմենտները սպասում են: Այստեղ մեծ օգնություն կլինի, եթե գլխավոր պատուհանի բաղկացուցիչ էլեմենտները աշխատեն զուգահեռաբար: Օրինակում գլխավոր պատուհանը առաջնային հոսքում է գտնվում, իսկ մնացածը կարող են թողարկվել ֆոնային հոսքում, օգտագործելով `async / await` ասինխրոն զուգահեռացումը: Օգտվենք հին ֆորմ օրինակից (**հարգանք անցյալին**).

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using System.Threading.Tasks;
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        Text = "albdarb";
        Button bt = new Button();
        bt.Size = new Size(100, 100);
        bt.Text = "Start";
        bt.Parent = this;
        bt.Click += bt_Click;
        TextBox tb = new TextBox();
        tb.Location = new Point(150, 150);
        tb.Size = new Size(400, 50);
        tb.Parent = this;
    }
    async void bt_Click(object sender, EventArgs e)
    {
        Text = await f();
    }
    Task<string> f()
    {
        return Task.Run
             (  () =>
                {
                  Thread.Sleep(7000);
                  return "barev";
                }
             );
    }
    // void bt_Click(object sender, EventArgs e)
    // {
    //    Text = f();
    // }
    // string f()
    // {
    //    Thread.Sleep(7000);
    //    return "barev";
    // }
}
```

.....

# Հայցային ծրագրավորում (Query Programming)

## 51. LINQ to Objects

[Шилдт-637; Троелсен-452; Troelsen-493; Нэш-525]

- LINQ - Language-Integrated Query
- Հնարավորություն է տրվում ստանալ տվյալներ ինֆորմացիոն կուտակիչից:
- LINQ – նման է SQL հայցերին, սակայն այն **խիստ տիպայնացված** է: Հայցերը վերածվում են **ընդլայնված մեթոդների** հաջորդականության, որոնք ընդունում են լյամդա արտահայտությունները: System.Linq անունի տարածքում գտնվող Enumerable և Queryable **ստատիկ** կլասները ընդգրկում են հայցերին ծառայող ընդլայնված մեթոդներ: (Нэш-528)
- Ֆունկցիոնալ ծրագրավորումը նոր թափ է ստացել շարփ լեզվում: Դա առավել ևս զգացվում է բազաների հետ աշխատանքի ժամանակ, որովհետև հայցերը դրանք ֆունկցիոնալ պրոցեսներ են և հիմնականում նրանց մոտ չի կիրառվում օբյեկտային մոտեցումը:
- Հայցը անցնում է **երկու էտապ**: Սկզբից ձևավորվում է հայցը և հետո ստանում ինֆորմացիան:
- LINQ - ով ձևավորվող հայցերը իրականացնող տվյալների աղբյուրը պետք է ունենա իրականացում IEnumerable կամ IEnumerable<T> ինտերֆեյս: Այդ թվարկման միջոցով հայցողին կարող է ըստ հերթականության տվյալներ տրամադրի: Օրինակ շարփում բոլոր զանգվածները ունեն նշված ինտրեֆեյսները:
- Հայցերի ներդված բառերը.

```
ascending           by          descending     equals
from                group       in             into
join                let         on             orderby
select              where
```

- Հայցը սկսվում է from ով և վերջանում select կամ group:
  - where – ստուգում է պայմանը:
  - orderby - կատարում է վերադասավորում, որի ընթացքում ascending և descending օգտագործվում է ըստ աճման կամ ըստ նվազման համար:
  - select - որոշում է վերադարձնող արտահայտությունը: Կարող է լինել մեկից ավել:
  - from - որոշում է ինֆորմացիայի աղբյուրները, որոն կարող են լինել մեկից ավելին:
  - group ….by …բանալի - հնարավորություն և տալիս հայցը խմբավորել ըստ բանալի բառի: Արդյունքում ստացվում է էլեմենտների հաջորդականություն , որը ունի IGrouping<Tkey,TElement> տիպ:
  - into – պահում է ժամանակավոր արդյունք, որպեսզի select կամ group կատարելուց հետո օգտագործվի որպես հայցի շարունակություն:
  - let – երբեմն անհրաժեշտ է լինում հայցում պահել ժամանակավոր տվյալներ:
  - join – միացնում և համատեղում է երկու հայցերը հաջորդաբար (տարբեր աղբյուրներից):
- var – ընդհանուր փոփոխականի տիպ է , որը որոշվում է թարգմանման ժամանակ:

Հայցում օգտագործվող միջանկյալ (օրինակ n) փոփոխականի տիպը պետք է համնկնի կուտակիչի էլեմենտի տիպի հետ: Եթե կուտակիչը չունի IEnumerable<T> ընդհանրացումը, ապա միջանկյալ փոփոխականի տիպը պետք է նշվի:

```
var posNums = from  n in nums:
```

կամ

```
IEnumerable<int> posNums = from  n in nums:
```

……

# 52. LINQ to object - from, where, select, orderby

[Шилдт-653,644,646,649; Нэш-530,535,534]

- **from** – որոշում է ինֆորմացիայի աղբյուրները, որոնք կարող են լինել մեկից ավելին:
- **where** – ստուգում է պայմանը:
- **select** – որոշում է վերադարձնող արտահայտությունը: Կարող է լինել մեկից ավել:

```
using System; //Шилдт - 639
using System.Linq;
//using System.Collections.Generic; // for collection
class M
{
    static void Main()
    {
        int[] nums = { 1, -2, 3, 0, -4, 5 };
        var posNums = from n in nums
                      where n > 0
                      select n; //return
        foreach (int i in posNums)
            Console.Write(i + " ");
        Console.WriteLine();
         // nums[1] = 777;
         foreach (int i in posNums)
              Console.Write(i + " ");
         Console.WriteLine();
    }
}
//.....
```

- **orderby** - կատարում է վերադասավորում, որի ընթացքում ascending և descending օգտագործվում է ըստ աճման կամ ըստ նվազման համար:

```
using System; //Шилдт - 646
using System.Linq;
class M
{
    static void Main()
    {
        int[] nums = { 1, -2, 3, 0, -4, 5 };
        var posNums = from n in nums
                      where n > 0
                      orderby n descending
                      select n;
        foreach (int i in posNums)
            Console.Write(i + " ");
        Console.WriteLine();
        nums[1] = 777;
        foreach (int i in posNums)
            Console.Write(i + " ");
        Console.WriteLine();
    }
}
```

.....

# 53. LINQ to object – group

[Шилдт-655; Нэш-538]

- **group … by …** **բանալի**; - հնարավորություն է տալիս հայցը խմբավորել ըստ հատուկ բառի: Արդյունքում ստացվում է էլեմենտների հաջորդականություն, որը ունի IGrouping<TKey,TElement> տիպ:
- Խնդիր, անհրաժեշտ է տրված կոլեկցիայից դուրս բերել ենթատեքստով նշված խումբ էլեմենտներ:

```
// Demonstrate the group clause //Шилдт-655;
using System;
using System.Linq;
class A
{
      static void Main()
      {
            string[] website = { "Name.A.com", "Name.B.net", "Name.C.net",
                                 "NameD.com", "NameE.org", "NameF.org",
                                   "Name.G.tv", "Name.H.net", "Name.I.tv" };
            var webadres = from addr in website
            group addr by addr.Substring(addr.LastIndexOf('.'));
            foreach(var s in webadres)
                  Console.WriteLine(s.Key);
      }
}
```

- `Substring()` – **`string`** տիպը ունի մեթոդ, որը հնարավորություն է տալիս վերադարձնել **ենթատեքստ,** որտեղից սկսված կատարվում է տեքստի հետ աշխատանք:
- `LastIndexOf()` – **`string`** տիպը ունի մեթոդ, որը որոշում է **վերջին** հանդիպած տեքստի տեղը:

```
//.....................................................................

using System;
using System.Linq;
class A
{
    static void Main()
    {
        string[] websites = { "NameA.com", "NameB.net", "NameC.net",
                              "NameD.com", "NameE.org", "NameF.org",
                               "NameG.tv", "NameH.net", "NameI.tv" };
        var webAddrs = from addr in websites
                       // where addr.LastIndexOf('.') != -1
                      group addr by addr.Substring(addr.LastIndexOf('.'));
        foreach (var sites in webAddrs)
        {
            Console.WriteLine("Web sites grouped by " + sites.Key);
            foreach (var s in sites)
                Console.WriteLine(" " + s);
            Console.WriteLine();
        }
    }
}
```

.....

# 54. LINQ to object – into, let

[Шилдт-657,659; Нэш-541,537]

- **into** – պահում է ժամանակավոր **արդյունք**, որպեսզի select կամ group կատարելուց հետո օգտագործվի որպես հայցի շարունակություն: **Խնդիր՝** անհրաժեշտ է ղեկավարել ձևավորված հայցի նույնատիպ պարունակության քանակի հսկում:

```
using System; // Demonstrate the into clause //Шилдт-657;
using System.Linq;
class A
{
    static void Main()
    {
        string[] website = {"NameA.com", "NameB.net", "NameC.net", "NameD.com", "NameE.org",
                            "NameF.org", "NameG.tv", "NameH.net", "NameI.tv" };
        var webadres = from addr in website
                       group addr by addr.Substring(addr.LastIndexOf('.'))
                           into w
                           where w.Count() > 2 select w;
        foreach (var s in webadres)
            Console.WriteLine(s.Key);

}   }
//.............................................
using System;
using System.Linq;
class A
{
    static void Main()
    {
        string[] websites = {"NameA.com", "NameB.net", "NameC.net", "NameD.com", "NameE.org",
                             "NameF.org","NameG.tv", "NameH.net", "NameI.tv" };
        var webAddrs = from addr in websites
                       group addr by addr.Substring(addr.LastIndexOf('.'))
                           into w
                           where w.Count() > 2 select w;
        foreach (var sites in webAddrs)
        {
            Console.WriteLine("Web sites grouped by " + sites.Key);
            foreach (var s in sites)
                Console.WriteLine(" " + s);
            Console.WriteLine();
        }
}   }
```

❖ **let** – երբեմն անհրաժեշտ է լինում հայցում պահել ժամանակավոր **տվյալներ**:

- Խնդիր, անհրաժեշտ է string զանգվածից վերցնել բոլոր սիմվոլները և սորտավորել:

```
using System; // Use a let clause and a nested from clause. //Шилдт-659
using System.Linq;
class LetDemo
{   static void Main()
    {
        string[] strs = { "alpha", "beta", "gamma" };
        var chrs = from s in strs
                   let chrArray = s.ToCharArray()
                   from ch in chrArray orderby ch select ch;
        foreach (char c in chrs)
            Console.Write(c + " ");
}   }
```

.....

# 55. LINQ to object - join

[Шилдт-660; Нэш-531]

- **join** – միացնում և համատեղում է երկու հայցերը հաջորդաբար (**տարբեր աղբյուրներից**):

```
// Demonstrate join //Шилдт-661
using System;
using System.Linq;
class Item
{
    public string Name { get; set; }
    public int ItemNumber { get; set; }
    public Item(string n, int inum)
    {
        Name = n;
        ItemNumber = inum;
    }
}
class InStockStatus
{
    public int ItemNumber { get; set; }
    public bool InStock { get; set; }
    public InStockStatus(int n, bool b)
    {
        ItemNumber = n;
        InStock = b;
    }
}
class Temp
{
    public string Name { get; set; }
    public bool InStock { get; set; }
    public Temp(string n, bool b)
    {
        Name = n;
        InStock = b;
    }
}
class JoinDemo
{
    static void Main()
    {
        Item[] items = {
        new Item("Toyota", 111),
        new Item("Hammer", 222),
        new Item("BMW", 333),
        new Item("Jip", 444)
        };
        InStockStatus[] statusList = {
        new InStockStatus(111, true),
        new InStockStatus(222, false),
        new InStockStatus(555, true),
        new InStockStatus(666, true)
        };
        var v = from n in items
                join m in statusList
                on n.ItemNumber equals m.ItemNumber
                select new Temp(n.Name, m.InStock);
        foreach (Temp t in v)
```

```
            Console.WriteLine("{0}\t{1}", t.Name, t.InStock);
    }
}
```

❖ LINQ to object - անանուն տիպեր [Шилдт-663]

```
// Demonstrate join Anonymous Types //Шилдт-663
using System;
using System.Linq;
class Item
{
    public string Name { get; set; }
    public int ItemNumber { get; set; }
    public Item(string n, int inum)
    {
        Name = n;
        ItemNumber = inum;
    }
}
class InStockStatus
{
    public int ItemNumber { get; set; }
    public bool InStock { get; set; }
    public InStockStatus(int n, bool b)
    {
        ItemNumber = n;
        InStock = b;
    }
}
class JoinDemo
{
    static void Main()
    {
        Item[] items = {
        new Item("Toyota", 111),
        new Item("Hammer", 222),
        new Item("BMW", 333),
        new Item("JIP", 444)
        };
        InStockStatus[] statusList = {
        new InStockStatus(111, true),
        new InStockStatus(222, false),
        new InStockStatus(555, true),
        new InStockStatus(888, true)
        };
        var v = from n in items
                join m in statusList
                on n.ItemNumber equals m.ItemNumber
                select new { N = n.Name, I = m.InStock };
         foreach (var t in v)
         Console.WriteLine("{0}\t{1}", t.N, t.I);
    }
}
```

.....

# 56. LINQ to Objects – Հայցային մեթոդներ

[Шилдт-669; Нэш-545]

- Ցանկացած հայց ունի իրեն համարժեք **հայցային մեթոդները** և հայցեր կարելի է ձևավորել նրանց միջոցով: Մեթոդները գտնվում են ինչպես `System.Linq.Enumerable` կլասում (**ստատիկ** կլաս իր **ընդլայնված** մեթոդներով), որը ունի `INumerable<T>` ինտերֆեյս, այնպես էլ հայցեր կարելի է ձևավորել նաև `System.Linq.Queryable` կլասի միջոցով (նույնպես **ստատիկ** կլաս իր ընդլայնված մեթոդներով), որը իրականացնում է `IQueryable<T>` ինտերֆեյսը:
- Նշենք `Enumerable` կլասի մեթոդները.

| Query Keyword | Equivalent Query Method |
|---|---|
| select | Select(*selector)* |
| where | Where(*predicate)* |
| orderby | OrderBy(*keySelector) or OrderByDescending(keySelector)* |
| join | Join(*inner, outerKeySelector, innerKeySelector, resultSelector)* |
| group | GroupBy(*keySelector)* |

- Գոյություն ունեն նաև լրացուցիչ մեթոդներ, որոնք կարող են կատարել տարբեր գործողություններ: Օրինակ - հաշվել մինիմում, մաքսիմում, միջին թվաբանական (Min(), Max(), Sum(), Average())և այլն:
- Հայցերը հիմնականում կատարվում են **ուշացած** ռեժիմում, երբ այն կարդում ենք, այլ ոչ թե հայցի ձևավորման ժամանակ: Կան լրացուցիչ մեթոդներ, որոնք հայցը կատարում են միանգամից, հայցի ձևավորման ժամանակ: Այդ մեթոդներից են Count(), ToArray(), ToList(): //

```
using System; //lambda //Шилдт-670
using System.Linq;
class M
{
    static void Main()
    {
        int[] nums = { 1, 2, 3, 10, 4, 5 };
        var posNums = nums.Where(n => n > 3).Select(n => n);
        //var posNums = nums.Where(n => n > 3).Select(r => r * 10);
        //var posNums = nums.Where(n => n > 3).OrderByDescending(j => j);
        foreach (int i in posNums)
            Console.Write(i + " ");
        string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                              "hsNameD.com", "hsNameE.org", "hsNameF.org",
                              "hsNameG.tv", "hsNameH.net", "hsNameI.tv" };
        var webAddrs = websites.Where(w => w.LastIndexOf('.') != -1).GroupBy
            (x => x.Substring(x.LastIndexOf(".")));
        foreach (var sites in webAddrs)
        {
            Console.WriteLine("Web sites grouped by " + sites.Key);
            foreach (var site in sites)
                Console.WriteLine(" " + site);
        }
            // Console.WriteLine("The maximum value is " + nums.Max());
            // Console.WriteLine("The first value is " + nums.First());
            // Console.WriteLine("The sum is " + nums.Sum());
            // Console.WriteLine("The average is " + nums.Average());
            // if(nums.All(n => n > 0))
                  //Console.WriteLine("All values are greater than zero.");
            // if(nums.Any(n => (n % 2) == 0))
                  //Console.WriteLine("At least one value is even.");
            // if(nums.Contains(3))
                  //Console.WriteLine("The array contains 3.");
    }
}
```

# 57. PLINQ - Հայցի զուգահեռ իրականացում

[Шилдт-917; Троелсен-728; Troelsen-580]

- PLINQ : Հնարավորություն է տալիս **հայցերի կատարման զուգահեռացում**: Հայցի ձևավորման ժամանակ կատարվում է **անալիզ** և եթե զուգահեռությունը **օգտակար** է, ապա օգտվում է PLINQ -ից, հակառակ դեպքում հայցը կատարվում է հաջորդական:
- Անհրաժեշտ **ընդլայնված** մեթոդները գտնվում էն `ParallelEnumerable` **ստատիկ** կլասում:
- **Զուգահեռ** աշխատանք կազմակերպելու համար ուղղակի կանչվում է `AsParallel()` մեթոդը:
- `WithDegreeOfParallelism()` մեթոդը ցուցում է, թե քանի **պրոցեսորից** ոչ ավելի պետք է մասնակցեն զուգահեռացմանը:
- `AsOrdered()`մեթոդը հնարավորություն է տալիս արդյունքը ցուցադրել **ինչպես որ կա կուտակիչում**:
- Օրինակում բացասական թվերի դուրս բերման հաջորդականության **խախտումը** ապացուցում է զուգահեռությունը:

```
using System; // Шилдт-918
        int[] data = new int[100_000_000];
        for (int i = 0; i < data.Length; i++)
            data[i] = i;
        data[1000] = -1;
        data[14000] = -2;
        data[15000] = -3;
        data[676000] = -4;
        data[8024540] = -5;
        data[9908000] = -6;
        data[99080000] = -7;
        var n = from val in data.AsParallel()
                        where val < 0
                        select val;
        foreach (var v in n)
                Console.Write(v + " ");
```

❖ class `CancellationTokenSource` [Шилдт-920; Троелсен-729; Troelsen-581]

- Հայցի `WithCancellation()` մեթոդը հնարավորություն է տալիս հայցի **դադարեցում**: Այն իրականացվում է `CancellationTokenSource` կլասի `Cancal()` մեթոդով և մշակվում է `OperationCanceledException` բացառություն միջոցով:

```
CancellationTokenSource obToken = new CancellationTokenSource();
int[] data = new int[100_000_000];
for (int i = 0; i < data.Length; i++)
    data[i] = i;
data[1000] = -1;  data[14000] = -2;  data[15000] = -3;
data[676000] = -4;  data[8024540] = -5;
data[9908000] = -6;
data[99080000] = -7;
var n= from val in data.AsParallel().WithCancellation(obToken.Token)
        where val< 0
        select val;
Task obTask = Task.Factory.StartNew(() =>
{
    obToken.Cancel();//Thread.Sleep(500);
});
try
{
    foreach (var v in n)
        Console.Write(v + " ");
}
catch (OperationCanceledException e)
{ Console.WriteLine(e.Message); } //obTask.Wait(); obTask.Dispose(); obToken.Dispose();
```

# 58. LINQ to XML

[Нейгел-1111; Covaci – 461]

- Անունի տարածք `System.Xml.Linq;`
- `XDocument` – ներկայացնում է ամբողջ XML դոկումենտը:
  `Save("sss.xml")` – պահպանում է XML ֆայլը:
  `Load("sss.xml")` – կարդում է XML ֆայլից:
- `XElement` - ներկայացնում է XML դոկումենտի էլեմենտները:
- `XNamespace` - ներկայացնում է XML դոկումենտի և էլեմենտների անունի տարածքը:
- `XComment` - ավելացնում է "կոմենտարիա" տեգ:
- `XAttribute` – տեգում ավելացնում է ատրիբուտ:
- `Descendants()`, ժառանգներ - մեթոդը վերադարձնում է **ֆիլտրացված կոլեկցիա**, ներառելով ժառանգված էլեմենտները: Օգտագործվում է հայցի ձևավորման ժամանակ:

```
using System;// Нейгел 1114
using System.Xml.Linq;
class Program
{
    static void Main(string[] args)
    {
        XElement x = new XElement("company","sony");
        Console.WriteLine(x);
        // XNamespace ns = "http://www.albdarb.com/ns/l";
        // XElement xe = new XElement(ns + "Company",
        XElement xe = new XElement("Company",//
new XElement("CompanyName", "AlbDarb"),
new XElement("CompanyAddress",
new XElement("Address", "123 dzmeruk bereq Street"),
new XElement("City", "Yerevan"),
new XElement("State", "Arabkir"),
new XElement("Country", "Armenia")));
        Console.WriteLine(xe.ToString());
    }
}
//.....................................................................
using System;// Нейгел 1115
using System.Xml.Linq;
class Program
{
    static void Main(string[] args)
    {
        XDocument xdoc = new XDocument();
        XComment xc = new XComment("Here is a comment.");
        xdoc.Add(xc);
        XElement xe = new XElement("Company",
new XAttribute("MyAttribute", "MyAttributeValue"),
new XElement("CompanyName", "AlbDarb"),
new XElement("CompanyAddress",
new XElement("Address", "123 dzmeruk bereq Street"),
new XElement("City", "Yerevan"),
new XElement("State", "Arabkir"),
new XElement("Country", "Armenia")));
        xdoc.Add(xe);
        Console.WriteLine(xdoc.ToString());
        xdoc.Save("sssNagel.xml");
        //XDocument x = XDocument.Load("sssNagel.xml");
        //Console.WriteLine(x);
    }
}
```

❖ Հայցի ձևավորում

```
using System;
using System.Linq;
using System.Xml.Linq;
class Program
{
    static void LINQtoXML()
    {
        var ob = new[]
        {
            new
            {
                Name = "Albert",
                LastName = "Darbinyan",
                age = 96
            },
            new
            {
                Name = "Ani",
                LastName = "Darbinyan",
                age = 21
            },
            new
            {
                Name = "Nune",
                LastName = "Darbinyan",
                age = 19
            }
        };

        var xmlob = new XElement("Root",
            from e in ob
            select
            new XElement("HighCode",
            new XElement("FirstName", e.Name),
            new XElement("LastName", e.LastName),
            new XElement("Age", e.age)));
        Console.WriteLine(xmlob);
        xmlob.Save("sss.xml");

        //XDocument x = XDocument.Load("sss.xml");
        //Console.WriteLine(x);
        //var q = from v in x.Descendants("FirstName")
        //        select v.Value;
        //Console.WriteLine("FirstName Count = " + q.Count());
        //Console.WriteLine("--------------------");
        //foreach (var item in q)
        //    Console.WriteLine(item);
    }
    static void Main(string[] args)
    {
        LINQtoXML();
    }
}
```

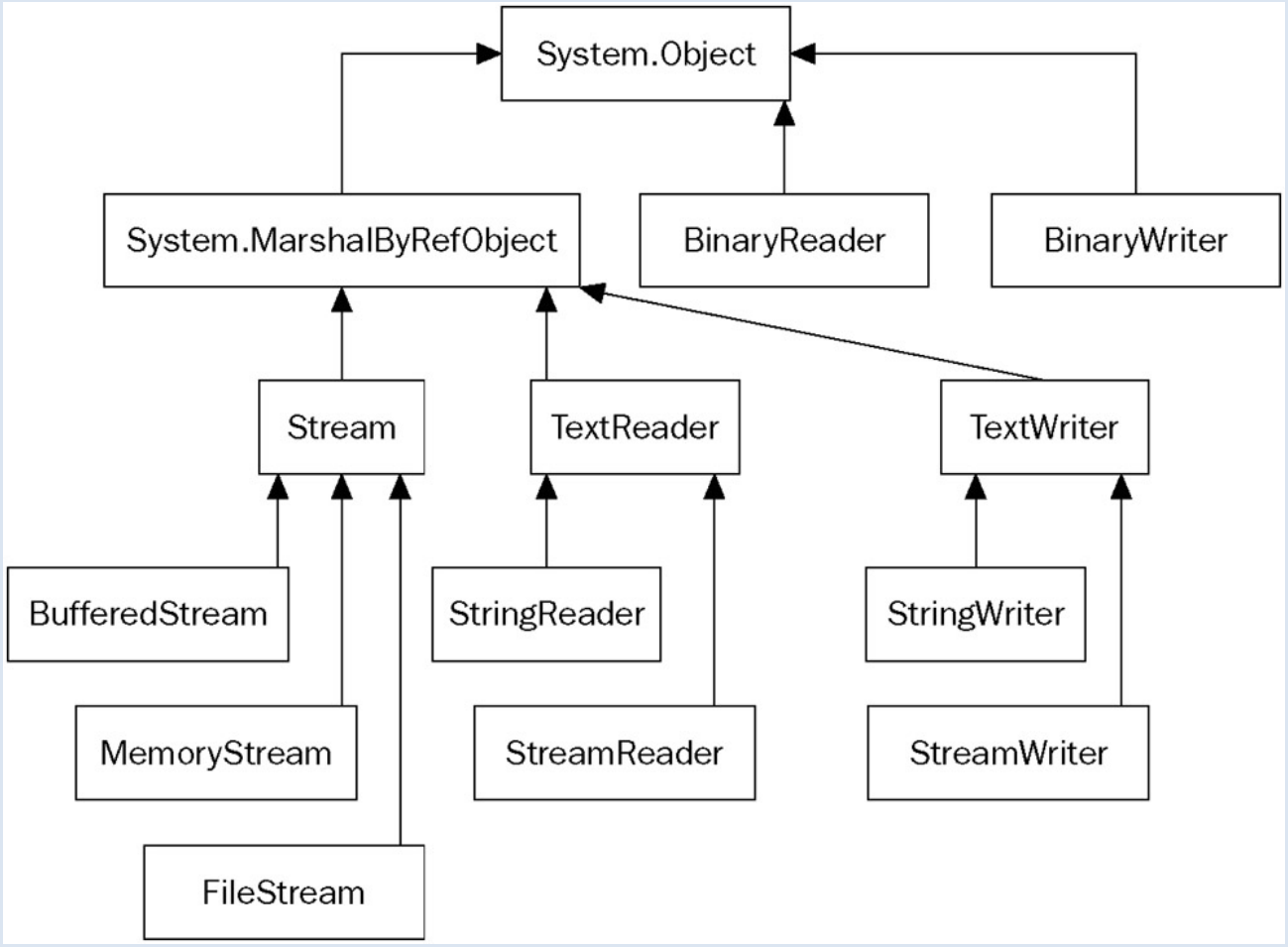
.....

# Մուտք/ ելք (Input/Output)

## 59. Տվյալների հոսքեր

[Нейгел-896; Шилдт-431]

- Երբ ծրագրերը աշխատում են տվյալների հետ, որոնք գտնվում են օպերատիվ հիշողությունում, ապա նրան դիմվում է անմիջականորեն: Սակայն երբ անհրաժեշտ է տվյալների դիմել, որոնք գտնվում են ֆայլում կամ ցանցային համակարգում, ապա անհրաժեշտ է տվյալները վերածել **տվյալների հոսքի** և աշխատել հաջորդականության հետ: Վերջին հաշվով այդ տվյալները պահվում են արտաքին հիշողությունում:
- Մուտք/ելքի ամենացածր մակարդակում տվյալների փոխանակումը կատարվում է **բայթերով**: Սակայն մարդու համար ավելի հարմար է տվյալների սիմվոլային տեսքը: Այդ դեպքում պետք է ունենալ սիմվոլայինից բայթի և հակառակ անցումներ: Անցումները կրում են բարդություններ, առավել ևս եթե գործ ունենք Unicode – ի հետ:
- Կարելի է կատարել հետևություն, որ բայթային հոսքերից բացի անհրաժեշտ է նաև տեքստային հոսքեր, որոնց անցումը պետք է թողնել պլատֆորմայի վրա:
- Ներդրված տվյալների հոսքերից են System անունի տարածքի Console.In, Console.Out և Console.Error հատկանիշները: Այսպես, երբ կանչվում է Console.WriteLine(), ապա ինֆորմացիան ավտոմատ տրվում է Console.Out հատկանիշին:
- NET համակարգում գործում են ինչպես բայթային, այնպես էլ տեքստային տվյալների հոսքերը: Թկուզ կարելի է ասել, որ տեքստայինը տրամաբանական կառույց է և միևնույնն է նա օգտվում է բայթայինից:
- Տվյալների հոսքերի հիմնական կլասները գտնվում են System.IO անունի տարածքում:



## ❖ Ֆայլային ինֆորմացիա և դիրեկտորիաներ

[Троелсен-740; Troelsen-724]

- System.IO
- Ապահովված է այս անունի տարածքում:
- Ֆայլային մուտք-ելք: Ֆայլը տվյալներ է պահում: Բայց, երբ դիմում ենք նրան, նա դառնում է հոսք:
- Նույնիսկ ցանցից ստացած ինֆորմացիան հանդիսանում է հոսք:
- Կատալոգների հետ կատարվող աշխատանքներ:

Object->Path, Directory, File  //static անդամներ
Object->MarshalByRefObject-> FileSysteminfo(abstract)->DirectoryInfo,  FileInfo - դիմում ենք օբյեկտներով:

```
using System;//FileInfo
using System.IO;
class M
{
    static void Main()
    {
        FileInfo fi = new FileInfo("C:\\1\\pic.gif");
        string name = fi.Name;
        string dir = fi.DirectoryName;
        string ext = fi.Extension;
        long length = fi.Length;
        DateTime crTime = fi.CreationTime;
        DateTime laTime = fi.LastAccessTime;
        Console.WriteLine(laTime);
        Console.WriteLine(crTime);
        Console.WriteLine(length);
        Console.WriteLine(ext);
        Console.WriteLine(dir);
        Console.WriteLine(name);
        fi.CopyTo("C:\\1\\arctic3.gif");
        fi.Delete();
    }
}
```

## ❖ class DriveInfo

[Troelsen-730]

```
DriveInfo[] myDrives = DriveInfo.GetDrives();
foreach (DriveInfo d in myDrives)
{
    Console.WriteLine("Name: {0}", d.Name);
    Console.WriteLine("Type: {0}", d.DriveType);
    if (d.IsReady)
    {
        Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
        Console.WriteLine("Format: {0}", d.DriveFormat);
        Console.WriteLine("Label: {0}", d.VolumeLabel);
    }
    Console.ReadKey();
}
```

.....

# 60. Աբստրակտ կլաս Stream

[Шилдт-432; Троелсен-752; Troelsen-737]

Մեթոդներ

**void Close ()** – Փակում է մուտք/ելք տվյալների հոսքը:

**void Flush ()** – Մուտք/ելք հոսքի պարունակությունը տեղադրում է ֆիզիկական սարքում:

**int ReadByte () –** Վերադարձնում է հաջորդ բայթի ամբողջ տիպի թվային տեսքը, մուտք/ելք հոսքից մուտք կատարելու համր: Եթե հանդիպում է **ֆայլի վերջ**, ապա վերադարձնում է **-1**:

**int Read (byte [ ] buf, int offset, int numBytes)** - Կատարում է մուտք, սկսած buf[offset]-ից Buf[numBytes] չափով: Վերադարձնում է հաջողված բայթերի քանակը:

**long Seek (long offset, SeekOrigin origin)** – Որոշում է ընթացիք տեղը, մուտք/ելք հոսքում նշված offset - ով և origin սկզբնական առժեքից: Վերադարձնում է մուտք/ելք հոսքի օգտագործման նոր տեղը:

**void WriteByte (byte b)** -Ուղղարկում է մեկ բայթ մուտք/ելք հոսքի մեջ:

**void Write (byte [ ] buf, int offset, int numBytes)** - Կատարում է ելք սկսած buf[offset]-ից Buf[numBytes] չափով:

Հատկանիշներ

**bool CanRead** - Ընդունում է `true`, եթե կարելի է հոսքից տվյալներ հանել: Միայն կարելի է կարդալ:

**bool CanSeek** - Ընդունում է `true`, եթե հոսքը կարող է ապահովել ընթացիք տեղը որոշող հայցը: Միայն կարելի է կարդալ:

**bool CanWrite** - Ընդունում է `true`, եթե կարելի է հոսք տվյալներ դնել: Միայն կարելի է կարդալ:

**long Length –** Պահում է հոսքի երկարությունը. Միայն կարելի է կարդալ:

**long Position** -Ներկայացնում է ընթացիք դիրքը հոսքում: Կարելի է կարդալ և գրել:

**int ReadTimeout** -Ներկայացնում է սպասման ժամանակը մուտքի ժամանակ: Կարելի է կարդալ և գրել:

**int WriteTimeout** -Ներկայացնում է սպասման ժամանակը ելքի ժամանակ: Կարելի է կարդալ և գրել:

### ❖ Ֆայլային մուտք /ելք

[Троелсен-753; Troelsen-738; Шилдт-441]

- **FileStream –** Գլխավոր կլաս: Ապահովում է բացումը, կարդալը, գրանցումը, փակելը:
- Այս կլասի կոնստրուկտորը կարող է առաջացնել IOException, FileNotFoundExeption իրավիճակ: Հիմնականում FileStream կլասի կոնստրուկտորը պետք է կանչել try բլոկում, որպեսզի սխալները մշակվի:
- Գրում կամ կարդում է ըստ բայթերի բազմության:
- Object->MarshalByRefObject->Stream(abstract)-> **FileStream,** MemoryStream, BufferedStream:
- Օբյեկտ ստեղծելիս կարելի է օգտվել հինգ կոնստուկտորներից, որտեղ կան հետևյալ թվարկումները FileMode, FileAccess, FileShare:

```
using System;//FileStream
using System.IO;
class M
{
    static void Main()
    {
        FileStream mm = new FileStream("alb.dat",
                        FileMode.OpenOrCreate, FileAccess.ReadWrite);
        for (int i = 0; i < 256; i++)
        {
            mm.WriteByte((byte)i);
        }
        mm.Position = 0;
        for (int i = 0; i < mm.Length; i++)
        {
```

```
            Console.Write(mm.ReadByte());
            Console.Write(" ");
        }
        mm.Close();
    }
}
```

❖ **Հոսք հիշողությամբ մուտք / ելք** [Шилдт-463]

- **MemoryStream**- Object->MarshalByRefObject->Stream(abstract)-> MemoryStream
- Հիշողության (RAM) մեջ ստեղծում է ֆայլ: Անչափ բարձրանում է արագագործությունը, սակայն ինֆորմացիայի կորստի մեծ վտանգ կա:

```
using System;
using System.IO;
class M
{
    static void Main()
    {
        MemoryStream ms = new MemoryStream();
        ms.Capacity = 277;
        for (int i = 0; i < 256; i++)
        {
            ms.WriteByte((byte)i);
        }
        ms.Position = 0;
        for (int i = 0; i < 277; i++)
        {
            Console.Write(ms.ReadByte());
            Console.Write(" ");
        }
        FileStream fs = new FileStream("alb.dat",
                                       FileMode.Create, FileAccess.ReadWrite);
        ms.WriteTo(fs);
        //byte[] bm = ms.ToArray();
        ms.Close();
    }
}
```

❖ **Բուֆերացված մուտք / ելք** [Нейгел-898]

- **BufferedStream** - Object-> MarshalByRefObject-> Stream(abstract)-> BufferedStream
- Եթե հոսքը բուֆերացված չէ, ապա կարելի է ստեղծել բուֆերացված մուտք/ելք , որը ավելի օպտիմալ է դիմում արտաքին սարքին:
- Close() հրամանը բուֆերից ուղղարկում է արտաքին ֆայլ: Սակայն, եթե բուֆերը լցվել է անկախ Close() հրամանի օգտագործման, բուֆերի պարունակությունը ուղղարկվում է արտաքին սարք:
- Օրինակում `i < 2048` արժեքից մեծի դեպքում **առանց Close()** հրամանի կատարում է գրանցում ֆայլի մեջ:

```
using System;
using System.IO;

class M
{
    static void Main()
    {

        FileStream df = new FileStream("t.dat", FileMode.Create);
        BufferedStream fb = new BufferedStream(df);
                for (int i = 0; i < 4; i++)
```

```
                    fb.WriteByte((byte)i);
        fb.Close();
        FileStream df2 = new FileStream("t.dat", FileMode.Open);
        // df2.Position = 0;
        for (int i = 0; i < 7; i++)
            Console.Write(df2.ReadByte() + " ");
        df.Close();
        df2.Close();
    }
}
//.....................................................

using System.IO;
class M
{
    static void Main()
    {
        FileStream df = new FileStream("alb.dat",
                                       FileMode.Create, FileAccess.ReadWrite);
        BufferedStream fb = new BufferedStream(df);
        byte[] str = { 127, 0x77, 0x4, 0x0, 0x0, 0x16 };
        fb.Write(str, 0, str.Length);
        fb.Close();
    }
}
```

❖ **Համեմատենք արագությունները**

```
using System;
using System.IO;
class M
{
    static void Main()
    {
        DateTime dt = DateTime.Now;
        MemoryStream ms = new MemoryStream();
        ms.Capacity = 1234567890;
        for (int i = 0; i < 1234567890; i++)
            ms.WriteByte((byte)i);
        ms.Close();
        Console.WriteLine("Memory Stream = " + (DateTime.Now - dt));
        //---------------------------
        DateTime dt2 = DateTime.Now;
        FileStream fs = new FileStream("fs.dat",FileMode.Create);
        for (int i = 0; i < 1234567890; i++)
            fs.WriteByte((byte)i);
        fs.Close();
        Console.WriteLine("Files  Stream = " + (DateTime.Now - dt2));
        //---------------------------
        DateTime dt3 = DateTime.Now;
        FileStream df = new FileStream("buf.dat", FileMode.Create);
        BufferedStream fb = new BufferedStream(df);
        for (int i = 0; i < 1234567890; i++)
            fb.WriteByte((byte)i);
        fb.Close();
        Console.WriteLine("Buffer Stream = " + (DateTime.Now - dt3));
    }
}
```

.....

# 61. Տեքստային մուտք / ելք

[Троелсен-754; Troelsen-740; Шилдт-449]

- Մուտք ելքը կատարվում է տեքստային ֆորմատով և լռելիությամբ Unicode – ով է:
- Object->MarshalByRefObject->TextWriter(abstract)-> StreamWriter, StringWriter
- Object->MarshalByRefObject->TextReader(abstract)-> StreamReader, StringReader

**StreamWriter**

- Close() - փակում է օբյեկտը և ազատում է նրա հետ կապված ռեսուրսները, բուֆերը ավտոմատ մաքրվում է:
- Flush() – որը վերաորոշված է: Մաքրում է բուֆերը և գրում է դիսկի վրա, բուֆերը չի փակում
- Write()- Նման է կոնսոլին, բայց գրում է դիսկի վրա - առանց Enter ի:
- WriteLine() - Նման է կոնսոլին, բայց գրում է դիսկի վրա Enter ընդունելով:

```
using System.IO;
class M
{
    static void Main()
    {
        FileInfo f = new FileInfo("c:\\1\\alb.txt");
        StreamWriter ww = f.CreateText();
        //  FileStream df = new FileStream("c:\\1\\111.txt", FileMode.Create);
        //  StreamWriter ww = new StreamWriter(df);
        ww.WriteLine("aaaaaaaaaaaaaaaaaaaa......");
        ww.WriteLine("bbbbbbbbbbbbbbbbbbbb.....");
        for (int i = 0; i < 10; i++)
            ww.Write(i + " ");
        //ww.Flush();
        ww.Close();
}   }
```

**StreamReader**

- Peek() – վերադարձնում է հաջորդ սիմվոլը չտեղափոխելով հոսքում մարկերը:
- Read() – կարդում է սիմվոլ փոխելով հոսքում մարկերը: Վերադաձնում է սիմվոլի `int` տիպը:
- ReadBlock() – Կարդում է նշված քանակով սիմվոլներ, սկսած որոշակի տեղից:
- ReadLine() – Կարդում է մինչև Enter:
- ReadToEnd() – Կարդում է բոլորը:

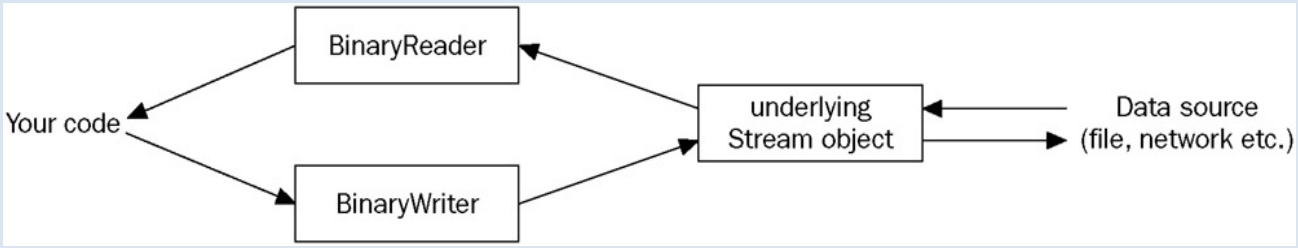
```
using System;
using System.IO;
class M
{
    static void Main()
    {
        StreamReader s = File.OpenText("c:\\1\\alb.txt");
        string i = null;
        while ((i = s.ReadLine()) != null)
            Console.WriteLine(i);
        // int j;
        // while ((j = s.Read()) != -1)
        // Console.Write(j + " ");
        // string all=s.ReadToEnd();
        // Console.WriteLine(all);
        s.Close();
}   }
```

- **StringWriter, StringReader** – Նման են **StreamReader** -ին հետևյալ տարբերությամբ – մուտք/ելքը կատարվում է բուֆերից և տեքստային ֆայլը պահվում է օպերատիվ հիշողությունում (կփորձենք **XmlSerializer** դասում):
- **System.Text.Encoding** - Կարող է դիսկի վրա տեքստային ֆայլը պահել 8 բիթով: // դաս **XmlSerializer**

# 62. Բինար մուտք / ելք

[Нейгел-897; Шилдт-454; Троелсен-758; Troelsen-744]

- **BinaryReader և BinaryWriter** - Այս կլասները իրենք **հոսքեր չեն իրականացնում,** բայց կարող են օբյեկտներին հնարավորություններ տրամադրել: Այլ խոսքով կարող են **ֆորմատավորել տվյալները** այնպես, որ կարելի է գրել - կարդալ անմիջական փոփոխականից: Այսինքն, BinaryReader и BinaryWriter կատարում են միջնորդի դեր:
- Բազային “տվյալների հոսքերը” աշխատում են **բայթերով:** Նշված կլասները կարող են հոսքից վերցնել բայթեր և ասենք վերածել int, long:




Object->BinaryReader, BinaryWriter

**BinaryWriter** - գլխավոր անդամները

- BaseStream – հոսքն է, որի հետ աշխատում է BinaryWriter օբյեկտը:
- Close() – փակում է հոսքը:
- Flush() – Մաքրում է բուֆերը:
- Seek() – փոխում է մարկերի դիրքը:
- Write() – գրանցում է տվյալը հոսք:

**BinaryReader** - գլխավոր անդամները

- BaseStream - հոսքն է, որի հետ աշխատում է BinaryReader օբյեկտը:
- Close() – փակում է BinaryReader օբյեկտը:
- PeekChar() –վերադարձնում է հաջորդ սիմվոլը չտեղափոխելով հոսքում մարկերը:
- Read() – կարդում է բայթերի հոսք և պահում է զանգվածում:
- ReadXXX() – (ReadByte, ReadInt32) – նույնը որոշակի տիպի համար:

```
using System;
using System.IO;
class M
{
    static void Main()
    {
        FileStream fs = new FileStream("c:\\1\\alb.dat",
                FileMode.OpenOrCreate, FileAccess.ReadWrite);
        BinaryWriter bw = new BinaryWriter(fs);
        int myInt = 99;
        float myFloat = 09456.789F;
        bool myBool = true;
        char[] myCharArray = { 'H', 'A', 'V', 'I', 'K' };
        bw.Write(myInt);
        bw.Write(myFloat);
        bw.Write(myBool);
        bw.Write(myCharArray);
        bw.BaseStream.Position = 0;
        BinaryReader br = new BinaryReader(fs);
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadSingle());
        Console.WriteLine(br.ReadBoolean());
        for (int i = 0; i < 5; i++)
            Console.WriteLine(br.ReadChar() + " " + br.PeekChar());
```

```
        // bw.BaseStream.Position = 0;
        // char[] chr = br.ReadChars(5);
        // for (int i = 0; i < 5; i++)
            //Console.Write(chr[i]);
        Console.WriteLine();
        bw.Close();
        br.Close();
        fs.Close();
    }
}
```

.....

## 63. Ֆայլային մարկերի տեղաշարժ - Seek(), Position

[Шилдт-461]

- long Seek(long offset, SeekOrigin origin)

✓ SeekOrigin.Begin – offset երկարությունը սկսում է սկզբից:

✓ SeekOrigin.Current – offset երկարությունը սկսում է ընթացիք տեղից:

✓ SeekOrigin.End – offset երկարությունը սկսում է վերջից:

```
using System;
using System.IO;
class RandomAccessDemo
{
    static void Main()
    {
        FileStream f = null;
        char ch;
        try
        {
            f = new FileStream("random.dat", FileMode.Create);
            for (int i = 0; i < 26; i++)
                f.WriteByte((byte)('A' + i));
            f.Seek(4, SeekOrigin.Begin); // seek to 5th byte
            ch = (char)f.ReadByte();
            Console.WriteLine("Fifth value is " + ch);
            for (int i = 0; i < 26; i += 2)
            {
                f.Seek(i, SeekOrigin.Begin); // seek to ith character
                ch = (char)f.ReadByte();
                Console.Write(ch + " ");
            }
            //for (int i = 0; i < 26; i += 2)
            //{
            //    f.Position = i; // seek to i-th character via Position
            //    ch = (char)f.ReadByte();
            //    Console.Write(ch + " ");
            //}
            Console.WriteLine();
        }
        catch (IOException exc)
        {
            Console.WriteLine("I/O Error\n" + exc.Message);
        }
        finally
        {
            if (f != null)
               f.Close();
        }
}   }
```

# 64. Ասինխրոն մուտք / ելք

[Liberty-454; albdarb; Троелсен2008-712]

- Մինչև հիմա խոսքը գնում էր սինխրոն մուտք-ելքի մասին, այսինքն այլ գործով չէր կարող զբաղվել ծրագիրը և պետք է սպասեր մինչև մուտք-ելքի ավարտը: Եթե օգտագործենք Stream կլասի BeginRead(), BeginWrite() մեթոդները, որը ապահովում է ասինխրոն մուտք-ելք, ապա թույլատրվում է ծրագրի **զուգահեռ** աշխատանք: Երբ մուտք-ելքը ավարտվում է, հետադարձ կապի միջոցը տեղեկացնում է դրա մասին - **callback**:
- Բացի երեք արգումենտներից, կարելի է ոչ պարտադիր ձևով նշել դելեգատ - հետադարձ կապի միջոցը և օբյեկտի վիճակ: Դելեգատի մեթոդը կանչվում է մուտք/ելքի ավարտից հետո:
- Սակայն առաջանում է մի պրոբլեմ՝ callback կանչող ծրագրի հատվածը չի հետևում callback–ի աշխատանքին: Այսինքն, եթե callback կանչող ծրագրի հատվածը ավելի շուտ է ավարտվում, ավտոմատ ավարտվում է նաև callback-ի աշխատանքը **(ֆոնային հոսք):**
- **Օրինակում**՝ ծրագիրը սկզբից ստեղծում է `alb.txt` ֆայլ և գրանցում `buffer` վեկտորի թվային պարունակությունը BeginWrite() -ով: Այնուհետև BeginRead() -ով կատարում է ընթերցում ֆայլից: Մուտք/ելքը իրականացվում է զուգահեր ֆոնային ռեժիմում և որպես callback օգտագործվում է ff() ff2() մեթոդները: Զուգահեռությունը ապացուցվում է **##### -ների և ++++++ ների միջոցով**: Ֆայլի կարդացումը ուղղեկցվում է նաև ff() մեթոդում buffer թվային վեկտորի string տիպի վերածումով: Որպեսզի ապացուցենք ֆայլի գրանցման ֆոնայնությունը `//* -`ից սկսած Main() –ի հրամանները դարձնել “քոմենտ”: Այդ դեպքում alb.txt ֆայլը կլինի դատարկ, որովհետև Main() –ի ավարտը կդադարեցնի ֆոնայինի աշխատանքը, որը չի հասցնի գրանցում սկսել ֆայլում:

```
using System;
using System.IO;
using System.Text;
class M
{
    static Stream rw;
    static byte[] buffer = new byte[128];
    public static void Main()
    {
        for (int i = 0; i < buffer.Length; i++)
            buffer[i] = (byte)i;
        rw = File.OpenWrite("alb.txt");
        rw.BeginWrite(buffer, 0, buffer.Length, new AsyncCallback(ff2), null);
        rw.Close();// *
        rw = File.OpenRead("alb.txt");
        rw.BeginRead(buffer, 0, buffer.Length, new AsyncCallback(ff), null);
        for (long i = 0; i < 30; i++)
            Console.WriteLine("########" + i);
        foreach (var item in buffer)
            Console.Write(item + " "); // output byte format
        Console.Write("End Main");
    }
    static void ff(IAsyncResult ar)
    {
        Console.WriteLine("end READ from file");
        //String s = Encoding.ASCII.GetString(buffer, 0, rw.EndRead(ar));
        string s = Encoding.ASCII.GetString(buffer, 0, buffer.Length);
        Console.WriteLine("Read in ff() = " + s); // output string format
        for (long i = 0; i < 50; i++)
            Console.WriteLine("                  +++++++++" + i);
    }
    static void ff2(IAsyncResult ar)
    {            Console.WriteLine("end WRITE in file");    }
}
```

# 65. Ցանցային մուտք / ելք

[Liberty-458; albdarb]

- Սկզբունքորեն չի տարբերվում **ֆայլային մուտք/ելքից,** հիմնված է հոսքի վրա ստեղծված սոքետների միջոցով: Կան տարբեր պրոտոկոլներ– օրինակ TCP/IP:
- Կլիենտ, սերվեր:
- Ճիշտ է TCP/IP – ն ստեղծված է ցանցով կապնվելու համար, բայց կարելի է իմիտացիա անել երկու պրոցես մեկ համակարգչի վրա և աշխատացնել կլիենտ սերվեր համակարգ:
- IP հասցեն Yahoo –ի: 216.114.108.245
- Բացի հասցեից միանում ենք նաև կոնկրետ պորտի հետ (0 - 65535): Լռելիությամբ վեբ բրաուզերները կապնվում են 80 պորտի հետ:
- Սոկետ ստեղծելուց հետո, ծրագրավորողը պետք է կանչի Start() մեթոդը:
- Երբ սերվերը պատրաստ է պատասխանի կլիենտի հայցին, պետք է կանչել AcceptSocket():
- Կապի համար օգտվում ենք Socket և TcpListener կլասներից:

❖ **Ցանցային server** [Liberty ; albdarb]

- Ցանցային սերվեր ստանալու համար տվյալների հոսքի TCP/IP պրոտոկոլով պետք է ստեղծել TcpListener կլասի օբյեկտ: Որպես պորտ վերցնենք 65000:
- TcpListener tl = new TcpListener(65000);
  Որից հետո նա կարող է լսել tl.Start();
  Սպասենք մինչև որևէ կլիենտի հայցի:
  Socket sc = tl.AcceptSocket();
- AcceptSocket( ) մեթոդը սինխիոն է:
- Եթե կապը հաստատվել է, ապա կարելի է կլիենտին ուղղարկել ֆայլ:
  if (sc.Connected) { }
  NetworkStream ns = new NetworkStream(sc);
  StreamWriter sw = new StreamWriter(ns);

```
// server
using System.Net.Sockets;
using System.IO;
        TcpListener tl = new TcpListener(65000);
        tl.Start();
        Console.WriteLine("Lsum em");
        Socket sc = tl.AcceptSocket();
        Console.WriteLine("Client connected");
        ff2(sc);
        sc.Close();
    void ff2(Socket sc)
    {
        NetworkStream ns = new NetworkStream(sc);
        StreamWriter sw = new StreamWriter(ns);
        StreamReader sr = new StreamReader("C:\\1\\t.txt");
        string s;
        s = sr.ReadToEnd();
        sw.WriteLine(s);
        sw.Flush();
        sr.Close();        ns.Close();        sw.Close();
    }
```

❖ **Ցանցային client** [Liberty ; Либерти2002-604]

- Կլիենտը ստեղծում է TcpClient կլաս օբյեկտ, TCP/IP պրոտոկոլով:
  TcpClient ss;
  ss = new TcpClient("localHost", 65000);
- Ստեղծվում է NetworkStream կլասի օբյեկտ և օգտվում ենք GetStream() մեթոդից:

```
    NetworkStream ns=ss.GetStream();
    StreamReader sr=new StreamReader(ns);
```
- Ցիկլով կարելի է կարդալ տվյալներ, քանի դեռ նրանք կան:

```
do
{       s = sr.ReadLine();   }while(s!=null);
// client
using System.Net.Sockets;
using System.IO;
        TcpClient ss;
        try
        {            ss = new TcpClient("localHost", 65000);          }
        catch
        {
            Console.WriteLine("server ??????");
            return;
        }
        NetworkStream ns = ss.GetStream();
        StreamReader sr = new StreamReader(ns);
        try
        {
            string s;
            s = sr.ReadToEnd();
            Console.WriteLine(s);
        }
        catch
        {            Console.WriteLine("exeption");         }
        ns.Close();
```

❖ **Ինտերնետ ցանցային կապ** [Шилдт-1014]

- System.Net – Անունի տարածք: Այս անունի տարածքի կլասները հնարավորություն են տալիս կապնվել ինտերնետում հայց-պատասխան համակարգով:

✓ WebRequest - աբստրակտ կլաս: WebResponse - աբստրակտ կլաս

✓ HttpWebRequest – ժառանգ: HttpWebResponse - ժառանգ

- Նշված անունի տարածքը կարող է ապահովել և սինխրոն և **ասինխրոն** կապ: Սինխրոնի դեպքում ծրագիրը սպասում է հայցին, չի կարող զբաղվել այլ գործով և ավելի պարզ է կիրառման տեսակետից: Ասինխրոնի դեպքում ծրագիրը կարող է **զուգահեռ** այլ աշխատանքով զբաղվել:

```
using System.Net; //Шилдт-1018
using System.IO;
        int ch;
        HttpWebRequest req = (HttpWebRequest)WebRequest.Create("http://www.google.com");
        HttpWebResponse resp = (HttpWebResponse)req.GetResponse();
        Stream istrm = resp.GetResponseStream();
        for (int i = 1; ; i++)
        {
            ch = istrm.ReadByte();
            if (ch == -1)
               break;
            Console.Write((char)ch);
            if ((i % 400) == 0)
            {
                Console.Write("\nPress Enter.");
                Console.ReadLine();
            }
        }
        resp.Close();
```

# 66. Սերիալիզացիա, Դեսերիալիզացիա

[Троелсен-762; Troelsen-748]

- “Սերիալիզացիան” հնարավորություն է տալիս հիշել կլասի **օբյեկտները ամբողջությամբ** և **պարզեզվում է** պահպանման պրոցեսը, քանի որ, առանձին փոփոխականների հիշելը ավելի **աշխատատար** պրոցես է: Կարելի է նշել, որ սերիալիզացիայով տվյալների պահպանումը **ավելի քիչ տեղ** է զբաղեցնում քան մուտք/ելքով իրականացումը:
- `Serialize` – նշանակում է օբյեկտի վերափոխումը գծային հաջորդականության հոսքի: Հակառակը պրոցեսը դա – `Deserialize:` Այստեղ կարող են տարբեր հարցեր առաջանալ: Եթե կա օբյեկտների կապվածություն, այն նույնպես պետք է հաշվի առնվի: Այդ դեպքում ստեղծվում է նրանց հարաբերության **գրաֆը**: Դա նման չէ կլասիկ օբյեկտային ծրագրավորման պատկանում է կամ ընդգրկում է հարաբերությանը: Այն ունի “**պահանջում է**” կամ “**կապվածություն ունի**” հարաբերությունից:
- Ժառանգականության ժամանակ անհրաժեշտ է բոլոր կլասներ ունենան `[Serializable]`, հակառակ դեպքում կարտածվի `SerializationException` բազառիկ իրավիճակ:
- `[Serializable]` ատրիբուտը դրվում է այն կլասի վրա, որը հաջողականացվում է: Եթե կա առանձին անդամ, որը պետք չէ հիշել, ապա նրա վրա դրվում է `[NonSerialized]` ատրիբուտը:
- Դեսերիալիզացիան հնարավորություն է տալիս օբյեկտը, որը պահպանվել է արտաքին միջավայրում կարդալ ծրագրային կոդում:
- **`BinaryFormatter`** սերյալիզացիա

```
using System.Runtime.Serialization.Formatters.Binary;
BinaryFormatter bf=new BinaryFormatter();
FileStream fs=new FileStream("c:\\111.out", FileMode.Create);
bf.Serialize(fs,this);
fs.Close();
```

```
// BinaryFormatter
using System.Runtime.Serialization.Formatters.Binary;
A ob = new A();
ob.n = 99;
ob.SSSerialize();
ob.n = 88;
A ob2 = new A();
ob2=ob2.DDDeSerialize();
Console.WriteLine(ob2.n + ob2.s);
[Serializable]
class A
{
    public int n;
    [NonSerialized] //
    public string s = "xxx";
    public void SSSerialize()
    {
        FileStream fs = new FileStream("c:\\1\\alb.out", FileMode.Create);
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(fs, this);
        fs.Close();
    }
    public  A DDDeSerialize()
    {
        FileStream fs = new FileStream("c:\\1\\alb.out", FileMode.Open);
        BinaryFormatter bf = new BinaryFormatter();
        return (A)bf.Deserialize(fs);
    }
}
```

.....

# 67. XmlSerializer

[Троелсен-772; Troelsen-752; metanit.com/sharp(pg.10)]

- **`XmlSerializer`** սերիալիզացիայի ժամանակ փակ դաշտերը չեն ընդգրկվում սերիալիզացիայում ինչպես `BinaryFormatter` և `SoapFormatter` կլասների օգտագործման ժամանակ:

```
using System;// Kort pg.308
using System.Xml.Serialization;
using System.IO;

[Serializable]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        XmlSerializer serializer = new XmlSerializer(typeof(Person));
        string xml;
        using (StringWriter stringWriter = new StringWriter())
        {
            Person p = new Person
            {
                FirstName = "John",
                LastName = "Doe",
                Age = 42
            };
            serializer.Serialize(stringWriter, p);
            xml = stringWriter.ToString();
        }
        Console.WriteLine(xml);
        using (StringReader stringReader = new StringReader(xml))
        {
          Person p = (Person)serializer.Deserialize(stringReader);
          Console.WriteLine("{0} {1} {2}", p.FirstName, p.LastName, p.Age);
        }
    }
}
```

.....

# 68. JSON Serializer

[Perkins-609; metanit.com/sharp(pg.10)]

- Կոնվերտացնենք XML -ից JSON: Դրա համար սկզբից պետք է ավելացնել NuGet փաթեթի միջոցով Newtonsoft.Json հղումը:
- Tools ➪ NuGet Package Manager ➪ Manage NuGet Packages for Solution. Ընտրել Newtonsoft.Json package:

```
using System;// Benjamin Perkins  - 609
using System.Xml.Linq;
using Newtonsoft.Json;
class Program
{
    static void Main(string[] args)
    {
        XDocument xdoc = new XDocument();
        XComment xc = new XComment("Here is a comment.");
        xdoc.Add(xc);
        XElement xe = new XElement("Company",
new XAttribute("MyAttribute", "MyAttributeValue"),
new XElement("CompanyName", "AlbDarb"),
new XElement("CompanyAddress",
new XElement("Address", "123 dzmeruk bereq Street"),
new XElement("City", "Yerevan"),
new XElement("State", "Arabkir"),
new XElement("Country", "Armenia")));
        xdoc.Add(xe);
        Console.WriteLine(xdoc.ToString());
        xdoc.Save("sss.xml");
        XDocument x = XDocument.Load("sss.xml");
        Console.WriteLine(x);
        string j = JsonConvert.SerializeXNode(x);
        Console.WriteLine(j);
        x = JsonConvert.DeserializeXNode(j);
        Console.WriteLine(x);
    }
}
//.....
```

- Կլասի օբյեկտների սերիալիզացիայի համար օգտագործենք `DataContractJsonSerializer` կլասը: Դրա համար կցենք References –ում `System.Runtime.Serialization` հավաքագրման բլոկը:

metanit.com/sharp(pg.10)

```
using System;
using System.IO;
using System.Runtime.Serialization.Json;
using System.Runtime.Serialization;
[DataContract]
public class Person
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public int Age { get; set; }

    public Person(string name, int year)
    {
```

```
        Name = name;
        Age = year;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person ob1 = new Person("Ani", 25);
        Person ob2 = new Person("Nune", 23);
        Person[] people = new Person[] { ob1, ob2 };

        DataContractJsonSerializer jsonFormatter = new
DataContractJsonSerializer(typeof(Person[]));

        using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))
        {
            jsonFormatter.WriteObject(fs, people);
        }

        using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))
        {
            Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);

            foreach (Person v in newpeople)
            {
                Console.WriteLine("Anun: {0} --- Tariq: {1}", v.Name, v.Age);
            }
        }
    }
}
```

.....

# Կոլեկցիաներ (Collections)

## 69. Կոլեկցիա Ցուցակ - `List<T>`

[Троелсен-361; Troelsen-391; Нейгел-288,288; Nagel – 195; Шилдт-961]

- `List<T>` սկզբունքորեն չի տարբերվում `ArrayList` -ից, կամ ավելի ճիշտ նրա ընդհնրացված տեսակն է: Այն ստեղծում է ընդհանրացված դինամիկ զանգվածներ:
- Հիմնական ժառանգված ինտերֆեյսները.
  `IList<T>, ICollection<T>, IEnumerable<T>`
- Հաջորդական էլեմենտների կուտակիչ, որը կարող է **դինամիկ փոխել չափը**:
- `List<T>` ունի կոնստուկտորներ, որոնք հիմանկանում տարբերվում են կոլեկցիայի սկզբնական չափի սահմանումով: Կարելի է ունենալ նախնական լռելիությամբ չափ: Կարելի է **կոնստրուկտորի պարամետրում** նշել անհրաժեշտ չափը: Եթե նախնական չափ չի նշվում, ցուցակը սկզբից լինում է դատարկ: Հետո էլեմենտների ավելացման հետ ավելացվում է նոր տեղեր, սակայն **միայն 2 –ի աստիճանների** չափով: Երբ փոխվում է ցուցակի տարողությունը ամբողջ ցուցակը տեղադրվում է հիշողության նոր մասում:
- Գլխավոր անդամներն են.

✓ `void Add(T item)` - ավելացնում է էլելմենտ կոլեկցիայի վերջում:
✓ `void Insert(int index, T item)` - մեթոդը համապատասխան տեղում ավելացնում է գրանցում:
✓ `int Capacity { get; set; }` – հատկանիշի միջոցով կարելի է փոփոխել ցուցակի տարողությունը:
✓ `bool Remove(T item)` - հեռացնում է էլեմենտ:
✓ `void RemoveAt(int index)` - հեռացնում է էլեմենտը ըստ ինդեքսի:
✓ `void RemoveRange(int index,int count)`-հեռացնում է էլելմենտներ սկսած `index` տեղից `count` չափով:
✓ `T[] ToArray()` - վերածում է զանգվածի:
✓ `void Sort()` - սորտավորում է:
✓ `void Reverse()` – շրջում է անդամները:
✓ `void Reverse(int index, int count)`- շրջում է անդամները ըստ տեղի և քանակի:
✓ `int IndexOf(T item)` – գտնում է `item` առաջին պատահած անդամը, չհայտնաբերելու դեպքում վերադարձնում է -1:
✓ `int LastIndexOf(T item)`- գտնում է վերջին `item` պատահած անդամը, չհայտնաբերելու դեպքում վերադարձնում է -1:
✓ `int Count { get; }` հատկանիշը ցուցադրում է էլեմենտների քանակը:
✓ `T this[int index] { get; set; }` ինդեքսատոր, որը հնարավորություն է տալիս ըստ ինդեքսի ստանալ և ներմուծել տվյալներ:
✓ `TrimExcess()`- մեթոդը կոլեկցիայի չափը հավասարեցնում է անդամների քանակին:

```
using System;
using System.Collections.Generic;
class A
{
    public string name;
    public int age;
}
class Program
{
    static void Main(string[] args)
    {
      List<A> p = new List<A>()
      {
            new A {name= "ani", age=28},
            new A {name= "nune", age=25},
```

```
            new A {name= "tatik", age=77}
      };
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("qanak = " + p.Count);
        Console.WriteLine("chap = " + p.Capacity);
        Console.WriteLine("---------Add------Insert----Reverse------");
        p.Add(new A { name = "darb", age = 38 });
        p.Insert(3, new A { name = "alb", age = 58 });
        p.Reverse();
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("------------ToArray -----------");
        A[] ob2 = p.ToArray();
        for (int i = 0; i < ob2.Length; i++)
            Console.WriteLine(ob2[i].name + " " + ob2[i].age);
        Console.WriteLine("-----------RemoveAt--------------");
        p.RemoveAt(3);
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("-------------------------");
        Console.WriteLine("chap = " + p.Capacity);
        Console.WriteLine("-----------TrimExcess--------------");
        p.TrimExcess();
        Console.WriteLine("chap = " + p.Capacity);
        p.Capacity = 100;
        Console.WriteLine("-----------insert Capacity 100--------------");
        Console.WriteLine("chap = " + p.Capacity);
        Console.WriteLine("------------RemoveRange(1,2)-------------");
        p.RemoveRange(1,2);
        Console.WriteLine("qanak = " + p.Count);
    }
}
//----- Sort()-----------
using System;
using System.Collections.Generic;
class A: IComparer<A>
{
    public string name;
    public int age;
    public int Compare(A firstPerson, A secondPerson)
    {
        if (firstPerson.age > secondPerson.age)
            return 1;
        if (firstPerson.age < secondPerson.age)
            return -1;
        else
            return 0;
    }
}
class Program
{
    static void Main(string[] args)
    {
      List<A> p = new List<A>()
      {
      new A {name= "ani", age=28},
      new A {name= "nune", age=25},
      new A {name= "anna", age=77}
      };
```

```
        p.Add(new A { name = "darb", age = 38 });
        p.Insert(3, new A { name = "alb", age = 58 });
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("------------Sort-----------");
        p.Sort(new A());
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
    }
}
```

❖ **Կոլեկցիա ArrayList** [Шилдт- 932; Троелсен-345; Troelsen-374]

- Կլաս `ArrayList` –ը `List<T>` -ի ոչ ընդհանրացված տեսակն: Դա նշանակում է `ArrayList` -ը կարող է պահել տարբեր տիպի տվյալներ:
- Հիմանկանում `ArrayList` –ը `List<T>` -ի հրամանները նույնն են:

✓ Չհամնկնող հրամաններ. `T[] ToArray(), TrimExcess()` փոխարինվում է `TrimToSize()` –ով:

```
using System;
//using System.Collections.Generic;
using System.Collections;
class A
{
    public string name;
    public int age;
}
class Program
{
    static void Main(string[] args)
    {
        ArrayList p =new ArrayList()
        //List<A> p = new List<A>()
        {
            new A {name= "ani", age=28},
            new A {name= "nune", age=25},
            new A {name= "anna", age=77}
        };
        p.Add(new A { name = "darb", age = 38 });
        p.Insert(3, new A { name = "alb", age = 58 });
        p.Reverse();
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
        //A[] ob2 = p.ToArray(); // error
        Console.WriteLine("qanak = " + p.Count);
        Console.WriteLine("chap = " + p.Capacity);
        Console.WriteLine("-----------RemoveAt--------------");
        p.RemoveAt(3);
        foreach (A ob in p)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("-------------------------");
        Console.WriteLine("chap = " + p.Capacity);
        p.Capacity = 100;
        Console.WriteLine("-----------insert Capacity 100--------------");
        Console.WriteLine("chap = " + p.Capacity);
        Console.WriteLine("------------RemoveRange(1,2)-------------");
        p.RemoveRange(1, 2);
        Console.WriteLine("qanak = " + p.Count);
        p.TrimToSize();
        Console.WriteLine("chap = " + p.Capacity);
}   }
```

## 70. Կոլեկցիա Ստեկ - Stack, Stack<T>

[Троелсен-362; Troelsen-393; Нейгел-302; Nagel – 206; Шилдт-945, 977 ]

- Ստեկը տվյալների կազմակերպման կառույց է: Այն կապված ցուցակի պարզեված տեսակ է: Ավելացում և կարդացումը կատարվում է միայն ցուցակի վերջից:
- Ստեկում տվյալի ավելացման կամ դուրս բերման ժամանակ չի կատարվում տվյալների հիշողության տեղաշարժ, այլ կատարվում է ստեկի ցուցիչի փոփոխություն:
- Ստեկի օգտագործման պրակտիկ օրինակներից է ֆունկցիաների կանչի կազմակերպումը և անհրաժեշտ տվյալների պահպանումը:
- Ստեկ կոնտեյները աշխատում է LIFO (last in, first out) սկզբունքով: Գլխավոր անդամներն են .

✓ Push() և Pop() մեթոդներով կատարվում է ինֆորմացիայի գրանցում և հեռացում:
✓ Peek() մեթոդով դիտարկվում է ստեկի սպասվող անդամը:
✓ Count հատկանիշը վերադարձնում է էլեմենտների քանակը ստեկում:
✓ Contains() - ստուգում է ստեկում էլեմենտի առկայությունը և հայտնաբերման դեպքում վերադարձնում է true: (Պարզ տիպեր: Բարդ տիպերի համար պետք է վերաորոշել Equals -ը)
✓ Clear() – մաքրում է ստեկը ամբողջությամբ:
✓ Ստեկը իրականացնում է IEnumerable<T>, IEnumerable, ICollection ինտերֆեյսները:

```
using System;
//using System.Collections;
using System.Collections.Generic;
class A
{
    public string name;
    public int age;
    public override string ToString()
    {
        return name.ToString()+" "+ age.ToString();
    }
    public override bool Equals(object obj)
    {
        A temp;  //
        temp = (A)obj;  //
        // if (((A)obj).age == this.age&&(((A)obj).name==this.name))
        if (temp.age == this.age&&temp.name==this.name)  //
            return true;
        else
            return false;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Stack<A> p = new Stack<A>();
        // Stack p = new Stack();
        p.Push(new A {name= "ani", age=14});
        p.Push(new A {name= "nune", age=15});
        p.Push(new A {name= "anna", age=16});
        p.Push(new A {name= "marina", age=17});
        Console.WriteLine(p.Peek());
        Console.WriteLine(p.Pop());
        Console.WriteLine(p.Peek());
        Console.WriteLine(p.Pop());
        A ob = new A {name="nune",age=55 }; //override bool Equals(object obj)
        Console.WriteLine(p.Contains(ob));
        try
        {
```

```
        Console.WriteLine(p.Peek());
        Console.WriteLine(p.Pop());
        }
        catch(InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }
        p.Clear();
        Console.WriteLine(p.Count);
    }
}
```

❖ Հարց՝ ինչու Stack<T> կոլեկցիան չունի TrimExcess(), չունի capasity:

.....

## 71. Կոլեկցիա Հերթ - Queue, Queue<T>

[Троелсен-363; Troelsen-394; Нейгел-298; Nagel – 202; Шилдт-947, 979]

- Հերթը տվյալների կազմակերպման կառույց է: Այն **կապված ցուցակի** սահմանափակ տեսակ է: Ավելացում կարատվում է միայն ակզբից, իսկ կարդում ենք միայն վերջից:
- Հերթի օգտագործման պրակտիկ տեխնիկակն օրինակ է, տպող սարքին դիմելը, որը դրվում է հերթի տպման համար:
- Հերթ կոնտեյները աշխատում է FIFO (first in, firs out) սկզբունքով: Գլխավոր անդամներն են .

✓ Enqueue() - ավելացնում է էլեմենտ կոնտեյների վերջում:
✓ Dequeue() – կարդում է էլեմենտը և միաժամակ հեռացնում:
✓ Peek() – կարդում է սպասվող էլեմենտը:
✓ Count հատկանիշը - վերադարձնում է էլեմենտների քանակը:
✓ TrimExcess() – փոքրացնում է հերթի չափը: **Dequeue() -ն** հեռացնում է էլեմենտը սակայն չի փոքրացնում պարունակությունը:
✓ Հերթը իրականացնում է IEnumerable<T>, IEnumerable, ICollection ինտերֆեյսները:
✓ Հերթը չի իրականացնում IList<T> ինտերֆեսը, դրա համար հնարավոր չէ դիմել հերթի էլելմենտներին ինդեքսի միջոցով:
✓ Հերթը `public Queue()` լռելիությամբ կոնստրուկտրով տեղ է հատկացնում, որը աշխատանքի ժամանակ ընդլայնվում է: Հերթը ունի նաև պարամետրերով կոնստրուկտոր, որոնցից է `public Queue(int capacity)`: `capacity` պարամետրում նշվում նախնական հիշողության չափը, որը նույնպես ընդլայնվող է:

```
using System;
using System.Collections.Generic;
class A
{
    public string name;
    public int age;
    public override string ToString()
    {
         return name.ToString() + " " + age.ToString();
    }
}
class Program
{
    static void Main(string[] args)
    {
        //Console.WriteLine(GC.GetTotalMemory(false));
        //Console.WriteLine(GC.GetTotalMemory(false));
        Queue<A> p = new Queue<A>();
        //Queue<A> p = new Queue<A>(64000);
        p.Enqueue(new A { name = "ani", age = 44 });
```

```
        p.Enqueue(new A { name = "nune", age = 55 });
        p.Enqueue(new A { name = "anna", age = 66 });
        p.Enqueue(new A { name = "marina", age = 77 });
        Console.WriteLine(p.Count);
        Console.WriteLine(p.Peek());
        Console.WriteLine(p.Dequeue());
        Console.WriteLine(p.Peek());
        Console.WriteLine(p.Dequeue());
        Console.WriteLine(p.Count);
        try
        {
            Console.WriteLine(p.Peek());
            Console.WriteLine(p.Dequeue());
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
             p.TrimExcess();
        }
        //Console.WriteLine(GC.GetTotalMemory(true));
    }
}
```

.....

## 72. Կոլեկցիա Hashtable, Բառարան Dictionary<TKey,TValue>

[Шилдт-939]

- Hashtable կլասը ստեղծում է կոլեկցիա “հեշ” աղյուսակի միջոցով: Հեշավորման ժամանակ կատարվում է **հատուկ բառի** ընտրություն, որը ծառայում է որպես հատուկ **ինդեքս** աղյուսակում տվյալ փնտրելու համար: Հեշ կոդի ստացումը կատարվում է ավտոմատ և **հասանելի չէ** ծրագրավորողին: Հեշավորման առավելություններից է փնտրման **ժամանակի կայունությունը** անկախ աղյուսակի մեծությունից:
- Hashtable կլասը իրականացնում է IDictionary, ICollection, IEnumerable, ISerializable, ICloneable:

```
using System;
using System.Collections;
class M
{
    static void Main()
    {
        //Hashtable ht = new Hashtable(64); // capacity
        Hashtable  ht = new Hashtable();
        ht.Add("Ani", 27);
        ht.Add("Nune", 25); // ht.Add("Ani", 10); // error
        ht.Add("Albert", 99);
        Console.WriteLine(ht.Count);
        ht.Remove("AAA");
        // foreach (int v in ht.Values)
        foreach (string v in ht.Keys)
            Console.WriteLine(v);
        ICollection c = ht.Keys;
        foreach (string s in c)
            Console.WriteLine(s + "=" + ht[s]);
        Console.WriteLine(ht["Ani"]); // indexer
    }
}
```

❖ **Կոլեկցիա Բառարան - `Dictionary<TKey,TValue>`**

[Троелсен-366; Troelsen-397; Нейгел-310; Nagel – 211; Шилдт-969]

- `Dictionary<TKey,TValue>` կլասը `Hashtable` կլասի **ընդհանրացված** տեսակն է:
- Բառարանը հնարավորություն է տալիս կուտակել տվյալներ TKey,Tvalue բանալի և արժեք համակարգով: Ձևավորվող կոլեկցիան պետք է ունենա իրարից տարբերվող բանալու արժեքներ:
- Բառարանը նախատեսում է **արագ փնտրման** կազմակերպում ըստ բանալու: List<> կոլեկցիայի նման կարելի է ավելացնել կամ հեռացնել էլեմենտ: Սակայն բառարանը աշխատում է ավելի **արտադրողական** և չունի **էլեմենների տեղաշարժի** խնդիր, որը գոյություն ունի List<> կոլեկցիայում:
- Այն տիպը, որ որոշում է **բանալին**, ցանկալի է **վերաորոշի** իր Object կլասի `GetHashCode()` -ը: Ջուգահեռ անհրաժեշտ է վերաորոշել Equals() –ը մեթոդը: Ցանկացած ժամանակ, երբ անհրաժեշտ է ձևավորել էլեմենտի տեղը, բառարանը օգտվում է `GetHashCode()` կոդի վերադարձրած արժեքից, որի միջոցով ձևավորում է ինդեքս: Պետք է հիշել, որ ինդեքսը թվային միջակայքում բաշխվում է հավասարաչափ որպես **պարզ թիվ** հետագայում արագ փնտրմանը օգնելու համար:
- string տիպ կարելի է օգտագործել որպես բանալի, որովհետև string -ում GetHashCode() -ը և Equals() -ը **վերաորոշված** է: Կարելի է ավելացնել, որ `int` տիպը նույնպես կարելի է օգտագործել որպես Key, սակայն `int` –ը GetHashCode() –ը վերաորոշել է **ըստ արժեքի** և կարող է դժվարություններ առաջանալ GetHashCode() –ի տված թիվը որպես ինդեքս օգտագործման ոչ հավասարաչափ բաշխման պատճառով: Կարելի հետևություն անել, որ `int` տիպը **ցանկալի չէ** օգտագործել որպես բանալի:
- `Dictionary<TKey, TValue>` կլասի գլխավոր անդամներից են .

✓ `void Add(TKey key, TValue value)` - ավելացնում է բառարանում տվյալ `TKey,Tvalue` զույգով: Եթե բանալին կա արդեն բառարանում, ապա առաջանում է արտակարգ իրավիճակ:

✓ `bool ContainsKey(TKey key)` – վերադարձնում է `true`, եթե կա տվյալ բանալիյով գրանցում:

✓ `bool ContainsValue (TValue value)` – վերադարձնում է `true`, եթե կա տվյալ արժեքը բառարանում:

✓ `bool Remove (TKey key)` – հեռացնում է բառարանից տվյալ ըստ բանալու և վերադարձնում է `true` հայտնաբերելու դեպքում: Եթե բանալին չի հայտնաբերվում վերադարձնում է `false`:

✓ `Keys` հատկանիշը վերադարձնում է բանալիների կոլեկցիա:

✓ `Values` հատկանիշը վերադարձնում է արժեքների կոլեկցիա:

✓ `TValue this[TKey key] { get; set; }` – ինդեքսատոր, որը կազմակերպում է բառարանի ձևավորումը ոչ թե ինդեքսով, այլ բանալու արժեքով:

```
using System;
using System.Collections.Generic;
class M
{
    static void Main()
    {
        // Dictionary<string, int> dic = new Dictionary<string, int>(64); // capacity
        Dictionary<string, int> dic = new Dictionary<string, int>();
        dic.Add("Ani", 27);
        dic.Add("Nune", 25);
        dic.Add("Albert", 47);
        Console.WriteLine("count =" + dic.Count);
        ICollection<string> c = dic.Keys;
        foreach(string  s in c)
            Console.WriteLine(s +"="+dic[s]);
        //foreach (var s in dic)
        //    Console.WriteLine(s.Key + "=" + s.Value);
        Console.WriteLine(dic["Ani"]); // indexer
    }
}
//----------------------
```

```
using System;
using System.Collections.Generic;
class M
{
    static void Main()
    {
        Dictionary<int, string> dic = new Dictionary<int, string>();
        dic.Add(1, "Ani");
        dic.Add(5, "Nune");
        dic.Add(7, "Albert");
        ICollection<int> c = dic.Keys;
        foreach (int n in c)
            Console.WriteLine(n + "=" + dic[n]);
        Console.WriteLine("Values Collections.................");
        ICollection<string> c2 = dic.Values;
        foreach (string s in c2)
            Console.WriteLine(s);
        Console.WriteLine("Remove 17.................");
        dic.Remove(7);
        ICollection<int> c3 = dic.Keys;
        foreach (int n in c3)
            Console.WriteLine(n + "=" + dic[n]);
        Console.WriteLine(dic.ContainsKey(5));
        Console.WriteLine(dic.ContainsValue("aaa"));
    }
}
```

## 73. Կոլեկցիա SortedDictionary<K,V>,SortedList<K,V>

[Троелсен-366; Шилдт-972,974; Нейгел-316,308; Nagel – 209]

- Եթե անհրաժեշտ Dictionary<TKey,TValue> բառարանի սորտավորված կոլելկցիա ըստ բանալու, ապա կարելի է օգտվել SortedDictionary<TKey,TValue> կամ SortedList<TKey,TValue> կլասներից:
- Քանի, որ SortedList<TKey,TValue> կլասը հիմնված է ցուցակի վրա, որի հիմքում է **զանգվածը**, իսկ SortedDictionary<TKey,TValue> -ն հիմնաված է բառարանի սկզբունքի վրա, ապա նրանք ունեն տարբերություններ: [Нейгел-316]

✓ SortedList<> կլասը ավելի **քիչ հիշողություն** է զբաղացնում քան SortedDictionary<> -ը:

✓ SortedDictionary<> կլասը ավելի **արագ է էլեմենտ ավելացնում** և **հեռացնում** քան SortedList<> -ը:

✓ Եթե անհրաժեշտ է ավելացնել էլեմենտներ, որոնք արդեն սորտավորված են, ապա այդ դեպքում SortedList<> -ը ավելի արագ է աշխատում, եթե կոլեկցիայի չափի փոփոխության կարիք չկա:

```
using System;
using System.Collections.Generic;
class M
{
    static void Main()
    {
       // SortedDictionary<string, int> dic = new SortedDictionary<string, int>();
        SortedList<string, int> dic = new SortedList<string, int>();
        dic.Add("Ani", 27);
        dic.Add("Nune", 25);
        dic.Add("Albert", 20);
        ICollection<string> c = dic.Keys;
        foreach (string s in c)
            Console.WriteLine(s + "=" + dic[s]);
}   }
```

## 74. Կոլեկցիա HashSet<T> [Нейгел-317; Шилдт-982]
## SortedSet<T> [Троелсен-364; Troelsen-394; Нейгел-317; Шилдт-980]

- Կոլեկցիաները, որոնք **չեն պարունակում կրկնվող** էլեմենտներ կոչվում են բազմություններ (set): Net.4.0 ում ներդրվել են երկու այդպիսի բազմություններ `SortedSet<>` և `HashSet<>`: `HashSet<>` -ը պարունակում է ոչ սորտավորված կրկնվող էլեմենտներ **(ըստ "հաշ"** կոդի չեն կրկնվում), իսկ `SortedSet<>` -ը սորտավորված և չկրկնվող էլեմենտներ:
- `SortedSet<>` կլասը հարմար է նրանով, որ եթե կոլեկցիայում էլեմենտներ են ստեղծվում, ավելացվում կամ հեռացվում, ապա կատարվում է **ավտոմատ վերադասավորում**:
- Որպեսզի կատարվի վերադասավորման ապահովում, անհրաժեշտ է `SortedSet<T>` -ի կոնստրուկտորում նշվի սորտավորման օբյեկտ, որի տիպում ընդգրկված լինի `IComparer<>` ինտերֆեսի իրականացում:
- `SortedSet<>` և `HashSet<>` կլասները ժառանգում են `ISet<T>, ICollection<T>, IEnumerable<T>` ինտերֆեյսները:
- `SortedSet<>` և `HashSet<>` որոշ անդամներ՝

✓ `void` Add(`T` item) - ավելացնում է էլեմենտ կոլեկցիայի վերջում:
✓ `int` Count { `get`; } հատկանիշը ցուցադրում է էլեմենտների քանակը:
✓ `void` Reverse() – շրջում է անդամները:
✓ Ունի նաև ստանդարտ հրամաններ՝ First(), Last(), Contains(), Average(), Min(), Max(), Sum(), Remove() Clear() և այլն:

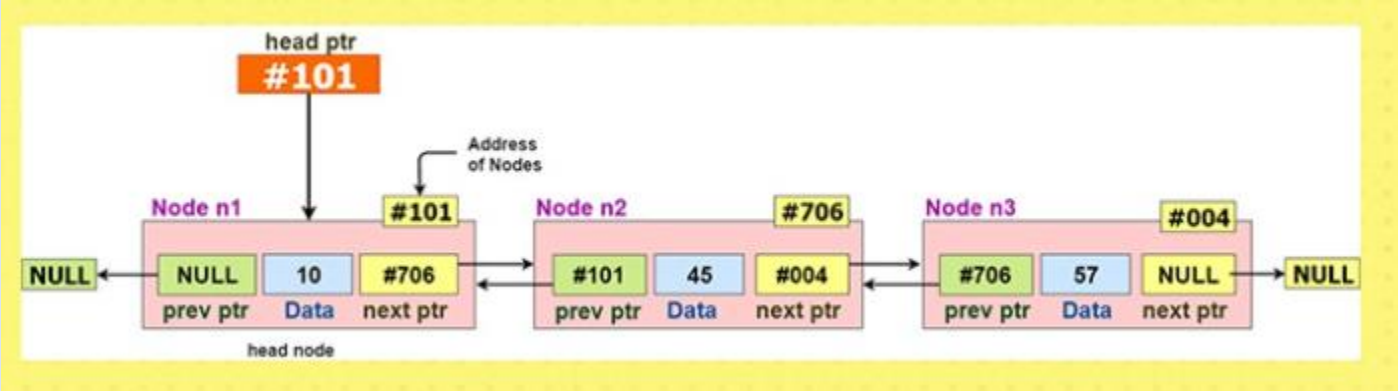
```
using System;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {
        HashSet<int> hs = new HashSet<int>();
        hs.Add(6);
        hs.Add(7);
        hs.Add(6); // չի ավելացնում
        foreach (int v in hs)
            Console.WriteLine(v);
        Console.WriteLine("qanak = " + hs.Count);
}   }
//--------------
using System;
using System.Collections.Generic;
// struct A
class A
{
    public string name;
    public int age;
}
class Program
{
    static void Main(string[] args)
    {
        HashSet<A> hs = new HashSet<A>()
        {
            new A {name= "ani", age=28},
            new A {name= "nune", age=25},
            new A {name= "nune", age=25},
            new A {name= "anna", age=77}
        };
        foreach (A ob in hs)
            Console.WriteLine(ob.name + " " + ob.age);
```

```
        Console.WriteLine("qanak = " + hs.Count);
        // A ob3 = new A { name = "darb", age = 38 }; // ըստ “հաշ” կողի
        // A ob4 = new A { name = "darb", age = 38 }; // ըստ “հաշ” կողի
        // Console.WriteLine(ob3.GetHashCode());
        // Console.WriteLine(ob4.GetHashCode());
        // hs.Add(ob3);
        // hs.Add(ob3);
        // hs.Add(ob4);
        hs.Add(new A { name = "darb", age = 38 });
        foreach (A ob in hs)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("qanak = " + hs.Count);
    }
}
//------------- SortedSet -----------------
using System;
using System.Collections.Generic;
class SortAge : IComparer<A>
{
    public int Compare(A first, A second)
    {
        if (first.age > second.age)
            return 1;
        if (first.age < second.age)
            return -1;
        else
            return 0;
    }
}
class A
{
    public string name;
    public int age;
}
class Program
{
    static void Main(string[] args)
    {
        SortedSet<A> ss = new SortedSet<A>(new SortAge())
        {
            new A {name= "ani", age=28},
            new A {name= "nune", age=25},
            new A {name= "nune", age=25},
            new A {name= "anna", age=77}
        };
        foreach (A ob in ss)
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("qanak = " + ss.Count);
        Console.WriteLine("---------Add---------Reverse------");
        ss.Add(new A { name = "darb", age = 38 });
        ss.Reverse();// d’not reverse
        foreach (A ob in ss)
            Console.WriteLine(ob.name + " " + ob.age);
        foreach (A ob in ss.Reverse())
            Console.WriteLine(ob.name + " " + ob.age);
        Console.WriteLine("qanak = " + ss.Count);
    }
}
```

# 75. Կոլեկցիա LinkedList<T>

[Нейгел-303; Nagel – 208; Шилдт-965; Албахари-337; Albahari-366]

- `LinkedList<T>` կապված ցուցակ - իրենից ներկայացնում է երկողմանի կապված ցուցակներ, որտեղ ցանկացած էլեմենտ կապված է և հղում է **դեպի հաջորդ և դեպի նախորդ** էլեմենտներ: Այս ձևով կատարվում է արագ տեղաշարժ ցուցակում և հետ և առաջ:




- `LinkedList<T>` կապված ցուցակ կազմված է հանգույցներից – Nodes: Նրա թերություններից է, որ ցուցակում էլեմենտնրի հետ աշխատանքի թույլատրություն ստանալու համար անցնում ենք հանգույցներով հաջորդաբար, մեկը մյուսից հետո:
- Կապված ցուցակի առավելություններից է շատ **արագ էլելմենտի ավելացում** ցուցակի ցանկացած մասում: Ավելացման համար անհրաժեշտ է ուղակի կապի հաստատում իրենից **հաջորդող** և **նախորդող** էլելմենտնրին:
- `LinkedList<T>` կլասում պարփակված են (inkapsulation) `LinkedListNode<>` օբյեկտներ, որտեղ պահվում են տվյալները:
- `LinkedListNode<>` կլասի չորս հատկանիշներն են.

✓ `LinkedList<T> List { get; }` – ստանում է հղում հենց իրեն, ցուցակին
✓ `LinkedListNode<T> Next { get; }` – ստանում է հղում հաջորդ հանգույցին
✓ `LinkedListNode<T> Previous { get; }` - ստանում է հղում նախորդ հանգույցին
✓ `T Value { get; set; }` – հանգույցի արժեքն է

- `LinkedList<T>` կլասում կա նաև հատկանիշներ որոնք նշում են “կապված ցուցակի” արաջին և վերջին հանգույցները.

✓ `LinkedListNode<T> First { get; }`
✓ `LinkedListNode<T> Last { get; }`

- `LinkedList<T>` կլասի որոշ անդամներ.

✓ `LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value)`-ավելացնում է `value` արժեք `node` -ում նշված տեղից հետո:
✓ `LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value)`-ավելացնում է `value` արժեք `node` -ում նշված տեղից առաջ:
✓ `LinkedListNode<T> AddFirst(T value)` - ավելացնում է `value` արժեք ցուցակի սկզբում:
✓ `LinkedListNode<T> AddLast(T value)`- ավելացնում է `value` արժեք ցուցակի վերջում:
✓ `bool Remove(T value)` – հեռացնում է `value` արժեքով առաջին պատահած հանգույցը:
✓ `void RemoveFirst()` – ցուցակից հեռացնում է առաջին հանգույցը:
✓ `void RemoveLast()` - ցուցակից հեռացնում է վերջին հանգույցը:
✓ `LinkedListNode<T> Find(T value)` - վերադարձնում է ցուցակի `value` փնտրվող արժեքներից առաջի պատահածը:
✓ `LinkedListNode<T> FindLast(T value)` – վերադարձնում է ցուցակի `value` փնտրվող արժեքներից վերջինը:
✓ `int Count { get; }` հատկանիշը վերադարձնում է ցուցակի գրանցումների քանակը:

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {
        LinkedList<string> ll = new LinkedList<string>();
        Console.WriteLine(ll.Count);
        ll.AddFirst("ani");
        //mark1=ll.Last;
        ll.AddFirst("nune");
        ll.AddFirst("anna");
        ll.AddFirst("alb");
        Console.WriteLine(ll.Count);
        LinkedListNode<string> node;
        for (node = ll.First; node != null; node = node.Next)
            Console.Write(node.Value + " ");
        Console.WriteLine();
        foreach (string o in ll)
            Console.Write(o + " ");
        Console.WriteLine();
        for (node = ll.Last; node != null; node = node.Previous)
            Console.Write(node.Value + " ");
        Console.WriteLine();
        ll.Remove("anna");
        foreach (string o in ll)
            Console.Write(o + " ");
        Console.WriteLine();
        Console.WriteLine(ll.Count);
        ll.AddLast("alb");
        ll.AddLast("darb");
        foreach (string o in ll)
            Console.Write(o + " ");
        Console.WriteLine();
        Console.WriteLine(ll.Contains("ani"));
        node = ll.First;
        ll.AddAfter(node,"aaaa");
        foreach (string o in ll)
            Console.Write(o + " ");
        Console.WriteLine();
        //LinkedListNode<string> current = ll.Find("alb");
        LinkedListNode<string> current = ll.FindLast("alb");
        ll.AddBefore(current, "bbbb");
        foreach (string o in ll)
            Console.Write(o + " ");
        Console.WriteLine();
        ll.Clear();
        Console.WriteLine("After Clear count = "+ll.Count);
    }
}
```

.....

# 76. Զուգահեռ աշխատանքի կոլեկցիաներ

[Нейгел-325, Nagel – 329; Шилдт-983]

- Անունի տարածք `System.Collections.Concurrent` (NET Framework 4.0):
- Այն պարունակում է կոլեկցիաներ, որոնք ապահոված են ծրագրային հոսքերի նկատմամբ և կարող են օգտագործվել ծրագրի զուգահեռ կատարման ժամանակ: Դա նշանակում **է երկու և ավելի հոսքեր** կարող են դիմել միևնույն կոլեկցիային:
- Նրանք աշխատում են ապահով այն իմաստով, որ նրանց մեթոդները վերադարձնում են `false`, եթե **անհնար է** որևէ գործողություն տվյալ հոսքի սահմաններում:
- Զուգահեռ կոլեկցիաներ.

✓ `BlockingCollection<T>` - բլոկավորող կոլեկցիա: Այն համարվում է **ընդհանուր** ձև և լռելիությամբ օգտագործում է բլոկավորող հերթ `ConcurrentQueue<T>`: Այն իրականացնում է բլոկավորում և սպասում է ավելացման կամ ջնջման հնարավորության:
Անդամներից են.
`void Add(T item)`- Ավելացնում է էլեմենտ: Եթե կոլեկցիան սահմանափակ է, ապա բլոկավորվում է մինչև տեղ ազատվի:
`T Take( )` – հեռացնում է էլեմենտ կոլեկցիայից:
Եթե չկա անհրաժեշտություն հոսքի երկար սպասման և ցանկալի չէ հայցը հանել, օգտագործվում է մեթոդներ, որտեղ կարելի է նաև հսկել ժամանակը:

```
bool TryAdd(T item);
bool TryTake(out T item);
bool TryAdd(T item, TimeSpan timeout);
bool TryTake(out T item, TimeSpan timeout);
```

✓ `ConcurrentBag<T>` - կլասը չի որոշում էլեմենտերի կարդալու և գրանցելու կարգավորվածություն: Օգտագործում է `void Add(T item)`, `bool TryPeek(out T result)`, `bool TryTake(out T result)` մեթոդները:

✓ `ConcurrentDictionary<TKey, TValue>` - իրականացնում է զուգահեռ բառարան: Օգտագործում է, `bool TryAdd(Tkey key, Tvalue value)`, `bool TryGetValue(Tkey key, out Tvalue value)`, `bool TryRemove(Tkey key, out Tvalue value)`, `bool TryUpdate(Tkey key, Tvalue newValue, Tvalue comparisonValue)`: Քանի որ բանալու հիմքի վրա է աշխատում, ապա այն չի իրականացնում `IProducerConsumerCollection<T>` ինտերֆեսը:

✓ `ConcurrentQueue<T>`- իրականացնում է զուգահեռ հերթ: Ներքին էլեմենտները պահվում են կապված ցուցակներով: Հերթի էլեմենտներին կարելի է դիմել `void Enqueue(T item)`, `bool TryDequeue(out T result)`, `bool TryPeek(out T result)` մեթոդներով: Քանի որ, կլասը օգտագործում է `IProducerConsumerCollection<T>` ինտերֆեսը, ապա TryAdd(), TryTake() մեթոդները ուղղակի կանչում են Enqueue(), TryDequeue() մեթոդները:

✓ `ConcurrentStack<T>` - իրականացնում է զուգահեռ ստեկ: Ներքին էլեմենտները պահվում են կապված ցուցակներով: Նման է զուգահեռ հերթին: Օգտագործում է `bool TryPeek(out T result)`, `bool TryPop(out T result)`, `int TryPopRange(T[] items)` մեթոդները:

- `IProducerConsumerCollection<T>` ինտերֆեյսը օգտագործվում է հիմնականում բոլոր կլասներում: Այն զարգացնում է **`IEnumerable`, `IEnumerable<T>`, `ICollection`** ինտերֆեյսները: Բացի դրանից այնտեղ որոշված է մեթոդներ, որոնք իրականացնում են **առաջարկող-օգտագործող** շաբլոնը: Նրանցից մեկը կոլեկցիան է լրացնում, միուսը օգտվում է կոլեկցիայից.

✓ `bool TryAdd(T item)` վերադարձնում է `true`, եթե կոլեկցիայում էլեմենտ է ավելացվում:

✓ `bool TryTake(out T item)` վերադարձնում է `true`, եթե կոլեկցիայում էլեմենտ է ջնջվում:

- Զուգահեռ կոլեկցիաները հաճախ օգտագրծվում են Task Parallel Library կամ PLINQ -ի հետ:

- Օրինակներում ստեղծվում է սահմանափակ տարողությամբ 11 էլեմետ պարունակող հերթ:
- Queue<char> կոլեկցիան զուգահեռություն չի ապահովում, դրա համար ծրագիրը կառաջացնի արտակարգ իրավիճակ:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
class Program
{
    static Queue<char> bc;
    static void Producer()
    {
        for (char ch = 'A'; ch <= 'Z'; ch++)
        {
            bc.Enqueue(ch);
            Console.WriteLine("..................... Producing " + ch);
        }
    }
    static void Consumer()
    {
        for (int i = 0; i < 26; i++)
            Console.WriteLine("Consuming " + bc.Dequeue());
    }
    static void Main(string[] args)
    {
        bc = new Queue<char>(11);
        Parallel.Invoke(Producer, Consumer);
        // Producer();
        // Consumer();
    }
}
//---------------------------------
```

- BlockingCollection<char> կլասը թույլատրում է անկախ հոսքերին միաժամանակ դիմել հերթ կոլեկցիային: (BlockingCollection<> կլասը լռելիությամբ դառնում է հերթ տիպի):

```
using System;
using System.Threading.Tasks;
using System.Collections.Concurrent;
class Program
{
    static BlockingCollection<char> bc;
    static void Producer()
    {
        for (char ch = 'A'; ch <= 'Z'; ch++)
        {
            bc.Add(ch);
            Console.WriteLine("Producing " + ch);
        }
    }
    static void Consumer()
    {
        for (int i = 0; i < 26; i++)
            Console.WriteLine("                        Consuming " + bc.Take());
    }
    static void Main(string[] args)
    {
        bc = new BlockingCollection<char>(11);
```

```
        Parallel.Invoke(Producer, Consumer);
        bc.Dispose();
    }
}
//-----------------------------------
```

- `BlockingCollection<T>` ունի նաև `void CompleteAdding()` մեթոդը: Մեթոդի կանչը նշանակում է, որ կոլեցիայում չի կարելի ավելացնել էլեմենտ: Դա բերում է նրան, որ `bool IsAddingCompleted { get; }` կամ `bool IsCompleted { get; }` հատկանիշները ընդունում են `true` արժեք:
- Հաջորդ օրինակը օգտագործում է CompleteAdding( ), IsCompleted, TryTake( ) անդամները և զուգահեռացման համար օգտագործվում է `Task` կլասը:

```
using System; //Шилдт 985
using System.Threading.Tasks;
using System.Collections.Concurrent;
class Program
{
    static BlockingCollection<char> bc;
    static void Producer()
    {
        int count = 0;
        for (char ch = 'A'; ch <= 'Z'; ch++)
        {
            count++;
            bc.Add(ch);
            Console.WriteLine("Producing... " + ch);
            if(count==5)
            bc.CompleteAdding();
        }
    }
    static void Consumer()
    {
        char ch;
        while (!bc.IsCompleted)
        {
            if (bc.TryTake(out ch))
                Console.WriteLine("Consuming... " + ch);
        }
    }
    static void Main(string[] args)
    {
        bc = new BlockingCollection<char>(44);
        Task Prod = new Task(Producer);
        Task Con = new Task(Consumer);
        Con.Start();
        Prod.Start();
        Console.ReadKey();
        Con.Dispose();
        Prod.Dispose();
        bc.Dispose();
    }
}
.....
```

# 77. Կլաս կոլեկցիաների արտադրողականությունը

[Нейгел-333; Nagel – 316]

- Կլաս կոլեկցիաները կարող են կատարել նույն աշխատանքը, սակայն նրանք կարող են տարբերվել ըստ զբաղեցրած հիշողության ծավալի և ըստ էլեմենտների աշխատանքի արագության:
- Ընդունված է մեծատառ O -ով նշել աշխատանքի արագությունը:

✓ O(1); O(nlog n); O(log n); O(n); $O(n^2)$

| COLLECTION | ADD | INSERT | REMOVE | ITEM | SORT | FIND |
|---|---|---|---|---|---|---|
| `List<T>` | O(1) or O(n) if the collection must be resized | O(n) | O(n) | O(1) | O (n log n), worst case O(n ^ 2) | O(n) |
| `Stack<T>` | `Push`, O(1), or O(n) if the stack must be resized | n/a | `Pop`, O(1) | n/a | n/a | n/a |
| `Queue<T>` | `Enqueue`, O(1), or O(n) if the queue must be resized | n/a | `Dequeue`, O(1) | n/a | n/a | n/a |
| `HashSet<T>` | O(1) or O(n) if the set must be resized | `Add` O(1) or O(n) | O(1) | n/a | n/a | n/a |
| `SortedSet<T>` | O(1) or O(n) if the set must be resized | `Add` O(1) or O(n) | O(1) | n/a | n/a | n/a |
| `LinkedList<T>` | `AddLast` O(1) | `Add After` O(1) | O(1) | n/a | n/a | O(n) |
| `Dictionary <TKey, TValue>` | O(1) or O(n) | n/a | O(1) | O(1) | n/a | n/a |
| `SortedDictionary <TKey, TValue>` | O(log n) | n/a | O(log n) | O(log n) | n/a | n/a |
| `SortedList <TKey, TValue>` | O(n) for unsorted data, O(log n) for end of list, O(n) if resize is needed | n/a | O(n) | O(log n) to read/ write, O(log n) if the key is in the list, O(n) if the key is not in the list | n/a | n/a |

- O(l) նշանակում է ժամանակը կախված չէ կոլեկցիայի էլեմենտների քանակից: Օրինակ. ArrayList ունի մեթոդ Add() որի կատարման ժամանակը O(l) կարգի է:
- O(n) նշանակում է վատագույն դեպքում անհրաժեշտ է N ժամանակ աշխատանքի կատարման համար և օպերացիաների կատարման ժամանակը ավելանում է գծային ձևով: Օրինակ. եթե անհրաժեշտ է կոլեկցիայի չափը մեծացնել, ապա կատարվում է կոլեկցիայի կոպիա և դա համարվում է O(n) բարդության:
- O(log n) նշանակում է ամեն էլեմենտի ավելացման հետ ավելանում է նրա հետ կատարման օպերացիաները, սակայն ոչ թէ գծային, այլ լոգարիթմական:
- `SortedDictionary<TKey,TValue>` կլասը O(log n) բարդություն ունի, երբ անհրաժեշտ է կոլեկցիան ավելացնել էլեմենտով, իսկ `SortedList<TKey,TValue>` կլասը O(n) բարդություն նույն օպերացիայի համար: Ստացվում է, որ `SortedDictionary<>` ավելի արագ է աշխատում, որովհետև ներքին ծառի ստրուկտուրա ունի, քան `SortedList<>` -ը, որը ցուցակի կառույց է:

# Debug

## 78. Պրեպրոցեսոր

[Шилдт-528; Нейгел-105; Covaci – 498]

- **Պայմանական թարգմանություն**:
- Պրեպրոցեսորի հրամանները չեն թարգմանվում որպես կատարման կոդի: Նրանք անհրաժեշտ են թարգմանման ժամանակ ծրագրի տարբեր տարբերակների ստացման համար:
- Պրեպրոցեսորի հրամանները կոչվում են "**դիրեկտիվաներ**": Նրանց միջոցով կարելի է ստանալ ծրագրի տարբեր վերսիաներ: Պրեպրոցեսորի հրամանները օգնում են նաև Debug –ի աշխատանքի ժամանակ: Բոլոր հրամանները սկսվում են # վանդականիշով և **չեն ավարտվում** կետ ստորակետով: Ավարտման սիմվոլը դա <Enter> սիմվոլն է:
- Պրեպրոցեսորի հրամաններն են - `#define #elif #else #endif #endregion #error #if #line #pragma #region #undef #warning`:

❖ **`#define, #undef`**

✓ `#define DDD` - հայտարարում է փոփոխական, որը արժեք չունի և կարող է օգտագործվել միայն թարգմանությանը օժանդակելու համար:

✓ `#undef DDD` – չեղյալ է հայտարաում փոփոխականը:

✓ Հիմանականում օգտագործվում է `#if` դիրեկտիվայի հետ:

✓ Նշված պրեպրոցեսորի հրամանները դրվում են մինչև ծրագրի կոդը:

```
//------------
```

❖ **`#if, #elif, #else, #endif`**

✓ Պայմանական հրամաններ են, որոնք նշում են ծրագրի որ մասն է թարգմանվում: Պայմանը համարվում է իրական, եթե հայտարաված է **`#define`** –ի միջոցով իդենտիֆիկատոր և այն օգտագործվում է պայմանի մեջ: Իդենտիֆիկատրի արկայությունը նշանակում է `true`:

✓ Պայմանական հրամանները կարող են աշխատել !; ==; !=; &&; || տրամաբանության հետ:

✓ **`#elif, #else`** միաժամանկ չեն աշխատում: Կամ մեկը կամ մյուսը:

```
#define ddd
#define bbb
//#undef ddd
#undef bbb
using System;
class Program
{
    static void Main(string[] args)
    {
      #if ddd && bbb == false
        Console.WriteLine("barev versia");
      //#elif bbb
      //        Console.WriteLine("elif");
      #else
            Console.WriteLine("demo");
      #endif
    }
}
//------------
```

❖ **DEBUG , TRACE** հաստատուններ

- Թարգմանիչը ունի ներքին հաստատուններ DEBUG, TRACE անուններով, որոնց կարելի է օգտագործել պրեպրոցեսորի դիրեկտիվաների միջոցով:
- DEBUG, TRACE անունները կարելի է ակտիվացնել => (VS 2022 Framework) Debug->ApplPropertis->Build->CheckBox նշում:

```
using System;
class Program
{
    static void Main(string[] args)
    {
#if DEBUG
        Console.WriteLine("Debug mode");
#else
        Console.WriteLine("Not debug");
#endif
#if TRACE
        Console.WriteLine("Trace mode");
#else
        Console.WriteLine("Not trace");
#endif
        Console.ReadKey();
    }
}
//------------
```

❖ **#error, #warning**

- Դադարացնում է թարգմանությունը կամ զգուշացնում է : Օգտագործվում է Debug –ի ժամանկ:
- Դիրեկտիվայից հետո նշվում է արտահայտություն:
- Error List պատուհանը ակտիվացվում է View -> Error List հրամանով:

```
#define ddd
//#undef ddd
using System;
class Program
{
    static void Main(string[] args)
    {
        #if ddd
        Console.WriteLine("barev");
        #else
        Console.WriteLine("else");
        #error sxal ka
         //#warning egheq zgush
        #endif
    }
}
```

❖ **#line ->** Error List -ում համարակալման (Line) թիվն է փոփոխում:

- Կարող է փոփոխել տողի համարը և նշել ֆայլի անունը, երբ կատավում է թարգմանչի կողմից սխալի կամ զգուշացման հաղորդագրություն: Այլ խոսքով որտեղ դրված է `#line 155` հրամանը, դրանից հետո ծրագրի հարմանները շարունակվում են նշված թվից:

✓ `#line 155 "prog.cs"`– նշանակում է նշված հրամանի տողը պետք է ընդունի 155 համար նշված ֆայլում:

✓ `#line default` – հրամանը բերում է նախնական համարակալման:

✓ `#line hidden` – հնարավորություն է տալիս Debug -ի քայլային դիտարկման (F10,F11) ժամանակ բաց թողնել հրամանները մինչև հաջորդ `#line default` հրամանը:

```
#line 100
using System;
class Program
{
    static void Main(string[] args)
    {
        //#line  10
        Console.WriteLine("barev");
        Console.WriteLine("albdarb.com");
        //#error sxal ka
        #line default
        #warning egheq zgush
        #line hidden
        for (int i = 0; i < 8; i++)
            Console.WriteLine("aaa");
    }
}
```

❖ **#region  #endregion**

- Հնարավորություն է տալիս ծրագրի հատվածը փակել տեսադաշտից: Թարգմանիչը միևուս նշան է դնում, եթե փակում ենք և + նշան բացման դեպքում:

```
using System;
class Program
{
    static void Main(string[] args)
    {
#region
        Console.WriteLine("barev");
        Console.WriteLine("albdarb.com");
#endregion
        for (int i = 0; i < 8; i++)
            Console.WriteLine("aaa");
}   }
```

❖ **#pragma**

- Հնարավորություն է տալիս թարգմանչի զգուշացումը ղեկավարել: Կարելի է զգուշացումը չեզոքացնել: Կարելի է վերականգնել:

✓ `#pragma warning disable` – հանվում է զգուշացումը:
✓ `#pragma warning restore` – վերականգնում է զգուշացումը:
✓ `#pragma checksum "filename" "globally unique identifier (GUID)" "bytes"` – օգտագործվում է հատուկ դեպքերում, օրինակ ASP.NET նախագծերում “checksum” ստանալու համար: Իրականում թարգմանիչը հաշվում է հատուկ “checksum” և տեղադրում է program database (PDB) ֆայլում: “checksum” նախատեսված է ծրագրի ստացված կոդի համեմատումը **օրիգինալի** հետ:

```
using System;
class Program
{
    static void Main(string[] args)
    {
#pragma warning disable
        int a = 999;
        Console.WriteLine("barev");
#pragma warning restore
        string s = "";
        Console.WriteLine("albdarb.com");
}   }
```

.....

# 79. Conditional ատրիբուտ

[Албахари-539]

- [Conditional] ատրիբուտը նշում է թարգմանիչին, որ այն կարող է չեզոքացնել ցանկացած մեթոդի հղում, եթե ատրիբուտի տրված սիմվոլը որոշված չէ `#define` պրեպրոցեսորի հրամանով:
- [Conditional] ատրիբուտը փոխարինում է `#if, #endif` պրեպրոցեսորի հրամանին, ավելի հարմար ծրագրային հատված գրելու և ատրիբուտների որոշ առավելություններից օգտվելու համար:
- Անունի տարածք - `System.Diagnostics`:
- [Conditional] ատրիբուտը կատարման ժամանակ անտեսվում է: Այն ծառայում է միայն որպես թարգմանչի հրաման:
- Առաջին օրինակում օգտագործվում է `#if, #endif` պրեպրոցեսորի հրամանները, որի թերությունը կայանում է նրանում, որ կարելի է փակել միայն մեթոդի կաչնը: Այլ խոսքով կլասի մեթոդի որոշումը մտնում են թարգմանության հրամանների կազմի մեջ և ստիպված ենք կրկնակի անգամ օգտագործել `#if, #endif` պրեպրոցեսորի հրամանները այն չընդգրկելու համար:

```
#define dd
#define bb
using System;
//#if bb
class B
{
    public B()
    {
        Console.WriteLine("Es B classi construktor em");
    }
}
//#endif
class Program
{
//#if dd
    static void ff()
    {
        Console.WriteLine("ff() method em");
    }
//#endif
    static void Main(string[] args)
    {
#if dd
        ff();
#endif
#if bb
        B ob = new B();
#endif
        Console.ReadKey();
    }
}
//------------------------
```

- Կլասը կարող է ընդունել[Conditional] ատրիբուտ, որը պետք է ժառանված լինի `Attribute` կլասից սակայն այն չի ազդում անդամների չեզոքացման վրա:

```
#define dd
#define bb
#undef bb
using System;
using System.Diagnostics;
```

```
[Conditional("bb")]
class B:Attribute
{
    public B()
    {
        Console.WriteLine("Es B classi construktor em");
    }
}
class Program
{
    [Conditional ("dd")]
    static void ff()
    {
        Console.WriteLine("ff() method em ");
    }
    static void Main(string[] args)
    {
        ff();
        B ob = new B();
        Console.ReadKey();
    }
}
```

- [Conditional] ատրիբուտը անիմաստ է դառնում ծրագրի կատարման ժամանակ մեթոդի դինամիկ կանչի դեպքում այն ընդգրկել, թէ ոչ: Նշված էֆեկտին կարելի է հասնել օգտագործելով լյամբդա արտահայտություն, որին ղեկավարում որևէ փոփոխականի կիրառություն: [Албахари-522] (տնային, գրել օրինակը)

.....

# 80. Debug

[Уотсон- 166; Нейгел-501]

- Debug –ը, դա հատուկ ծրագիր է, որը միավորելով ծրագրին, հեշտացնում է ծրագրում սխալների հայտնաբերումը, սակայն ծանրացնում է նախագծվող ծրագրի աշխատանքը: Debug –ի ժամանակ հիշվում է ծրագրի սիմվոլային մասը, որպեսզի թարգմանիչը տեղյակ լինի, թէ ինչ է կատարվում հրամանների ցանկացած հատվածում: Նշված ինֆորմացիան պահվում է .pdb ֆայլում:
- Ծրագիրը կարելի է նաև հետևել միջանկյալ Console.WriteLine() – ներով: Սակայն նա չունի մեծ հնարավորություններ:
- Debug –ի գլխավոր իմաստն է ծրագրում նշել **կանգառի կետեր** և դիտարկել փոփոխականների արժեքները:
- Կանգառի կետերը կարելի է նշել F9 –ով, կամ հրամանի ձախ եզրում մկնիկի կրկրնակի քլիկով:
- Debug->Windows->Breakpoints –ով կարելի է դիտարկել բոլոր կանգառի կետերը, որի միջոցով ստեղծվում է հետևյալ հնարավորությունները.

✓ Ղեկավարել կանգառի կետով անցնելու քանակը: Կամ կատարել կանգառ օգտագործման ինչ որ քանակից հետո: Կատարել կանգառ նշված քանակով:
Hit Count – փոփոխականով նշվում է անցնելու քանակը: // ?
Conditoins – նշվում է պայմանը, որով կարելի է մտցնել ցանկացած արտահայտություն այն փոփոխականներից, որոնք գտնվում են տվյալ ընդհատված կետի տեսանելիության դաշտում: Օրինակ, Breakpoints աջ քլիկով կարելի է ստանալ նեմուծման մաս: Նշել պայման Conditoins մասում տեսանելի փոփոխականով և Actions -ում գրել անհրաժեշտ հրաման – Debug.Write("esiminch"):

✓ Կանգառի կետը լինի միայն տվյալների համար և ոչ թէ հրամանների համար: // ?

- F11, Shift-F11, F10

✓ F11 - Step Into – անցնում է հաջորդ օպերատրին:

✓ Shift-F11 - Step Out – դուրս է գալիս մեթոդից:

✓ F10 - Step Over - անցնում է հաջորդ օպերատրին, սակայն բաց է թողնում մեթոդները:

✓ #line hidden – հնարավորություն է տալիս Debug -ի քայլային դիտարկման (F10,F11) ժամանակ բաց թողնել հրամանները մինչև հաջորդ #line default հրամանը:

**Օրինակ 1.**

```
using System;
class Program
{
    static void f()
    {
        int k = 2;
        while (k > 0)
        {
            Console.WriteLine("k=" + k);
            k--;
        }
    }
    static void Main(string[] args)
    {
        for (int i = 0; i < 2; i++)
        {
            Console.WriteLine("i= " + i);
            int j = 2;
            while (j > 0)
            {
                f();
                Console.WriteLine("j=" + j);
                j--;
```

```
            }
        }
    }
}
//--------------------
```

- Debug->Windows-> Autos, Locals և Watch պատուհաններով կարելի է հետևել փոփոխականների արժեքները:
- ✓ Autos – նշվում են տվյալ կետում գտնվող և **նախորդող օպերատրում** մասնակցող բոլոր փոփոխականները և արժեքները:
- ✓ Locals – տվյալ կանգառի կետի **բոլոր լոկալ** փոփոխականները, որոնք գտնվում են տեսանելիության տիրույթում:
- ✓ Watch – կարելի դիտարկել **նշված փոփոխականի** արժեքը: Օգտագործվում է Enter որպես հրամանի վերջ:
- ✓ Immediate – (անմիջապես) թույլատրում է փոփոխականներին տալ այլ արժեքավորում կամ կատարել լրացուցիչ կոդի ներմուծում: Օգտագործվում է Enter որպես հրամանի վերջ: [Уотсон- 183]
- ✓ Command – ով թույլատրում է օգտագործել ծրագրի աշխատանքի ժամանակ տարբեր օպերացիաների ընդգրկում: Օրինակ ընտրել գործիքների պանելից էլեմենտներ: // ? [Уотсон- 183]
- ✓ Call Stack – Ցույց է տալիս ճանապարհը, որին հասել է ընդհատված կետը: Նշվում է անունի տարածքը, տիպը, մեթոդը և տվյալ ընդհատված կետի հրամանի համարը: [Уотсон- 184]
- Վերցնենք որևէ ծարգրի օրինակ և կիրառենք կանգառի կետեր դիտարկելով փոփոխականների արժեքները:

**Օրինակ 2.** Կատարել Debug->Windows->Breakpoints նշումներ և հետևել ծրագիրը:

```
class A
{
}
namespace N
{
    class Program
    {
        static A ob = new A();
        static void f()
        {
            int a = 99;
            System.Console.WriteLine(a);
        }
        static int n;
        static int p { get; set; }
        static void Main(string[] args)
        {
            f();
            int c = 11;
            int b = 10;
            int a = 999;
            System.Console.WriteLine(a + b);
            double d = 5.5;
            b = 8;
            n = 123;
        }
    }
}
```

.....

# 81. Debug և Trace կլասներ

[Уотсон- 166; Албахари-540; Covaci – 504]

- `Debug` և `Trace` կլասները ստատիկ կլասներ են, որոնք գտնվում են `System.Diagnostics` անունի տարածքում: Նրանք փոխարինում են ծրագրավորողի Debug աշխատանքին **ծրագրային կողով միջամտման** և տվյալների ստացման համար:
- Debug ռեժիմում աշխատում են և `Trace` կլասի և `Debug` կլասի հրամանները: `Debug` կլասի անդամները ունեն `[Conditional"DEBUG"]` ատրիբուտը, որը կարելի է ղեկավարել `#define` պրեպրոցեսորի հրամանով: Լռելիությամբ ընդունվում է թարգմանչի նշածը:
- Release ռեժիմում աշխատում են միայն `Trace` կլասի հարամանները: `Trace` կլասի անդամները ունեն `[Conditional"TRACE"]` ատրիբուտը, որը նույնպես կարելի է ղեկավարել `#define` պրեպրոցեսորի հրամանով: Լռելիությամբ ընդունվում է թարգմանչի նշածը:
- `Debug` և `Trace` կլասները հնարավորություն են տալիս ինչպես `Output` պատուհանում արտածել տվյալներ, այնպես էլ կոդի կատարման ժամանակ ստեղծել **կանգառի կետեր** և շարունակել Abort, Retry, Ignore հնարավորությունով:
- Output պատուհանը ակտիվացվում է View -> Output հրամանով:
- Նշենք որ, կոմպիլյատորի `Debug` թարգմանության ժամանակ աշխատում են և `Debug` և `Trace` կլասների անդամները: `Rleace` թարգմանության ժամանակ աշխատում է միայն `Trace` կլասի անդամները: `Start Without Debugging` (Ctrl F5) ոչինչ չի արտածվում `Output` պատուհանում, սակայն աշխատում է դիալոգային պատուհանը:
- `Debug` և `Trace` կլասները ունեն ստատիկ անդամներ.

✓ `void WriteLine(string message)` – արտածում է հայտարարություն `Output` պատուհանում որպես հղորդագրություն:

✓ `void WriteLineIf(bool condition, object value)` – արտածում է `value`, եթե `condition` փոփոխականը `true` է:

✓ `void Assert(bool condition, string message)` – նշում է կանգառի կետ (**Breakpoint**) և եթե պայմանը տալիս է `false`, ապա առանձին կատարման դիալոգային պատուհանում և թարգմանիչի `Output` պատուհանում տալիս է հաղորդագրություն: **Այլ խոսքով կանգառի կետի ծրագրային ձևն է**:

✓ `void Fail(string message)` – Նման է `Assert()` –ին, սակայն **պայմանի** հնարավորություն չունի: Նույնպես կանգառի կետի ծրագրային ձևն է:

```
using System;
using System.Diagnostics;
public static class Program
{
    public static void Main()
    {
        bool b = true;
        bool b2 = false;
        int a = 99;
        string s = "alb";
        Trace.WriteLine("TTTTTTTTT");
        Debug.Write(a);
        Debug.WriteLine("DDDDDDDDDD");
        Debug.WriteLineIf(b,"xxx message", s);
        //------------------
        Debug.Assert(b2, "DebugAssert");
        Trace.Assert(b2, "TraceAssert");
        Debug.Fail("DebugFail");
        Trace.Fail("TraceFail");
}   }
```

- Պարզ է, եթե հանեք Debug->ApplPropertis->Build->DEBUG և TRACE CheckBox –ի ակտիվացումը, կամ ավելացնենք #undef DEBUG և #undef TRACE, ապա ոչին չի արտածվի:

.....

## 82. TraceListener [Албахари-542; Covaci – 504]

- ✓ `Debug` և `Trace` կլասները ունեն **`Listeners`** հատկանիշ, որը իրենից ներկայացնում է **`TraceListener`** կլասի ստատիկ կոլեկցիա: Այն հնարավորություն է տալիս ուղղորդել մշակվող կամ ցուցադրվող ինֆորմացիան դեպի տարբեր աղբյուրներ: `Trace.Listeners.Clear()` հրամանի դեպքում ցուցադրում չկա նույնիսկ `Output` -ում:
- ✓ `TraceListenerCollection Listeners { get; }` – հնարավորություն է տալիս `TraceListenerCollection` կոլեկցիայի միջոցով մշակել `Fail()` մեթոդի պարունակությունը:
- ✓ Լռելիությամբ ստեղծվում է `DefaultTraceListener` կոլեկցիա, որը ղեկավարում է `Abort, Retry, Ignore` հնարավորությունը:
- ✓ Կարելի է ստեղծել սեփական **`Listener`**-ներ, ժառանգելով **`TraceListener`** կլասից, կամ օգտվել ստանդարտ **`Listener`**-ներից: Ստանդարտ **`Listener`**-ներ են.
- ✓ `TextWriterTraceListener` – գրանցում է `Stream, TextWriter` կամ ֆայլի մեջ: Տվյալ դեպքում քանի որ, կատարվում է տվյալների հոսքի քեշավորում, հնարավոր է տվյալների ոչ ժամանակին գրանցում: Դրա համար օգտագործվում է `Flush()` կամ `Close()`մեթոդները: `Flush()` –ը կատարում է գրանցում, իսկ `Close()`–ը բացի գրանցումից փակում է նաև ֆայլի դիսկրիպտորը:
- ✓ `ConsoleTraceListener` – ուղղարկում է կոնսոլային ելք:
- ✓ `EventLogTraceListener` – գրանցում է `Windows event log`:
- ✓ `WebPageTraceListener` – գրանցում է ACP.NET Web էջ:
- ➢ Օրինակում **`Listener`**-ները ուղղարկում է կոնսոլ և ֆայլ:

```
using System;
using System.Diagnostics;//[Covaci - 504]
using System.IO;
public static class Program
{   public static void Main()
    {   Trace.Listeners.Clear();
        Trace.WriteLine("Trace started !!!!!!!!!!!!!!!!" + DateTime.Now.ToString());
        ConsoleTraceListener cons = new ConsoleTraceListener();
        Trace.Listeners.Add(cons);
        Stream traceStream = File.Create("TraceFile.txt");
        TextWriterTraceListener traceListener = new TextWriterTraceListener(traceStream);
        Trace.Listeners.Add(traceListener);
        Trace.WriteLine("Trace started !!!!!!!!!!!!!!!!" + DateTime.Now.ToString());
        Trace.Fail("TraceFail   ######################");
        Trace.Flush();
        Trace.AutoFlush=true;
        Trace.Close();
        Console.ReadKey();
}  }
```

- ➢ Հաջորդ օրինակում նեկայացվում է ստանդարտ **`Listener`**-ներ: Սկզբից մաքրվում է ստանդարտ **`Listener`**-ները: Այնուհետև ավելացվում է երեք **`Listener`**-ներ: Առաջին, գրանցում է ֆայլում: Երկրորդ, ցուցադրում է կոնսոլի վրա: Երրորդ, կատարում է գրանցում `Windows event log` –ում:

```
using System; //[Албахари-524] ??
using System.Diagnostics;
public static class Program
{   public static void Main()
    {   Trace.Listeners.Clear();
        Trace.Listeners.Add(new TextWriterTraceListener("trace.txt"));
        System.IO.TextWriter tw = Console.Out;
        Trace.Listeners.Add(new TextWriterTraceListener(tw));
        ////if (!EventLog.SourceExists("DemoApp"))
           //// EventLog.CreateEventSource("DemoApp", "Application");
        // Trace.Listeners.Add(new EventLogTraceListener("DemoApp"));
        Trace.Fail("TraceFail   ######################");      Trace.Close();
}   }
```

# 83. Debugging և Release թարգմանություն

[Нейгел-497]

- Ծրագրի արտադրական (**Release**) կողը և աշախատանքային (**Debug**) կողը տարբերվում են իրարից: “Դեբագի” ժամանկ անհրաժեշտ է սխալների հայտնաբերում: Իսկ ծրագրի արտադրական-կոմերցիոն ձևը պետք է լինի **քիչ ռեսուրս օգտագործող** և **արագագործ**:
- Ծրագրի օպտիմիզացիան (**Release** ռեժիմ) թարգմանիչի կարևոր բաղկացուցիչներից է: Այն կատարում է բավարար մեծ աշխատանք ծրագրի էֆեկտիվությունը բարձրացնելու համար և ֆիրմաները օպտիմիզացիոն հնարքները չեն տարածում, այլ պահում են **գաղտնի**: Երբ թարգմանում ենք **Release** ռեժմում, ապա կողը նույն արդյունք է ստանում հանելով ծրագրում **որոշ ավելորդ** անցումներ կամ նույնիսկ հրամաններ:

❖ Օրինակում, `var b2 = 7 * 7;` ծրագրային մասը ավելի օպտիմալ կոդ է, քան նրա `f(int a)` մեթոդով իրականացումը:
Կամ `Console.WriteLine("albdarb")` կողը ավելի ռացիոնալ է, քան
```
string s = "albdarb";
Console.WriteLine(s);
```

```
//---------------
using System;
class Program
{
    static int f(int a)
    {
         return a * a;
    }
    static void Main(string[] args)
    {
        var b = f(7);
        Console.WriteLine(b);
        var b2 = 7 * 7;
        Console.WriteLine(b2);
        //----------------
        string s = "albdarb";
        Console.WriteLine(s);
        Console.WriteLine("albdarb");
    }
}
```

- Նշված օրինակում ցանկալի կլինի կոդի օպտիմիզացիան կատարվի JIT կամպիլիատրի աշխատանքի ժամանակ, այլ ոչ թե թարգմանման, քանի որ հնարավոր է `Debuge` ռեժիմում անհրաժեշտ լինի **ընդհատման կետեր** իրականացնել նշված հատվածներում, իսկ թարքմանիչը հանի համապատասխան հատվածները:

❖ Օրինակ 2. (**Framework** ռեժիմ)

- Օրինակը `Debug` և `Release` ռեժիմներում տարբեր արդյունքներ է տալիս: Օպտիմիզատրը `Release` ռեժիմում որոշում է կանչել ob -ի դեստրուկտորը, որը դեռ դուրս չի եկել **տեսանելիության** տիրույթից: Սակայն, եթե հանենք քոմենթը, ապա կստացվի օբյեկտը դեռ կիրառելի է և `Release` -ը չի աշխատացնի դեստրուկտորը: `Debuge` –ը օպտիմիզատոր չունի և բնական է դեստրուկտոր չի կանչվում, քանի դեռ չի ավարտվել `Main()` –ը, որովհետև գործում է տեսանելիության տիրույթը:

```
using System;
class A
{
    ~A()
    {
        Console.Write("......... " +this.GetHashCode());
```

```
    }
}
public static class Program
{
    public static void Main()
    {
        A ob = new A();
        GC.Collect();
        //Console.WriteLine(ob.ToString());  // ակտիվացնելու դեպքում օպտիմիզատրը չի աշխատի
        Console.ReadKey();
    }
}
//---------------------------------------
```

❖ Օրինակ 3. (**Framework** ռեժիմ)

- Օրինակի աշխատանքի ժամանակ կարելի է փորձել **Debuge** և **Release** ռեժիմները և արդյունքում կտեսնենք **Release** ռեժիմը օպտիմալացնում է ծրագիրը: `GC.Collect()` հրամանը ժամանակ հաշվող ծրագիրը հեռացնում է, քանի որ `Timer` օբյեկտը երկար ժամանակ մնում է առանց հղման:

```
// Kort-220
using System;
using System.Threading;
public static class Program
{
    public static void Main()
    {
        Timer t = new Timer(TimerCallback, null, 0, 2000);
        Console.ReadKey();
    }
    private static void TimerCallback(Object o)
    {
        Console.WriteLine("In TimerCallback: " + DateTime.Now);
        GC.Collect();
    }
}
//---------------------------------------
```

❖ Օրինակ 4.

- Ծրագրի `CIL` հարմանները դիտարկենք `ILDasm` –ով և համեմատենք `Debuge` և `Release` ռեժիմները.

```
// Debug Code size         44
// Release Code size       36

using System;
namespace AlbDarb
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 55;
            int b = 77; // չօգտագործվող փոփոխական
            Console.WriteLine("a value = "+a);
            string s = "albdarb"; // անիմաստ փոփոխական
            Console.WriteLine(s);
        }
    }
}
```

DEBUG

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       44 (0x2c)
  .maxstack  2
  .locals init ([0] int32 a, // 3 փոփոխական է հայտարարված
           [1] int32 b,
           [2] string s)
  IL_0000:  nop
  IL_0001:  ldc.i4.s   55
  IL_0003:  stloc.0
  IL_0004:  ldc.i4.s   77
  IL_0006:  stloc.1
  IL_0007:  ldstr      "a value = "
  IL_000c:  ldloca.s   a
  IL_000e:  call       instance string [mscorlib]System.Int32::ToString()
  IL_0013:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0018:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_001d:  nop
  IL_001e:  ldstr      "albdarb"
  IL_0023:  stloc.2
  IL_0024:  ldloc.2
  IL_0025:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_002a:  nop
  IL_002b:  ret
}
// end of method Program::Main
```

RELEASE

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       36 (0x24)
  .maxstack  2
  .locals init ([0] int32 a) // 1 փոփոխական է հայտարարված
  IL_0000:  ldc.i4.s   55
  IL_0002:  stloc.0
  IL_0003:  ldstr      "a value = "
  IL_0008:  ldloca.s   a
  IL_000a:  call       instance string [mscorlib]System.Int32::ToString()
  IL_000f:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0014:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0019:  ldstr      "albdarb"
  IL_001e:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0023:  ret
}
// end of method Program::Main
```

.....

# Անվտանգություն (Security)

## 84. Անվտանգություն

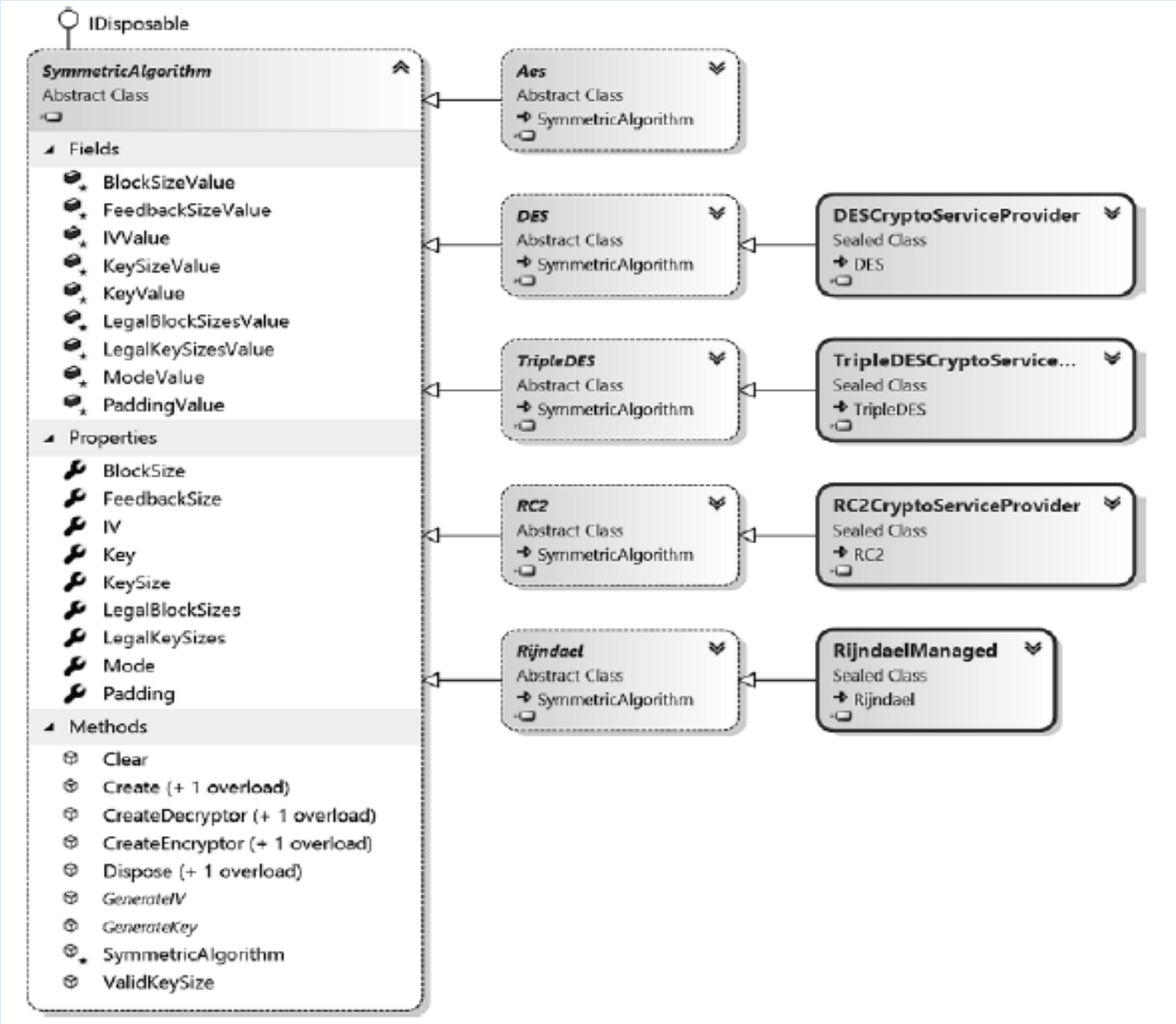
[Прайс-353; Albahari – 851]

- **(Cryptography)** : Տվյալները և ծրագրերը ունեն անհրաժեշտություն իրենց օգտագործման ընթացքում լինեն **պաշտպանված**: Ծրագրային կոդում կարելի է ներառել այնպիսի տեխնոլոգիական հնարավորություններ, որոնք կարող են ստեղծել անվտանգության իրականացում, որը կպաշտպանի չնախատեսված և ոչ ցանկալի միջամտումներից: Անվտանգության անհրաժեշտություն առաջանում է, երբ տվյալ է տեղափոխվում ինֆորմացիոն միջավայրում (ասենք ցանցերում) կամ երբ օգտագործողներ են առաջանում ստեղծված ինֆորմացիոն միավորի տարածքում:
- **“Շիֆրացում” և “Դեշիֆրացում”**, երբ անհրաժեշտ է երկողմանի ձևով սովորական տեքստը վերածել հատուկ անծանոթ տեքստի և հակառակը:
- **“Հեշավորում”**, դա տվյալի միակողմանի ձևով “հեշ” արժեքի ձևավորումն է, որը ապահովում է անվտանգ ծածկագրերի ստեղծումը և չնախատեսված ձևով տվյալի **փոփոխման** կամ վնասման ապահովումը:
- **Թվային ստորագրություն**, երբ հնարավորություն է ստեղծվում ստուգել տվյալ ուղղարկողի ով լինելը: Այն կատարվում է բաց բանալու “վերիֆիկացիայի միջոցով”:
- **“Առւթենտիկացիա”**, երբ կատարվում օգտագործողի ճանաչում տվյալների **հսկման ցուցակով**:
- **“Ավտորիզացիա”**, երբ անրաժեշտ է ղեկավարել ինֆորմացիան տիրելու **թույլատրության աստիճանը**, ելնելով նրա խմբի և կատարման մասնակցության չափանիշներից:
- Կոդի անվտանգության ապահովումը կատարվում է տարբեր ալգորիթմների միջոցով և հաճախ օգտագործվում է **բանալու (kye)** գաղափարը: Ալգորիթմները լինում են **սիմետրիկ** և **ոչ սիմետրիկ**: Սիմետրիկ ալգորիթմները ունեն **ընդհանուր** բանալի, որը օգտագործվում է և շիֆրացման և դեշիֆրացման ժամանակ: Ոչ սիմետրիկ ալգորիթմների ժամանակ օգտագործվում է այսպես ասած **փակ և բաց** բանալիների զույգ, որտեղ շիֆրացման ժամանակ օգտագործվում է բաց բանալին, իսկ դեշիֆրացման ժամանակ փակ բանալին: Նշվածը ճիշտ է, եթե կատարվում է **տվյալների** շիֆրացում: Երբ անհրաժեշտ է թվային ստորագրությամբ շիֆրացում, ապա կատարվում է **լրիվ հակառակ** աշխատանք, տվյալները շիֆրավորվում են փակ բանալով, իսկ դեշիֆրացումը կատարվում է բաց բանալիով:
- **Վեկտորային ինիցալիզացիա** (IV - Initialization Vector): Շատ հավանական է, երբ կատարվող տվյալների շիֆրացումը պարունակում են տեքստային կրկնություններ, որոնք գուշակելի դառնան անցանկալի միջամտողի համար: Օրինակ, անգլերենում հաճախ է կրկնվում `the` նախդիրը, որը ենթադենք շիֆրավորվել է `hQ2` տառերով: Պարզ է նրան հայտնաբերելը դառնում է հեշտ: When the wind blew hard the umbrella broke. // տեքստը
  `5:s4&hQ2aj#D f9d1d£8fh"&hQ2s0)an DF8SFd#][1` // շիֆրացված տեքստը
  Այսպիսի բացահայտումից խուսափելու համար օգտագործվում է տվյալների **բլոկների** բաժանման մեթոդը: Առաջին բլոկը շիֆրացնելուց հետո գեներացվում է նաև **բայթերի զանգված,** որը փոխանցվում է հաջորդ բլոկ սիմվոլնրի շիֆրացումը այլ հերթականության բերելու համար:
- **Salt:** Այսպես ասած “աղը” իրենից ներկայացնում է **պատահական բայթերի զանգված** է, որը միակողմանի “հեշ” ֆունկցիաների անվտանգության լրացուցիչ տվյալներ են: Քանի որ, միշտ նույն ծածկագրին նույն **հեշն** է տրամադրվում, ապա այդպիսի գուշակվող ծածկագրերը կարելի է ուժեղացնել ծածկագրին “կանկատենացիայով” ավելացնելով մինչև հեշավորումը, պատահական թվերի գեներատրով արժեք: Երբ օգտագործողը **ավտորիզացվում** է համակարգ և մուտքագրում է ծածկագիրը կատարվում է նշված “աղ” կոդի տարանջատում և այնուհետև համեմատվում բազայում գտնվող հեշի արժեքի հետ: Պարզ է համընկման դեպքում ծածկագիրը ճիշտ է:
- **Կլաս `Rfc2898DeriveBytes`:** Նախատեսված է **բանալու և ինիցիալիզացիայի վեկտրի** գեներացիայի համար: Այն ընդունում է ծածկագիր, “աղ” և իտերացիայի հաշվիչ և `GetBytes` մեթոդի միջոցով գենրացնում է `key` և `Initialization Vector`:

# 85. Տվյալների շիֆրացում և դեշիֆրացում

[Прайс-356 ; Нейгел-674; Албахари – 839; Albahari – 855; Covaci – 528]

- .Net Core պլատֆորմայով իրականացվում է տարբեր շիֆրացման ալգորիթմներ, որոնք կարող են լինել **սիմետրիկ և ասիմետրիկ**: Եթե ալգորիթմները իրականացվում են օպերացիոն համակարգի միջոցով, ապա օգտվում ենք `CryptoServiceProvider` կլասից: Եթե օգտվում ենք .NET միջավայրից, ապա օգտվում ենք `Managed` կլասից:
- **Ներկայացնենք** որոշ կլասներ սիմետրիկ ալգորիթմներից, որնցից առավել տարածվածներից է **AES**(Advanced Encryption Standard) ալգորիթմը:




❖ **Սիմետրիկ կրիպտոգրաֆիա:** [Прайс-357; Албахари – 839; Albahari-855]

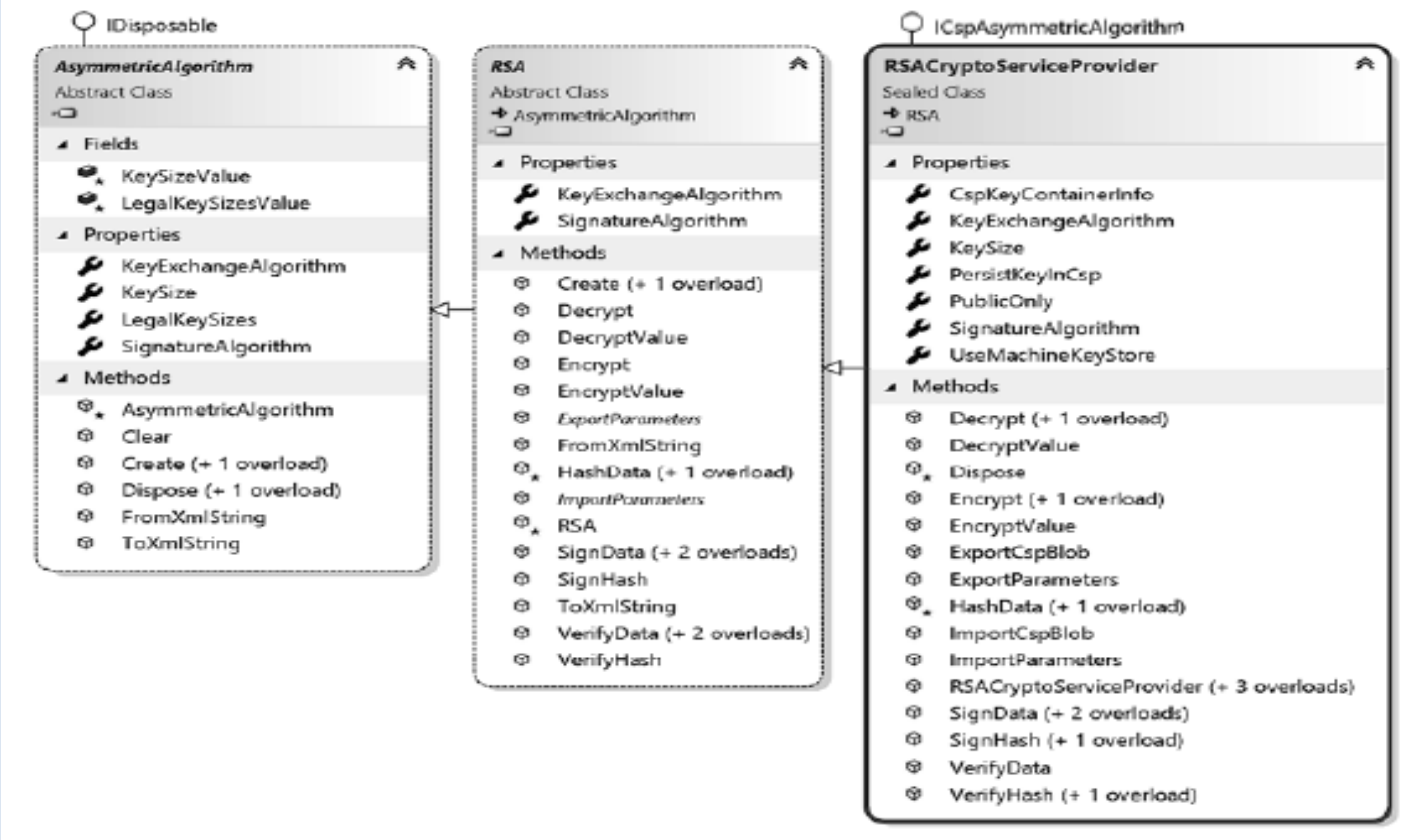
- (Encryption) Օրինակը կատարվում է սիմետրիկ AES ալգորիթմով, որտեղ օգտվում ենք `CryptoStream` կլասից, որը հնարավորություն է տալիս աշխատել մեծ քանակով բայթերի հետ և տվյալները տեղադրվում են **տվյալների հոսքում**: (Ասիմետրիկի դեպքում մշակվում է ավելի սահմանափակ քանակությամբ տվյալներ և այն տեղադրվում **բայթերի զանգվածում**):
- Շիֆրացման արդյունքի համար օգտագործվում է ժամանակավոր տեղ `MemoryStream` կլասով և օգտագործվում է `ToArray()` մեթոդը հոսքից զանգված ստանալու համար:
- Շիֆրացված բայթերի զանգվածի կոնվերտացիայի համար օգտագործվում է `Base64` կոդավորումը:
- Օրինակը սկզբից փորձենք որևէ ծածկագրով և հետո կիրառենք սխալ ծածկագիր:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
public static class P
{
    static readonly byte[] salt = Encoding.Unicode.GetBytes("abcd"); // >=8 byte
    static readonly int iterations = 1000;  // >=1000
    public static string f_Encrypt(string s, string password)
    {
        byte[] plainBytes = Encoding.Unicode.GetBytes(s);
        var aes = Aes.Create();
        var rfc = new Rfc2898DeriveBytes(password, salt, iterations);
        aes.Key = rfc.GetBytes(32); // 256-bit key
        aes.IV = rfc.GetBytes(16);  // 128-bit key
        var ms = new MemoryStream();
        using (var cs = new CryptoStream(ms, aes.CreateEncryptor(), CryptoStreamMode.Write))
        {
            cs.Write(plainBytes, 0, plainBytes.Length);
        }
        return Convert.ToBase64String(ms.ToArray());
    }
    public static string f_Decrypt(string s, string password)
    {
        byte[] cryptoBytes = Convert.FromBase64String(s);
        var aes = Aes.Create();
        var rfc = new Rfc2898DeriveBytes(password, salt, iterations);
        aes.Key = rfc.GetBytes(32);
        aes.IV = rfc.GetBytes(16);
        var ms = new MemoryStream();
        using (var cs = new CryptoStream(ms, aes.CreateDecryptor(), CryptoStreamMode.Write))
        {
            cs.Write(cryptoBytes, 0, cryptoBytes.Length);
        }
        return Encoding.Unicode.GetString(ms.ToArray());
    }
}
class M
{
    static void Main()
    {
        Console.Write("Enter encrypt Text: ");
        string message = Console.ReadLine();
        Console.Write("Enter a password: ");
        string password = Console.ReadLine();
        string s = P.f_Encrypt(message, password);
        Console.WriteLine($"Encrypted text: {s}");
        Console.Write("Enter the password: ");
        string password2 = Console.ReadLine();
        try
        {
            string s2 = P.f_Decrypt(s, password2);
            Console.WriteLine($"Decrypted text: {s2}");
        }
        catch
        {
            Console.WriteLine("wrong password!");
        }
}   }
```

# 86. Ասիմետրիկ կրիպտոգրաֆիա

[Албахари – 843; Albahari – 861]

- Ասիմետրիկ կրիպտոգրաֆիայի ժամանակ շիֆրացումը և դեշիֆրացումը կատարվում է տարբեր բանալիներով: Այն պահանջում է հատուկ ձևավորվող զույգ բանալիներ, **բաց բանալի և գաղտնի բանալի**: Բաց բանալին կատարում է շիֆրացումը-ուղղարկողը, իսկ փակ բանալին, դեշիֆրացումը-ստացողը:
- Ենթադրենք երկու համակարգիչներ պետք է ասիմետրիկ շիֆրով տվյալներ ուղղարկեն: Ասենք A համակագիչը (կլիենտ) պետք է ասիմետրիկ ծածկագրումով տվյալ ուղղարկի B համակարգչին (բանկ) ընդհանուր **բաց ցանցով**: Դրա համար B համակարգիչը գեներացնում է բաց և գաղտնի բանալիներ և բաց բանալին ուղղարկում է A համակարգչին: A համակարգիչը ստացած բաց բանալով կատարում է շիֆրացում և ուղղարկում է B համակարգչին: B համակարգիչը դեշիֆրացնում է իր մոտ գտնվող, բոլորից գաղտնի փակ բանալիով: Եվ ահա ինչ կարող է տեսնել ինֆորմացիա գողացողը: Նա կարող է բռնացնել բաց բանալի կամ այդ բաց բանալով շիֆրացված տվյալը, սակայն չունենալով փակ բանալին, նա չի կարող տիրապետել շիֆրացված տվյալներին:
- Ներկայացնենք որոշ կլասներ ասիմետրիկ ալգորիթմներից, որնցից առավել տարածվածներից է **RSA (Rivest Shamir Adleman)** ալգորիթմը:




- Օրինակը առանց բանալիներ, որտեղ օգտագործվում է `RSACryptoServiceProvider` կլասը, որի լռելիությամբ կոնստրուկտորը թույլատրում է կրիպտոգրաֆիայի 1024 բիթ: Ընդհանրապես նշված կլասը թույլատրում է 2 –ի աստիճաններով “ֆլագներ” սկսած 512 արժեքից:

```
using System;
using System.Security.Cryptography;
public static class Program
{
    public static void Main()
    {
        byte[] data = { 1, 2, 3, 4, 5 };
        using (var rsa = new RSACryptoServiceProvider())// ()standart long = 1024bit
        {
```

```
                //RSACryptoServiceProvider(2048)
                byte[] ens = rsa.Encrypt(data, true);
                foreach(byte v in ens)
                    Console.Write(v+" ");
                Console.WriteLine();
                byte[] des = rsa.Decrypt(ens, true);
                foreach (byte v in des)
                    Console.Write(v + "    ");
        }
    }
}
```

- Հաջորդ օրինակը ստեղծում է բաց և գաղտնի բանալիներ, օգտվելով `RSACryptoServiceProvider` կլասի `FromXmlString()` և `ToXmlString()` մեթոդներից, որոնք ունեն XML տեքստային ֆորմատ: `ToXmlString()`մեթոդի պարամետրի `true` արժեքի դեպքում ընդգրկում է փակ և բաց բանալիները, իսկ `false` արժեքի դեպքում միայն բաց բանալին:

```
using System;
using System.IO;
using System.Security.Cryptography;

class Program
{
    static void Main(string[] args)
    {
        byte[] data = { 1, 2, 3, 4, 5 };
        using (var rsa = new RSACryptoServiceProvider())
        {
            File.WriteAllText("PublicKeyOnly.xml", rsa.ToXmlString(false));
            File.WriteAllText("PublicPrivateKey.xml", rsa.ToXmlString(true));
        }
        string publicKeyOnly = File.ReadAllText("PublicKeyOnly.xml");
        string publicPrivateKey = File.ReadAllText("PublicPrivateKey.xml");
        byte[] encrypted, decrypted;
        using (var rsaPublicOnly = new RSACryptoServiceProvider())
        {
            rsaPublicOnly.FromXmlString(publicKeyOnly);
            encrypted = rsaPublicOnly.Encrypt(data, true);
            foreach (byte v in encrypted)
                Console.Write(v + " ");
            Console.WriteLine();
            //decrypted = rsaPublicOnly.Decrypt(encrypted, true); // error
            //foreach (byte v in decrypted)
            //    Console.Write(v + "    ");
        }
        using (var rsaPublicPrivateKey = new RSACryptoServiceProvider())
        {
            // Օգտագործենք փակ բանալին դեշիֆրացիայի համար
            rsaPublicPrivateKey.FromXmlString(publicPrivateKey);
            decrypted = rsaPublicPrivateKey.Decrypt(encrypted, true);
            foreach (byte v in decrypted)
                Console.Write(v + "    ");
        }
        Console.WriteLine();
    }
}
```

- `RSACryptoServiceProvider` կլասի `ExportCspBlob()` և `ImportCspBlob()` մեթոդները բայթ զանվածի տեսքով նույնպես կարող են ստեղծել բաց և փակ բանալիներ:

# 87. Տվյալների հեշավորում

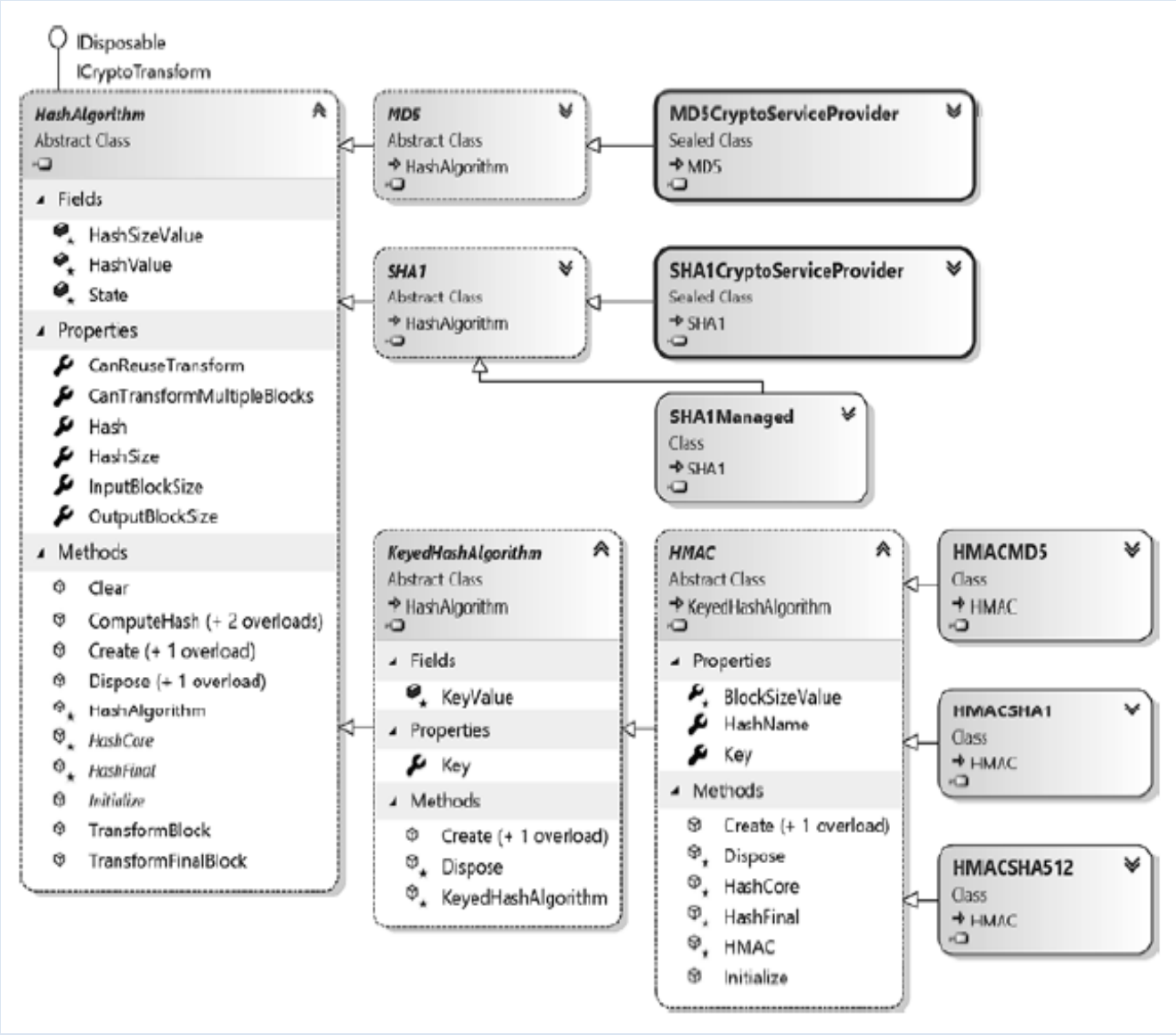
[Прайс-361; Covaci – 538; Албахари – 837; Albahari – 853]

- Տերմիններ:
- **Հեշավորումը,** դա փոփոխական չափ ունեցող երկուական տվյալների արտապատկերումն է մի այլ երկուական ֆիքսված չափերով տվյալների: **Միևնույն** կառուցվածքով տվյալները տալիս են **նույն հեշ** արժեքը:
- **Տվյալների անվնտություն:** Օգտագործվում են այն դեպքում, երբ անհրաժեշտ է տվյալների տեղափոխությունը տեղի ունենա առանց կորուստների: Ուաղղարկողը հաշվում է կրիպտոգրաֆիկ հեշ և ուղարկում է տվյալը, հեշը և մեթոդիկան, որպեսզի ստացողը հեշը կարողանա հաշվել: Ստացողը կարող է հեշավորել ստացված տվյալները ստացված մեթոդով և համեմատել ստացված հեշի հետ: Համնկնման դեպքում ստացված տվյալը համարվում է ճիշտ: Պազ է, եթե ինֆորմացիա վնաս տվողը կարող է նաև կիրառել իր ալգորիթմը ճանապարհին և ուղղարկել նաև ալգորիթմը և այս ձևով խաբել ստացողին:
- **Տվյալների իսկությունը:** Օգտագործվում է, երբ անհրաժեշտ է ունենալ ստացվող տվյալնների իսկական լինելու հանգամանքը: Այն աշխատում է հետևյալ ձևով – ուղղարկողը ստանում է կրիպտոգրաֆիկ հեշ և ստորագրում է այն իր սեփական փակ բանալիով: Ստացողը նույնպես հեշավորում է և հետո վերծանում է ստացվող ստորագրությունը, օգտագործելով ուղղարկողի բաց բանալին և համեմատում է հեշերը:
- **Ծածկագրի պահպանում:** Ծածկագիրը սովորական ձևով պահելը համրավում է անապհով ձև և եթե պահպանման տեղը դարձել է հասանելի վնասարարի կողմից, ապա արդյունքը պարզ է: Ծածկագրերը պաշտպանելու համար նրանք սովորաբար հեշավորվում են և ծածկագրի փոխարեն հեշն է պահվում:
- **Կրիպտոգրաֆիկ հեշի** առավելությունը կայանում է նրանում, որ երկու տարբեր մուտքերի գեներացված հեշի նույն լինելը քիչ հավանական է և երկու շատ քիչ միմյանցից տարբերվող ծածկագրերը կարող են գեներացվել միմյանցից միանգամայն մեծ տարբերությամբ հեշեր:
- Գոյություն ունի երկու տիպի հեշավորման ալգորիթմներ, **բանալիով և առանց բանալի:** Առանց բանալի հեշավորումը նախատեսված է տեղափոխման ժամանակ ուղղակի տվյալների կորուստից խուսափելու համար, իսկ բանալիով հեշավորման դեպքում կատարվում է նաև աուդենտիֆիկացիա MAC (Message Authentication Code) ալգորիթմով:
- **Տվյալների ինդեքսավորում:** Տվյալների ինդեքսավորումը հեշի միջոցով արագացնում է փնտրման ժամանակը:

- Հեշավորման ալգորիթմներ ընտրելիս հաշվի պետք է առնել, թէ որքան հաճախ երկու տարբեր մուտքեր կունենան նույն հեշը:
- Նշենք որոշ հեշավորման ալգորիթմներ.

MD5 (Message Digest) - 16 բայթ, աշխատում է արագ, սակայն հեշի համընկման հավանականությունը մեծ է:
SHA (Secure Hashing Algorithm) - պաշտպանված հեշավորումով ալգիրթմներ:
SHA1 – 20 բայթ, ցանկալի չէ օգտագործել ինտերնետում:
SHA2 - երկրորդ սերնդի հեշեր, որոնք ունեն տարբեր հեշի մեծություն.
SHA256 - 32 բայթ
SHA384 - 48 բայթ
SHA512 - 64 բայթ

- .Net Core պլատֆորմում հասանելի են հեշավորման ալգորիթմներ: Որոշ ալգորիթմներ չեն պահանջում բանալիներ, որոշ մասը պահանջում է սիմետրիկ բանալի, որոշ մասը ասիմերտրիկ բանալիներ:




- Օրինակում ցուցադրվում է, որ նույնիսկ եթե երկու տարբեր օգտագործողներ գրանցվում են նույն ծածկագրով, ապա միևնույնն է, `solt` -ով և հեշավորումով ստացված ծածկագրերը ստացվում են տարբեր:

```
using System;
using System.Security.Cryptography;
using System.Text;
using static System.Console;
using System.Collections.Generic;
public class User
{
    public string Name { get; set; }
    public string Salt { get; set; }
    public string SaltedHashedPassword { get; set; }
}
public static class Protector
{
    private static Dictionary<string, User> Users = new Dictionary<string, User>();

    public static User Register(string username, string password)
    {
        var rng = RandomNumberGenerator.Create();
        // salt գեներացիա
        var saltBytes = new byte[16];
        rng.GetBytes(saltBytes);
        var saltText = Convert.ToBase64String(saltBytes);
```

```
        var sha = SHA256.Create();
        // գեներացիա salt -ացված և հեշացված ծածկագիր
        var saltedPassword = password + saltText;
        var saltedhashedPassword = Convert.ToBase64String(
                                   sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));
        var user = new User
        {
            Name = username,
            Salt = saltText,
            SaltedHashedPassword = saltedhashedPassword
        };
        Users.Add(user.Name, user);
        return user;
    }
    public static bool CheckPassword(string username, string password)
    {
        if (!Users.ContainsKey(username))
            return false;
        var user = Users[username];
        var sha = SHA256.Create();
        // գեներացիա salt -ացված և հեշացված ծածկագիր
        var saltedPassword = password + user.Salt;
        var saltedhashedPassword = Convert.ToBase64String(
        sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));
        return (saltedhashedPassword == user.SaltedHashedPassword);
   }
}
class M
{
    static void Main()
    {
        WriteLine("User Ani password= aaa");
        var v = Protector.Register("Ani", "aaa");
        WriteLine($"Name: {v.Name}");
        WriteLine($"Salt: {v.Salt}");
        WriteLine($"Salted and hashed password: {v.SaltedHashedPassword}");
        WriteLine();
        Write("Enter a different username: ");
        string username = ReadLine();
        Write("Enter a password: ");
        string password = ReadLine();
        var user = Protector.Register(username, password);
        WriteLine($"Name: {user.Name}");
        WriteLine($"Salt: {user.Salt}");
        WriteLine($"Salted and hashed password: {user.SaltedHashedPassword}");
        bool correctPassword = false;
        while (!correctPassword)
        {
            Write("Enter a username: ");
            string loginUsername = ReadLine();
            Write("Enter a password: ");
            string loginPassword = ReadLine();
            correctPassword = Protector.CheckPassword(loginUsername,loginPassword);
            if (correctPassword)
                WriteLine($"Correct! {loginUsername}");
            else
                WriteLine("Invalid");
        }
}   }
```

# 88. Թվային ստորագրություն

[Прайс-365; Албахари – 845; Albahari-863]

- Երբ անհրաժեշտ է համոզված լինել, որ տվյալները ուղղարկված են **վստահելի աղբյուրից**, ապա օգտագործվում է թվային ստորագրություն համակարգը: Վերջին հաշվով ստորագրվում է ոչ թե տվյալները, այլ նրա **հեշ** նշանակությունը: Օրինակ SHA256 ալգորիթմով գեներացվում է "հեշ" և ստորագրում ենք այն օգտագործելով RSA:
- Ստորագրության օրինակ SHA256 և RSA ալգորիթմներով. Օգտագործենք RSA կլասի ToXmlString() и FromXmlString() մեթոդները, որոնք կատարում են սերիալիզացիա և դեսերիալիզացիա RSAParameters կառույցը, որը ընդգրկում է փակ և բաց բանալիներ: Նշված մեթոդները RSA կլասում ընդգրկվել են ընդլայնված (Extension) մեթոդների տեխնոլոգիայով:

```
using System;
using System.Security.Cryptography;
using System.Text;
using static System.Console;
using System.Xml.Linq;
public static class Protector
{
    public static string PublicKey;
    public static string ToXmlStringExt(this RSA rsa, bool includePrivateParameters)
    {
        var p = rsa.ExportParameters(includePrivateParameters);
        XElement  xml = new XElement("RSAKeyValue"
        , new XElement("Modulus", Convert.ToBase64String(p.Modulus))
        , new XElement("Exponent", Convert.ToBase64String(p.Exponent)));
        return xml?.ToString();
    }
    public static void FromXmlStringExt(this RSA rsa, string parametersAsXml)
    {
        var xml = XDocument.Parse(parametersAsXml);
        var root = xml.Element("RSAKeyValue");
        var p = new RSAParameters
        {
            Modulus = Convert.FromBase64String(root.Element("Modulus").Value),
            Exponent = Convert.FromBase64String(root.Element("Exponent").Value)
        };
        rsa.ImportParameters(p);
    }
    public static string GenerateSignature(string data)
    {
        byte[] dataBytes = Encoding.Unicode.GetBytes(data);
        var sha = SHA256.Create();
        var hd= sha.ComputeHash(dataBytes);// hashedData
        var rsa = RSA.Create();
        PublicKey = rsa.ToXmlStringExt(false); // գեներացնում է PublicKey  // *
        return Convert.ToBase64String(rsa.SignHash(hd, HashAlgorithmName.SHA256,
                                      RSASignaturePadding.Pkcs1));
    }
    public static bool ValidateSignature(string data, string signature)
    {
        byte[] dataBytes = Encoding.Unicode.GetBytes(data);
        var sha = SHA256.Create();
        var hd = sha.ComputeHash(dataBytes); // hashedData
        byte[] sb = Convert.FromBase64String(signature);// signatureBytes
        var rsa = RSA.Create();
        rsa.FromXmlStringExt(PublicKey);
        return rsa.VerifyHash(hd, sb, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    }
```

```
}
class M
{
    static void Main()
    {
        Write("Enter some text to sign: ");
        string data = ReadLine();
        var signature = Protector.GenerateSignature(data);
        WriteLine($"Signature: {signature}");
        WriteLine("Public key used to check signature:");
        WriteLine(Protector.PublicKey); // *
        // Սեղծենք առանձին ստորագրություն փոխելով 1 -ին սիմվոլը օրինակ՝ X
        WriteLine("input 1-st signature char");
        char c =Convert.ToChar(Console.ReadLine());
        signature = signature.Replace(signature[0], c);
        if (Protector.ValidateSignature(data, signature))
            WriteLine("Correct! Signature is valid.");
        else
            WriteLine("Invalid signature.");
    }
}
```

.....

## 89. Աուդենտիկացիա և ավտորիզացիա

[Прайс-371; Нейгел-665]

- **Աուդենտիկացիան** իրենից ներկայացնում է օգտագործողի **ով լինելու** որոշման համակարգ: Աուդենտիկացիայի պրոցես է համարվում, երբ օրինակ՝ կատարվում է անհատականության տվյալների վերիֆիկացիայի համեմատում որևէ անուն ձեռք բերած ինֆորմացիոն աղբյուրում: Տվյալները հիմնականում լինում են անուն և ծածկագիր: Աուդենտիկացիան հնարավորություն է տալիս **հովանավոր** տվյալների ավելի բացված տեղեկությունների ստացում և վստահված ռեգիստրացիայից օգտվելով շարունակել աշխատանքը:
- **Ավտորիզացիան** դա հնարավորություն է, երբ համարվում ես վստահավաձ խմբի անդամ և թույլատրություն ես ստանում տարբեր **ռեսուրսներից օգտվելու**:
- Աուդենտիկացիան և ավտորիզացիան օգտվում է System.Security.Principal անունի տարածքից օգտվելով IIdentity և IPrincipal ինտերֆեյսներից:

```
using System;
using System.Security.Cryptography;
using System.Text;
using static System.Console;
using System.Collections.Generic;
using System.Threading;
using System.Security.Principal;
using System.Security.Claims; // պնդում
// using System.Security;
// using System.Security.Permissions;

public class User
{
    public string[] Roles { get; set; }
    public string Name { get; set; }
    public string Salt { get; set; }
    public string SaltedHashedPassword { get; set; }
}
public class Protector
{
    private static Dictionary<string, User> Users = new Dictionary<string, User>();
```

```
    public static void RegisterSomeUsers()
    {
        Register("alb", "psw", new[] { "Admins" });
        Register("ani", "a", new[] { "Programmer", "TeamLeads" });
        Register("eve", "e");
    }
    public static void Register(string username, string password, string[] roles = null)
    {
        // генерация соли
        var rng = RandomNumberGenerator.Create();
        var saltBytes = new byte[16];
        rng.GetBytes(saltBytes);
        var saltText = Convert.ToBase64String(saltBytes);
        // генерация соленого и хешированного пароля
        var sha = SHA256.Create();
        var saltedPassword = password + saltText;
        var saltedhashedPassword = Convert.ToBase64String(
            sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));
        var user = new User
        {
            Name = username,
            Salt = saltText,
            SaltedHashedPassword = saltedhashedPassword,
            Roles = roles
        };
        Users.Add(user.Name, user);
    }
    public static void LogIn(string username, string password)
    {
        if (CheckPassword(username, password))
        {
            var identity = new GenericIdentity(username, "HighCode");
            var principal = new GenericPrincipal(identity,  Users[username].Roles);
            Thread.CurrentPrincipal = principal;
        }
    }
    public static bool CheckPassword(string username, string password)
    {
        if (!Users.ContainsKey(username))
            return false;
        var user = Users[username];
        // повторная генерация соленого и хешированного пароля
        var sha = SHA256.Create();
        var saltedPassword = password + user.Salt;
        var saltedhashedPassword = Convert.ToBase64String(
            sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));
        return (saltedhashedPassword == user.SaltedHashedPassword);
    }
}
class M
{
    static void Main()
    {
        Protector.RegisterSomeUsers(); // համակարգում կատարվում է Users -ի գրանցում
        Write($"Enter your user name: ");
        string username = ReadLine();
        Write($"Enter your password: ");
        string password = ReadLine();
        Protector.LogIn(username, password);
        if (Thread.CurrentPrincipal == null)
```

```
        {
            WriteLine("Log in failed.");
            return;
        }
        var p = Thread.CurrentPrincipal;
        WriteLine($"IsAuthenticated: {p.Identity.IsAuthenticated}");
        WriteLine($"AuthenticationType: {p.Identity.AuthenticationType}");
        WriteLine($"Name: {p.Identity.Name}");
        WriteLine($"IsInRole(\"Admins\"): {p.IsInRole("Admins")}");
        WriteLine($"IsInRole(\"Programmer\"): {p.IsInRole("Programmer")}");
        if (p is ClaimsPrincipal)
        {
            WriteLine($"{p.Identity.Name} has the following claims:");
            foreach (Claim claim in (p as ClaimsPrincipal).Claims) // պնդում
            {
                WriteLine($" {claim.Type}: {claim.Value}");
            }
        }
        Console.ReadKey();
    }
}
```

.....

# Ամփոփում (Summary)

## 90. C#.NET ծրագրավորման OOP, կոմպոնենտ և Runtime էտապները

[albdarb]

- Ինչպես նշվեց C#1 կուրսի առաջին դասում, մենք ստեղծեցինք **ծրագրավորման մոդելը**, որը ուներ երկու բաղկացուցիչ: Առաջինը՝ դա խնդիրն է, որը մոդելավորելով ըստ քայլերի հաջորդականության ստանում ենք **ալգորիթմ** և երկրորդը՝ նրա **իրականացումը**, որը կատարվում է համակարգիչ անվամբ **տեխնիկական միջավայրում**:
- Ամբողջ դասավանդման ընթացքում մենք կատարել ենք աստիճանական զարգացում՝ սկսել ենք լեզվի սինտաքսիս/սեմանտիկայից, անցնելով՝ ալգորիմական, ֆունկցիոնալ, օբյեկտ կողմնորոշված և կոմպոնենտ **աբստրակցիային**, որը խնդրի **մոդելավորումն** ու ալգորիթմական իրականացումն է և վերջացրել ենք Runtime –ով (ղեկավարվող կոդ, CLR, պրոցես, հոսք), որը խնդրի **տեխնիկական** իրականացումն է:
- Եթե ծրագրավորումը հաեմատենք մաթեմատիկա գիտության հետ, ապա կարող ենք ասել, որ մաթեմատիկան, դա **աբստրակցիա** է սիմվոլային լեզվով (ծրագրավորման մոդելում՝ սինտաքսիս, OOP, կոմպոնենտ), իսկ նրա փորձարկումը (իրականացումը), դա ֆիզիկան է, որը արտապատկերում է մատերիան իր **ժամանակատարածքային** միջավայրում (ծրագրավորման մոդելում՝ Runtime):

##########.....END…. C# 2(.NET)……..##########

## Թեստերի Օրինակներ.

❖ Ծրագիրը թարգմանվել է Debug ռեժիմում և կարտածի

```
using System;
class Program
{
    static void Main(string[] args)
    {
#if DEBUG
        Console.Write("Debugmode ");
#else
        Console.Write("Notdebug ");
#endif
#if TRACE
        Console.Write("Tracemode ");
#else
        Console.Write("Nottrace ");
#endif
#if RELEASE
        Console.Write("Releasemode ");
#else
        Console.Write("Notrelease ");
#endif
        Console.ReadKey();
    }
}
```

a) Notdebug Tracemode Notrelease

b) Debugmode Tracemode Notrelease

c) Notdebug Nottrace Notrelease

d) Notdebug Nottrace Releasemode

e) չունի այդպիսի ռեժիմ

❖ Ծրագիրը կարտածի

```
using System;
using System.IO;
class M
{
    static void Main()
    {
        FileStream mm = new FileStream("d:\\t.dat", FileMode.OpenOrCreate);
        for (int i = 0; i < 5; i++)
            mm.WriteByte((byte)i);
        mm.Position = 0;
        for (int i = 0; i < 7; i++)
            Console.Write(mm.ReadByte()+" ");
        mm.Close();
    }
}
```

a) 0 1 2 3 4

b) 0 1 2 3 4 -1 -1

c) 0 1 2 3 4 5 6

d) 0 1 2 3 4 0 0

e) ոչ, կտա թարգմանման սխալ

❖ HashSet<> [checkbox]

a) կոլեկցիան բացառում է կրկնվող էլեմենտներ

b) ISet<T>, ICollection<T>, IEnumerable<T> ինտերֆեյսները

c) կլոեկցիայում օգտագործվում է Reverse() մեթոդը, որը "ռեվերսում" է էլեմենտները

d) կարող է վերադասավորել էլեմենտները IComparer<> ինտերֆեյսի միջոցով

e) կոլեկցիայում անդամ ավելացվում է ցանկացած տեղից:

❖ Ծրագիրը կարտածի

```
using System;
using System.Linq;
class Item
{
    public string Name { get; set; }
    public int N{ get; set; }
    public Item(string n, int inum)
    {
        Name = n;
        N = inum;
    }
}
class Status
{
    public int N { get; set; }
    public bool InStock { get; set; }
    public Status(int n, bool b)
    {
        N = n;
        InStock = b;
    }
}
class Join
{
    static void Main()
    {
        Item[] items = {
        new Item("Toyota", 4444), new Item("Hammer", 7777), new Item("BMW", 8888)
        };
        Status[] statusList = {
        new Status(4444, true), new Status(7777, false), new Status(5555, true)
        };
        var v = from n in items
                join m in statusList
                on n.N equals m.N
         select new { NN = n.Name, I = m.InStock };
         foreach (var t in v)
            Console.Write(t.NN+" "+ t.I+" ");
    }
}
```

a) Toyota True Hammer False

b) Toyota 4444 Hammer 777

c) Toyota 4444 Hammer 777 BMW 8888

d) Toyota True BMW true

e) ոչ, կտա թարգմանման սխալ

……