

C#sharp1 (basics + OOP)

Last Update - 2022.12.21

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 (Թեստ քննությունը “մինուս բոնուս” համակարգով -> 25 հարց 50 միավոր; 70 րոպե)

Ալգորիթմական C#sharp (Algorithmic C#)

1. Ծրագիր և Համակարգիչ [\[albdarb\]](#)
2. Թարգմանիչներ: Console Application – (թվային մուտք) [\[albdarb\]](#)
3. Ներդրված տիպեր [\[Шилдт-68; Нейгел-78; Троелсен-110; Troelsen-64\]](#)
Native-Sized [nint, nuint](#) [\[Albahari – 246; Price – 283\]](#) **N**
Երկուական համակարգ [\[albdarb\]](#)
4. Օպերացիաներ և Օպերատորներ [\[Шилдт-97\]](#)
Վերագրման օպերացիա [\[Шилдт- 106\]](#)
Թվաբանական օպերացիաներ (+, -, *, /, %) [\[Шилдт- 97\]](#)
Մեծացման և փոքրացման օպերացիաներ, (++, --) [\[Шилдт- 98\]](#)
Համեմատման օպերացիաներ (<, >, ==, !=, <=, >=) [\[Шилдт-101; Троелсен-142; Troelsen-98\]](#)
Տրամաբանական օպերացիաներ (&&, ||, !) [\[Шилдт-101; Троелсен-143; Troelsen-102\]](#)
Տիպերի վերափոխարկման օպերատոր () [\[Шилдт-89\]](#)
5. Պայմանական օպերատորներ [if, if...else](#) [\[Шилдт-121; Троелсен-142; Troelsen-97\]](#)
(? - ternary) [\[Шилдт-117; Троелсен-143; Troelsen-100\]](#)
6. [switch](#) օպերատոր [\[Шилдт-125; Троелсен-144; Troelsen-102\]](#)
7. Ցիկլիկ օպերատորներ - [while, do... while, for](#) [\[Шилдт-129; Троелсен- 141,139; Troelsen-94\]](#)
8. Անցումային օպերատորներ – [break, continue, goto](#) [\[Шилдт-139\]](#)
9. Բիթային օպերացիաներ (|, &, ~, ^, <<, >>) [\[Шилдт-107\]](#)
10. C/C# պարզ տարբերություններ [\[Шилдт-97, 121\]](#)
Using Digit Separators [\[Troelsen-74\]](#) **N**
11. Կոտրակային թվեր [\[Прайс – 81; Albahari – 48\]](#)
System.Numerics անունի տարածք [\[Troelsen-72\]](#) **N**
12. Զանգված: Պրակտիկ օրինակներ [\[Троелсен-150; Troelsen-111; albdarb\]](#)

Ֆունկցիոնալ ծրագրավորում (Functional programming)

13. Ծրագրավորման լեզուների զարգացման էտապները [\[albdarb\]](#)
14. Մեթոդներ [\[Шилдт-155; Troelsen-120\]](#)
մեթոդի պարամետրեր, արգումենտներ [\[Шилдт-162\]](#)
մեթոդի վերադարձվող արժեք, return օպերատոր [\[Шилдт-158\]](#)
լոկալ մեթոդներ [\[Solis – 90; Троелсен-169; Troelsen-121\]](#)
պրակտիկ օրինակներ [\[albdarb\]](#)
15. Գերբեռնված մեթոդներ [\[Шилдт-235; Троелсен-168; Troelsen-133\]](#)
16. Լոկալ փոփոխականներ, բլոկ { }, մեթոդ և ստեկ [\[albdarb; Solis - 124\]](#)
17. Ռեկուրսիա [\[Шилдт-257\]](#)

C#sharp հիմունքներ (C # Basics)

18. Կլասներ, Օբյեկտներ [Шилдт-147; Троелсен-194; Troelsen-171]
Օբյեկտի ստեղծում և մեթոդի կանչ [albdarb]
19. Թույլատրության մոդիֆիկատորներ [Троелсен-219; Troelsen-196 Шилдт-210]
Պարփակվածություն [Троелсен-215; Troelsen-192; Шилдт-210]
20. Կոնստրուկտոր, լռելիությամբ կոնստրուկտոր [Шилдт-166; Троелсен-197; Troelsen-174]
Պարամետրով և գերբեռնված կոնստրուկտորներ [albdarb]
21. Ինիցիալիզատոր, կոմպոզիցիա, ագրեգացիա [albdarb; Троелсен- 236; Troelsen-217]
22. Անունի տարածք, Պսևդոանուն [Шилдт-513; Троелсен-510; Troelsen-597; Нейгел-92]
global:: [Шилдт-524]
23. Չափային և հղիչային տիպեր [Нейгел-371; Троелсен-178; Troelsen-147; Рихтер-150]
24. Boxing և Unboxing [Рихтер-156; Шилдт-370; Нейгел-217]
new չափային տիպերի համար [Троелсен-113; Troelsen-67]
25. null արժեք, ? և ?? սիմվոլներ, ?. (հարցական կետ) [Шилдт-695; Нейгел-211; Троелсен-186; Troelsen-155]
26. Ստրուկտուրաների կազմակերպումը [Нейгел-130; Шилдт-391; Троелсен-175; Troelsen-142]
System.ValueType [Troelsen-148]
Պրակտիկ օրինակ. Լոկալ տվյալների կուտակիչ [albdarb]
27. Կոպիայի կոնստրուկտոր [Liberty-66; Либерти-97; albdarb]
28. Կլասի ստատիկ անդամներ [Троелсен-207; Troelsen-184; Шилдт-260; Solis - 135]
Ստատիկ անդամներ և հիշողություն [albdarb]
static լոկալ մեթոդներ [Troelsen-122] N
29. Ստատիկ կոնստրուկտոր: Ստատիկ կլաս [Троелсен-211,213; Troelsen-188, 190; Нейгел-126; Шилдт-265]
30. Singleton պատերն [Нэш-96; Шевчук -83; GOF-157]
31. this հղում [Троелсен-201; Troelsen-178; Шилдт -174]
32. this որպես մեթոդի արգումենտ [Liberty-66; Либерти-98; albdarb]
33. Կոնստրուկտոր և this [Троелсен-202; Troelsen-179; Нейгел-128; Шилдт-245]
34. Զանգված [Шилдт -139; Троелсен-150; Troelsen-111; Нейгел- 184; Нэш-248; Рихтер-416]
Անհամաչափ զանգված [Троелсен-153; Troelsen-114; Нейгел- 188; Нэш-258]
35. foreach ցիկլ [Троелсен-140; Troelsen – 95; Нейгел-197]
System.Range, System.Index [Troelsen – 118; Olsson-100; Albahari-13] N
36. System.Array կլաս [Троелсен-155; Troelsen-117]
37. const հաստատուն, readonly հաստատուն [Рихтер-210; Троелсен-237; Troelsen-218; Нейгел-129; Нэш-63]
Read-Only Structs, Read-Only Members [Troelsen - 145] N
38. enum (թվարկում) հաստատուն [Шилдт -397; Троелсен-170; Troelsen-135; Нейгел-90]
System.Enum կլաս [Троелсен-173; Troelsen-138]
39. string տիպ [Нейгел-266; Троелсен-120; Troelsen – 75]
String.Format() մեթոդ [Шилдт – 812; Нейгел-215; Covaci – 150]
DateTime և TimeSpan [Troelsen - 72]
40. System.Text.StringBuilder կլաս [Троелсен-127; Troelsen – 83]
41. Main(string [] args) մեթոդ [Троелсен-100; Troelsen – 51; Шилдт-254; Нейгел-95]
Command-Line արգումենտներ [Troelsen-55]
Application Error Code սպեցիֆիկացիա [Troelsen-53] N
Top-Level հրամաններ [Troelsen - 52] N
42. Հատկանիշ [Шилдт-313; Троелсен-224; Troelsen – 201; Рихтер-263]
Ռեֆակտորինգ [albdarb]
43. Ավտոմատ հատկանիշներ [Троелсен-230; Troelsen – 209; Нэш -69; Шилдт-318; Нейгел-124]
init [Albahari-108; Troelsen-215] N
44. Օպերատորների գերբեռնում [Шилдт-269; Троелсен-422; Troelsen-417; Нейгел-219]
45. Ինդեքսատոր [Троелсен-417; Troelsen-411; Шилдт-303]
Առանց կուտակիչ ինդեքսատոր [Шилдт-310]
46. Գերբեռնված ինդեքսատոր [Шилдт-307; Троелсен-420; Troelsen-415]
Բազմաչափ ինդեքսատոր [Троелсен-421; Troelsen-415; Шилдт-310]
47. Մեթոդի հղող պարամետրեր - ref, out [Шилдт-222; Троелсен-159,161; Troelsen-125,127; Нэш-63]

- հղող պարամետրեր [in](#) [Troelsen-128] N
- [ref](#) լոկալ փոփոխական [Albahari-67] N
- [ref](#) - ? ternary օպերատոր [Troelsen-101] N
- [ref](#) Structs [Troelsen-146] N
- 48. [ref](#), [out](#) հղիչային տիպերի համար [Шилдт-227; Троелсен-183]
- 49. Մեթոդի պարամետր – [params](#) [Троелсен-163; Troelsen – 229; Шилдт-229]
- 50. Մեթոդի անվանական արգումենտ [Шилдт-247; Троелсен-165; Troelsen – 230; Нейгел-121]
- Մեթոդի ոչ պարտադիր պարամետր [Шилдт-252]
- 51. [partial](#) կլասներ, [partial](#) մեթոդներ [Нэш-91; Нейгел-133; Троелсен-240; Troelsen – 221]
- 52. [var](#) -ոչ հստակ տիպայնացում [Троелсен-135; Troelsen-89; Шилдт-663; Нейгел-72]
- Անանուն տիպեր [Троелсен-438; Troelsen – 436; Нейгел-130]
- 53. Tuples [Троелсен-189; Troelsen-161; Albahari – 207; Гриффитс; Nagel]
- Deconstructing Tuples [Albahari – 102] N
- Tuple Equality/Inequality [Troelsen-163] N
- [switch](#) expression with Tuples [Troelsen-109] N
- 54. [record](#) [Troelsen – 222; Albahari – 214] N
- [record struct](#) [Price – 243; Albahari-212; Троелсен-241] N
- 55. Ընդլայնված մեթոդներ [Троелсен-434; Troelsen – 429; Шилдт -678; Нэш-477]
- 56. Կանոնավոր արտահայտություններ [Албахари –963; Albahari – 987; Нейгел-277]

Օբյեկտ կողմնորոշվածություն (Object Oriented)

- 57. Ժառանգականություն [Троелсен-222; Troelsen-228; Шилдт-329]
- [protected](#) [Троелсен-251; Troelsen-238; Шилдт-336]
- Կոմպոզիցիա “has a”(Containment/Delegation) [Troelsen-243] N
- 58. Ժառանգում և [base](#), [new](#) [Шилдт-343; Нейгел-144; Троелсен-266,249; Troelsen-236; 255]
- 59. Բազային կոնստրուկտորի պարամետրերի փոխանցում [Шилдт-339; Троелсен-249; Troelsen-236; Нейгел-149]
- 60. Տիպերի վերափոխարկում [Троелсен-130,428; Troelsen-84,423; Нейгел-228]
- Կոնվերտացիաներ [Covaci-122]
- 61. Հղում բազային կլասով [Шилдт-351; Троелсен-428; Troelsen-257; Нейгел-233]
- Ժառանգական կապվածությամբ տիպերի վերափոխարկում [Troelsen-257; 424]
- 62. Վիրտուալ մեթոդներ, [virtual](#), [override](#) [Шилдт-355, Троелсен-256; Troelsen-247]
- [virtual](#) և կոմպիլանտ [return](#) տիպ [Albahari - 119] N
- Պոլիմորֆիզմ [Шилдт-43; Троелсен-217; Troelsen-246]
- [Non-Virtual Interface](#) (NVI) պատերն (պոլիմորֆ) [Нэш -423]
- 63. Ժառանգականություն և [is](#) / [as](#) պատկանելիություն [Troelsen-259,260] N
- Discards և Pattern Matching [Troelsen-262] N
- 64. Տիպերի վերափոխարկում [implicit](#), [explicit](#) [Троелсен-429; Troelsen-427; Нейгел-230]
- 65. Մեկուսացված կլասներ և մեթոդներ, [sealed](#) [Троелсен-245,259; Troelsen-232; 249; Шилдт-367]
- 66. Ներդրված կլասներ [Троелсен-254; Troelsen-244; Liberty-101; Либерти-141]
- 67. Աբստրակտ կլասներ [Троелсен-260; Troelsen-250; Шилдт-363]
- Աբստրակտ պոլիմորֆիզմ [Troelsen-252]
- 68. [Template Method](#) պատերն [Фримен-310; Шевчук -270; GOF-373]
- 69. [Visitor](#) պատերն [Шевчук – 276; Нэш -498; GOF-379]

Օբյեկտ տիպ (Object type)

- 70. [Object](#) կլաս և նրա անդամները [Троелсен-273; Troelsen – 263; Нэш-114; Нейгел-135]
- 71. [Object](#) կլասի անդամների վերավորոշում
- ToString() մեթոդի վերավորոշում [Троелсен-276; Troelsen – 266; Нейгел-137]
- Equals() մեթոդի վերաորոշում [Троелсен-276; Troelsen – 266; Нэш-445; Рихтер-172]
- GetHashCode() մեթոդի վերաորոշում [Троелсен-277; Troelsen – 267; Нэш-445; Рихтер-172]
- 72. [Object](#) կլասի [static](#) անդամները [Троелсен-279; Troelsen – 269; Нейгел-218]
- 73. [Object](#).MemberwiseClone [Нэш-115,427]
- 74. [Prototype](#) պատերն [Шевчук – 72; GOF-146]

75. **dynamic** տիպ [Троелсен-600; Troelsen-668; Нэш-557; Шилдт-703; Нейгел-359]

object որպես ընդհանուր տիպ [Шилдт -372]

Performing **switch** Statement Pattern Matching [Troelsen-105] N

76. **IronPython** (Պրակտիկ) [albdarb]

Ինտերֆեյս (Interface) [Object oriented]

77. Ինտերֆեյս [Шилдт-375; Троелсен-306; Troelsen – 297]

78. Ինտերֆեյսի հղում [Шилдт -381; Троелсен-313]

Interface-Default Implementations [Troelsen-306] N

79. Ինտերֆեյսի ստատիկ անդամներ [Troelsen-308] N

80. Ինտերֆեյսի **is** և **as** պատկանելիության ձևեր [Троелсен-314; Troelsen-305]

81. Ինտերֆեյսը մեթոդի պարամետր և վերադարձվող արժեք [Троелсен-315; Troelsen-308, 310]

82. Ինտերֆեյսների ժառանգականություն, բազմաժառանգում [Шилдт -387; Троелсен-322; Troelsen-317; 320]

83. Ինտերֆեյսների բացահայտ իրականացում [Троелсен-320; Troelsen-314; Шилдт-388]

84. Ինտերֆեյս **ICloneable** [Троелсен-331; Troelsen-329; Шилдт-779]

85. Ինտերֆեյս **IEnumerable** [Троелсен-326; Troelsen-322; Шилдт – 1001]

86. Ինտերֆեյս **IEnumerator** [Троелсен-326; Troelsen-322; Шилдт – 1001]

foreach և **Ienumerator** [Нейгел-197]

87. Ինտերատորներ, օպերատոր **yield** [Троелсен-328; Troelsen-325; Шилдт-1003]

88. Անվանական ինտերատորներ [Троелсен-330; Troelsen-327; Шилдт-1007]

89. Ինտերֆեյս **IComparable** [Троелсен-335; Troelsen-334; Шилдт-778]

Դելեգատ (Delegate) [Runtime]

90. Դելեգատներ [Шилдт-473; Троелсен-379; Troelsen-452; Нейгел-241]

Խմբային դելեգատներ [Шилдт-476; Троелсен-389; Troelsen-462; Нейгел-251]

91. **Delegate** և **MulticastDelegate** բազային կլասներ [Троелсен-383; Troelsen-454; Нэш - 288]

92. Անանուն մեթոդներ [Шилдт-484; Нэш-314; Троелсен-406; Troelsen-478; Нейгел-254]

93. Lambda արտահայտություն [Шилдт-488; Нэш-503; Нейгел-255; Троелсен-408; Troelsen-482]

Single-line method [Troelsen-415; Troelsen-490] N

Natural types for lambdas N

Deconstructing Tuples with "Single-line" lambda [Troelsen-165] N

switch Expressions [Troelsen-108] N

94. Covariance և Contravariance դելեգատներ [Шилдт-481]

95. **Strategy** պատերն [Нэш-306; Шевчук - 265; Фримен-37; GOF-362]

Իրադարձություն (Event) [Runtime]

96. Իրադարձություն - **event** [Троелсен-396; Troelsen-467; Шилдт-494]

Իրադարձության իրականացում լրամբնարտահայտությամբ և անանուն մեթոդով [Шилдт -504]

97. Multicast **event** [Шилдт-496]

Իրադարձության **add** և **remove** արքեստրներ [Шилдт-500; Гриффитс-510]

98. BCL գրադարանի իրադարձություններ [Шилдт-506; Троелсен-403; Troelsen-476; Liberty-249; Либерти-321]

EventHandler դելեգատ [Шилдт-508; Троелсен-405; Troelsen-478]

99. **Observer** պատերն [Шевчук –228 ; Фримен -71; Sarcar-269; GOF-339]

IObservable<T> և **IObserver<T>** ինտերֆեյսներ [Шилдт - 781]

Բացառիկ իրավիճակ (Exception) [Runtime]

100. Բացառիկ իրավիճակներ [Шилдт-403; Троелсен-280; Troelsen-271; Нейгел-419]

101. **Exception** կլաս [Шилдт-418; Троелсен-282,288; Troelsen-273; 279; Нейгел-427]

102. Արհեստական բացառիկ իրավիճակ, **throw** [Шилдт-414; Троелсен-285; Troelsen-277]

throw as Expression - Anywhere [Troelsen-280] N

103. Բացառիկ իրավիճակի Rethrowing [Шилдт-415; Троелсен-300; Troelsen-292]

104. Բազմակի **catch** – ի օգտագործում [Троелсен-297; Troelsen-289; Нейгел-423]

105. Custom բացառիկ իրավիճակներ [Шилдт-422; Троелсен-293; Troelsen-285]

106.checked, unchecked [Шилдт -428 ; Троелсен-133; Troelsen-86; Нейгел-209]

Ընդհանրացում (Generic) [Type]

107. Կլասների ընդհանրացումներ [Нэш-309; Шилдт-575; Нейгел-161; Рихтер-302; Троелсен-372; Troelsen-403]

108. Ընդհանրացում և where սահմանափակում [Шилдт-585; Троелсен-374; Troelsen-406]

109. Generic - մերողներ [Шилдт-607; Троелсен-369; Troelsen-400; Нэш-316]

Կոմպարյանտություն [albdarb]

110. Generic - դեկլատներ [Нэш-318; Шилдт-610; Троелсен-393; Troelsen-466]

111. Action<> և Func<> դեկլատներ [Троелсен-394; Troelsen-467; Нейгел- 247; Covaci - 216]

112. Generic – ինտեֆեյսներ [Шилдт -612; Нейгел -171]

Covariance և Contravariance [Шилдт -626]

113. Կլասի հետաձգված ինիցիալիզացիա - Lazy<> [Троелсен-504; Troelsen-364; Гриффитс-873]

albdarb.com

user: books psw: “sharp” (ռուսերեն գրականություն)

user: books psw: “sharpeng” (անգլերեն գրականություն)

user: books psw: “pattern”

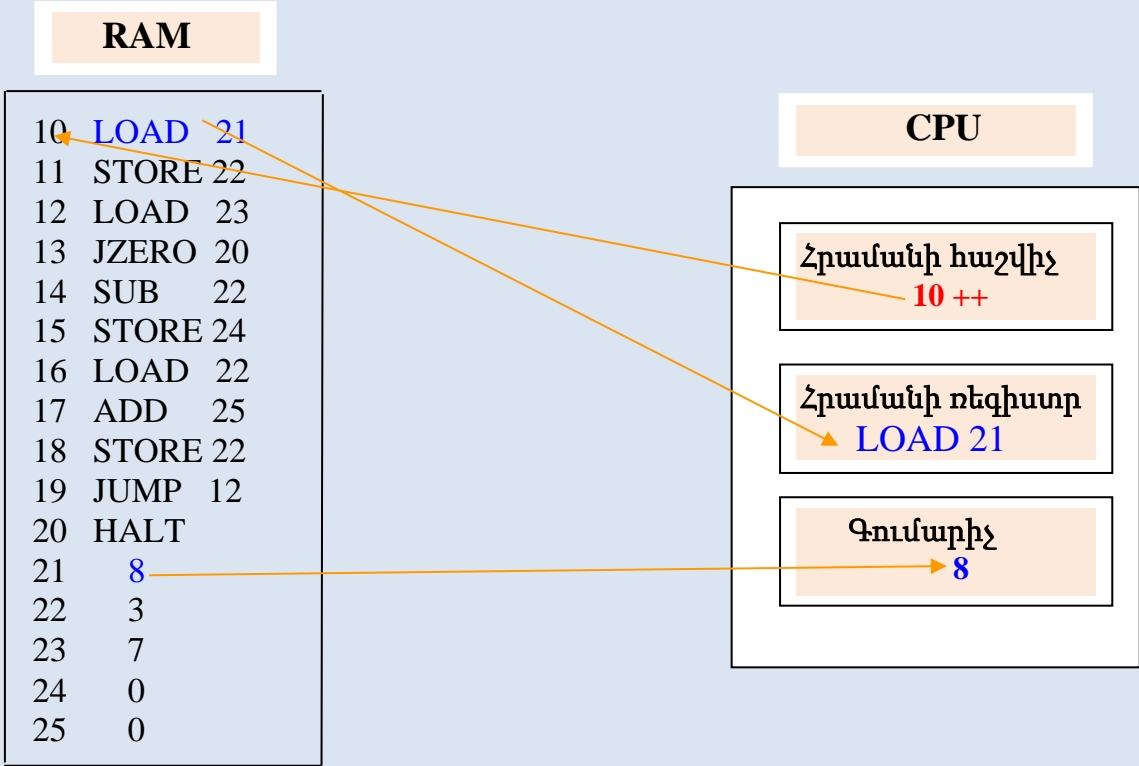
ԳՐԱԿԱՆՈՒԹՅՈՒՆ

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, Phillip Japikse – “Pro C# 9 with .NET 5”, 2021.
4. Andrew Troelsen, Phil Japikse – “Pro C# 10 with .NET 6”, 2022.
5. Нейгел К...- “C# 4.0 и платформа .NET 4 для профессионалов”, 2011.
6. Christian Nagel – “Professional C# And . NET”, 2021
7. Нэш Трей – “C# 2010. Ускоренный курс для профессионалов”, 2011.
8. Рихтер Дж. - “CLR via C#. Программирование на платформе NET Framework 4.5 на языке C#”, 2013.
9. Гриффитс Йен – “Программирование на C# 8.0”, 2021.
10. Ian Griffiths - Programming C#10 Build Cloud, Web, and Desktop Applications, 2022
11. Албахари Джозеф, Албахари Бен – “C# 7.0. Справочник. Полное описание языка”, 2018.
12. Joseph Albahari - C# 10 in a Nutshell, 2022.
13. Марк Дж. Прайс – “C#8 и NET Core, Разработка и оптимизация”, 2021.
14. Mark J. Price - C#11 and .NET 7, Modern Cross-Platform Development Fundamentals, 2022.
15. Шевчук А ... - “ Design Patterns via Csharp”, 2015.
16. GOF - Гамма Э.... “ Паттерны объектно-ориентированного проектирования”, 2020.
17. Фримен Эрик...- “Паттерны проектирования”, 2021.
18. Vaskaran Sarcar – “Design Patterns in C#”, 2020.
19. Daniel Solis – “Illustrated C# 7”, 2018.
20. Mikael Olsson – “C# 8 Quick Syntax Reference”, 2019.
21. Tiberiu Covaci...”MCSD Certification Toolkit (Exam 70-483): Programming in C#”, 2013.
22. Jesse Liberty – “Programming C#”, 2001.

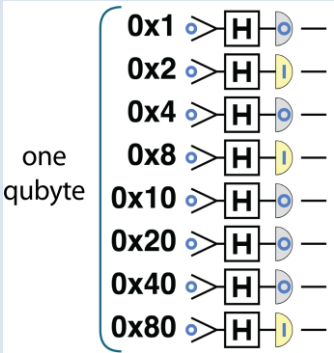
Ալգորիթմական C#

1. Ծրագիր և Համակարգիչ [albdarb]

- Ներկայացնենք մոդել, որը հիմք կդառնա հետագա ծրագրավորման հիմունքները կառուցելու համար: Այլ խոսքով դա կլինի ծրագրավորման մոդելի “աքսիոմատիկան”:
- Համակարգիչների կառուցվածքը տվել է Ֆոն Նեյմանը դեռ 40 ական թվականներին: Նա նշել է, որ ապագա համակարգիչը պետք է ունենա ղեկավարման մաս (այժմ CPU), հիշողություն (այժմ RAM), մուտք/ելքի մաս: Բացի դրանից, այն պետք է լինի *էլեկտրոնային, հաշվի երկուական համակարգով և հրամանները կատարի հաջորդաբար*:



- Համակարգիչը աշխատում է 2-ական (բինար) համակարգով: Չմոռանանք, որ մարդը 10 –ական համակարգն է ընտրել իր ձեռքերի մատերի քանակի պատճառով և ավելացնեմ, որ մարդկությունը մինչև ագրարային հեղափոխությունը (10 000 տարի մ.թ.ա.) հաշվում էր 1 –ական համակարգով, “ունար” համակարգ (ասենք որքան որս է արել այդքան էլ նշան է պահում) [albdarb]: Մոտակա ապագայում համակարգչի բինար համակարգը կփոխարինվի Qubit համակարգով (QPU), որը համարվում է *քվանտային* համակարգիչների հիմքը: Նշենք մեկ բայթի պատահական Qubit ներկայացումը՝ (կստացվի 256 զուգահեռ տարբերակներ)



- Բինար համակարգում ծրագրի հրամանները կատարվում են պրոցեսորի միջոցով, հիմնականում հաջորդաբար: Նրանք պետք է գտնվեն օպերատիվ հիշողությունում: Ծրագրի *հրամանները* և *տվյալները* ստանում են հասցեներ և սկզբից ծրագրի կատարման առաջին հրամանի հասցեն տեղադրվում է հրամանի հաշվիչում: Հաջորդ քայլում հաշվիչի պարունակության հասցեի համապատասխան հրամանը բերվում է հրամանի ռեգիստր: Սկսվում է կատարվել ռեգիստրում

գտնվող հրամանը, միաժամանակ կատարվում է շատ կարևոր գործողություն, **հրամանի հաշվիչի պարունակությունը ավելանում է մեկով**, որը նշում է հաջորդ հրամանի հասցեն: Այս ձևով ապահովվում է ծրագրի ավտոմատ կատարումը: Ավելացնենք, որ համակարգիչը գլխավորապես կատարում է միայն գումարում եւ ցանկացած գործողության կատարումը կազմակերպվում է գումարիչի միջոցով: Օրինակ, եթե անհրաժեշտ է 22 հացեի պարունակության 3 թիվը տանել 23 հասցե, ապա այն սկզբից բերվում է գումարիչ, հետ նոր միայն այնտեղից 23 հասցե: Կամ, օրինակ, եթե անհրաժեշտ է հանում կատարել, հանվող թիվը վերածվում է լրացուցիչ կոդի եւ գումարման գործողությունով ստացվում է հանում:

- Ծրագրավորման տեսանկյունից կարելի է ասել, որ ծրագիրը բաղկացած է հրամաններից (ADD, LOAD, HALT), տվյալներից (8, 3, 7, 0, 0) և հիշողության հասցեներից (10, 11, 12, 13 ...):
- **Մինիմալ ինֆորմացիոն միավորը**, որը կարելի է գրել կամ կարդալ հիշողությունից, դա 8 բիթն է: Այդ պատճառով, եթե անհրաժեշտ է նույնիսկ գրել 1 բիթանոց տվյալ, միննույնն է պետք է զբաղեցնել 8 բիթ կամ մեկ բայթ տարածք:

2. Թարգմանիչներ: Console Application – (թվային մուտք) [albdarb]

- Ցանկացած խնդիր ծրագրավորելու համար անհրաժեշտ է թարգմանիչ: Թարգմանիչը որևէ լեզվով գրած ծրագիրը թարգմանում է երկուական կոդի, որպեսզի համակարգիչը կարողանա այն կատարել: Ինչպես թարգմանիչն է հիշողություն զբաղեցնում, այնպես էլ որևէ լեզվով գրած ծրագիրը եւ նրա թարգմանված 2- ական կոդը նույնպես պահանջում են հիշողություն:
- Թարգմանիչները լինում են երկու տեսակ: Ինտերպրետատոր (նախկին Basic լեզուն, ժամանակակից բրաուզերները – Google Chrom, Microsoft Intrnet Explorer, Edge) կամ կոմպիլիատոր(C, Pascal, C++): Ինտերպրետատորի ժամանակ ծրագիրը թարգմանվում է մաս մաս ու տեղում կատարվում: Ծրագրի ամեն նոր կանչի ժամանակ կատարվում է նոր թարգմանում: Կոմպիլիատորի ժամանակ ծրագիրը թարգմանվում է ամբողջությամբ միայն մեկ անգամ և կատարվում թարգմանված ծրագրի բազմակի օգտագործում: Ինտերպրետատորը աշխատում է **դանդաղ**, սակայն օգնում սխալները շուտ հայտնաբերել: Այն հարմարվում է տեխնիկական միջոցներին: Կոմպիլիատորը ավելի պրոֆեսիոնալ մոտեցում է պահանջում եւ նրա թարգմանած ծրագիրը աշխատում է **ավելի արագ**:
- C#sharp լեզուն կարելի է ասել **ավելի շատ** կոմպիլիատոր է քան ինտերպրետատոր: Ավելի շուտ, C# դա dotNET համակարգի լեզու է: Այն աշխատում է նոր սկզբունքով: Այդ սկզբունքը կոչվում է **դեկավարվող կոդ, որը կանցնենք ավելի ուշ**:
- Console Application -ի օրինակ. (անդուր սև սպիտակ մուտք/ելքի տեսքով)

```
using System;
    Console.WriteLine("barev bolorin"); // էլք
    int a = 999; // փոփոխականի հայտարարում
    int b = 222;
    int c = a + b;
    Console.WriteLine("The sum of " + a + " and " + b + " = " + c); // էլքի C++ տարբերակ
    Console.WriteLine("The sum of {0} and {1} = {2}", a, b, c); // էլքի C տարբերակ
    Console.WriteLine($"The sum of {a} and {b} = {c}"); // էլքի C#7 տարբերակ
    string aa = Console.ReadLine(); // մուտք
    a = Convert.ToInt32(aa); // կոնվերտացիա սիմվոլային տողից - թիվ
    //a=Int32.Parse(aa); // կոնվերտացիայի այլ տարբերակ
    b = Convert.ToInt32(Console.ReadLine()); // մուտք
    //b=Int32.Parse( Console.ReadLine() ); // մուտք
    c = a + b;
    Console.WriteLine(c); // էլք
    Console.ReadKey(); // ծրագրի ՀԱՐՄԱՐ սպասում արդյունքը դիտելու համար
```

3. Ներդրված տիպեր

[[Heйpeл-78](#); [Шилдт-68](#); [Троелсен-110](#); [Troelsen-64](#)]

- Ծրագրավորումը, դա լուծման նպատակով խնդրի ալգորիթմային ներկայացումն է, որը կատարվում է համակարգչի միջոցով: Խնդրի լուծման համար անհրաժեշտ է տվյալների մշակում, որը իր հերթին բերում է տվյալների պահպանման (հիշման) անհրաժեշտության: Տվյալների պահումը եւ մշակումը կատարվում է **տիպերի** միջոցով: Ծրագրավորման լեզուները ունեն մեծ քանակով տիպեր, որոնք թույլ են տալիս տվյալները օգտագործել օպտիմալ ձևով:
- C# լեզուն խիստ տիպային լեզու է: Տիպային լեզուների առավելություններից է համարվում հիշողության **օպտիմալ օգտագործումը**: Մյուս առավելություններից է կոդի **սխալների** արագ հայտնաբերումը, որը լավ կլինի կատարվի թարգմանության ժամանակ: Ժամանակակից լեզուներում երկրորդ չափանիշը ավելի առաջնային է համարվում եւ կոչվում է կոդի **հսկման** բարձրացում: Ներկայացնենք C# լեզվի ներդրված տիպերը (built-in types):

byte	8bit	0-255	
sbyte	8bit	-128 to 127	
short	16bit	-32768 to 32767	
ushort	16bit	0 to 65535	
int	32bit	-2147483648 to 2147483647	
uint	32bit	0 to 4294967295	
long	64bit	
ulong	64bit	0 to 18446744073709551615	
float	32bit	10 ³⁸ - ճշտությունը 7-8 կարգի
double	64bit	10 ³⁰⁸ - ճշտությունը 15-16 կարգի
decimal	128bit	10 ²⁸ - ճշտությունը 28-29 կարգի
char	16bit	(unicode) - 2 ¹⁶ = 65000	սիմվոլներ
bool	8bit	ընդունում է միայն (true or false)	արժեքներ

//-----	
string	աշխատանք տողերի հետ
var	ոչ հստակ տիպ, որը որոշվում է արժեքավորման ժամանակ
object	ընդհանուր տիպայնացում թարգմանության ժամանակ
dynamic	ընդհանուր տիպայնացում կատարման ժամանակ

- **// Տնային: Հարց: float: Ինչպես կարելի է 32 բիթով ստանալ 10³⁸ թիվ:**
- Պարզվում է, որ գոյություն ունեն նաև մեծ քանակով (տասնյակ հազարավոր) այլ տիպեր, որոնք չեն համարվում առաջին անհրաժեշտության տիպեր: Նրանք չեն ստանում կապույտ գույն եւ շատ ավելի դանդաղագործ են: Պարզ է, որ ներդրված տիպերը ավելի արագագործ են, ուղղակի բոլորին հնարավոր չէ ընդգրկել ներդրված մասում (աշխատում է էնեգիայի պահպանման օրենքը՝ մի բան պիտի զիճես, որպեսզի մի այլ բան ստանաս):
- **byte, sbyte, short, ushort, int, uint, long, ulong** տիպերը աշխատում են ամբողջ թվերի հետ: Կա պայմանավորվածություն: Տիպին հատկացվող բիթային տարածքի ամենաձախ բիթը համարվում է թվի նշանի բիթ եւ եթե այն զրո է, ապա թիվը դրական է: Եթե այն մեկ է, ապա թիվը բացասական է: **sbyte, short, int, long** թույլատրում են աշխատել նաև բացասական թվերի հետ: **byte, ushort, uint, ulong** տիպերը աշխատում են միայն դրական թվերի հետ:
- **float, double, decimal** տիպերը աշխատում են կոտորակային թվերի հետ: Նրանք պարունակում են շատ մեծ թվեր: Մեծ թվերի հետ աշխատանքը կազմակերպվում է հետևյալ հնարքով, - թիվը կարելի է ներկայացնել երկու բաղկացուցիչով: Առաջինը կոչվում է կոտորակային մաս: Երկրորդը կոչվում է “կարգայնություն” եւ ներկայացվում է 10 ի աստիճաններով: Օրինակ.
56789= 0,56789 * 10⁵:
Թվերի այսպիսի ներկայացման դեպքում կարելի է շատ փոքր կորուստով աշխատել մեծ թվերի հետ, կատարելով մաթեմատիկական գործողություններ:

- Վերցնենք **float** տիպը: Այն ունի 32 բիթ հիշողության տարածք (նույնը ունի նաև **int** տիպը, սակայն այն 4 միլիարդից ավել չի կարող պահել): 10^{38} աստիճան ստանալու համար 26 բիթ հատկացվում է “մանտիսին” և 6 բիթ կարգայնությանը: 6 բիթ ուղղակի, որպեսզի 38 թիվը 2 ականով հնարավոր լինի ներկայանել:
- **decimal** տիպը նորույթ է: Նրա բիթային տարածքը շատ մեծ է (128բիթ), սակայն բիթերի մեծ մասը հատկացվում է մանտիսին “սուպեր” ճշտություն ապահովելու համար (երևի որոշ բնագավառներում անհրաժեշտ է):
 - Ըստ մեծության կոտորակային թվերը դասավորվում են **decimal**, **float**, **double** հերթականությամբ:


```
double doub = 1.0E308; //փոքրձեղ
float fl=1.0e38f;
decimal dec = 1.0E28m;
```
 - **char** տիպը նախատեսվում մեկ նիշ (սիմվոլ) պահելու համար: C# ում այն ունի 16 բիթ տարածք (նախկին լեզուներում 8 բիթ էր), որը հնարավորություն է տալիս հաղթահարել լեզվային սիմվոլների սահմանափակումները: Այլ խոսքով, առանց միջնորդ ծրագրերի (KDWin - մադելին), կարելի է աշխատել աշխարհի բոլոր լեզուների հետ: Այստեղ նույնպես ունենք հիշողության կորուստ, սակայն շահում ենք ավելի “կոնֆորտ” աշխատանքում:
 - **bool** տրամաբանական տիպ է: Այն ունի վերացականության հատկություն: Ինչպես գիտենք, համակարգիչը կառուցված է տրամաբանական էլեմենտներից: Նրանք են “և” “կամ” “ոչ”: Այլ խոսքով համակարգչի ԴՆԹ –ն կազմված է 3 էլեմենտից: Նշված էլեմենտներից կազմված են սխեմաները, որոնք կատարում են տրամաբանական աշխատանքներ: Ահա այս տրամաբանական մոտեցումը կիրառվում է նաև ծրագրավորման մեջ: Ծրագրի որևէ հատված մեզ կարող է ասել ինչ որ բան ճիշտ է, թե ոչ: Ճիշտ է գաղափարը պայմանավորվենք **true** անվանել, ճիշտ չէ գաղափարը պայմանավորվենք **false** անվանել, պարզ է համապատասխան տարածք տրամադրելով ծրագրի այս պահվածքին:
 - **string** և **object** տիպերը կանցնենք ավելի ուշ: Նշենք, որ **string** տիպը աշխատում է տողերի հետ, որոնք կարելի է համարել սիմվոլների խումբ: Իսկ **object** տիպը ամենատարողունակ տիպն է և կարող է հանդես գալ ցանկացած տիպի փոխարեն:
 - ❖ **default** հրամանը նշանակում է տիպի լռելիությամբ արժեք: [Troelsen-66]


```
int a = default; // C#7.1
Console.Write(default(string)+default(int)+default(char)+default(double)+
default(bool)); Վերադարձնում է (ոչինչ)0space0False:
```
 - `Console.WriteLine(sizeof(decimal));` //??Ոչ միայն չդեկլարվող կոդում: // Troelsen սխալ
 - ❖ **Native-Sized Integers** (C# 9 core) **nint**, **nuint** [Albahari – 246; Price – 283]
 - **nint**, **nuint** տիպերը ունեն հատուկ չափ, որը համապատասխանում է կատարման պրոցեսի կարգայնությանը և իր նշանակությունը օգտագործվում է **կատարման ժամանակ** (հայտարարման ժամանակ պահում է **int**): Այն կարող է պահպանել հազեցման (overflow) անվտանգությունը: Բարձրացնել էֆեկտիվությունը և **ցուցիչներով** թվաբանական օպերացիաների կատարումը (C#2):


```
Console.WriteLine($"int.MaxValue = {int.MaxValue:N0}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue:N0}");
nint x = 1234567890, y = 1234567890;
// nint y = 1234567890L; // error
// long y = 1234567890L; // ok
nint z= x * y;
Console.WriteLine(z);
```
 - հազեցման (overflow) անվտանգությունը (**կհ**)


```
nint x = 123456, y = 123456;
checked // անիմաստ է
{
    nint z2 = x * y;
    Console.WriteLine(z2);
}
```

4. Օպերացիաներ և օպերատորներ

[Шилдт-97]

- Պայմանավորվենք, որ համակարգչի տեսակետից **օպերացիաները** ավելի թույլ գործողություններ են կատարում, իսկ **օպերատորները** ավելի բարդ: Վեջին հաշվով նրանք նույնն են: Նշենք նաև, որ ցանկացած հրաման ավարտվում է կետ ստորակետով (;) :
- ❖ **Վերագրման օպերացիա** (=) [Шилдт- 106]
 - $c=a+b$; Վերագրման օպերացիան դա մաթեմատիկական նույնություն չէ: Այսինքն, եթե հավասարության աջ մասը փոփոխենք, ապա ձախ մասը համապատասխանորեն կփոփոխվի: Իսկ, եթե ձախ մասն ենք փոխում աջ մասը չի փոխվում: Մաթեմատիկայում այդպես չէ: Նշված երևույթը կապված է ծրագրավորման առանձնահատկություն հետ: Այն խիստ կախված է համակարգչի ներքին կառուցվածքից, հիշողության կազմակերպումից և ռեսուրսներից:
- ❖ **Թվաբանական օպերացիաներ** (+, -, *, /, %) [Шилдт- 97]
 - Ցանկացած թվային տիպի հետ կարելի է կատարել թվաբանական գործողություններ, որոնք նման են մաթեմատիկականին:
 - / - բաժանում գործողությունը ամբողջ տիպերի դեպքում (byte, short, int, long) պահում է միայն թվի ամբողջ մասը: % գործողությունը կատարում է բաժանում եւ պահում է մնացորդը: Նշենք որ, կոտարակային թվերի դեպքում / - բաժանում կատարվում է նորմալ և % գործողությունը նույնպես պահում է մնացորդը:

```
using System; // Գտնել եռանիշ թվի թվանշանների գումարը:
```

```
class Program
```

```
{
    static void Main(string[] args)
    {
        int a = 567;
        int a1, a2, a3;
        a1 = a / 100;
        a2 = a % 100 / 10;
        a3 = a % 10;
        Console.WriteLine(a1 + a2 + a3);
    }
}
```

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

❖ **Համեմատման օպերացիաներ** (<, >, ==, !=, <=, >=) [Шилдт-101; Троелсен-142; Troelsen-98]

❖ **Տրամաբանական օպերացիաներ** (&&, ||, !) [Шилдт-101; Троелсен-143; Troelsen-102]

- Ֆոն Նեյմանի առաջարկած համակարգչի կառուցվածքի հիման վրա, հրամանները կատարվում են հաջորդաբար, իսկ թարգմանիչում դա արտահայտվում է վերնից ներքև գաղափարով (այսպես ասած ազդում է “գրավիտացիան”): Հրամանների կատարման հաջորդականությունը ուղղեկցվում է պայմաններով և ցիկլերով: Պայմանի ստեղծման ժամանակ օգտագործվում է նոր օպերացիաներ, որոնք փոփոխականների միջև կատարում են ինչպես համեմատում (<, >, ==, !=, <=, >=), այնպես էլ տրամաբանական միացում կամ խմբավորում (&&, ||, !) “և” “կամ” “ոչ” համակարգով:
- Հրամանները ունեն կատարման առավելության հետևյալ հաջորդականությունը՝ սկզբից () կլոր փակագծեր, այնուհետև գալիս է տրամաբանական “և” (&&), վերջապես տրամաբանական “կամ” (||):
- ✓ Օրինակը ապացուցում է, որ && ավելի առաջնային է, քան || օպերացիան.

```
using System;
```

```
class Program
```

```
{
    static void Main(string[] args)
    {
        int a = 999;
        char c = 'A';
        bool b = (16 > 8) || (a > 6) && (c == 'B');
        Console.WriteLine(b); //True
    }
}
```

❖ **Մեծացման և փոքրացման օպերացիաներ** (++, --), *Պրեֆիքս և պոստֆիքս* [Шилдт- 98]

- ++ գործողությունը համակարգչում մեկով ավելանալու գաղափարն է առաջ տանում, որը նաև ապահովում է պրոցեսորի (CPU) ավտոմատ իրականացումը, ծրագրի հրամանների կատարման հաջորդականության տեսանկյունից: ++ գործողությունը կոչվում է “Increment”, որի հակառակ գործողությունը (փոփոխականի պարունակությունը մեկով պակասացնելը) կոչվում է “Decrement”:
- (++, --) գործողությունները կարելի է կիրառել ինչպես փոփոխականից առաջ, այնպես էլ հետո (prefix, postfix): Եթե (++, --) գործողությունները ուղղեկցվում են այլ հրամանների հետ, ապա այդ ժամանակ սկսում է դեր խաղալ (++, --) գործողությունների տեղը և որոշվում է, սկզբից կատարել այլ հրամանը հետո (++, --), թե հակառակը:
- a++; a=a+1; a+=1;
- ✓ Նշված երեք հարմանները կատարում են նույն գործողությունը, ավելացնում են a փոփոխականը մկով: Եթե անհրաժեշտ է ավելացնել փոփոխականի արժեքը այլ թվով, ապա օգտագործվում է երկրորդ կամ երրորդ տարբերակը: a=a+5; կամ a+=5;
- ✓ Օրինակում ծրագիրը կարտածի 5 6 6 7 հաջորդականությունը:

```
using System;
    int a = 4;
    a++;
    Console.WriteLine(a);
    Console.WriteLine(++a);
    Console.WriteLine(a++);
    Console.WriteLine(a);
```

❖ **Տիպերի վերափոխման օպերատոր** () [Шилдт-89]

- Տիպերի վերափոխումը, որը օգտագործում է () փակագծերը, կարելի է անվանել օպերատոր, այլ ոչ թե օպերացիա, իր կարևորության և բարդության հետ կապված: Տվյալները ստանալով տիպեր, ծրագրի կատարման ժամանակ անցնում են տարբեր թույլատրելի տարածքներով: Նշված տարածքները կարող են լինել տարբեր տիպեր: Անցումները տարածքներում կարող են լինել մեծից փոքր և փոքրից մեծ: Փոքրից մեծ անցումը բնական է կատարվում առանց որևէ միաջամտման, իսկ մեծից փոքրի դեպքում, հնարավոր է տեղի ունենա տվյալների կորուստ: Դրա համար, եթե ծրագրավորողը օգտագործում է կլոր փակագծեր () տիպերի վերափոխարկման օպերատորը, ապա թարգմանիչը շարունակում է աշխատանքը, չհամարելով այն սխալ:
- Նշենք, որ տիպերի վերափոխարկում կարելի է կատարել ոչ բոլոր տիպերի միջև (օրինակ bool ից int հնարավոր չէ), այլ կարելի է միայն միևնույն ընտանիքի տիպերի միջև (օրինակ բոլոր թվային տիպերի միջև - byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal):

```
using System;
    int a = 1;
    long b = 2;
    b = a; // ok
    //a = b; // error
    a = (int)b; //ok
    short b1 = 0;
    short b2 = 0;
    //short b3 = b1 + b2; // error **
    short b3 = (short)(b1 + b2); // ok
    double d = 3 / (float)2;
```

❖ **// error ****

- 8 և 16 բիթային ամբողջ տիպերը (byte, sbyte, short, ushort) չունեն իրենց համար նախատեսված թվաբանական օպերացիաները և այդ պատճառով նրանք ոչ ակնհայտ (implicit) ձևով վերածվում են ավելի մեծ տիպի: Արդյունքը չի կարող վերագրվել 8 կամ 16 բիթային տիպերով փոփոխականներին, եթե չկատարվի **տիպերի վերափոխարկում**: [Albahari - 46]

5. Պայմանական օպերատոր if, if....else, (? - ternary)

[Шилдг-121, 117; Троелсен-142,143; Troelsen-97]

- Պայմանական օպերատորը հնարավորություն է տալիս կատարել ծրագրի հրամանների կատարման հաջորդականության ճյուղավորում կախված պայմանից:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int a=9;
        char c = 'A';
        if(a==9 && c=='A')
            Console.WriteLine("barev");
        else
            Console.WriteLine("again barev");
    }
}
```

- if պայմանական օպերատորում թույլատրվում է օգտագործել վերագրման (=) օպերատոր: Անհրաժեշտ է օգտվել () կլոր փակագծերից:

```
int a;
if ((a = 8) > 0)
    Console.WriteLine("barev");
```

- if օպերատորը ունի առանձնահատկություն: Ձևավոր փակագծերը պարտադիր են, եթե բլոկը ունի միակ հրաման տիպի հայտարարումով: Հաջորդ օրինակը առանձնահատուկ դեպք է.

```
static void Main(string[] args)
{
    int a=9;
    if (a > 0)
        // { //ok
            int b=5; //error
        // } //ok
}
```

❖ Պայմանական օպերատոր (? - ternary operator) [Шилдг- 117; Troelsen-100]

- (?) պայմանական օպերատորը if... else օպերատորի նման կատարում է ծրագրում ճյուղավորման աշխատանք: Ինչպես նաև ունի առանձնահատկություն, օգտագործվել ֆունկցիաների (մեթոդների) պարամետրերում և առանձնահատուկ տեղերում:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int a = 5;
        int c = (a > 0) ? 44 : 99;
        Console.WriteLine(c);
        Console.WriteLine((a < 0) ? 55 : 77);
    }
}
```

.....

6. switch օպերատոր

[Шилдг-125; Троелсен-144; Troelsen-102]

- **switch** օպերատորը փոխարինում է **if... else** օպերատորին, եթե անհրաժեշտություն կա միաժամանակ ստուգել մեծ քանակով պայմաններ: Կարող է որպես պարամետր օգտագործել բոլոր ներդրված տիպերը **byte, int, long, double, char, bool, string** և այլն:
- C# -ում **switch** օպերատորի օգտագործումը հիմնականում նման է մյուս լեզուներին: Ստանում է կլոր փակագծերում () տվյալ, որը որպես պայման ճյուղավորվում է **case** –երի միջոցով:
- C# -ում **switch** օպերատորը **case** –ի և **default** –ի վերջում պարտադրվում է **break** հրամանը, եթե չի օգտագործվում **goto** թռիչքային անցման հրամանը:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int n;
        for (; ; )
        {
            Console.Write("tiv = ");
            n = Convert.ToInt32(Console.ReadLine());
            switch (n)
            {
                case 1: Console.WriteLine("11111");
                        // goto mm;
                        break;

                case 2:
                        Console.WriteLine("22222");
                        // mm: Console.WriteLine("22222");
                        break;

                case 3: Console.WriteLine("33333");
                        break;

                default: Console.WriteLine("Default");
                        break;
            }
        }
    }
}
```

- Օրինակը ցույց է տալիս, թե ինչպես կարելի է **case** -ում ոչինչ չգրել: Այս դեպքը թույլատրելի է և հնարավորություն է տալիս մի քանի պայմաններ կատարեն նույն գործողությունը:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.Write("tiv = ");
        int n = Convert.ToInt32(Console.ReadLine());
        switch (n)
        {
            case 1:
            case 2:
            case 3:
                Console.WriteLine("barev");
                break;
        }
    }
}
```


7. Ցիկլիկ օպերատորներ - while, do... while, for

[Шилдг-129; Троелсен-141,142; Troelsen-94]

- Ցիկլիկ օպերատորը հնարավորություն է տալիս ծրագրի որևէ հատված կրկնել և այդ հնարավորությունն է ստիպում համակարգչին լուծել բարդ խնդիրներ: (Առաջին ծրագրավորողը համարվում է Ադա Բայրոնը, ով և առաջարկել է ցիկլի գաղափարը):
- **while** օպերատորը ստուգում է պայմանը և կատարում ցիկլը:
- **do... while** օպերատորը սկզբից կատարում է ցիկլը հետո ստուգում է պայմանը:
- **for** օպերատորը օգտագործվում է այն դեպքում, երբ առկա է որևէ փոփոխականի սկզբնական արժեք, նրա մեծության ստուգման պայման և փոփոխականի ինքնամեծացում կամ փոքրացում:
- Հավերժ ցիկլեր կարելի է ստանալ **while(true)** կամ **for(;;)** հրամաններով:

// do... while

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int a = -10;
        do
        {
            Console.WriteLine("barev");
        } while (a > 0);
    }
}
//-----
```

// while, for // factorial 5!

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int n = 5;
        int fact = 1;
        int i = 1;
        while (i <= n)
        {
            fact= fact * i;
            i++;
        }
        Console.WriteLine(fact);
        fact= 1;
        for (int ii = 1; ii <= n; ii++)
            fact = fact * ii;
        Console.WriteLine(fact);
    }
}
//-----
```

❖ n թվի թվանշանների գումար

```
using System;
class M
{
    static void Main(string[] args)
    {
        for (; ; )
        {
            int sum = 0;
            Console.Write("tiv = ");
            int n = Convert.ToInt32(Console.ReadLine());
            int tiv = 0;
            while (n > 0)
            {
                tiv = n % 10;
                n = n / 10;
                Console.Write(tiv + " ");
                sum += tiv;
            }
            Console.WriteLine("gumar = " + sum);
        }
    }
}
//-----
```

using System; // n թվի թվանշանների առանձնացում

```
class M
{
    static void Main(string[] args)
    {
        for (; ; )
        {
            Console.Write("tiv = ");
            int n = Convert.ToInt32(Console.ReadLine());
            int tvanshan = 0;
            int nn = n;
            int k = 0; // թվանշանների քանակ
            while (n > 0)
            {
                n = n / 10;
                k++;
            }
            Console.WriteLine("tvanshanneri qanak = " + k);
            int kk = 1; // թվի կարգայնություն
            for (int i = 1; i < k; i++)
                kk *= 10;
            Console.WriteLine("kargaynutiun = " + kk);
            for (int i = 0; i < k; i++)
            {
                tvanshan = nn / kk;
                nn = nn % kk;
                kk = kk / 10;
                Console.Write(tvanshan + " ");
            }
            Console.WriteLine();
        }
    }
}
//-----
```

- ❖ Ֆիբոնաչիի շարք (1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...)
- ✓ Շարքի առաջին երկու անդամները հավասար են 1: Մնացած անդամները որոշվում են նախորդ երկու անդամների գումարով:

```
// Ֆիբոնաչի
using System;
class M
{
    static void Main(string[] args)
    {
        for (; ; )
        {
            Console.Write("tiv = ");
            int n = Convert.ToInt32(Console.ReadLine());
            int a1 = 1;
            int a2 = 1;
            int fib = 0;
            Console.Write(a1+" ");
            Console.Write(a2+" ");
            for (int i = 2; i < n;i++ )
            {
                fib = a1 + a2;
                Console.Write(fib+" ");
                a2 = a1;
                a1 = fib;
            }
            Console.WriteLine();
        }
    }
}
//-----
```

- Ցիկլերը, ինչպես **if** օպերատորը, ունեն այն առանձնահատկությունը, որ ձևավոր փակագծեր պարտադիր են, եթե ցիկլը ունի տիպի հայտարարումով միակ հրաման: Եթե տիպը հայտարարվում է և տեղում կատարվում է արժեքավորում (կամ նույնիսկ չկա արժեքավորում), ապա մեքենայական հրամանի տեսակետից, դրանք երկու հրաման են մեկ տողում: Հաջորդ օրինակը այդ առանձնահատուկ դեպքն է.

```
class M
{
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
            // {
                int a=99; //error
            //}
    }
}
//-----
```

- ❖ Պարզ թվի որոշում – (տնային)

8. Անցումային օպերատորներ – break, continue, goto

[Шилдг-139]

- Ցիկլներում թույլատրվում է օգտագործել **break** օպերատորը, որը ցիկլից դուրս է բերում ծրագրի հատվածը:
- continue** օպերատորի կատարման ժամանակ բաց է թողնվում նրան հաջորդող բոլոր հրամանները, սակայն ցիկլից դուրս չի գալիս ծրագիրը:
- goto** օպերատորը կատարում է թռիչք ծրագրի ցանկացած մասից ցանկացած այլ մաս, նշված “մետկայով”: Ժամանակակից լեզուներում խորհուրդ չի տրվում օգտագործել այս օպերատորը, որովհետև կա ծրագրավորման դառը փորձ, այսպես կոչված “սպագետտի կոդ”, որի իմաստը կայանում է հետևյալում, երբ շատ **goto** - ներ ենք օգտագործում, ապա ծրագրային կոդը խճճվում է՝ բերելով ծրագրի խափանման:

```
class Program // break
{
    static void Main(string[] args)
    {
        int a = 5;
        while (true)
        {
            System.Console.WriteLine("barev");
            a--;
            if (a <= 0)
                break;
        }
    }
}
```

```
//-----
class Program // continue
{
    static void Main(string[] args)
    {
        int a = 5;
        while (a>0)
        {
            System.Console.WriteLine("barev");
            a--;
            if (a >=3)
                continue;
            System.Console.WriteLine("#####");
        }
    }
}
```

```
//-----
class Program // if + goto = Ցիկլ
{
    static void Main(string[] args)
    {
        int a = 5;
mm:
        if (a > 0)
        {
            System.Console.WriteLine("barev");
            a--;
            goto mm;
        }
    }
}
```

9. Բիթային օպերացիաներ (|, &, ~, ^, <<, >>)

[Шилдг-107]

- Կա հետևյալ պատմությունը: Pascal և C լեզուների **պայքարի** ժամանակ հաղթող դուրս եկավ C լեզուն, որը գրավեց և արտադրական ծրագրավորումը և ուսման մեջ կիրառումը: C լեզվի հաղթանակի հիմքերում է ընկած այնպիսի բարդ հնարավորություններ, ինչպիսին են բիթային օպերացիաները, կամ ասենք հասցեներով աշխատելու հնարավորությունները ցուցիչների և հղիչների միջոցով: Նշված գործողությունները համարվում էին ասեմբլեր լեզվի յուրահատկություններ և ուսուցման տեսանկյունից տանում են դժվարությունների:
- Բիթային օպերացիաները նույնպես կիրառվում է C#sharp լեզվում: Այն հարավորություն է տալիս աշխատել բիթերի մակարդակով, որը կարելի է կիրառել ինչպես տրամաբանական խնդիրներում, այնպես էլ տարբեր արտաքին միջավայրներում (ասենք արտաքին սարքեր), որտեղ գործում են միջավայրի անհատական հրամանները: Այդ հրամանները շատ ավելի հեշտ ղեկավարվում են բիթային օպերացիաների միջոցով: Օրինակ արտաքին սարքին կարելի է ուղղարկել տարբեր կոդեր, որոնք այդ սարքի համար դառնում են հրամաններ: Այդպիսի ծրագրային իրականացումը կոչվում է սարքի “driver” ծրագիր:

✓ | բիթային “կամ” աշխատում հետևյալ ձևով՝

0 կամ 0 հավասար է 0;

0 կամ 1 հավասար է 1;

1 կամ 0 հավասար է 1;

1 կամ 1 հավասար է 1;

✓ & բիթային “և” աշխատում հետևյալ ձևով՝

0 և 0 հավասար է 0;

0 և 1 հավասար է 0;

1 և 0 հավասար է 0;

1 և 1 հավասար է 1;

✓ ^ բիթային բացառում ըստ “կամ” ի աշխատում հետևյալ ձևով (XOR)

0 ^ 0 հավասար է 0;

0 ^ 1 հավասար է 1;

1 ^ 0 հավասար է 1;

1 ^ 1 հավասար է 0;

✓ <<, >> բիթային շեղում աջ կամ ձախ

հնարավորություն է տալիս ըստ նշված տասական թվի չափի, 2 ական կողմ տեղաշարժել աջ կամ ձախ: Երկու դեպքում էլ աջից կամ ձախից կատարվում է տվյալի կորուստ, անհայտ բիթերում լրացնելով զրոներ, եթե դրական թիվ է: Բացասական թվերի դեպքում հատուկ մոտեցում է պետք:

✓ ~ բիթային ժխտում (NOT)

Նշված երկուական կոդում զրոները փոխվում է մեկերով, իսկ մեկերը զրոներով: Պարզ է, եթե դրական թիվ է ժխտվում, ապա նրա նշանը նույնպես ժխտվում է, որի պատճառով թիվը դառնում է բացասական: Համակարգիչը բացասական թվերը ներկայացնում է իր ձևով:

```
//.... 0000 0001 = 1
```

```
//... .0000 0000 = 0
```

```
//.... 1111 1111 = -1
```

```
//.... 1111 1110 = -2
```

```
//.... 1111 1101 = -3
```

```
//.... 1111 1100 = -4 // բացասական թվերը համարգչում պահվում են լրացուցիչ կոդով
```

```
using System;
```

```
int a = 5; //..... 0000 0101
```

```
int b = 3; //..... 0000 0011
```

```
int c = a & b; // բիթային տրամաբանական "և" օպերացիա
```

```
Console.WriteLine(c); //..... 0001 =1
```

```
c = a | b; // բիթային տրամաբանական "կամ" օպերացիա
```



```

Console.WriteLine(c); //..... 0111 = 7
c = a ^ b; // բիթային տրամաբանական բացառում ըստ "կամ" օպերացիայի
Console.WriteLine(c); //..... 0110 = 6
c = a >> 1; // բիթային տրամաբանական շեղում աջ մեկ բիթ
Console.WriteLine(c); //..... 0010 = 2
c = a << 1; // բիթային տրամաբանական շեղում ձախ մեկ բիթ
Console.WriteLine(c); //..... 1010 = 10
c = ~b; //..... 1111 1100 = -4 // բիթային ժխտման օպերացիա
Console.WriteLine(c); // -4 // .....1111 1100 = fc
Console.WriteLine(string.Format("{0:x}", c)); // ffffffff
//.... 0000 0001 = 1
//... .0000 0000 = 0
//.... 1111 1111 = -1
//.... 1111 1110 = -2
//.... 1111 1101 = -3
//.... 1111 1100 = -4 //..000100=4 //..111011 // հակադարձ կոդ // ..111011+1= 111100=-4 //լրացուցիչ կոդ

```

10. C/C# պարզ տարբերություններ

[Шилдт-97, 121]

- ❖ Նշենք C# -ի որոշ նորույթ առավելություններ C/C++ լեզվի նկատմամբ, նրա պրոցեդուրային մասով: [Troelsen – 79] Verbatim **string**: Կարելի է գրել մի քանի տող, որը պահում է ընդհանուր տեքստի արտաքին տեսքը և տողում թույլատրում է “Enter”:

```

Console.WriteLine(@"text
                        text 2");

կամ` Console.WriteLine($"@text") // C#8

```

- C -ում թույլատրում է չարժեքավորված լոկալ փոփոխականի հետ աշխատել, իսկ C# -ում, ոչ:
- C -ում թույլատրում է բլոկում փոփոխականի անունը համընկնի բլոկից դուրս, ուղղակի նա ծածկում է նրան: Այն C# -ում չի թույլատրվում:
- C- ում % թվաբանական գործողությունը միայն ամբողջ թվերին է թույլատրվում: C# -ում նաև կոտորակային թվերին:
- C# -ում տրամաբանական արտահայտությունը վերադարձնում է միայն **true** կամ **false**: Այդ դեպքում (==) հնարավոր չէ խառնել (=) – ի հետ, որովհետև պայմանը միշտ վերադարձնում է **true** կամ **false**:
- Տրամաբանական արտահայտությունը կարող է ունենալ կրճատված (&&,||) և ոչ կրճատված (&,) պայմաններ: (&,) համարվում է բիթային օպերացիա և կախված արտահայտությունից միաժամանակ կարող է դառնալ ոչ կրճատված տրամաբանական օպերացիա: Ոչ կրճատված նշանակում է տրամաբանական արտահայտության բոլոր անդամները ստուգվում են, եթե նույնիսկ արտահայտության արդյունքը հայտնի է լինում ավելի շուտ:

```

using System;
int a = 9;
if (4 > 8 && 9 < 0 && 6 > 0 && 88 == 88 && a++ > 8)
//if (4 > 8 & 9 < 0 & 6 > 0 & 88 == 88 & a++ > 8)
    Console.WriteLine("barev");
Console.WriteLine(a);

```

- C# -ում **switch** օպերատորի պարամետր ընդունում է նաև **string** և **enum**: C# -ում **break** - ը համարվում պարտադիր: Եթե **break** չկա, ապա կարելի է օգտագործել **goto case n**:
- **for** – ի պայմանում տեսանելիության տիրույթը նույնպես ընդգրկվում է բլոկի սահմանում: C- ում այդպես չէ:
- ✓ Օրինակում նշվածը չի բերում սխալի, իսկ C լեզվում երկրորդ **int** հայտարարումը պետք էր հանել:

```

for (int i = 0; i < 8; i++)
{
    }

```

```
for (int i = 0; i < 8; i++)
{
}
```

- `int @for = 55; int @new = 19;` Կարելի է `@` նշանի օգնությամբ հատուկ անունը օգտագործել որպես փոփոխականի անուն: (`@` և `_` սիմվոլները թույլատրված են որպես փոփոխականի 1-ին սիմվոլ):

❖ Using Digit Separators (New 7.0) [Troelsen-74]

- `_` սիմվոլը C#7 -ից սկսած թույլատրվում է նաև հանդիսանալ թվի թվանշանների մեջ որպես հասկանալի տարանջատիչ:

```
using System;
int decimalNotation = 2_000_000;
int binaryNotation = 0b0001_1110_1000_0100_1000_0000;
int hexadecimalNotation = 0x001A_8480;
Console.WriteLine(decimalNotation == binaryNotation); // => true
Console.WriteLine(decimalNotation == hexadecimalNotation); // => false
```

11. Կոտորակային թվեր

❖ [Пафяк – 81; Troelsen-67; Albahari – 48]

- Կոտորակային թվեր ներկայացնող `double`, `float` և `decimal` տիպերը ունեն ճշտության և թվերի ներկայացման տարբեր մոտեցումներ:

```
using System;
double a = 0.1;
double b = 0.2;
if (a + b == 0.3)
    Console.WriteLine("equal");
else
    Console.WriteLine("NOT equal");
decimal a2 = 0.1m;
decimal b2 = 0.2m;
if (a2 + b2 == 0.3m)
    Console.WriteLine("equal");
else
    Console.WriteLine("NOT equal");
Console.WriteLine("-----");
Console.WriteLine(int.MinValue);
Console.WriteLine(int.MaxValue);
Console.WriteLine($"{decimal.MinValue:N0}");
Console.WriteLine(decimal.MaxValue);
Console.WriteLine($"{double.MinValue:N0}");
Console.WriteLine(double.MaxValue);
Console.WriteLine(double.NaN);
Console.WriteLine(double.NegativeInfinity);
Console.WriteLine(double.PositiveInfinity);
Console.WriteLine(double.Epsilon);
```

- `double` -ի և `float` -ի ճշտությունը ցածր է, որովհետև թվերը ներկայացվում է անվերջ կոտորակների գումարով: Օրինակ 0.1 թիվը ներկայացվում է հետևյալ ձևով.

4	2	1	.	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
0	0	0	.	0	0	0	1	0	0	1	0	0	1	0

- `double` և `float` -ը անվերջ կոտորակների գումարի ներկայացման պատճառով կարող է ունենալ NaN, NegativeInfinity, PositiveInfinity, Epsilon արժեքներ:

12. Զանգված: Պրակտիկ խնդիրներ զանգվածներից

[Troelsen-150; Troelsen-111; albdarb]

- Զանգվածը տվյալների կազմակերպման այնպիսի ձև է, երբ անհրաժեշտ է միևնույն տիպի տվյալները պահել համարակալաված ձևով: Զանգվածների հետ աշխատանք կատարվում է **ինդեքսով**: Հայտարարման ձևը՝

```
int [ ] myArray = new int [5]; // վեկտոր
```

```
int [,] myArray = new int [8,8]; // մատրիցա
```

- Զանգվածը հայտարարման ժամանակ ընդունում է ըստ տիպի **լեւիիության արժեքներ**:
- Զանգվածի սահմանները գտնվում են հսկման տակ և չափը նշվում է **Length** հատկանիշի միջոցով:

```
using System; //պատահական թվերի վեկտոր
```

```
class M
```

```
{
    static void Main()
    {
        int[] A = new int[8];
        Random r = new Random();
        for (int i = 0; i < 8; i++)
            A[i] = r.Next(10);
        for (int i = 0; i < A.Length; i++)
            Console.Write(A[i] + " ");
    }
}
```

```
using System; // մաքսիմումի փնտրում
```

```
class M
```

```
{
    public static void Main()
    {
        Random r = new Random();
        int[] A = new int[10];
        for (int i = 0; i < 10; i++)
            A[i] = 50-r.Next()%100;
        int max=A[0];
        for (int i = 0; i < 10;i++ )
            Console.Write(A[i] + " ");
        for (int i = 0; i < 10; i++)
            if (A[i] > max)
                max = A[i];
        Console.WriteLine();
        Console.WriteLine("max = " + max);
    }
}
```

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

```
using System; // վերադասավորում
```

```
class M
```

```
{
    static void Main()
    {
        int n=16;
        int[] A = new int[n];
        Random r = new Random();
        for (int i = 0; i < n; i++)
            A[i] = r.Next(100);
        for (int i = 0; i < n; i++)
            Console.Write(A[i] + " ");
        Console.WriteLine();
        int a;
        for (int v = 0; v < n;v++ )
```

```

        for (int i = 0; i < n - 1; i++)
        {
            if (A[i] > A[i + 1])
            {
                a = A[i];
                A[i] = A[i + 1];
                A[i + 1] = a;
            }
        }
        for (int i = 0; i < n; i++)
            Console.Write(A[i] + " ");
        Console.WriteLine();
    }
}

//Տնային հատվածը
        a = A[i];
        A[i] = A[i + 1];
        A[i + 1] = a;
իրականացնել առանց a փոփոխականի
//-----
❖ Մատրիցա
using System; //մատրիցայի ցուցադրում
class M
{
    static void Main()
    {
        int[,] A = new int[8,8];

        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
                Console.Write(A[i, j]+ " ");
            Console.WriteLine();
        }
    }
}

//-----
using System; //մատրիցայի ցուցադրում անհրաժեշտ “պրոբեյնով”
class Program
{
    static void Main(string[] args)
    {
        int n = 8;
        Random r = new Random();
        int[,] A = new int[n, n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                A[i, j] = r.Next(1, 100);
        Console.WriteLine();
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (A[i, j] > 9)
                    Console.Write(A[i, j] + " ");
                else
                    Console.Write(" " + A[i, j] + " ");
            }
            Console.WriteLine();
        }
    }
}

```



```

using System; // նավակի հարվածներ
class M
{
    static void Main()
    {
        int[,] A = new int[8,8];
        Console.Write("navi y= ");
        int a = Convert.ToInt32(Console.ReadLine()); // կոորդինատ i
        Console.Write("navi x= ");
        int b = Convert.ToInt32(Console.ReadLine()); // կոորդինատ j
        for (int k = 0; k < 8; k++)
        {
            A[a-1, k] = 1;
            A[k, b-1] = 1;
        }
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
                Console.Write(A[i, j]+" ");
            Console.WriteLine();
        }
    }
}

```

```

using System; // շախմատ, փղի քայլեր
class M
{
    static void Main()
    {
        int[,] a = new int[8, 8];
        Console.Write("x=");
        int x = Convert.ToInt16(Console.ReadLine());
        Console.Write("y=");
        int y = Convert.ToInt16(Console.ReadLine());
        int ii = 0;
        int jj = 0;
        while (x + ii < 8 && y + jj < 8)
        {
            a[x + ii, y + jj] = 1;
            ii++;
            jj++;
        }
        ii = 0;
        jj = 0;
        while (x - ii >= 0 && y - jj >= 0)
        {
            a[x - ii, y - jj] = 1;
            ii++;
            jj++;
        }
        ii = 0;
        jj = 0;
        while (x + ii < 8 && y - jj >= 0)
        {
            a[x + ii, y - jj] = 1;
            ii++;
            jj++;
        }
        ii = 0;
    }
}

```

```

    jj = 0;
    while (x - ii >= 0 && y + jj < 8)
    {
        a[x - ii, y + jj] = 1;
        ii++;
        jj++;
    }
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
            Console.Write(a[i, j] + " ");
        Console.WriteLine();
    }
}

```

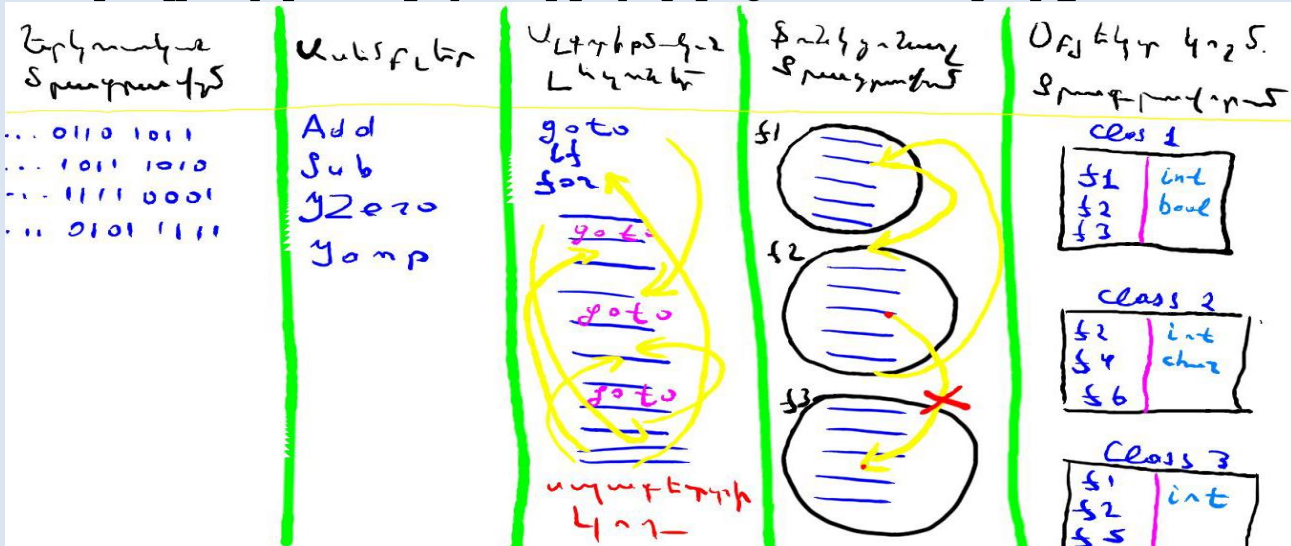
// (տնային) վերադասավորել մատրիցայի գլխավոր անկյունագիծը
 // (տնային) վերադասավորել մատրիցայի օժանդակ անկյունագիծը
 // (տնային) ցուցադրել ձիու հարվածները մատրիցայում
 // (տնային) ցուցադրել թագուհու հարվածները մատրիցայում
 // (տնային) Լոտո 36: 9x6 մատրիցայում պատահական ձևով լրացնել 36 չկրկնվող թվեր:

Տնային մատրիցաներից

- Թամբային կետ: Տրված է ամբողջթվային $M \times N$ չափի մատրիցը: Որոշել և արտածել մատրիցի այն տարրը, որը իր տողում մեծագույնն է, իսկ սյունում փոքրագույնը: Եթե այդպիսի տարր չկա, ապա արտածել “NO” հաղորդագրությունը:
- Տրված է $N \times N$ չափի քառակուսային մատրից: Առանց լրացուցիչ մատրից կիրառելու տեղերով փոխել $A_{1,1}$ և $A_{N,N}$, $A_{1,2}$ և $A_{N,N-1}$ և այլն տարրերը, այսինքն օժանդակ անկյունագծի նկատմամբ իրականացնել հայելային արտապատկերում: Արտածել ստացված մատրիցը:
- Տրված է $M \times N$ չափի մատրից: Արտածել մատրիցի կենտ համարով տողերի տարրերը ձախից աջ հաջորդականությամբ, իսկ զույգ համարովները՝ աջից ձախ հաջորդականությամբ:
- Տրված է $M \times N$ չափի մատրիցը: Հաշվել և արտածել մատրիցի ամեն տողերի փոքրագույն տարրերից մեծագույնի արժեքը:
- Տրված է A մատրիցա: Գտնել B վեկտորի տարրերը, որտեղ $B[i]$ -ն ստացվում է A մատրիցայի i -րդ տողի առաջին բացասական տարրից հետո եկող անդամների գումարից:

Ֆունկցիոնալ ծրագրավորում

13. Ծրագրավորման լեզուների զարգացման էտապները [albdarb]



- Ծրագրավորումը սկզբից եղել է **երկուական** տեսքի հրամաններով: Ծանրագույն ժամանակներ, երբ ծրագիրը ստեղծվում էր մեքենայական լեզվով: Արագ անհրաժեշտություն առաջացավ մարդկայնացնել ծրագրավորումը եւ այն բերեց լեզվի հնարման անհրաժեշտության: Այն կոչվեց **ասեմբլեր** լեզու: Ասեմբլերի ամեն հրաման համապատասխանում է երկուական լեզվի մեկ հրամանի: Լեզվի առկայությունը պահանջեց **թարգմանիչի անհրաժեշտության** եւ այսպես առաջացավ ասեմբլեր լեզուն:
- Հաջորդ էտապը առաջացավ արտադրողականության բարձրացման բնական պահանջի վրա: Գյուղատնտեսությունում այն կոչվում է ռենտայի օրենք՝ մեկ կարտոֆիլ գցիր ուրը ստացիր: Ծրագրավորումը այն օգտագործեց հետևյալ ձևով՝ մեկ հրամանը կկատարի մկից ավել մեքենայական հրամաններ: Հաջորդ լեզուները անվանվեցին բարձր կարգի լեզուներ կամ **ալգորիթմական լեզուներ**, իրենց հետ բերելով սեփական թարգմանիչները:
- Ինչ է **ֆունկցիոնալ** ծրագրավորում: Նա անհրաժեշտություն էր ալգորիթմական ծրագրավորումից հետո: Քանի որ, ծրագրավորման խնդիրները անընդհատ պահանջում են ավելի շատ հրամանների առկայություն, ապա ալգորիթմական լեզուները ծրագրի հրամանների խճճման պատճառով մի պահից սկսած չէին կարող բավարարել խնդրի լուծման աշխատանքային պահանջները: Նույնիսկ ընդունելի անուն ստացավ պրոբլեմը, որը անվանվեց **“սպագետի”** կոդ: Խնդրի լուծման համար առաջարկվեց ծրագրի հրամանները խմբավորել, ասենք ֆունկցիաների ձևով եւ ստանալ ոչ միայն որոշ թռիչքների խզում, այլ նաև կոդի բազմակի օգտագործման հնարավորություն ֆունկցիաների ձևով: Ֆունկցիոնալ ծրագրավորումը ստացավ տարբեր անվանումներ եւ զարգացումներ, որոնցից են կառուցվածքային կամ մոդուլային անվանումները: Պետք է հիշել մի շատ կարևոր աքսիոմա՝ լեզուները առաջանում են, որովհետև հին ձևով անհնար է “ապրել”, ծրագրի հրամանների անվերջ աճման պահանջի պատճառով: Ցանկացած նոր էտապ **“քառսից”** դուրս գալու հնարք է: Այլ խոսքով ամեն էտապ պայքարում է **էնտրոպիայի** դեմ: (էնտրոպիա – քառսի չափման միավոր):
- Հաջորդ էտապը կոչվում է **օբյեկտ-կողմնորոշված** ծրագրավորում: Այն բարձրացնում է ծրագրավորման **աբստրակցիան**: Քանի որ, ֆունկցիոնալ ծրագրավորումը հրամանների մեծ քանակի դեպքում բերում է խճճում ֆունկցիաների մակարդակով, անհրաժեշտություն առաջացավ այդ ֆունկցիաները **պարփակել** ծրագրավորման մի այլ միավորի մեջ, որը անվանվեց **կլաս**: Միավորումը իրականացվեց նորամուծությամբ, այն է, ֆունկցիաների հետ ընդգրկվեցին նաև տվյալներ:
- Չպետք է մոռանալ, որ այժմ ծրագրավորումը գտնվում է օբյեկտ կողմնորոշված տիրույթում, սակայն կարելի է ավելացնել, որ Microsoft -ի կողմից առաջարկված **դեկլարատիվ կոդի** միջավայրում ծրագրավորումը իր ուղղությունն է հարթում նոր ժամանակներում: Microsoft –ի նոր “պրոյեկտի” անունը կոչվում է .NET կամ **dotNET**:

14. Մեթոդներ

[Шилдт-155; Troelsen-120]

Մեթոդի պարամետրեր, արգումենտներ [Шилдт-162]

Մեթոդի վերադարձվող արժեք [Шилдт-159]

return *օպերատոր* [Шилдт-158]

Լոկալ մեթոդներ [Troelsen-121; Solis - 90]

- Ծրագրի ֆունկցիոնալ գործողությունների շարքին է պատկանում մեթոդը: Մեթոդի պարզագույն ներկայացման ձևը՝

```
void ff()  
{  
    // հրամաններ  
}
```

- ✓ Որտեղ մեթոդը ունի անուն `ff`, վերադարձվող արժեք `void` եւ կլոր փակագծեր (`:`): `void` գրվում է այն դեպքում, երբ մեթոդը ոչինչ չի վերադարձնում:
- Ընդանրապես ցանկալի է եւ գործողության հնարավորությունները մեծ են, եթե մեթոդը կարող է ստանալ տվյալներ, որոնք կոչվում են մեթոդի պարամետրեր: Եթե մեթոդը վերադարձնում է արժեք, ապա նշվում է նրա տիպը: Մեթոդը պարամետրեր ստանում է կլոր փակագծերում եւ արժեք վերադարձնում է `return` հրամանով: Մեթոդի վերադարձվող արժեքի տիպը (`int, bool double...`), որը գրվում է մեթոդի անունից ձախ, պետք է գտնվի տիպային համաձայնության մեջ `return` հրամանի հետ:

```
int ff(int a, char c)  
{  
    // հրամաններ  
    return արտահայտություն; // int տիպ պետք է լինի  
}
```

- Ծրագիրը իր ստարտը սկսում է `Main()` մեթոդի կատարումով, որը ենթարկվում է օպերացիոն համակարգի կանչին:

```
class Program  
{  
    static void Main()  
    {  
    }  
}
```

❖ Պարամետր, արգումենտ և վերադարձվող արժեք

- Պայմանավորվենք: Եթե մեթոդը նկարագրվում է, ապա այն ունի **պարամետրեր** տվյալներ ստանալու համար: Եթե մեթոդը կանչվում է, ապա պարամետրերը կկոչվեն **արգումենտներ**:

```
using System;  
class Program  
{  
    static int ff(int a, char c, int b) // պարամետրեր  
    {  
        Console.WriteLine("barev !");  
        Console.WriteLine(c);  
        return a + b;  
    }  
    static void Main()  
    {  
        int sum = ff(333, 'W', 555); // արգումենտներ  
        Console.WriteLine("gumar = " + sum);  
    }  
}
```

```
}
• return հրաման կարելի է օգտագործել նաև void ոչինչ չվերադարձվող արժեքի հետ: Այն չունի հաջորդող տվյալ կամ հրաման և ուղղակի դուրս է բերում ծրագիրը մեթոդից:
```

```
using System;
class Program
{
    static void f()
    {
        return;
        Console.WriteLine("barev"); // չի կատարվի
    }
    static void Main(string[] args)
    {
        f();
    }
}
```

• Մեթոդը արգումենտում կարող է ստանալ այլ մեթոդ, եթե արգումենտի մեթոդի վերադարձվող արժեքի տիպը համընկնում է մեթոդի պարամետրի տիպին:

```
using System;
class Program
{
    static void f(int a) //մեթոդի նկարագրություն
    {
        Console.WriteLine(a);
    }
    static int f2()
    {
        return 2021;
    }
    static string f()
    {
        return "albdarb.com";
    }
    static void Main(string[] args)
    {
        f(f2()); // մեթոդի կանչ արգումենտով
        Console.WriteLine(f()); // մեթոդի կանչ արգումենտով
    }
}
```

❖ **Լոկալ մեթոդներ** [Solis – 90; Троелсен-169; Troelsen-121]

- C#7 -ից սկսած մեթոդը կարող ընդգրկել այլ մեթոդի նկարագրություն, որը կոչվում է լոկալ մեթոդ: Դա կատարվում է այն դեպքում, երբ լոկալ մեթոդին դիմում է միայն մեկ մեթոդ:
- Լոկալ մեթոդի անունը կարող է համընկնել մեթոդի անվան հետ: Տվյալ դեպքում կանչը չի դառնում ռեկուրսիվ (ռեկուրսիա – կանցնենք ավելի ուշ):

```
using System;
class M
{
    static void Main(string[] args)
    {
        int a = 1;
        f(); // թույլատրվում է մեթոդի կանչ մինչև վ հայտարարումը
        void f()
        {
            Console.WriteLine("barev = "+a);
        }
        f();
    }
}
```



```
}
```

❖ Սորտավորման ծրագիրը մեթոդների միջոցով

- Օրինակում ցուցադրվում է, թե ինչպես մեթոդը զանգվածի արժեքները վերադարձնում է պարամետրի միջոցով:

```
using System;
```

```
class Program
```

```
{
```

```
    static void frand(int[] ar)
```

```
    {
```

```
        Random r = new Random();
```

```
        for (int i = 0; i < 10; i++)
```

```
            ar[i] = r.Next(100);
```

```
    }
```

```
    static void sortBuble(int[] arr)
```

```
    {
```

```
        Console.WriteLine("BubbleSort");
```

```
        for (int k = 0; k < 10; k++)
```

```
            for (int i = 0; i < 9; i++)
```

```
                if (arr[i] > arr[i + 1])
```

```
                {
```

```
                    arr[i + 1] = arr[i] + arr[i + 1]; // տվյալների փոխատեղում
```

```
                    arr[i] = arr[i + 1] - arr[i]; // առանց միջանկյալ
```

```
                    arr[i + 1] = arr[i + 1] - arr[i]; // փոփոխականի
```

```
                }
```

```
    }
```

```
    static void sortSelection(int[] arr)
```

```
    {
```

```
        Console.WriteLine("SelectionSort");
```

```
        for (int i = 0; i < arr.Length - 1; i++)
```

```
        {
```

```
            int k = i;
```

```
            for (int j = i + 1; j < arr.Length; j++)
```

```
                if (arr[k] > arr[j])
```

```
                    k = j;
```

```
            if (k != i)
```

```
            {
```

```
                int temp = arr[k];
```

```
                arr[k] = arr[i];
```

```
                arr[i] = temp;
```

```
            }
```

```
        }
```

```
    }
```

```
    static void output(int[] arr)
```

```
    {
```

```
        for (int i = 0; i < 10; i++)
```

```
            Console.Write(arr[i] + " ");
```

```
        Console.WriteLine();
```

```
    }
```

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

```
    {
```

```
        int[] ar = new int[10]; // Քվանտային տեղեպարտիայի տեսակի նմանություն
```

```
        frand(ar);
```

```
        output(ar);
```

```
        sortBuble(ar);
```

```
        output(ar);
```

```
        sortSelection(ar);
```

```
        output(ar);
```

```
    }
```

```
}
```

15. Գերբեռնված մեթոդներ

[Шилдт-235; Троелсен-168; Troelsen-133]

- Ինչպես գիտենք տվյալ տեսանելիության տիրույթում նույն անունով փոփոխական չի թույլատրվում հայտարարել: Նույնը չի վերաբերվում մեթոդներին: Կարելի է նույն անունով մեթոդ ստեղծել, եթե կա հետևյալ սահմանափակումները՝ մեթոդների պարամետրերը պետք է լինեն տարբեր: Կամ ավելի ճիշտ մեթոդները ունենան տարբեր քանակի պարամետրեր, կամ եթե նույն քանակի են, ապա տարբերվեն տիպով: Վերադարձվող արժեքի տիպը որոշիչ դեր չի խաղում:
- Գերբեռնված մեթոդները հնարավորություն են տալիս ինչպես ունենալ ավելի հասկանալի ծրագրային տեսք, այնպես էլ էֆեկտիվ օգտագործել համակարգչային ռեսուրսները:
- Վերցնենք `Console.WriteLine(...)` մեթոդը: Այն ունի 19 գերբեռնված տարբերակներ:

```
class Program
```

```
{
    static long f(long a, long b)
    {
        return a + b;
    }
    static double f(double a, double b)
    {
        return a + b;
    }
    static void Main(string[] args)
    {
        System.Console.WriteLine(f(4.4, 6.7));
        System.Console.WriteLine(f(4, 6));
    }
}
```

❖ Ծրագրի կատարման արագության տարբերության ապացույց

```
using System;
```

```
class Program
```

```
{
    static void f(long a, long b)
    {
        long sum = 0;
        DateTime dt = DateTime.Now;
        Console.WriteLine("es long em");
        for (int i = 0; i < 200_000_000; i++)
            sum = a + b;
        DateTime dt2 = DateTime.Now;
        Console.WriteLine(dt2 - dt);
    }
    static void f(decimal a, decimal b)
    {
        decimal sum = 0;
        DateTime dt = DateTime.Now;
        Console.WriteLine("es decimal em");
        for (int i = 0; i < 200_000_000; i++)
            sum = a + b;
        DateTime dt2 = DateTime.Now;
        Console.WriteLine(dt2 - dt);
    }
    static void Main(string[] args)
    {
        f(4.4m, 6.7m);
        f(4, 6);
    }
}
```

16. Լոկալ փոփոխականներ, բլոկ { }, Մեթոդ և Ստեկ

[albdarb; Solis - 124]

- Ցանկացած մեթոդի փոփոխականներ կոչվում են **լոկալ փոփոխականներ**: Նրանք անպայման պետք է արժեքավորվեն մինչև օգտագործվելը: Լոկալ փոփոխականները վերանում են մեթոդի ավարտից հետո (թիթեռի կյանք):
- Մեթոդի ցանկացած մասում կարելի է դնել ձևավոր փակագծեր { }: Այն կոչվում է **բլոկ**: Նույնիսկ պայմանական օպերատորների եւ ցիկլերի ձևավոր փակագծերը բլոկներ են: Նրանք ստեղծում են **տեսանելիության տիրույթ**: Ներքին բլոկից դեպի արտաքին բլոկը տեսանելի է, իսկ հակառակը, ոչ:

```
class Program
```

```
{
    static void Main(string[] args)
    {
        int x;
        // System.Console.WriteLine(x); // error // արժեքավորված չէ
        int a = 99;
        { // բլոկ
            // int a = 11; // error // int a կրկնվում է
            // System.Console.WriteLine(b); //error// տեսանելիության տիրույթում չէ
            int b = 88;
            {
                int c = 77;
                System.Console.WriteLine(a);
                System.Console.WriteLine(b);
                System.Console.WriteLine(c);
            }
        }
    }
}
```

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

```
class Program
```

```
{
    static void Main(string[] args)
    {
        for (int i = 0; i < 3; i++)
            //{
                int a = 999; // error
            //}
    }
}
```

❖ **Մեթոդ և ստեկ հիշողություն** [albdarb; Solis – 124; Liberty - C++ 24hours, pg.95]

- Մեթոդների լոկալ փոփոխականները պահվում են հիշողության այն մասում, որը կոչվում է **ստեկ** հիշողություն: Ստեկը հիշողության տրամաբանական կազմակերպման ձև է, որտեղ տվյալները պահվում եւ կարդացվում են հատուկ ձևով՝ հաջորդաբար ներառված տվյալներից “առաջինը կարդում ենք վերջում, իսկ վերջինը կարդում ենք առաջինը” սկզբունքով (ավտոմատի “մագազինի” նման): Ստեկ հիշողությունը նույնպես գտնվում է RAM ում: Այն ձևավորվում է ծրագրի թարգմանման ժամանակ: Նրա տարածքը սահմանափակ է: Աշխատում է LIFO սկզբունքով: Last in First Out:
- Երբ կանչվում է մեթոդը, ապա կատարվում է հետևյալ գործողությունների հաջորդականությունը.
 - Ստեկում հիշվում է մեթոդի կանչի հաջորդ հասցեն ծրագրի հետագա աշխատանքի համար:
 - Ստեկում հատկացվում է վրադարձվող արժեքի տիպի չափով տեղ, **return** օպերատորի տվյալի համար:
 - Ստեկ է ուղղարկվում մեթոդի պարամետրերը:
 - Ստեկում տեղադրվում է մեթոդի բոլոր լոկալ փոփոխականները:
- Մեթոդի տվյալները ինչ հաջորդականությամբ գրանցվում են ստեկում, հակառակ հաջորդականությամբ ջնջվում են:

17. Ռեկուրսիա

[Шилдг-257]

- Ռեկուրսիան, դա ծրագրավորման հնարավորություն է, երբ մեթոդը կանչում է ինքն իրեն: Այս ձևով կարելի է ապահովել ցիկլի աշխատանքի մի այլ ձև: Եթե միջոցներ չձեռնարկվի, ապա մեթոդը իրեն կկանչի ավներջ, կամ ավելի ճիշտ մինչև հատկացված հիշողության (ստեկ հիշողության) լցվելը: Հիմանականում կիրառվում է մեթոդի ռեկուրսիվ կանչը մեթոդի կանչը հսկող փոփոխականի փոքրացման միջոցով:

```
using System; // ռեկուրսիվ անվերջ կանչ
```

```
class A
{
    public void f()
    {
        Console.Write("barev ");
        f();
    }
}

class Program
{
    static void Main(string[] args)
    {
        A ob = new A();
        ob.f();
    }
}
```

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

```
using System; // ռեկուրսիվ 5 բարեւ
```

```
class Program
{
    static void f(int n)
    {
        if (n > 0)
        {
            Console.WriteLine("barev");
            n--;
            f(n); // կամ
            // f(--n);
            // f(n--); // Ինչու ճիշտ չի աշխատում
        }
        else return;
    }
    static void Main(string[] args)
    {
        int a = 5;
        f(a);
    }
}
```

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

```
using System; // թվի ֆակտորիալ ռեկուրսիվ
```

```
class Program
{
    static void Main(string[] args)
    {
        for ( ; ; )
        {
            Console.Write("0-exit; >0 tiv = ");
            int a = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

```

        if (a == 0) // exit
            return;
        int result = fact(a);
        Console.WriteLine(result);
    }
}
static int fact(int n)
{
    Console.WriteLine("barev " + n);
    if (n == 1)
        return 1;
    return n * fact(n - 1);
}
}
//-----

```

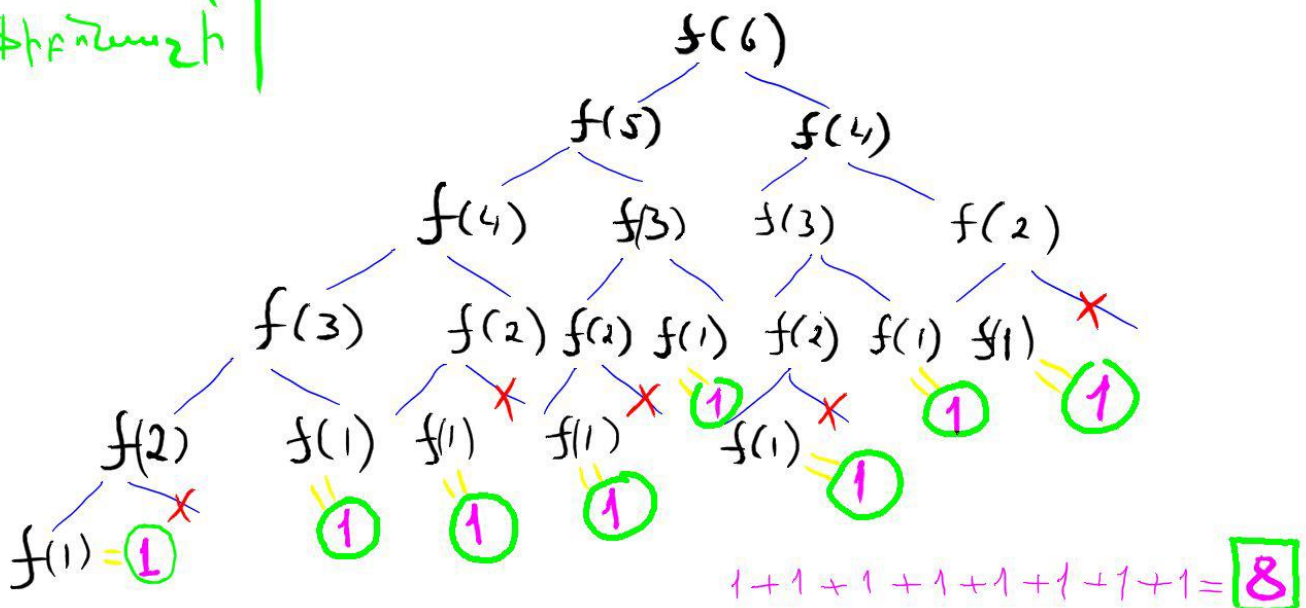
❖ Ֆիբոնաչի շարքի (1 1 2 3 5 8 13 21 34) անդամի որոշումը ռեկուրսիվ մեթոդով.

```

using System;
class Program
{
    static public int f(int n)
    {
        if (n > 2)
            return f(n-1)+f(n-2);
        else
            return 1;
    }
    static void Main(string[] args)
    {
        for (; ; )
        {
            int a = Convert.ToInt32(Console.ReadLine());
            Console.Write("fibonacci " + a + " =");
            Console.WriteLine(f(a));
        }
    }
}

```

Ռեկուրսիվ
Ֆիբոնաչի | $\rightarrow \text{return } f(n-1) + f(n-2); // f(6) = 8$



✓ Ռեկուրսիվ $f(2)$ կամ $f(1)$ վերադարձնում են մեկ արժեք: Նրանցից հետո չի շարունակվում հաշվարկը:

```
// ? ternary operator with return
using System;
class Program
{
    static int f(int n)
    {
        return (n > 1) ? n * f(n - 1) : 1; // factorial
        //return (n > 2) ? f(n - 1) + f(n - 2) : 1; // fibonacci
    }
    static void Main(string[] args)
    {
        int a = 5;
        Console.WriteLine(f(a));
    }
}
//-----
```

❖ Գտնել Կատալանի շարքի թիվը n -րդ անդամով // 1, 2, 5, 14, 42, 132

✓ Catalan number => $C_0=1$; $C_{n+1}=2(2n+1)*C_n/(n+2)$

```
using System;
class Catalan
{
    static int f(int n)
    {
        if (n == 0)
            return 1;
        return 2 * (2 * n + 1) * f(n - 1) / (n + 2);
    }
    static void Main()
    {
        int n;
        while (true)
        {
            Console.Write("n= ");
            n = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine(f(n - 1));
        }
    }
}
```

// Տնային աշխատանք, կիրառել մեթոդներ

1. 🦋 Նշված դաշտից ձին քայլի պատահական ազատ դաշտերով լինելով մեկ անգամ:
2. 🦋 Նշված դաշտից ձին քայլի պատահական ազատ դաշտերով լինելով մեկ անգամ կիրառելով ձիու քայլերի շարժումային էվրիստիկա:
3. 🦋 Նշված դաշտից ձին քայլի պատահական ազատ դաշտերով լինելով մեկ անգամ կիրառելով լրիվ հաշվարկ հետ քայլ անելու սկզբունքով:
4. 🦋 Պատահական տեղադրել շախմատի տախտակի վրա թագուհիներ այնպես, որ միմյանց չհարվածեն:
5. 🦋 Պատահական տեղադրել շախմատի տախտակի վրա թագուհիներ այնպես, որ միմյանց չհարվածեն, կիրառելով թագուհու դաշտեր գրավելու շարժումային էվրիստիկան:
6. 🦋 Պատահական տեղադրել շախմատի տախտակի վրա թագուհիներ այնպես, որ միմյանց չհարվածեն, կիրառելով լրիվ հաշվարկ հետ քայլ անելու սկզբունքով:

C#sharp հիմունքներ

18. Կլասներ, օբյեկտներ

[Шилдг-147; Троелсен-194; Troelsen-171]

- Կլասները եւ օբյեկտները համարվում են օբյեկտ-կողմնորոշված ծրագրավորման հիմնային կառուցվածքային բլոկներ: Մինչև օբյեկտ-կողմնորոշվածը, ծրագրավորումը անցել է հետևյալ էտապները: Երկուական ծրագրավորում (011101), ասեմբլեր, ավգորիթմիկ լեզուներ (Algol, Fortran, Cobol), պրոցեդուրային կամ ֆունկցիոնալ լեզուներ (C, Pascal), օբյեկտ կողմնորոշված լեզուներ (C++, Java, C#), դեկլարատիվ կողով լեզուներ (Java, C#, Swift): Ցանկացած էտապից մյուսին անցնելու համար անհրաժեշտ է օբյեկտիվ պայմաններ: Այդպիսի պայմաններից է ծրագրային հրամանների **անվերջ մեծացումը**: Օբյեկտ-կողմնորոշված ծրագրավորումը դա հիմնային անցում է: Այն **քառսից** դուրս գալու հնարք է, որի հիմքում ընկած է **մարդկային** հասարակության **մեծ փորձը** բարդ խնդիրների լուծման համար: Այն մոդելավորվում էր որպես օբյեկտներ, շրջապատը եւ խնդիրները **պարզեցնելու** համար: Ստեղծվեց մարդկային լեզուն եւ խոսքը, որը պարունակում է ինչպես գործողություն (բայ) այնպես էլ առարկա (գոյական): Ահա այս մոդելը արտապատկերվել է ծրագրավորման զարգացման նոր էտապի մեջ ու հետաքրքիրը կայանաում է նրանում, որ այն տվել է **արդյունք**: Ծրագրավորումը բերվեց մի վիճակի, երբ խնդիրը կատարվում է **նկարագրությունների** միջոցով: Որպես ծրագրավորման միավոր ստեղծվեց **կլասներ**, որոնց թույլատրվեց պահել եւ **տվյալ** եւ **գործողություն** միաժամանակ: Կլասը դարձավ մարդկային վարկեցողության արտապատկերող բարդ խնդիրներ լուծելու համար:
- Ենթադրենք մոդելավորում ենք աթոռ կլասը: Որպես տվյալներ կարելի է ներկայացնել նյութերը, ասենք փայտ, “պլաստմասս”, “ռեզին”, երկաթ, կտորեղեն, “գուպկա” եւ այլն: Որպես գործողություն կարելի է ներկայացնել ամրացումներ, կոնֆորտացնող միջոցներ, նստելու հնարավորություն, պաշտպանական գործողության հնարավորություն (մի խփեք ուրիշի գլխին):
- Կլասը կատարեց նոր ծառայություն, այն թույլ է տալիս սահմանափակ քանակով ներդրված տիպերի հետ զուգահեռ ստեղծել անսահմանափակ քանակությամբ նոր տիպեր: (user-defined types):
- Կլասի տեխնիկական կիրառման համար օգտագործվում է օբյեկտ գաղափարը: Կարելի է համարել, որ կլասը դա շաբլոն է, իսկ օբյեկտը նրա իրականացումը: Օրինակ, առանձնատան նախագիծը դա կլաս է (այլապես ասած շաբլոն), իսկ շինարարությունից հետո առաջացած տունը, դա օբյեկտն է: Օբյեկտը մեծ նյութականություն է եւ ծրագրի տեսանկյունից՝ զբաղեցրած հիշողություն:
- Կլասը կարող է պարունակել տվյալներ (**int** a, **char** c, **bool** b այլն) կամ այլ կլասի օբյեկտներ (կոմպոզիցիա), որոնց անվանում են **դաշտեր** եւ **գործողություններ** (ֆունկցիաներ), որոնցից են մեթոդները, հատկանիշները, ինդեքսատորները, իրադարձությունները եւ այլն:
- Թարգմանման ժամանակ կատարվում է հիշողության հատկացում միայն ֆունկցիաների հրամանների համար, իսկ տվյալների համար, ոչ: Կլասի տվյալների համար հիշողություն հատկացվում է ծրագրի կատարման ժամանակ եւ միայն օբյեկտի ստեղծման դեպքում:
- Օբյեկտ ստեղծվում է հետևյալ ձևով՝ A ob = **new** A(); Կլասի անդամներին դիմում ենք օբյեկտ եւ կետի միջոցով՝ ob.ff() կամ ob.a: Կարող են լինել բազում օբյեկտներ տվյալ կլասի համար: Ցանկացած օբյեկտի ստեղծվելու ժամանակ կատարվում է տվյալ օբյեկտի տվյալների համար սեփական հիշողության առանձնացում: Սակայն օբյեկտները չունեն սեփական մեթոդները: Այդ մեթոդները բոլորի օբյեկտների համար նույնն են:
- Ընդունված կարգ է, օբյեկտ կողմնորոշված ծրագրավորման լեզուներում գործողություն կատարող ֆունկցիան, եթե գտնվում է կլասի մեջ, այն անվանում են մեթոդ:

```
class A
{
    int a; // տվյալ
    public void ff() //գործողություն
```



```

    {
        a = 44;
        System.Console.WriteLine(a);
    }
}
class M
{
    static void Main()
    {
        A ob = new A(); // օբյեկտի ստեղծում
        ob.ff(); // կլասի անդամի օգտագործում
    }
}

```

❖ Օբյեկտի ստեղծում և մեթոդի կանչ

- Օբյեկտի ստեղծման ժամանակ թույլատրվում է կետով կանչել անդամ մեթոդները: Այդ ժամանակ մեթոդի վերադարձվող արժեքի տիպը պետք է հաընկնի ստացող օբյեկտի տիպին:

```

using System;
class A
{
    public B f()
    {
        return new B();
    }
}
class B
{
    public int f2()
    {
        return 999;
    }
    public void f3()
    {
        Console.WriteLine("barev");
    }
}
class Program
{
    static void Main(string[] args)
    {
        A ob1 = new A();
        B ob2 = new A().f();
        int a = new A().f().f2();
        Console.WriteLine(a);
        new A().f().f3();
        //-----
        A ob = new(); // C#9
    }
}

```

19. Թույլատրության մոդիֆիկատորներ, Պարփակվածություն

[Троелсен- 219,215; Troelsen-196, 192; Шилдт-210]

- Օբյեկտ-կողմնորոշված ծրագրավորումը առանձնանում է տվյալների պաշտպանվածության կամ պարփակվածության (**encapsulation**) մեծացման հնարավորությունով: Այն պահում է բնության “էներգիայի պահպանման օրենքը”: Տալիս է հնարավորություն ավելորդ միջամտումներից խուսափելու համար: (Հիշենք **կրիայի** պարփակվածությունը, որով նա գոյատևում է արդեն հարյուր միլիոնավոր տարիներ): Ինչպես ասում է Ստրաուստրուպը (C++ հեղինակ), պաշտպանվածությունը ուժեղացվում է ոչ թե վնասարարի դեմ, այլ պատահական վնասի դեմ, որը ավելի շատ է տարածված ծրագրավորման ասպարեզում: Այն ապահովվում է թույլատրության մոդիֆիկատորների օգնությամբ:
- Թույլատրության մոդիֆիկատորներ (Access Modifiers)
`public`
`private`
`protected` // կանցնենք ավելի ուշ
`internal` // կանցնենք ավելի ուշ
`protected internal` // կանցնենք ավելի ուշ
`private protected` // կանցնենք ավելի ուշ C#7.2
- Նշվածներից միայն `public` -ն է, որ չի բավարարում պարփակվածության պահանջներին:
- `class` -ը եւ `interface`, `struct`, `enum`, `delegate` (որոնք կանցնենք ավելի ուշ), նույնպես կարող են ունենալ թույլատրություն, միայն `public` եւ `internal`: Լռելիությամբ նրանք նշանակվում են `internal`:
`internal` - նշանակում է ընթացիկ հավաքագրման բլոկ:
`public` - նշանակում է արտաքին հավաքագրման բլոկ:
- **Կարևոր գաղափար:** Քանի, որ տվյալներ շատ կաեւոր են եւ նրանց վնասվելը (միտումնավոր կամ պատահական) թանկ է նստում, ապա ծրագրավորման մեջ ընդունված կարգ է, տվյալները փակել (`private`) և նրանց հետ աշխատելու համար ավելացնել բաց `public` մեթոդներ: Ծրագրավորման այսպիսի կազմակերպումը համարվում է **պարփակվածություն** կիրառման ձև: Այսպիսով առաջանում է պաշտպանվածության մի ձև, որ եթե պետք է վնասել տվյալը, ապա պետք է ծրագրային փոփոխություն կատարել մեթոդում, որը այդքան էլ հեշտ չէ:

```
using System;
class A // encapsulation
{
    private int a; // պարփակված տվյալ
    public void f_read() //գործողություն կարդալ
    {
        Console.WriteLine(a);
    }
    public void f_write() //գործողություն գրել
    {
        a = 999;
    }
}
class M
{
    static void Main()
    {
        A ob = new A(); // օբյեկտի ստեղծում
        ob.f_write();    // կլասի փակ անդամի արժեքավորում
        ob.f_read();     // կլասի փակ անդամի օգտագործում
    }
}
```

20. Կոնստրուկտոր

[Шилдг-147; Троелсен-197; Troelsen-174]

- Կլասներում շատ հաճախ անհրաժեշտ է լինում տվյալներին տալ նախնական արժեքներ: Այն հնարավոր է մեթոդի միջոցով եւ կանչել այդ մեթոդը յուրաքանչյուր օբյեկտի ստեղծման ժամանակ: Սակայն այդ գործողությունը հնարավոր նաև է կատարել հատուկ միջոցով, որ կոչվում է կոնստրուկտոր: Այն հնարավորություն է տալիս տվյալների նախնական արժեքավորումը **ավտոմատացնել** (չկանչելով մեթոդ): Այսպիսով, ցանկացած կլաս ունի մեթոդ, որի անունը **համընկնում է** կլասի անվան հետ: Այն կոչվում է կոնստրուկտոր եւ նախատեսված է կլասի դաշտերին (տվյալներին) **նախնական արժեքներ** վերագրելու համար: Կոնստրուկտորը աշխատում է, երբ **ստեղծվում է օբյեկտ**: Ավելին, կոնստրուկտորը **ռչինչ չի վերադարձնում**, նույիսկ **void**:
- Կոնստրուկտորը կարող է լինել **փակ կամ բաց**: Եթե **կոնստրուկտորը փակ է**, ապա հնարավոր չէ ստեղծել տվյալ կլասի օբյեկտը արտաքին միջավայրից: Պարզ է, որ կլասի անդամներին հնարավոր է դիմել այլ մեթոդներով, որոնցից են **կլասի ստատիկ անդամները** (կանցնենք հետո):
- Եթե կոնստրուկտորը չունի պարամետր, ապա կոչվում է **լռելիությամբ կոնստրուկտոր**: Օրենք՝ եթե ծրագրում ոչ մի կոնստրուկտոր չի նշված, ապա կոմպիլիտորը ինքն է ստեղծում իր լռելիությամբ կոնստրուկտորը:

```
using System;
class A
{
    int a;
    public A ()
    {
        a = 567;
        System.Console.WriteLine("barev");
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
    }
}
```

❖ Պարամետրով եւ գերբեռնված կոնստրուկտորներ

- Քանի, որ կոնստրուկտորը մեթոդ է, ապա այն կարող է ունենալ պարամետրեր եւ գերբեռնվել:
- Օրենք՝ եթե կա պարամետրով կոնստրուկտոր, ապա լռելիությամբ կոնստրուկտոր ստեղծելը մնում է ծրագրավորողի վրա: Օրինակում երևում է լռելիությամբ կոնստրուկտորի անհրաժեշտությունը:

```
class A
{
    int aa;
    public A(int a)
    {
        aa = a;
        System.Console.WriteLine(aa);
    }
    public A() // Բացակայության դեպքում կտա error !
    {
    }
}
class Program
{
    static void Main(string[] args)
    {
        A ob = new A(); //Ստիպում է լռելիությամբ կոնստրուկտոր գրել
        A ob2 = new A(2);
    }
}
```

21. Ինիցիալիզատորներ, կոմպոզիցիա

[albdarb; Троелсен-236; Troelsen-217]

- Տվյալներին նախնական արժեքներ տալը կոչվում է ինիցիալիզացիա: Ինիցիալիզացիան կարելի է կատարել ինչպես կոնստրուկտորի միջոցով, այնպես էլ վերագրման օպերատորով, կլասի դաշտին արժեքավորելով հայտարարման ժամանակ: Սկզբից կատարվում է ինիցիալիզատորը, նոր հետո կոնստրուկտորը: Ինիցիալիզատորը նույնպես աշխատում է օբյեկտի ստեղծման ժամանակ:
- Եթե որպես կլասի դաշտ (անդամ տվյալ) հանդես է գալիս այլ կլասի օբյեկտի ստեղծում, ապա դա կոչվում է **կոմպոզիցիա**: Եվ եթե օբյեկտի ստեղծման փոխարեն միայն **հղումն** է վերագրվում, ապա կոմպոզիցիան դառնում է **ագրեգացիա**:
- Կոմպոզիցիայի դեպքում կոնստրուկտորների կատարման հերթականությունը կատարվում հետևյալ ձևով՝ կզբից կոմպոզիցիան, հետո սեփական կոնստրուկտորը:
- Ինիցիալիզացիան չի կարող օգվել այլ դաշտի տվյալներից:

```
using System;
class A
{
    int a = 44;           // initializer
    B ob = new B();       // composition, containment, "has-a" relationship
    string s = Console.ReadLine(); // initializer // Ազդում է գրված տեղի վրա
    B ob2;                // aggregation
    public A()            // default constructor
    {
        Console.WriteLine("AAAAAAA");
    }
}
class B
{
    public B()
    {
        Console.WriteLine("BBBBBBBB");
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        // A ob2 = new A { a = 55 }; // this is initializer, when (a) verible is public
    }
}
class A
{
    public int a;
    public char c;
}
class Program
{
    static void Main(string[] args)
    {
        A ob = new A { a = 1, c = 'C' };
        System.Console.WriteLine(ob.a + " " + ob.c);
    }
}
```

22. Անունի տարածք, global ::

[Шилдг-513; Троелсен- 510; Troelsen-597; Нейгел-92,94]

- Ծրագրային կոդի **քարդասցման** հետ կապված կատարվում է տիպերի **խմբավորում** անունի տարածքում, որը հնարավորություն է տալիս **կոնֆլիկտներից** խուսափել: Անունի տարածքը համարվում է պարփակվածության (**encapsulation**) ձև: Օրինակ, եթե մարդուն օբյեկտ համարենք եւ տվյալ անուն ազգանունով կարող են շատ մարդիկ լինել, ապա բնակվելու հասցեն կդառնա անունի տարածք, որը կտարբերակի այդ մարդկանց:
- Նայեք “Տերմինատոր 1” ֆիլմը եւ կհասկանաք անունի տարածքը:
- Անունի տարածքը հայտարարվում է **namespace** հրամանով: Անունի տարածքից օգտվում են **using** – ով, կամ անմիջական կցում են կլասի անվան հետ:

```
using System;
using poghos;
using petros;
// using System.Threading;
namespace petros
{
    class A
    {
        public void f()
        {
            Console.WriteLine("barev petros");
        }
    }
}
namespace poghos
{
    class A
    {
        public void f()
        {
            Console.WriteLine("barev poghos");
        }
    }
}
namespace albdarb
{
    class Program
    {
        static void Main(string[] args)
        {
            // A ob = new A(); // error ?
            poghos.A ob1 = new poghos.A(); // ok
            ob1.f();
            //Thread.Sleep(3000);
            petros.A ob2 = new petros.A(); // ok
            ob2.f();
        }
    }
}
// .....
```

❖ Անունների տարածքի պսևիդոնում [Троелсен-513; Troelsen-600; Нейгел-94]

- Երբեմն անունի տարածքը ստացվում է երկար: Այն կարճացնելու համար օգտագործվում է պսևիդոնում, որից կարելի է օգտվել ծրագրում:

```
using NN = Armenia.Yerevan.Darbinyan.program;
class M
{
    public static int Main()
    {
        NN.M ob = new NN.M();
        System.Console.WriteLine(ob.ff());
        return 0;
    }
}
namespace Armenia.Yerevan.Darbinyan.program
{
    class M
    {
        public string ff()
        {
            return this.GetType().Namespace;
        }
    }
}
// .....
```

❖ Անունի տարածք և **global ::**

[Шилдг-524]

- Տվյալ ծրագրային պրոյեկտի սահմաններում կարելի է ունենալ եւ անունի տարածքի առկայություն եւ առանց անունի տարածքի ծրագրի հատված: Հնարավոր է նշված ծրագրի տարբեր հատվածները ներառեն նաև միևնույն անունով կլաս: Այդ կլասները տարբերակելու համար օգտվում են **global** հրամանից:

```
using System;
class A
{
    public A()
    {
        Console.WriteLine("global A");
    }
}
namespace ConsoleApplication1
{
    class A
    {
        public A()
        {
            Console.WriteLine("my A");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            A ob1 = new A();
            global::A ob2 = new global::A();
        }
    }
}
```

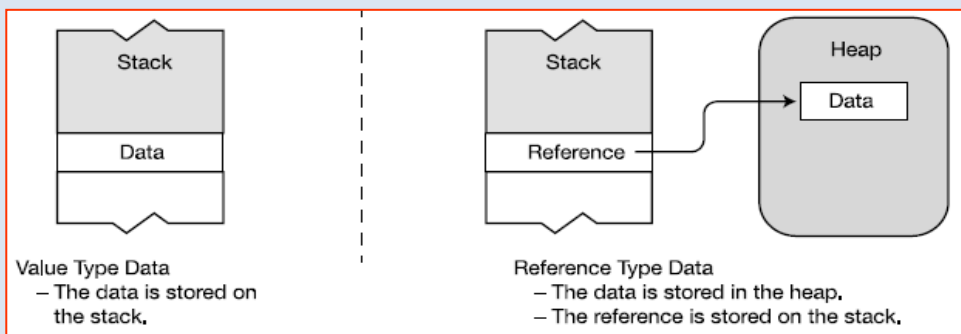
❖ (C#10)Թույլատրվում է հետևյալ հայտարարությունները

```
global using static System.Console;
global using aaa = System.Environment; // տեսանելի է պրոյեկտի բոլոր ֆայլերում
using System;
```

23. Չափային և հղիչային տիպեր

[Heյгел -371; Троелсен-178; Troelsen-147; Рихтер-150]

- C# լեզվում տիպերի բաժանման նոր մոտեցում է առաջարկվում: Տիպերը խմբավորվում են ըստ իրենց ֆիզիկական իրականացման չափանիշի՝ ստեկի եւ կույտի (դինամիկ հիշողություն): Կրկնենք: **Ստեկ (stack)**, դա հիշողության տրամաբանական կազմակերպման ձև է, որը աշխատում է հետևյալ մեթոդով՝ վերջի պահվածը կարդացվում է առաջինը: Ստեկը հատկացվում է թարգմանության ժամանակ եւ ազատվում է, երբ ծրագիրը հատում է բլոկի տեսանելիության սահմանը: Դինամիկ հիշողություն կամ կույտ (**managed heap**), դա ազատ հիշողությունն է, որը ծրագրին հատկացվում է կատարման ժամանակ: Կույտը ազատվում է նույնպես ավտոմատ ձևով հատուկ միջոցներով:
- C# -ում տիպերը բաժանվում են երկու խմբի, չափային տիպերի (value type), որոնց համար տեղ հատկացվում է ստեկում եւ հղիչային տիպերի (reference type), որոնք տեղավորվում են դինամիկ հիշողությունում՝ այսպես ասած կույտում:
Չափային տիպեր են՝ `int, long, bool, float, char, ... enum, struct`
Հղիչային տիպեր են՝ `class, Array, interface, string, delegate, object, dynamic`
- Այսպիսով, եթե ես մեկ անգամ նշենք, չափայինի դեպքում օբյեկտը կամ փոփոխականը ստեղծվում է ծրագրի թարգմանման ժամանակ՝ **ստեկում**: Հղիչայինի դեպքում՝ օբյեկտ ստեղծվում է ծրագրի կատարման ժամանակ դինամիկ հիշողությունում (Heap), **հղիչը թողնելով ստեկում**:
- Հղիչային տիպերի դեպքում **ռեսուրսների օգտագործումը մեծանում** է, որը բերում է արագության **անկման**: Չափային տիպերի դեպքում ծրագիրը ավելի արագագործ է, իսկ ծրագրավորման հնարավորությունները **սահմանափակ** են:



```
struct S //կանցնենք հետո
{
    int a; //stack
    string s; //heap
    public void f(int aa, string ss)
    {
        int b = 886; //stack
        string s2 = "albdarb"; //heap
        M ob = new M(); //heap
        S ob2 = new S(); //stack
    }
}
class A
{
    int a; //heap
    string s; //heap
    public void f(int aa, string ss)
    {
        int b = 886; //stack
        string s2 = "albdarb"; //heap
        int[,] Z = new int[8,8]; //heap
        A ob = new A(); //heap
        S ob2 = new S(); //stack
    }
}
```


24. boxing և unboxing

[Рихтер-156; Шилдт -370; Нейгел -212]

- Եթե տիպը դիտարկենք ծրագրի կատարման ժամանակային ընթացքում, ապա այն անցնում է տարբեր **ձևափոխման սահմաններով**: Տիպեր ավելի մեծ հնարավորությունից կարող է անցնել ավելի փոքրին եւ հակառակը: Անցումը համարվում է ավելի բարդ, եթե անցում կա չափայինից հղիչային եւ հակառակը: Այդպիսի անցումների մեխանիզմը կոչվում է **boxing/unboxing**: Առաջինի դեպքում չափային տիպը արհեստական ձևով **տեղափոխվում է կույտ**: Երկրորդի դեպքում կույտ տեղափոխված տիպը **վերադարձվում է ստեկ**: Օրինակ, boxing – ի անհրաժեշտություն կա, եթե մեթոդը չափային տիպ է ուղարկում մի այլ մեթոդի, որը ընդունում է միայն հղիչային տիպ:
- Նշված գործողությունները ապահովվում են արդեն գոյություն ունեցող միջոցներով: Դրանք են **object** տիպի օգտագործումը եւ **տիպերի վերափոխման ()** օպերացիան:

```
using System;
```

```
class M
{
    static void Main()
    {
        int a = 40;
        object b = a; // boxing
        b = 88;
        int c = (int)b; // unboxing
        Console.WriteLine(a + "\n" + b + "\n" + c);
        //.....
        // եթե կույտից վերադառնում ենք ոչ ճիշտ տիպով, ապա առաջանում է արտակարգ իրավիճակ,
        // որը կանցնենք ավելի ուշ:
        int s = 55;
        object ob = s;
        string ss = (string)ob;
    }
}
//.....
class M //Троелсен
{
    static void Main()
    {
        int a = 25;
        object ob = a;
        //object ob2 = 50666669977L;
        long a2 = (long)a;
        //long a3 = (long)ob; //expection ?
        long a4 = (long)(int)ob; // ok!!
        //long a5 = (int)ob; // ok!!
    }
}
```

❖ **new** չափային տիպերի համար [Троелсен-113; Troelsen-67; Шилдт -170]

- new** նշանակում է **կոնստրուկտորի աշխատանք** եւ նախատեսվում է լրելիությամբ նախնական արժեքները վերագրելու համար: Օրինակ. `int a = new int()` – նույնպես թույլատրելի է եւ `a` -ին տրվում է 0 արժեք:
- ✓ `ValueType ob = new int(); // ok`
- ✓ `int a = new(); // C#9`
- new** չափային տիպերում օգտագործելը նշանակում է C# լեզվի լրիվ օբյեկտային կողմնորոշվածության դրսևորում:

25. null արժեք, ? և ?? սինվոլներ

[Шилдг-695 ; Нейгел-211; Троелсен-186; Troelsen-155]

- Հղիչային տիպերին եթե ոչինչ չենք վերագրում, ապա **կարող ենք** վրագրել **null** արժեք: Եթե չափային տիպը լոկալ տիպ է, ապա նրան մինչև օգտագործելը պետք է անպայման արժեք վերագրել:
- Կարելի է չափային տիպը օգտագործել առանց նախնական վերագրման, եթե նրան տրվել է **null** արժեք: Այդպիսի վերագրում կարելի է կատարել երկու ձևով:
`int ? a=null; // Nullable<int> b=null; կանցնենք հետո (կհ)`
- Այսպիսի անհրաժեշտություն կարող է առաջանալ՝ օրինակ **տվյալների բազայի** ձևավորման ժամանակ, երբ որոշ դաշտեր արժեքներ չեն ձևավորում:
- **null** փոփոխականներում **HasValue** հատկանիշը հնարավորություն է տալիս ստուգել արժեքի գոյությունը կամ բացակայությունը:

```
Nullable<int> b = null;
Console.WriteLine(b);
int? a = null;
Console.WriteLine(a);
if (a.HasValue)
    Console.WriteLine("a has value");
else
    Console.WriteLine("a - no has value");
```
- Եթե ունար օպերացիաներում ցանկացած փոփոխական **null** է, ապա արդյունքը ստացվում է **null**:
`int? a = null;`
`int? b = a + 9;`
b – ն ստանում է **null** արժեք:
- ✓ Նշվածից խուսափելու համար օգտագործվում է **??** օպերատորը, որը պայմանական օպերատոր է եւ վերադարձնում է ձախ արտահայտությունը, եթե այն **null** չէ եւ աջը հակառակ դեպքում:
- ✓ `op1 ?? op2` համարժեք է `op1 == null ? op2 : op1` (ternary) արտահայտությանը:

```
int? a = null;
int? b = a + 9;
Console.WriteLine(b);
b = a ?? 886;
Console.WriteLine(b);
a = 5;
b = a ?? 886; //b = a ?? + 886;
Console.WriteLine(b);
int ? i = null;
int j = (int)i; // error // արգելում է տիպերի վերափոխարկումը
```

- ❖ **?. (հարցական կետ)** օպերատորը ստուգում է **null** լինելը նոր է դիմում օբյեկտին, որպեսզի խուսափենք **NullReferenceException** արտակարգ իրավիճակից, որը կառաջանա երբ դիմենք **null** օբյեկտին հին ձևով: (C#6.0)

```
string s=null;
int? c = s?.Length;
//int? c2 = s.Length; // error NullReferenceException
```

.....

26. Ստրուկտուրաներ

[Нейгел -130; Шилдт -391; Троелсен-175; Troelsen-142]

- Ստրուկտուրան իր օգտագործումը սկսել է C լեզվում: Այս կառուցվածքը սկզբից միայն կատարում էր տարբեր տիպերի հավաքագրում մեկ անվան տակ: Հաջորդ զարգացումը կիրառվեց C++ -ում, որտեղ ստրուկտուրան բացի տվյալներից ներառեց նաև ֆունկցիաներ եւ ուներ հավասար համարժեքություն կլասին: C# -ում ստրուկտուրան նոր սահմանում է ստանում:
- Ինչպես գիտենք՝ կլասի օբյեկտները ստեղծվում են կոյտում: Եթե կա անհրաժեշտություն այն ստեղծել **ստեկում**, ապա առաջարկվում է նոր չափային տիպ, որը կոչվում է **ստրուկտուրա**: Այլ խոսքով, ստրուկտուրան C# -ում կարելի է համարել կլասի **unboxing** իրականացում ավելի սակավ հնարավորություններով:
- ❖ Նշենք որոշ հատկություններ կապված ստրուկտուրաների հետ.
- ✓ Ստրուկտուրաներում չի կարելի որոշել **լռելիությամբ կոնստրուկտոր** (**ok-Framwork 2022, no-Core 2022**): Եթե օգտագործվում է պարամետրով կոնստրուկտոր, ապա կոնստրուկտորում տվյալների ինիցիալիզացիան **պարտադիր** է (**ok-Framwork 2022, no-Core 2022**):
- ✓ Կարելի է օգտագործել **new** հրամանը օբյեկտ ստեղծելու համար, (սակայն պարտադիր չէ): Եթե **new** չկա, ապա **նախնական ինիցիալիզացիան** կատարվում է ծրագրավորողի կողմից: Այսպիսով, օբյեկտ կարելի է ստեղծել երկու ձևով՝ `S ob = new S();` կամ `S ob;`
- ✓ Չի թույլատրվում փոփոխականներին **ինիցիալիզացիա** տեղում: Չի թույլատրվում **կոմպոզիցիա**, սակայն թույլատրվում **ագրեգացիա** (**ok-Framwork 2022, no-Core 2022**):
- ✓ Ստրուկտուրան **չի կարող ժառանգել** կլաս կամ այլ ստրուկտուրա, բայց կարող է ժառանգել **interface**: Ելնելով ժառանգականության սահմանափակումից, ստրուկտուրան չի օգտագործում մոդիֆիկատորներ **abstract**, **virtual**, **protected**: // կանցնենք հետո
- ✓ Ստրուկտուրան չունի **դեստրուկտոր** (կանցնենք հետո C#2 -ում):
- Ստրուկտուրայի համար չկա բազային կլաս **System.Structure**, սակայն այն առաջացել է **System.ValueType** աբստրակտ կլասից: `ValueType ob = new int();`
- ✓ Ստրուկտուրաները, ինչպես և կլասները տիպայնացված լեզուների այն առավելություններից են, որոնք հանդիսանում են ներդրված տիպերի շարունակություն և ավելացնում են տիպերի քանակը՝ հանդես գալով որպես օգտագործողի կողմից հայտարարված տիպեր:

```
using System;
```

```
struct S
```

```
{
```

```
    public int a;
```

```
    public void f()
```

```
    {
```

```
        Console.WriteLine(a);
```

```
        Console.WriteLine("barev");
```

```
    }
```

```
    public S(int aa)
```

```
    {
```

```
        a = aa; // եթե բացակայի, ապա error!
```

```
    }
```

```
    //public S() // error! // Framwork
```

```
    //{ }
```

```
}
```

```
class Program
```

```
{    static void Main(string[] args)
```

```
    {    S ob;
```

```
        ob.a = 99;
```

```
        ob.f();
```

```
        S ob2 = new S(44);
```

```
        ob2.f();
```

```
    }
```

```
}
```

- ❖ Եթե ստրուկտուրայի երկու օբյեկտ վերագրվում են մեկը մյուսին, ունենում ենք **երկու կոպիա**: Կլասի դեպքում **կոպիան մեկն է**, հղումները երկուսը: Օրինակ.

```
using System;
    S ob1 = new S();
    ob1.x = 100;
    S ob2 = new S();
    ob2 = ob1;
    Console.WriteLine(ob1.x);
    Console.WriteLine(ob2.x);
    ob2.x = 900;
    Console.WriteLine(ob1.x);
    Console.WriteLine(ob2.x);

struct S //class S // record S (կհ)
{
    public int x;
}
```

❖ System.ValueType [Troelsen-148]

- Պայմանավորվենք՝ բոլոր չափային տիպերը առաջանում են ValueType **աբստրակտ (կհ)** կլասից: ValueType –ի դերը կայանում է նրանում, որ ցանկացած նրան ենթակա տիպ (օրինակ ստրուկտուրա) տեղադրվի ստեկում: Ֆունկցիոնալ տեսակետից նրա առաքելությունն է object կլասի բոլոր վիրտուալ մեթոդները վերաորոշվի և օբյեկտները աշխատեն տվյալների հետ այլ ոչ թե հղման հետ:

❖ (Պրակտիկա) Օրինակ. Լոկալ տվյալների կուտակիչ [albdarb]

- Տվյալների կուտակիչ ստանալու համար անհրաժեշտ է, առաջին, ստեղծել կլաս կամ ստրուկտուրա կուտակիչի անհրաժեշտ դաշտերով: Երկրորդ, օգտվել զանգվածից, կիրառվող կլասը կամ ստրուկտուրան կուտակիչ դարձնելու համար:
- Այլ խոսքով: **Կուտակիչ = կլաս + զանգված:**

```
using System;
    int N = 25;
    string t = "";
    int x = 0; // Մուտքագրման քանակ
    int k = 0; // Ընդհանուր մուտքեր
    Anketa[] ob = new Anketa[N]; // Կլասի դեպքում դեռ հիշողություն չի զբաղեցնում
    //for(int i=0;i<N;i++) // class Anketa
    //    ob[i] = new Anketa();
    for ( ; ; )
    { Console.WriteLine(@"0-Exit; 1-input; 2-List; 3-search by age; 4-search by name
        5=Sort by Age; 6=sort by name; 7-delete by name ");
        t = Console.ReadLine();
        switch (t)
        { case "0":
            return;
            case "1":
                Console.Write("mutqagrvogh tvyalneri qanak = ");
                x = Convert.ToInt32(Console.ReadLine());
                for (int i = k; i < k + x; i++)
                { Console.Write("Enter name:");
                    ob[i].name = Console.ReadLine();
                    Console.Write("Enter age:");
                    ob[i].age = Convert.ToInt32(Console.ReadLine());
                }
                k = k + x;
                break;
            case "2":
                for (int i = 0; i < k; i++)
                { Console.Write(ob[i].name + "\t");
                    Console.WriteLine(ob[i].age);
                }
            }
        }
```


27. Կոպիայի կոնստրուկտոր

[Liberty-66; Либерти -97]

- Ինչպես գիտենք կլասի օբյեկտների վերագրման ժամանակ միայն հղումն է վերագրվում եւ երկու հղումները դիմում են հիշողության նույն օբյեկտին: Որպեսզի հնարավոր լինի ստանալ երկու առանձին կոպիաներ, պետք է օգտվել կոպիայի կոնստրուկտորից:
- C# լեզուն չի ապահովում կոպիայի կոնստրուկտոր, եւ այն դառնում է ծրագրավորողի պարտականությունը: Անհրաժեշտ է պարամետրի միջոցով ստանալ օբյեկտ եւ էլեմենտ առ էլեմենտ կատաել վերագրում:

```
A ob = new A();  
A ob1 = new A(ob);
```

```
using System;  
class A  
{  
    public int x = 20;  
    char c = 'w';  
    string s = "hello";  
    string s2 = "barev";  
    public A()  
    {  
    }  
    public A(A ob)  
    {  
        x = ob.x;  
        c = ob.c;  
        s = ob.s;  
        s2 = ob.s; // չի կարելի  
    }  
    public void ff()  
    {  
        Console.WriteLine(x);  
    }  
}  
class M  
{  
    static void Main()  
    {  
        A ob = new A();  
        A ob1 = new A(ob); // result ... 20 99  
        //A ob1 = ob; // result ... 99 99  
        ob.ff();  
        ob1.ff();  
        ob.x = 99;  
        ob.ff();  
        ob1.ff();  
    }  
}
```

28. Կլասի **static** անդամներ

[Троелсен-207; Troelsen-184; Шилдт -260; Solis -136]

- Կլասի դաշտերը եւ գործողությունները կարող են լինել **ստատիկ**: Հիմնականում լինում են ստատիկ անդամ տվյալներ եւ ստատիկ անդամ մեթոդներ: Ստատիկ տվյալները տարբերվում են ոչ ստատիկից նրանով, որ անկախ օբյեկտների քանակից տվյալ դաշտի **մեկ օրինակ** գոյություն ունի: Սա հնարավորություն է տալիս տարբեր օբյեկտների մեթոդներ աշխատեն միևնույն տվյալի հետ:
- Ստատիկ մեթոդները կարող են աշխատել **միայն ստատիկ** անդամների հետ: Իսկ սովորական մեթոդները՝ **ցանկացած** անդամի հետ:
- Կլասի ստատիկ անդամները օգտագործվում են կլասի անունով՝ **A.ff()**, այլ ոչ թե օբյեկտի միջոցով՝ **ob.ff()**: Որպեսզի ստատիկ մեթոդը աշխատի ոչ ստատիկ անդամի հետ, մեթոդը պետք է **ստեղծի օբյեկտ**:
- ❖ Օրինակը ցույց է տալիս ստատիկ մեթոդի աշխատանքը եւ ինչպես կարելի է ստատիկ մեթոդով դիմել ոչ ստատիկ անդամին: Ուղղակի դա կատարվում է օբյեկտ ստեղծելու ձևով:

```
//Static Methods
```

```
A.f1();
```

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

```
class A
```

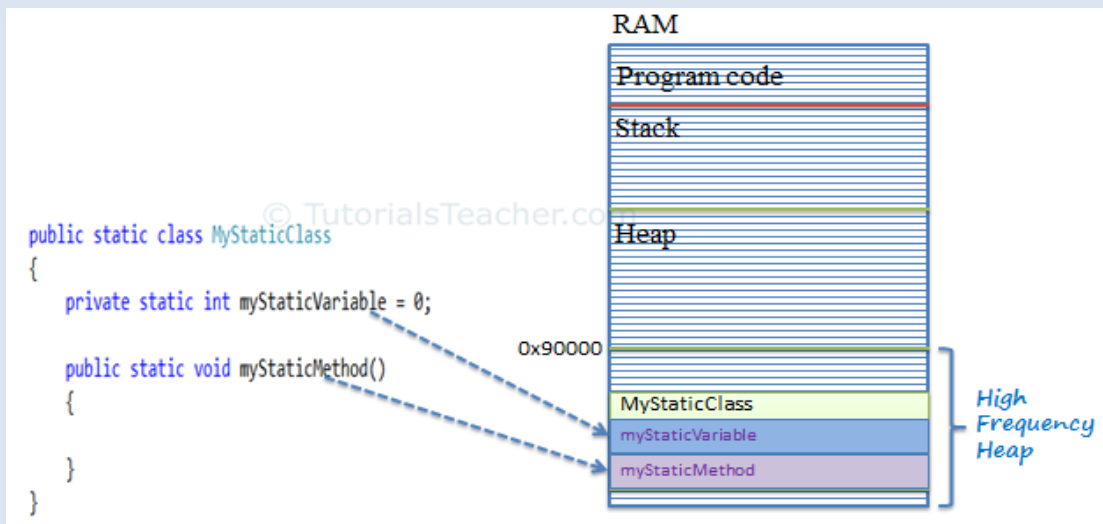
```
{  
    public static void f1()  
    {  
        Console.WriteLine("static f1()");  
        // f2(); // error // կա բացառություն  
        A ob = new A ();  
        ob.f2(); // ok  
    }  
    void f2()  
    {  
        Console.WriteLine("no static f2()");  
        // f1(); // ok  
    }  
}
```

- Հաջորդ օրինակը ցույց է տալիս, որ կլասը ներկայացված է օբյեկտ կոդմնորոշված **պատերնով**, որը պահում է տվյալ պահին օբյեկտների քանակը: Դրա համար անհրաժեշտ է ստատիկ դաշտ, որպես հաշվիչ եւ կոնստրուկտոր, որը կատարում է “ինքրեմենտ” օպերացիան: [Troelsen-191]

```
A.ff(); //Static Data  
A aa = new A();  
A.ff();  
A aa2 = new A();  
A.ff();  
  
class A  
{  
    static int ii = 0;  
    public A()  
    {  
        ii++;  
    }  
    public static void ff()  
    {  
        Console.WriteLine("{0} class A ", ii);  
    }  
}
```

- ❖ **Ստատիկ անդամներ և հիշողություն** [albdarb]

- Ստատիկ տվյալները պահվում են կույտում, որը կոչվում է High Frequency Heap: Ստատիկ տվյալները համարվում են կլասի բաղկացուցիչ, սակայն RAM –ում տեղ է հատկացվում **կատարման ժամանակ**՝ կամ օբյեկտի ստեղծումով կամ ստատիկով դիմելու դեպքում (**Framework**): Core –ում ստատիկ հիշողություն առանձնացվում է **միայն ստատիկին** դիմելու դեպքում: Ստատիկ տվյալները RAM –ում ընդգրկվելուց հետո պահպանվում են մինչև **ծրագրի ավարտը** եւ հեռացվում են կլասի հետ միասին: Նրանց վրա չի ազդում **տեսանելիության** տիրույթը եւ ավելցուկի **հավաքիչի** կանչը:



- Օրինակը **ապացուցում է**, որ ստատիկ տվյալների համար հիշողություն հատկացվում է ծրագրի **կատարման** ժամանակ և RAM –ում մնում են մինչև ծրագրի **ավարտը**: Հնարավոր չէ ջնջել GC -ով, որովհետև **ստեղծում չունի հղում**:

```

using System;
class A
{
    static long[, ] ar = new long[1000, 1000]; // Ազդում է օբյեկտի զբաղեցրած հիշողության թվի վրա
    // public static long x;
}
class M
{
    static void f()
    {
        //A.x = 44; // Core –ի դեպքում այլ արդյունք է ստացվում
        A ob = new A();
    }
    static void Main()
    {
        Console.WriteLine(GC.GetTotalMemory(true)); // ռեսուրս է օգտագործում
        Console.WriteLine(GC.GetTotalMemory(true));
        f();
        GC.Collect(); // ապացուցում է ստատիկի RAM -ում լինելը մինչև ծրագրի ավարտը: Հանել ստատիկը
        Console.WriteLine(GC.GetTotalMemory(false));
    }
}

```

❖ **static** լոկալ ֆունկցիաներ (New 8.0) [Troelsen-122] N (no video)

- Երբ հայտարարված է լոկալ մեթոդ, այն կարող է փոփոխել գլխավոր մեթոդի փոփոխականների արժեքները: C#8 Core –ի դեպքում թույլատրվում է լոկալ մեթոդը դարձնել ստատիկ: Այդ դեպքում լոկալ մեթոդը **չի կարող դիմել** գլխավոր մեթոդի փոփոխականներին, եթե նույնիսկ գլխավոր մեթոդը ստատիկ է:

```

f();
void f()
{
    int x = 4;
    int y = 3;
    static int Add(int a, int b)
    {
        // x++; // error, որովհետև Add() մեթոդը ստատիկ է
        return a + b;
    }
    Console.WriteLine(Add(x,y));
    Console.WriteLine(x-y);
}

```

29. Ստատիկ կոնստրուկտոր, Ստատիկ կլասներ

[Шилдг-265; Троелсен-211,213; Troelsen-188, 190; Хейгел-126]

- Կլասը կարող է ունենալ **ստատիկ կոնստրուկտոր** (միայն մեկ հատ): Այն երաշխավորում է, որ նա կաշխատի մինչև օբյեկտ ստեղծելը եւ միայն մեկ անգամ, անկախ օբյեկտների քանակից: Նա աշխատում է օբյեկտ ստեղծելուց առաջ, կամ ստատիկ անդամի առաջին դիմման ժամանակ: Նա չի կարող ունենալ ցանկացած մոդիֆիկատոր (**public**, **private**): Հիմնականում օգտագործվում է, եթե անհրաժեշտ է ստատիկ անդամներին տալ նախնական արժեք: Ստատիկ կոնստրուկտորը չի կարող ունենալ պարամետր:

```
class A
{
    public static void ff()
    {
        System.Console.WriteLine("barev");    }
    static A()
    {
        System.Console.WriteLine("static constructor");    }
}

class M
{
    static void Main()
    {
        //A ob = new A();
        A.ff();
    }
}
```

❖ Ստատիկ կլասներ

- C# 2005 –ում կլասը նույնպես կարող է լինել ստատիկ: Այդ դեպքում չի թույլատրվում **new**-ով ստեղծել կլասի օբյեկտ եւ պետք է ընդգրկի **միայն ստատիկ** անդամներ:

```
static class A
{
    // ստատիկ անդամներ    }
```

- Այսպիսով, կլասի օբյեկտի ստղծման արգելափակման համար կա երեք ձև՝ **private** կոնստրուկտոր, **static** կլաս, **abstract** կլաս (**կհ**):

```
class A
{
    private A(){    }
}

abstract class A    (կհ)
{
    }
```

- ❖ (VS 2017). Անունի տարածքում կարելի է օգտագործել նաև **ստատիկ կլասների ճանապարհը**:

```
using static System.Console;
using static NN.A;
namespace NN
{
    static class A
    {
        public static void f()
        {
            WriteLine("xxx");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        WriteLine("barev");
        f(); // NN.A.f();
    }
}
```

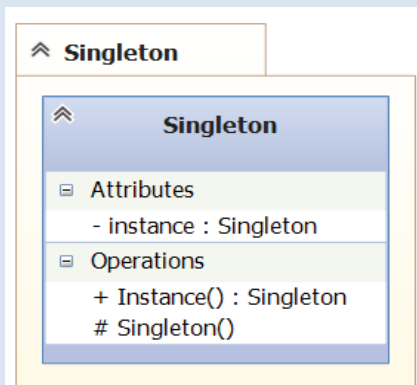
30. Մինգլտոն պատերն

[Hэш-96; Шевчук -83; GOF-157]

- Մինգլտոն պատերնը հանդիսանում է օբյեկտ կողմնորոշված ծրագրավորման կառույց, երբ անհրաժեշտություն է առաջանում ստեղծել **միայն մեկ օբյեկտ**, արգելելով բազմաթիվ օբյեկտների ստեղծման հնարավորությունը:
- Gamma E., ... “Design Patterns”: Նշված գրքով, որը գրվել է չորս հեղինակների կողմից (երբեմն անվանում են Gang of Four- GOF) 1995 թ. -ին, սկիզբ դրվեց ծրագրավորման ուղղության, որը կոչվում է Design Patterns: Պատերները, դրանք օբյեկտ կողմնորոշված ծրագրավորման պատրաստի բլոկներ են, որոնք արդեն մշակված են, ունեն մեծ անհրաժեշտություն եւ կիրառելի են տարբեր խնդիրներում: Պատերները ծրագրերը դարձնում են ավելի **ադապտիվ**, կոդի **փոփխելիությունը** ավելի հեշտ եւ որոնց շնորհիվ ծրագիրները դառնում են **ուղղեկցվող** կիրառման էտապում:
- Կարելի է օբյեկտի մեկ օրինակ ունենալ հետևյալ ձևով, որը OOP կառույց չէ.

```
class A
{
    public A()
    {
        System.Console.WriteLine("constructor");
    }
}
class M
{
    static void Main()
    {
        A ob1 = null;
        ob1 = new A(); // 1
        if (ob1 == null)
        {
            A ob2 =null;
            ob2 = new A(); // 2
            if (ob2 == null)
            {
                A ob3 = new A();
            }
        }
    }
}
```

- Մինգլտոն պատերնը, ի տարբերություն նշված օրինակի, կլասին է տալիս **հատկություն**, որը սահմանափակի օբյեկտների ստեղծման քանակը: Մինգլտոն փատերնը ստանալու համար կլասի կոնստրուկտորը պետք է լինի **փակ**: Այդ դեպքում արգելված է ստեղծել կլասի օբյեկտ արտաքին միջավայրից, սակայն փակ կոնստրուկտորով կլասները կարելի է օգտագործել **ստատիկ** անդամներով:
- Մինգլտոնը պատերների կազմում ամենապարզն է: Այն հնարավորություն է տալիս ստեղծել այսպես կոչված շաբլոն, որը ապահովում է տվյալ կլասի **միայն մեկ օբյեկտի** առկայություն:



- Մինգլտոն պատերնը ստեղծվում է հետևյալ ձևով: Փակվում է կոնստրուկտորը, որը արգելում է կլասի օբյեկտի ստեղծումը: Կատարվում է **public static** մեթոդով կլասի օբյեկտի ստեղծում եւ պայմանական օպերատորի միջոցով կազմակերպվում է այնպես, որպեսզի ստեղծվի կլասի միայն մեկ օբյեկտ:
- Նշենք պարզեցված օրինակ, որը ապահովում է մեկ օբյեկտի առկայությունը, սակայն լինելով պարզ ունի թերություն, միշտ աշխատում է կոնստրուկտոր եւ ստեղծվում է օբյեկտ նախկին անունով:

```
using System;
class A
{
    public static void ff()
    {
        A ob = new A();
    }
    A()
    {
        Console.WriteLine("constructor");
    }
}
class M
{
    static void Main()
    {
        //A ob = new A(); error !
        A.ff();
        A.ff();
        A.ff();
    }
}
```

- Նշված թերությունից խուսափելու համար բերենք “օրիգինալ” Մինգլտոն պաթերնը, որտեղ օգտագործվում է **if** պայմանական օպերատորը եւ **null** թույլատրելի հղումը:

```
using System;
class A
{
    static A instance = null;
    public static A ff()
    {
        if (instance == null)
        {
            instance = new A();
        }
        return instance;
    }
    A()
    {
        Console.WriteLine("constructor");
    }
}
class M
{
    static void Main()
    {
        A a = A.ff();
        A b = A.ff();
        A c = A.ff();
    }
}
```

31. this հղում

[Троелсен-201; Troelsen-178; Шилдт-174]

- **this** կարելի է անվանել միջոց, որը հղում է օբյեկտին: Կամ այլ խոսքով, պահում է օբյեկտի հասցեն:
- Կլասի ցանկացած օրինակ (օբյեկտ) դիմամիկ դեկլարվող հիշողությունում իր առանձին տեղն է զբաղեցնում: Մեթոդների համար կա հիշողության առանձին տեղ և գոյություն ունի միայն մեթոդի մեկ օրինակ: Այսպիսով ստացվում է այնպես, որ մեթոդը կարող է աշխատել տարբեր օբյեկտների տվյալների հետ: Երբ անհրաժեշտ է իմանալ, թե մեթոդը ստույգ որ օբյեկտի հետ գործ ունի, օգտագործվում է **this**: Մեթոդի պարամետրերի ցուցակում այն կա, բայց չի նշվում:
- **this** – ով կարելի է մեթոդի լոկալ փոփոխականի փոխարեն դիմել կլասի նույն անունով անդամ փոփոխականին: **this** – ով կարելի է փոխանցել ընթացիկ օբյեկտը այլ մեթոդի պարամետրին: **this** – ը օգտագործվում է **ինդեքսատորների** և **ընդլայնված մեթոդների** կազմակերպման ժամանակ: (**կհ**)
- **this** – ը կարող է կատարել լրացուցիչ բազմաթիվ գործողություններ: Ուղղակի պետք է հիշել, կամ այն ցույց է տալիս **հղում** (հիշողության հասցե), կամ հանդես է գալիս որպես **հատուկ միջոց** (հատուկ կամ հիմնային բառ) տարբեր ծրագրային հատվածներ կազմակերպելու համար:
- ❖ Փորձենք առանց **this** – ի մեթոդի լոկալ փոփոխականի փոխարեն դիմել կլասի նույն անունով անդամ փոփոխականին:

```
class A
{
    public int a;
    public void f(A ob)
    {
        int a;
        a = 55;
        ob.a = 44;
        System.Console.WriteLine(a);
        System.Console.WriteLine(ob.a);
    }
}
```

```
class M
{
    static void Main()
    {
        A ob = new A();
        ob.f(ob);
        System.Console.WriteLine(ob.a);
    }
}
```

- Այժմ նույնը կատարենք **this** – ով

```
class A
{
    int a;
    public void ff()
    {
        int a;
        a = 55;
        this.a = 44;
        System.Console.WriteLine(a);
        System.Console.WriteLine(this.a);
    }
}

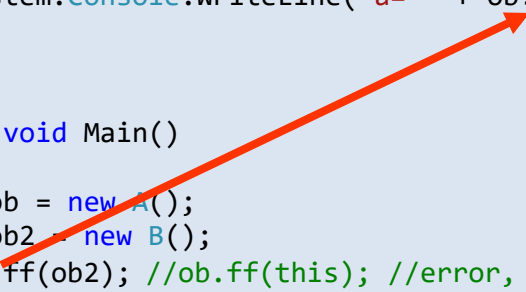
class M
{
    static void Main()
    {
        A ob = new A();
        ob.ff();
    }
}
```

32. this որպես մեթոդի արգումենտ

[Liberty-66; Либерти-98; albdarb]

- `this` –ը թույլատրվում է գրել նաև մեթոդի արգումենտում: Օրինակում երևում է, որ **A** կլասի `ff()` մեթոդը ստանում է **B** կլասի օբյեկտ, որին ուղղեկցող **A** կլասի հղումը տեղադրվում է `this` –ի մեջ: Քանի որ **A** կլասի `ff()` մեթոդը օգտագործում է **B** կլասի `ff2()` մեթոդը եւ նա պարտադրում է **A** կլասի օբյեկտ, ապա կարելի արդեն `this` – ի մեջ տեղակայված **A** կլասի օբյեկտը օգտագործել եւ այն փոխանցել արգումենտի միջոցով:

```
class A
{
    public int a = 77;
    public void ff(B bb)
    {
        bb.ff2(this); // ok, որովհետև ff() –ը static չէ և կարող է օգտագործել this
    }
}
class B
{
    public void ff2(A ob)
    {
        System.Console.WriteLine("a= " + ob.a);
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        B ob2 = new B();
        ob.ff(ob2); //ob.ff(this); //error, որովհետև Main() –ը static է
    }
}
```



33. Կոնստրուկտոր և this (Constructor Chaining)

[Троелсен-202; Troelsen-179; Нейгел-128; Шилдт-245]

- Կարելի է մի կոնստրուկտորի միջոցով կանչել այլ կոնստրուկտոր, **կողի ավելցուկից** խուսափելու համար: Սա `this`–ի օգտագործման եւս մի հնարավորություն է, սակայն այն օգտագործվում է որպես “**հանգույցային**” (հատուկ) բառ: Կոնստրուկտորների ցիկլային միմյանց կանչը բերում է թարգմանման սխալի:

```
class A
{
    public int n;
    public string anun;
    public float pi;
    public A()
    {
        anun = "Ani";
        pi = 3.14f;
    }
    public A(int nn) : this()
    {
        n = nn;
    }
}
class M
{
    static void Main()
    {
        A ob = new A(777);
        System.Console.WriteLine(ob.pi + " " + ob.anun + " " + ob.n);
    }
}
```

34. Զանգվածներ, Անհամաչափ զանգված

[Шилдг -139; Троелсен-150; Troelsen-111; Нейгел- 184; Нэш-248; Рихтер-416]

- Զանգվածը տվյալների կազմակերպման այնպիսի ձև է, երբ անհրաժեշտ է միևնույն տիպի տվյալները պահել **համարակալված** (կոլեկցիայի) ձևով: Զանգվածների հետ աշխատանք կատարվում է **ինդեքսով**: Հայտարարման ձևը՝

```
int [ ] myArray = new int [5];
```

 միաչափ զանգված (վեկտոր)

```
int [ ] myArray = { 2, 1, 6, 8, 3 };
```

 միաչափ զանգվածի հայտարարում եւ վերագրում

```
int [,] myArray = new int [8,8];
```

 երկչափ զանգված (մատրիցա)

```
int [,,] myArray = new int [8,8,8];
```

 բազմաչափ եռաչափ զանգված (խորանարդ)

- Զանգվածը հայտարարման ժամանակ ընդունում է ըստ տիպի **լռելիության** արժեքներ: Զանգվածի սահմանները գտնվում են հսկման տակ եւ չափը նշվում է **Length** հատկանիշի միջոցով:

❖ Անհամաչափ զանգված [Троелсен-153; Troelsen-114; Нейгел-188]

- Թույլատրվում է երկչափ զանգվածի (մատրիցայի) տողերի վեկտորների չափերը լինեն տարբեր: Այն կոչվում **անհամաչափ** զանգված: Օգտագործվում է հատուկ խնդիրների դեպքում հիշողության օպտիմալ կազմակերպման համար: Այդ դեպքում պետք է հայտարարման ժամանակ **ֆիքսել** տողերի չափերը եւ ամեն տողում հայտարարել **կամայական չափի** նոր վեկտոր:

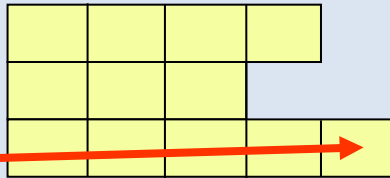
```
int[ ][ ] AA = new int[3][ ];
```

```
AA[0] = new int[4];
```

```
AA[1] = new int[3];
```

```
AA[2] = new int[5];
```

```
AA [2][4] = 44;
```



```
// երկչափ անհամաչափ զանգված
```

```
int[ ][ ] AA = new int[5][ ];
for (int i = 0; i < AA.Length; i++)
    AA[i] = new int[i + 8];
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < AA[i].Length; j++)
        Console.Write(AA[i][j]);
    Console.WriteLine();
}
Console.WriteLine(AA.Length); // 5
```

```
// եռաչափ անհամաչափ զանգված
```

```
int[ ][ ][ ] AAA = new int[2][ ][ ];
AAA[0] = new int[2][ ];
AAA[1] = new int[3][ ];
AAA[0][0] = new int[4];
AAA[0][1] = new int[5];
AAA[1][0] = new int[6];
AAA[1][1] = new int[7];
AAA[1][2] = new int[8];
foreach (int[ ][ ] arv in AAA)
{
    foreach (int[ ] v in arv)
    {
        foreach (int x in v)
        {
            Console.Write(x + " ");
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}
Console.ReadKey();
```


35. foreach – ցիկլ

[Троелсен-140 ; Troelsen – 95; Нейсел-197]

- **foreach** -ը ցիկլի կազմակերպման ձև է, որը հնարավորություն է տալիս կուտակիչից՝ զանգվածից կարդալ հաջորդաբար ինֆորմացիան, **սկզբից մինչև վերջ**: **foreach** ցիկլի միջոցով հնարավոր չէ կատարել զանգվածի անդամներին **արժեքավորում**: **for** -ի հետ համեմատած պետք չէ ինիցիալիզացնել ցիկլի փոփոխական, պետք չէ ստուգել բուլիան արտահայտություն եւ ցիկլի փոփոխականը ինքնամեծացնել: **foreach** -ի կիրառությունը հարմար է, երբ **ընթերցում ենք** զանգվածը կամ կոլեկցիան: Այն նաև մեծ կիրառություն ունի **սովյալների բազաներից** արժեքներ ստանալու համար:
- **foreach** -ը թարգմանության ժամանակ ներքին կարգով վերածվում է **while** ցիկլի իր **համարակալման** յուրահատուկ կառույցով: Այն մյուս ցիկլերի նման չի թարգմանվում մեքենայական կոդի պայմանական օպերացիայի (**if**) եւ թռիչքային հրամանի (**goto**):
- **foreach** -ը կարող է աշխատել **անհամաչափ** բազմաչափ զանգվածների հետ, իսկ **սովորական** մատրիցաների կամ բազմաչափ զանգվածների հետ **չի աշխատում** (եթե անհրաժեշտ է ըստ տողերի վերձանում): Միայն որպես **վեկտոր** է ընկալվում որպես համաչափ զանգված:

```
using System;
int[] ar = new int[8];
for (int i = 0; i < 8; i++)
    ar[i] = i;
//Console.WriteLine();
//for (int i = 0; i < 8; i++)
//    Console.Write(ar[i]+" ");
foreach (int v in ar)
    Console.Write(v + " ");
Console.WriteLine();
//int[,] AA = new int[4, 4]; // error
int[][] AA = new int[3][]; // Jagged Arrays
AA[0] = new int[4] {7,3,9,1};
AA[1] = new int[3] {6,2,5 };
AA[2] = new int[5] {8,4,0,1,5};
foreach(int [] arv in AA)
{
    foreach(int v in arv)
        Console.Write(v+" ");
    Console.WriteLine();
}
// Console.WriteLine(AA.Length); // 3 ??
Console.ReadKey();
```

❖ Using Indices and Ranges (New 8.0) Core [Troelsen – 118; Olsson-100; Albahari-13] N

- (C#8.0) Core վերսիայից սկսած **foreach** -ին թույլատրվում է կուտակիչից կարդալ ոչ միայն ամբողջական, այլ նաև կուտակիչի որոշ հատված:

```
using System;
int[] ar2 = { 1, 2, 3, 4, 5 };
Console.WriteLine(ar2[^2]); // 4
System.Range r = 1..3;
foreach (int v in ar2[r])    // or // foreach (int v in ar2[1..3])
    Console.Write(v);        // 23
Console.WriteLine();
for (int i = 1; i <= ar2.Length; i++)
{
    Index idx = ^i; //
    Console.Write(ar2[idx] + " ");
}
```

36. System.Array կլաս

[Троелсен-155; Troelsen-117; Либерти-192]

- Եթե հայտարարվում է զանգված, ապա այն կարող է օգտվել **Array** կլասի բոլոր անդամներից:
- ✓ **Length** – Հատկանիշ է, որը վերադարձնում է զանգվածի ընդհանուր **էլեմենտների քանակը**:
- ✓ **GetLength()** – Վերադարձնում է բազմաչափ զանգվածի տվյալ չափի **էլեմենտների քանակը**:
- ✓ **Rank** – Հատկանիշ է, որը ցուցադրում է բազմաչափ զանգվածի չափերի քանակը:
- ✓ **BinarySearch()** – Կատարում է փնտրում **սորտավորված** միաչափ զանգվածում և վերդարձնում է **վերջին** հայտնաբերված անդամի համարը:
- ✓ **IndexOf()** – Ստատիկ մեթոդ է, որը կատարում է էլեմենտի փնտրում միաչափ **չսորտավորված** զանգվածում և վերադարձնում է **առաջին** հայտնաբերված անդամի համարը:
- ✓ **LastIndexOf()** – Ստատիկ մեթոդ է, որը կատարում է էլեմենտի փնտրում միաչափ **չսորտավորված** զանգվածում և վերադարձնում է **վերջին** հայտնաբերված անդամի համարը:
- ✓ **Sort()** – Ստատիկ մեթոդ է, որը կատարում է **սորտավորում** միաչափ զանգվածում:
- ✓ **Reverse()** – Ստատիկ մեթոդ է, որը միաչափ զանգվածում կատարում է էլեմենտների փոխանակում **“ռեվերս”** ձևով (վերջին անդամը դարձնում է առաջին, նախավերջինը երկրորդ և այդպես շարունակ):
- ✓ **Clear()** – Ստատիկ մեթոդ է, որը զանգվածի էլեմենտների **քանակը զրոյացնում** է:
- ✓ **Copy()** – Ստատիկ մեթոդ է, որը մի զանգվածի էլեմենտների խումբը վերագրում է մի այլ զանգվածի:
- ✓ **SetValue()** – Վերագրում է զանգվածի տվյալ էլեմենտին արժեք:
- ✓ **IsFixedSize** – Հատկանիշ է, որը տեղեկացնում է կուտակիչի չափը **անփոփոխելի** է թե ոչ:

```
int N = 10;    int[] ar = new int[N];
Console.WriteLine("length = " + ar.Length); // 10
Console.WriteLine("rank = " + ar.Rank);      // 1
int[,] ar2 = new int[4,6];
Console.WriteLine("length = " + ar2.Length); // 24
Console.WriteLine("rank = " + ar2.Rank);     // 2
Console.WriteLine(ar2.GetLength(0));         // 4
Console.WriteLine(ar2.GetLength(1));         // 6
for (int i = 0; i < ar.Length; i++)
    ar[i] = i + 5;
foreach (int x in ar)
    Console.Write(x + " "); // 5 6 7 8 9 10 11 12 13 14
Array.Clear(ar, 2, 4);
ar.SetValue(99, 2); // ar[2]=99;
foreach (int x in ar)
    Console.Write(x + " "); // 5 6 99 0 0 0 11 12 13 14
Array.Sort(ar);
foreach (int x in ar)
    Console.Write(x + " "); // 0 0 0 5 6 11 12 13 14 99
ar[2] = 14;    ar[5] = 14;
foreach (int x in ar)
    Console.Write(x + " "); // 0 0 14 5 6 14 12 13 14 99
Console.WriteLine("binary = " + Array.BinarySearch(ar, 14)); // 8
Console.WriteLine("last = " + Array.LastIndexOf(ar, 14));    // 8
Console.WriteLine("indexOf = " + Array.IndexOf(ar, 14));      // 2
Array.Reverse(ar);
foreach (int x in ar)
    Console.Write(x + " "); // 99 14 13 12 14 6 5 14 0 0
int[] ar3 = new int[10];
Array.Copy(ar, ar3, 5);
foreach (int x in ar3)
    Console.Write(x + " "); // 99 14 13 12 14 0 0 0 0 0
Console.WriteLine(ar.IsFixedSize); // True
System.Collections.ArrayList arl= new System.Collections.ArrayList(); // C#2
Console.WriteLine(arl.IsFixedSize); // False // arl.Add(55); // arl.Count; // arl.Capacity;
```

37. `const` հաստատուն, `readonly` հաստատուն

❖ `const` հաստատուն: [Рихтер-210]

- Եթե անհրաժեշտ է փոփոխականի արժեքը դարձնել անփոփոխելի, ապա օգտագործվում է `const` հաստատունը: `const` կարող է լինել եւ կլասի տվյալների դաշտը եւ լոկալ փոփոխականը: `const` հաստատունի արժեքը հայտնի է լինում թարգմանման ժամանակ եւ պահպանվում է հիշողության հատուկ մասում: Չի թույլատրվում `const` հաստատունը նշել որպես մեթոդի պարամետր կամ վերադարձվող արժեք: `const` հաստատունը օգտագործվում է ստատիկ անդամի նման, այսինքն կլասի անունով:

❖ `readonly` հաստատուն [Троелсен-237; Troelsen-218; Нейгел-129; Нэш-63]

- Շատ դեպքերում հաստատունի արժեքը անհրաժեշտ է լինում վերագրել օգտագործման ժամանակ, այլ ոչ թե թարգմանման: Այդ դեպքում ինիցիալիզացիան կատարվում է կանստրուկտորում եւ կարող է կատարվել մեկից ավել: Ի տարբերություն `const` անդամի, `readonly` անդամին դիմում ենք օբյեկտով: `readonly` հաստատունը կարող է լինել ստատիկ:

```
A ob = new A();
Console.WriteLine(ob.ScreenWidth);
Console.WriteLine(ob.ScreenHeight);
// ob.ScreenHeight = 8888; //error!!
```

```
class A
{
    public readonly int ScreenWidth = 99; //ok!!
    public readonly int ScreenHeight;
    public A()
    {
        this.ScreenWidth = 1024;
        this.ScreenHeight = 768;
    }
}
```

- ❖ (Ապացույցը `Framework` -ով): Օրինակ, որը ապացուցում է, որ `const` -ը արժեքավորվում է թարգմանման ժամանակ, իսկ `readonly` -ն կատարման: Պետք է հաշվի առնել, որ GC -ն ի վիճակի չէ հայտնաբերել `const` -ի առկայությունը, քանի որ GC -ն աշխատում է օբյեկտների հետ: Պետք է հաշվի առնել, որ `A ob=new A();` հրամանը նույնպես տեղ է ավելացնում RAM -ում: Նշենք, որ `GetTotalMemory(true)` հրամանը մաքրում է ավելորդ ռեսուրսները, նոր ցուցադրում զբաղեցրած հիշողությունը:

```
Console.WriteLine(GC.GetTotalMemory(true));
Console.WriteLine(GC.GetTotalMemory(true));
A ob = new A();
Console.WriteLine(GC.GetTotalMemory(true));
```

```
class A
{
    readonly double d = 19; // Ազդում օբյեկտի զբաղեցրած հիշողության թվի վրա
    const double d2 = 19; // Չի ազդում օբյեկտի զբաղեցրած հիշողության թվի վրա
    // enum ee {red, blue = 7, black}; // Չի ազդում օբյեկտի զբաղեցրած հիշողության թվի վրա
}
```

❖ Read-Only Structs (New 7.2); Read-Only Members (New 8.0) [Troelsen - 145] N (no video)

- Կարելի է ստրուկտուրան դարձնել `readonly`: Այդ դեպքում դաշտերի փոփոխելը հնարավոր է միայն պարամետրով կոնստրուկտորով: Նաև թույլատրվում առանձին անդամներին ընտրությամբ դարձնել `readonly`:

```
S ob =new S(57);
Console.WriteLine(ob.x);
readonly struct S
{
    public readonly int x;
    public S(int a)
    {
        x = a;
    }
}
```

38. Հաստատուն **enum** (թվարկում)

[Шилдг-397; Троелсен-170; Troelsen-135; Нейсел-90]

- Թվարկումը չափային տիպ է, որը կարող է պահպանել թվային հաստատուններ սիմվոլային անվանումների միջոցով: Թվարկման էլեմենտների տիպեր կարող են լինել **byte**, **short**, **int**, **long**: Լռելիությամբ ընդունում է **int**: Թվարկումը կարող է ներդրվել կլաստում, կամ նրանից դուրս: Ինչպես հաստատունը, թվարկմանը նույնպես դիմում են կլասի անունով, կետով ավելացնելով անդամները:

```
enum ee2 : short { a, b, cc55 };
class A
{
    public enum ee { red, blue = 7, black } ;
}
class M
{
    public static void Main()
    {
        System.Console.WriteLine((int)A.ee.red);
        System.Console.WriteLine((int)A.ee.blue);
        System.Console.WriteLine((int)A.ee.black);
        System.Console.WriteLine(ee2.a);
        System.Console.WriteLine((int)ee2.b);
    }
}
```

❖ **System.Enum** կլաս

[Троелсен-173; Troelsen-138]

- Շարի լեզուն շարունակում է իր **սովորույթը**: Եթե կա գաղափար, ապա առաջարկում է նաև նրան ծառայող կլաս: Թվարկումը կարող է օգտվել **Enum** կլասից, որի անդամներից են.
- ✓ **GetUnderlyingType()** – վերադարձնում է օգտագործվող տիպը:
- ✓ **Format()** – թվարկման արժեքը նշված ֆորմատով փոխարկում է սիմվոլային համարժեքության:
- ✓ **GetValues()** – վերադարձնում է **զանգված**, որը պարունակում է թվարկման անդամները:
- ✓ **IsDefined** – հասկանիշ, որը հնարավորություն է տալիս իմանալ սիմվոլային տողը հանդիսանում է արդյոք թվարկման էլեմենտ:
- ✓ **GetName()** – վերադարձնում է անունը տված թվարկման արժեքի համար:

```
using System;
enum ee : byte
{
    red, blue = 57, black
}
class M
{
    public static void Main()
    {
        Console.WriteLine(Enum.GetUnderlyingType(typeof(ee)));
        Console.WriteLine(Enum.GetName(typeof(ee), 0));
        Console.WriteLine(Enum.IsDefined(typeof(ee), "red"));
        Console.WriteLine(Enum.Format(typeof(ee), ee.blue, "x")); //"d"-decimal //"G" - string
        Array ob = Enum.GetValues(typeof(ee));
        Console.WriteLine(ob.Length);
        foreach (ee k in ob)
            Console.WriteLine((int)k);
    }
}
```

.....

39. string տիպ

[Heйpeл-266; Tpoelceн-120; Troelsen - 75]

- `string` տիպը իրենից ներկայացնում է `char` տիպի սիմվոլների վեկտոր: Այն “immutable” է (անփոփոխելի), որը նշանակում է արժեքավորումից հետո փոփխման ենթակա չէ որպես սիմվոլների վեկտոր, սակայն ունի առանձնահատկություն, `string` – ի հետ աշխատանքը կատարվում է նրա կոպիայի հետ և որի շնորհիվ կարողանում է կատարել տրված արժեքների փոփոխություն:
- `string` –ը համարվում է `String` կլասի պսևդոդասանունը, որը `sealed` կլաս է (`կհ`):
- `string` –ի արժեքավորման հիմնական ձևը՝ `string ss = "barev !";`
- `string` –ը չունի լռելիությամբ կոնստրուկտոր:
`string s = new string(); // error`
- `string` –ը մի քանի պարամետրով կոնստրուկտորների հետ մեկտեղ ունի նաև `char` տիպի վեկտոր պարամետրով կոնստրուկտոր՝
`char[] ch = new char[10];`
`string s = new string(ch); //ok`
- `String` կլասի առաջարկվող **որոշ** անդամներ.
 - ✓ `Length` – Նշում է տողի սիմվոլների քանակը:
 - ✓ `Compare()` – Համեմատում է երկու տողեր, 0 (նույնը), -1 (սիմվոլը վերև), 1 (սիմվոլը ներքև) արդյունքով:
 - ✓ `Concat()` – Ստեղծում է նոր տող մեկ կամ ավելի տողերից – “կանկատենացիա”:
 - ✓ `Copy()` – Ստեղծում է նոր տող որպես նշված տողի կոպիա:
 - ✓ `Equals()` – Գերբեռնված `Equals()`, որը համեմատում է 2 տողերի նույնությունը:
 - ✓ `Join()` – Միացնում է տողերի վեկտորի անդամները միջնորդ տանաջատիչով:
 - ✓ `EndsWith()` – Նշում է արդյոք տողը վերջանում է նշված ենթատողով:
 - ✓ `IndexOf()` – Վերադարձնում է համընկնող ենթատողի առաջին հանդիպող համարը:
 - ✓ `LastIndexOf()` – Վերադարձնում է համընկնող ենթատողի վերջին հանդիպող համարը:
 - ✓ `Insert()` – Նրդրվում է ենթատող:
 - ✓ `Remove()` – Ջնջում է նշված դիրքից սիմվոլների քանակ:
 - ✓ `Substring()` – Տողից առանձնացնում է ենթատեքստ:
 - ✓ `ToLower()` – Վերադարձնում է տողի կոպիան փոքրատառերով:
 - ✓ `ToUpper()` – Վերադարձնում է տողի կոպիան մեծատառերով:
 - ✓ `Split()` – Տողը մասնատվում է ենթատողերի, օգտագործելով պարամետրում ներկայացված տարանջատիչ:
- Եթե երկու `string` փոփոխական ունեն նույն պարունակությունը, ապա նրանց հղումները թարգմանիչը նույնացնում է: Դա հնարավոր է (կամ անհրաժեշտ է), քանի որ `string` –ը աշխատում է կոպիայի հետ: Օգտագործենք `ReferenceEquals(s1, s2)` հրամանը, որը համեմատում է երկու փոփոխականների հասցեները, այն անհասկանալիորեն տալիս է `true`: Եթե վերագրենք տարբեր արժեքներ. `ss1 = "a"; ss2 = "a2";` ապա միայն այդ դեպքում `ReferenceEquals(ss1, ss2)` –ը կտա `false` (նայել վիդեո բացատրություն):

```
using System;
```

```
class M
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        string ss1 = "a";
```

```
        string ss2 = "a";
```

```
        Console.WriteLine(ReferenceEquals(ss1, ss2)); // true
```

```
        //Console.WriteLine(Equals(ss1, ss2)); // true
```

```
        //Console.WriteLine(ss1.Equals(ss2)); // true
```

```
        //Console.WriteLine(ss1 == ss2); // true
```

```
        string s1 = "abcd";
```

```

String s2 = "ABCD";
Console.WriteLine(s1[2]); // ok // c
//s1[2] = 'X'; //error
for (int i = 0; i < s1.Length; i++)
    Console.Write(s1[i]); // abcd
Console.WriteLine(string.Compare(s1, s2)); // -1
Console.WriteLine((int)('a')); // 97
Console.WriteLine((int)('A')); // 65
Console.WriteLine(string.Compare(s1, s2, true)); // 0
Console.WriteLine(string.Concat(s1, s2)); // abcdABCD
Console.WriteLine(s1 + s2); // abcdABCD
Console.WriteLine($"{s1}{s2}"); // abcdABCD // C#6
string s = string.Copy(s2);
Console.WriteLine(s); // ABCD // s = s2;
s = "...";
string[] a = { "alb", "ert", "dar", "bin", "yan" };
s1 = string.Join(s, a);
Console.WriteLine(s1); // alb...ert...dar...bin...yan
Console.WriteLine(s1.EndsWith("an")); // true
Console.WriteLine(s2.Insert(2, "barev")); // ABbarevCD
s = "kkwwkww.albdawrbww.com";
Console.WriteLine(s.IndexOf("ww")); // 2
Console.WriteLine(s.LastIndexOf("ww")); // 16
Console.WriteLine(s1.Remove(4, 20)); //alb.yan
Console.WriteLine(s2.Substring(1, 3)); //BCD
Console.WriteLine(s2.ToLower()); // abcd
Console.WriteLine(s1.ToUpper()); // ALB...ERT...DAR...BIN...YAN
s = "www#alb#darb com";
string[] ss = s.Split('#');
for (int i = 0; i < ss.Length; i++)
    Console.WriteLine(ss[i]); // www    alb    darb com
s = 1 + 2 + "g" + 3 + 4;
Console.WriteLine(s); // 3g34
}
}

```

- Հաջորդ օրինակը ցույց է տալիս, որ հիշողություն չի ավելանում `s = "aa"` երկրորդ հրամանի ավելացման դեպքում, քանի որ `string` փոփոխականը աշխատում է **կոպիայի** հետ: Ծրագիրը կարելի է աշխատացնել “կոմենտներ” դնելով և հիշողության զբաղեցրած ծավալը ստուգելով: (**Framework**)

```

using System;
class A
{
    string s = "12";
    public void f()
    {
        s = "aa";
        s = "bb";
        s = "aa"; //
    }
}
class Program
{
    static void Main()
    {
        A ob = new A();
        ob.f();
        Console.WriteLine("memory=" + GC.GetTotalMemory(true));
    }
}

```

❖ String.Format() մեթոդ

[Шилдг – 812; Нейгел -271; Covaci – 150]

- DateTime և TimeSpan [Troelsen - 72]
- Տողի ցուցադրման համար կարելի է օգտվել Format() մեթոդից, տվյալների տարբեր տեսքի հաղորդագրություններ ստանալու համար:

```
using System;
double temp = 15624.567;
int temp2 = 47;
Console.WriteLine(String.Format($"{temp:N2}")); // 15,624.57
Console.WriteLine(String.Format($"{temp:e}")); // 1.562457e+004
Console.WriteLine(String.Format($"{temp2:x}")); // 2f
Console.WriteLine(String.Format($"{temp2:c}")); // $47.00
Console.WriteLine(String.Format($"{temp2:p}")); // 4,700.00%
Console.WriteLine(String.Format("{0:10,0##.##00}", 23.6)); // 10,023.600
DateTime dt = DateTime.Now;
Console.WriteLine(dt); // 4/1/2020 3:48:29 PM
Console.WriteLine(String.Format($"{dt:d}")); // 4/1/2020
Console.WriteLine(String.Format($"{dt:D}")); // Wednesday, April 1, 2020
Console.WriteLine(String.Format($"{dt:t}")); // 3:48 PM
Console.WriteLine(String.Format($"{dt:T}")); // 3:48:29 PM
Console.WriteLine(String.Format($"{dt:U}")); //Wednesday, April 1, 2020 11:48:29 AM // Գրինվիչ
Console.WriteLine(String.Format("{0:hh:mm:ss - MM yyyy}", dt)); // 03:48:29 - 04 2020
Console.WriteLine(String.Format("{0:MMM yyy}", dt)); // Apr 2020
Console.WriteLine(String.Format("{0:MMMM yy}", dt)); // April 20
```

40. System.Text.StringBuilder կլասս

[Троелсен-127; Troelsen – 83]

- Ի տարբերություն `string` տիպի, որը աշխատում է **կոպիայի** հետ, `StringBulder` կլասը հնարավորություն է տալիս աշխատել **բնօրինակի** հետ, որը իր հերթին բերում է էֆեկտիվության տողի **բազմակի փոփոխության** ժամանակ:
- ❖ Օրինակը համեմատում է `StringBuilder` – ի եւ `string`– ի գրանցման **արագությունները**, որի արդյունքում `StringBuilder` –ը ստացվում է շատ **ավելի արագ**:

```
using System;
using System.Text;
class Program
{
    static int iter = 20000;
    static void Main(string[] args)
    {
        DateTime dt = DateTime.Now;
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < iter; i++)
            sb.Append("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n");
        Console.WriteLine(DateTime.Now - dt);
        DateTime dt2 = DateTime.Now;
        string s = "";
        for (int i = 0; i < iter; i++)
            s += "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n";
        Console.WriteLine(DateTime.Now - dt2);
        // StringBuilder sb = new StringBuilder();
        // Console.WriteLine(sb.Length);
        // Console.WriteLine(sb.MaxCapacity);
        // Console.WriteLine(sb.Capacity);
        // sb = new StringBuilder("12345678901234567");
        // Console.WriteLine(sb.Capacity);
        // sb.Append("darb");
        //// sb.Insert(5, "alb");
        //// sb.Remove(1, 3);
        //// sb.Clear();
        // Console.WriteLine(sb.ToString());
    }
}
```

- ❖ Հաջորդ օրինակը ցույց է տալիս, թե ինչպես կարելի է ստուգել զբաղեցրած հիշողությունը `string` –ի եւ `StringBulder` –ի համար: Ստացվում է, որ `string` –ը գործողությունների արդյունքում ավելի մեծ տարածք է վերցնում, եթե չի օգտագործվում `GC.Collect()` հիշողություն մաքրող հրամանը:

```
using System;
using System.Text;
class Program
{
    static void Main()
    {
        string s = "1234567890";
        for (int i = 0; i < 1000; i++)
            s = s + "abcdaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
        Console.WriteLine("memory=" + GC.GetTotalMemory(false)); // GC.GetTotalMemory(true)); low RAM
        //GC.Collect(); // ??
        //StringBuilder sb = new StringBuilder("1234567890");
        //for (int i = 0; i < 1000; i++)
        //    sb.Append("abcdaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
        //Console.WriteLine("memory=" + GC.GetTotalMemory(false));
    }
}
```


41. Main(string [] args) մեթոդ

[Троелсен-100; Troelsen – 51,52,53,55,57; Шилдт-255; Нейсел-95]

- Main մեթոդը կարող է ունենալ (string[] args) պարամետր, նույնիսկ օգտագործել params: Նշված պարամետրը ծառայում է, երբ անհրաժեշտ է ծրագիրը աշխատացնել **հրամանի տողով** (Windows -ում Start -> Run կամ VS2019 Core պրոյեկտից՝ Project->App->Debug->Application arguments =”HighCode”) եւ ցանկություն կա հրամանի տողում ներառել տվյալներ: Տվյալները այդ դեպքում իրենցից ներկայացնում են տողեր, որոնք տարանջատվում են դատարկ սիմվոլով (space): Նշված պարամետրերը ավտոմատ գրանցվում են (string[] args) զանգվածում, որը իր հերթին կարելի է օգտագործել ծրագրում:

- Environment.GetCommandLineArgs() մեթոդը կարող է փոխարինել (string[] args) –ին:

using System;

class M

```
{
    static void Main(string[] args) // public static void Main() // ok
    {
        // args[0] = "code"; // Runtime error
        string[] s;
        // s[0] = "code"; // Compile error
        for (int i = 0; i < args.Length; i++)
            Console.WriteLine(args[i]);
        // string[] s = Environment.GetCommandLineArgs();
        // Console.WriteLine(s[0]); // Project path
        Console.ReadKey();
    }
}
```

❖ Application Error Code սպեցիֆիկացիա (New 9.0) [Troelsen-53] N

- Քանի որ, ծրագիրը թողարկվում է օպերացիոն համակարգի միջոցով, ապա կա հնարավորություն ծրագրի ավարտի ժամանակ կատարվի տվյալների փոխանցում կանչող համակարգին՝ այսինքն օպերացիոն համակարգին: Հիմնականում Main() մեթոդը աշխատում է void վերադարձնող արժեքով, սակայն այն կարող է վերադարձնել int տիպ և C#7.1 –ից սկսած նաև Task<int>, Task տիպեր, որը ապահովում է արժեքի ասինխրոն վերադարձ (զուգահեռ աշխատանք), պահպանելով այլ լեզուների նմանություն և համապատասխանություն: Ըստ պայմանավորվածության, եթե 0 արժեք է վերադարձվում, ապա դա համարվում է ծրագրի բարեհաջող ավարտ և այլ թվերի դեպքում ծրագրը ունի պրոբլեմներ կամ սխալի առկայություն: Համարվում է, եթե նույնիսկ void է վերադարձվում, ապա միևնույն է 0 արժեք է վերադարձվում:
- Windows օպերացիոն համակարգով թողարկման դեպքում ծրագրի վերադարձվող արժեքի կապը պահվում է համակարգային միջավայրի %ERRORLEVEL% փոփոխականում, օգտագործելով ծրագրի պրոցեսի (կհ C#2) ExitCode հատկանիշը:
- Քանի որ, ծրագրի արժեքի վերադարձը կատարվում է միայն ավարտից հետո, ակնհայտ է, որ ծրագիրը իր աշխատանքի ժամանակ **չունի հնարավորություն տեսնել** և մշակել վերադարձվող սխալը:

❖ Top-Level Statements (New 9.0) [Troelsen-52] N

- Թույլատրվում է հրաժարվել Main() մեթոդի ներկայացումից իր կլաստով և այն կոչվում է Top-Level: Կան որոշ առանձնահատկություններ, որնցից են՝
 - ✓ Միայն ծրագրի **մեկ ֆայլ** կարող է օգտվել նրանից:
 - ✓ Ծրագիրը չի կարող ունենալ այլ մուտքի կետ: Եթե գոյություն ունի **Main()**, ապա այն կարող է աշխատել միայն մեկ պայմանում՝ Top-Level տարածքում պետք է **բացակայեն** հրամաններ:
 - ✓ Top-level ծրագիրը չի կարող ընդգրկվել որևէ **անունի տարածքում**:
 - ✓ Այն կարող է օգտվել (string[] args) **արգումենտից**:
 - ✓ **return** օպերատորով կարող է **արժեք վերադարձնել**:
 - ✓ Հայտարարված ֆունկցիաները համարվում են **լոկալ**:
 - ✓ Նոր տիպերի նկարագրությունները պետք է **ավելի ներքև** գտնվեն Top-level –ի հրամաններից:

42. Հատկանիշ (Properties)

[Шилдг-313; Троелсен-224; Troelsen – 201; Рихтер-263]

- **Օբյեկտային կողմնորոշված** ծրագրավորումը պարտադրում է, որպեսզի կլասի անդամ տվյալները լինեն **փակ**, եւ նրանց օգտագործումը կատարել բաց ֆունկցիաների միջոցով: C# -ում այսպիսի աշխատանքը մտցված է բազային համակարգ եւ իրագործվում է **հատկանիշի** (properties) միջոցով: Երբեմն նրան անվանում են **ինտելեկտուալ դաշտեր**, իսկ էությունը ավելի մոտ է **պարզեցված ամեթոդ** գաղափարին: Կամ այլ խոսքով հատկանիշը իրականացված է որպես մեթոդ, սակայն օգտագործվում է որպես դաշտ:
- Հատկանիշները իրագործվում են **set** եւ **get** աքսեսորների միջոցով: **set** – ը հնարավորություն է տալիս տվյալներին **վերագրել** արժեք: Այն օգտագործում է **value** հիմնային հրամանը, որը ներքին փոփոխական է եւ ծառայում է արժեք ստանալու համար: Հատկանիշի օգտագործման համար գրում ենք հատկանիշի անունը, վերագրման օպերատորը եւ անհրաժեշտ արժեքը: **get** – ը հնարավորություն է տալիս **ստանալ** հատկանիշի պարունակությունը: Այստեղ օգտագործվում է **return** հրամանը: Այս դեպքում հատկանիշի անունը պետք է գտնվի վերագրման օպերատորի աջ մասում: Պարզ է, որ ձախ մասը ստանում է հատկանիշի արժեքը: Նշենք, որ հատկանիշը կարող է աշխատել նաև կամ միայն **set** կամ միայն **get** տարբերակով:
- **set** եւ **get** աքսեսորները կարող են **ստանալ թույլատրության մոդիֆիկատորներ** (**private**, **protected**, եւ այլն), սակայն կստանա միայն որևէ մեկը եւ ավելի խստացված: Այլ խոսքով, եթե մոդիֆիկատորը դրված է մեկի վրա, ապա մյուս աքսեսորին չի թույլատրվում մոդիֆիկատոր դնել: Դա պարզ է, քանի որ հատկանիշի գլխավոր մոդիֆիկատորը կկորցնի իր իմաստը: Փորձել:
- Հատկանիշները նույնպես “**ինկապսուլացիայի**” մի ձև են, երբ անհրաժեշտ է տվյալները ծածկել (փակել, **private**) եւ նրանց դիմել բաց (**public**) գործողություններով: (Համեմատում => Գնացինք արաբական աշխարհ աղջիկ ուզելու):

```
class A
{
    string s;
    public string p
    {
        get
        {
            return s;
        }
        set
        {
            s = value;
        }
    }
}

class M
{
    static void Main()
    {
        A ob = new A();
        ob.p = "555";
        string z = ob.p;
        System.Console.WriteLine(z);
    }
}

.....
```

```
// օրինակ մեթոդի միջոցով
```

```
class A
{
    string s;
    public string ff()
    {
        return this.s;
    }
    public void ff2(string ss)
    {
        this.s = ss;
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        ob.ff2("5555");
        string z = ob.ff();
        System.Console.WriteLine(z);
    }
}
```

- ❖ Հատկանիշի առավելություններից է, երբ դաշտի վերագրումը եւ ընթերցումը կատարվում է **տարբեր տիպերով**: Այդ աշխատանքը կատարվում է տիպերի վերափոխարկումով եւ այն կարող է իրականացվել **ֆունային ռեժիմով** (գուգահեր), որը ավելի քիչ է ազդում է ընդհանուր ծրագրի աշխատանքային ընթացքի վրա: (C#2) [Olsson - 92]

```
class A
{
    private byte a;
    public int p
    {
        get { return (int)a; }
        set { a = (byte)value; }
    }
}
```

- ❖ **Properties As Expression-Bodied Members** (New 7.0) [Troelsen-204]

```
class A
{
    int myage;
    public int age
    {
        get => myage;
        set => myage = value;
    }
}
```

- ❖ **Ռեֆակտորինգ**

- ✓ Եթե անհրաժեշտ է արագ ստեղծել բաց դաշտի պարփակում հատկանիշի միջոցով, ապա դա կարելի է կատարել **string s** դաշտին սեխմելով աջ քլիկ եւ ընտրել **Quick Actions**:

```
class A
{
    string s;
    public string S { get => s; set => s = value; }
}
```

43. Ավտոմատ հատկանիշներ

[Троелсен-230; Troelsen – 209; Шилдт-318; Нэш -69; Нейгел-124]

- Եթե հատկանիշը օգտագործվում է միայն դաշտերին արժեք տալու եւ կարդալու համար, ապա նույն աշխատանքը կատարելու համար նախատեսված է ավելի պարզ ձև, որը կոչվում է ավտոմատ հատկանիշներ:

```
A ob = new A();
Console.WriteLine(ob.pp);
ob.pp = 999;
Console.WriteLine(ob.pp);
Console.WriteLine(ob.pp2);
// ob.x = 88; // error
A obinit = new A{ x = 999 }; // "Init-only"
// obinit.x = 777; // error
```

```
class A
{
    public int pp { get; set; }
    public string pp2 { get; } = "barev";
    public int x { get; init; } = 20; // "Init-only" property (readonly) // C#9 [Albahari-108; Troelsen-215]
}
```

- Հատկանիշում թույլատրվում միայն `get` (C#7):
- Հատկանիշը կարող է ընդունել լռելիությամբ արժեքներ:
- Ավտոմատ հատկանիշը արողջությամբ փոխարինում է դաշտին (կլասի `int a`), ունենալով որոշ առավելություններ: Օրինակ՝ կարելի է տալ թույլատրության մոդիֆիկատորներ կամ միայն `get` -ին կամ միայն `set` -ին: Կամ օրինակ՝ ստանալ որոշ **ատրիբուտներ (կհ)**, որը չի կարող ստանալ դաշտը:

- ❖ `init-only` հատկանիշին “alternative” կարելի է օգտագործել `read-only` հատկանիշ, որը արժեքավորվում է կոնստրուկտորի միջոցով:

```
A ob = new A ("aaa");
Console.WriteLine(ob.pp);
Console.ReadKey();
class A
{
    public string pp { get; }
    public A(string pp2) // alternative to init-only
    {
        this.pp = pp2;
    }
}
```

- ❖ **Refactoring** VS 2017

```
int b;
public int B
{
    get => b; set => b = value; // lambda
}
```

44. operator - Օպերատորների գերբեռնում

[Шилдг-269; Троелсен-422; Troelsen-417; Нейгел-219]

- Կլասների միջոցով ստեղծված օբյեկտների միջև հնարավոր չէ կատարել բազային տիպերի օպերացիաներ առանց **օպերատորների գերբեռնման**, բացի վերագրման օպերատորից եւ `==` համեմատումից: Բազային տիպերի օպերացիաներ հնարավոր է իրականացնել նաեւ **մեթոդների** ծրագրային իրականացումով՝ օգտագործելով պարամետրով օբյեկտ փոխանցման հնարավորությունը՝
`A ob = ob1.ff(ob2);`
- Օպերատորների գերբեռնումը իրականացվում է ներդրված **operator** հրամանի միջոցով՝
`public static A operator + (A ob1, A ob2)`
`{`
`return ob;`
`}`
- ✓ Այդ դեպքում `A ob = ob1+ob2;` հրամանը դառնում է հնարավոր:
- ❖ Նշենք օպերատորների գերբեռնման որոշ հատկություններ
- ✓ **public static** լինելը պարտադիր է: Օպերատորը, կախված թե ինչ օպերացիա է անում՝ ”ունար” թե ”բինար”, կարող է ընդունել մեկ կամ երկու պարամետր: “Ունար” օպերացիայի դեպքում (մասնակցում է մեկ օպերանդ) օպերատորի պարամետրը պետք է ունենա այն տիպը, որի համար որոշվում է: “Բինար” օպերացիայի համար (մասնակցում է երկու օպերանդ)՝ գոնե մեկ պարամետր պետք է լինի որոշվող կլասի տիպը:
- `+=, -=, *=, /=, %=, &=, |=, ^=, <=>, >=>`
- ✓ Նշված կրճատված օպերացիաները գերբեռնվում են ավտոմատ, եթե համապատասխան “բինար” օպերացիաները գերբեռնվել են:
- `==, !=` թույլատրվում է գերբեռնել, սակայն առանց գերբեռնման նույնպես նրանք կաշխատեն որովհետեւ `==` փոխարինում է `object` կլասի `Equals` անդամին: (կանցնենք)
- **Չի թույլատրվում** գերբեռնել հետևյալ օպերատորները՝
`(, [, =, new, default, checked, unchecked, typeof, ->, is, as, =>, ??, sizeof`
- ✓ `=` չի գերբեռնվում, որովհետև արդեն թույլատրվում նույն կլասի օբյեկտները վերագրել:
- ✓ `&&, ||`, չի գերբեռնվում, սակայն որոշ պայմաններ բավարարելու դեպքում այն դառնում է հնարավոր: (տնային) `nt[Шилдг-288]`
- ✓ `()` տիպերի վերափոխարկումը չի գերբեռնվում, սակայն կա նրան համապատասխանող օպերացիաներ `implicit` եւ `explicit`: (կանցնենք)
- ✓ `[]` զանգվածի ինդեքսացիոնա չի գերբեռնվում, որովհետև կա հատուկ միջոց, որը կոչվում է **ինդեքսատոր**:
- Օպերատորի պարամետրում արգելվում է օգտագործել `ref` կամ `out`: (կանցնենք)

```
using System; // overload opererator ++ +
class A
{
    int a = 50;
    double d = 2.22;
    string s = "AniNune ";
    public static A operator +(A op1, A op2)
    {
        A result = new A();
        result.a = op1.a + op2.a;
        result.d = op1.d + op2.d;
        result.s = op1.s + op2.s;
        return result;
    }
    public static A operator ++(A op)
    {
        A result = new A();
        result.a = ++op.a;
        return result;
    }
}
```

```

    }
    public void ff()
    {
        Console.WriteLine(a + " " + s + " " + d);
    }
}
class M
{
    public static void Main()
    {
        A ob1 = new A();
        A ob2 = new A();
        A ob = ob1 + ob2;
        ob.ff();
        ob++;
        ob.ff();
    }
}
//-----
using System; // overload operator == !=
class A
{
    public int a = 50;
    public static bool operator ==(A op1, A op2)
    {
        if(op1.a==op2.a)
            return true;
        else
            return false;
    }
    public static bool operator !=(A op1, A op2)
    {
        if (op1.a != op2.a)

            return true;
        else
            return false;
    }
}
class M
{
    public static void Main()
    {
        A ob1 = new A();
        A ob2 = new A();
        Console.WriteLine(ob1 == ob2);
        ob1.a = 99;
        Console.WriteLine(ob1 == ob2);
    }
}
// -----
using System; // overload operator true false
class A
{
    public int a;
    public static bool operator true (A op)
    {
        if(op.a==0)
            return true;
        else
            return false;
    }
}

```

```

public static bool operator false(A op)
{
    if (op.a!=0)

        return true;
    else
        return false;
}
}
class M
{
    public static void Main()
    {
        A ob = new A();
        ob.a = 10;
        if(ob)
            Console.WriteLine("Ayo");
        else
            Console.WriteLine("Voch");
        ob.a = 0;
        if (ob)
            Console.WriteLine("Ayo");
        else
            Console.WriteLine("Voch");
    }
}
//-----
using System; // overload opererator &
class A
{
    public int a = 50;
    public static bool operator & (A op1,A op2)
    {
        if(op1.a==0 & op2.a==0)
            return true;
        else
            return false;
    }
}
class M
{
    public static void Main()
    {
        A ob1 = new A();
        A ob2 = new A();
        if(ob1 & ob2)
            Console.WriteLine("1");
        else
            Console.WriteLine("0");
        ob1.a = 0;
        ob2.a = 0;
        if (ob1 & ob2)
            Console.WriteLine("1");
        else
            Console.WriteLine("0");
    }
}
// Տնայիններ
// overload operator * / % > <
// overload operator | ! ~ ^ >> <<

```

45. Ինդեքսատոր

[Троелсен-417; Troelsen-411; Шилдт-303]

- **Ինդեքսատորը** հնարավորություն է տալիս կլասի անդամներին դիմել **ինդեքսով**, օգտագործելով [] ինդեքսացիայի փակագծերը: Ինչպես գիտենք C++ - ում զանգվածի ինդեքսացիայի [] էլեմենտը կարելի է գերբեռնել **operator** հրամանով: C# - ում այս նույն էֆեկտին կարելի է հասնել ինդեքսատորի միջոցով, որի պատճառով [] սիմվոլների գերեռնումը **կորցնում** է իր իմաստը: Ինդեքսատորները իրականացվում են **this** -ի միջոցով, որը այստեղ հանդես է գալիս որպես **հատուկ** բառ, այլ ոչ թե հղում: Մյուս կողմից կարելի է ասել օբյեկտի հղում է, քանի որ ինդեքսատորը չի կարող լինել **static** եւ նրան դիմում ենք օբյեկտի ինդեքսի նշումով: Այստեղ գործում է կարծիքի “**երկակիություն**”:

(կոդմուրոշվեք, եթե ցանկանում եք):

```
class A
{
    public string this[int ii]
    {
        get { return }
        set { value }
    }
}
```

- Եթե հատկանիշներին անվանում են **խելացի դաշտեր**, ապա ինդեքսատորներին կարելի է անվանել **խելացի զանգվածներ**: Այլ խոսքով ինդեքսատորի միջոցով կարելի է ապահովել զանգվածների **պարփակվածությունը** (incapsulation)՝ այն փակելով եւ ինդեքսատորով դիմելու ձևով:
- Ինչպես երևում է նկարագրությունից, ինդեքսատորը հատկանիշների նման օգտագործում է **get**, **set** աքսեսորները: Եթե ինդեքսատորը գտնվում է ձախ վերագրման օպերատորից, կանչվում է **set** աքսեսորը: Եթե գտնվում է աջ, ապա **get** -ը: Ինդեքսատորի վերադարձնող տիպը որոշում է եւ **set** -ի եւ **get** -ի տիպը:
- Ինդեքսատորը թույլատրում է կամ **set** -ի կամ **get** -ի օգտագործում եւ հատկանիշների նման կարող է **խստացնել** աքսեսորների թույլատրության մոդիֆիկատորները վերադարձնող տիպի նկատմամբ, սակայն կամ **set** -ում կամ **get** -ում:
- Ինդեքսատորը կարելի է օգտագործել հետևյալ կերպ՝

```
A ob = new A();
ob[0] = "barev"; // set աքսեսորի աշխատանք
Console.WriteLine(ob[0]); // get աքսեսորի աշխատանք
```

```
using System;
```

```
class A
{
    int[] a = new int[5];
    public int this[int ii]
    {
        set
        {
            a[ii] = value;
        }
        get
        {
            return a[ii];
        }
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        for (int i = 0; i < 5; i++)
            ob[i] = i;
        for (int i = 0; i < 5; i++)
            Console.WriteLine(ob[i]);
    }
}
```


❖ Առանց կուտակիչի ինդեքսատոր: [Шилдт-310]

- Պարտադիր չէ ունենալ կուտակիչ ինդեքսատոր օգտագործելու համար: Օրինակում `get` -ում գրվում է ծրագրի ալգորիթմական իրականացում, որը մշակում է ինդեքսը և վերադարձնում արժեք:

```
class A
{
    public int this[int ii]
    {
        get
        {
            if ((ii >= 0) && (ii < 8))
                return ff(ii);
            else
                return -1;
        }
    }
    int ff(int p)
    {
        int result = 1;
        for (int i = 0; i < p; i++)
            result *= 2;
        return result;
    }
}
class M
{
    static void Main()
    {
        A ob = new A();

        for (int i = 0; i < 10; i++)
            System.Console.Write(ob[i] + " ");
    }
}
```

46. Գերբեռնված ինդեքսատոր

[Шилдт-307; Троелсен-420; Troelsen-415]

- Թույլատրվում է գերբեռնել ինդեքսատորը: Նրա ինդեքսը կարող է լինել `char`, `int`, `string`, `bool`, `decimal`, `object` տիպի:

```
using System;
class A
{
    string[] ss={"ani","nune","darbinyan"};
    public string this[int index]
    {
        get
        {
            return ss[index];
        }
        set
        {
            ss[index] = value;
        }
    }
    public string this[string index]
    {
        get
        {
            string s = "";
            for (int i = 0; i < 3; i++)
```

```

        if (ss[i] == index)
            s = ss[i];
        return s;
    }
    set
    {
        for (int i = 0; i < 3; i++)
            if (ss[i] == index)
                ss[i] = value;
    }
}
}
public class M
{
    public static void Main()
    {
        A ob = new A();
        for (int i = 0; i < 3; i++)
            Console.Write(ob[i]+ " ");
        Console.WriteLine();
        ob[0] = "a_index";
        ob["a_index"] = "xxx";
        Console.WriteLine(ob["xxx"]);
        for (int i = 0; i < 3; i++)
            Console.Write(ob[i] + " ");
        Console.WriteLine();
        //Console.WriteLine(ob["nune"]);
    }
}

```

❖ Բազմաչափ ինդեքսատոր

[Троелсен-421; Troelsen-415; Шилдт-310]

❖ Բազմաչափ ինդեքսատոր, այդ այն թույլատրելի է՝ օրինակ:

```

class A
{
    int[,] ar = new int[8,8];
    public int this[int i,int j]
    {
        set
        {
            ar[i,j] = value;
        }
        get
        {
            return ar[i,j];
        }
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        for (int i = 0; i < 8; i++)
            for (int j = 0; j < 8; j++)
                ob[i,j] = i;
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
                System.Console.Write(ob[i, j]+" ");
            System.Console.WriteLine();
        }
    }
}

```

47. Մեթոդի հղող պարամետրեր - ref, out, in

[Шилдг-222; Троелсен-159, 161; Troelsen-125,127; Hэм-63]

- Ինչպես գիտենք մեթոդը կարող է վերադարձնել ներդրված չափային տիպի միայն մեկ արժեք **return** –ի միջոցով: Եթե անհրաժեշտ է վերադարձնել **մեկից ավել արժեքներ**, ապա դա հնարավոր է պարամետրերի չափային տիպերի հղումների միջոցով, օգտագործելով **ref** եւ **out**:
- ref** –ի դեպքում փոփոխականները մեթոդից դուրս պետք է ստանան **նախնական արժեքներ**: C#7 -ից սկսած չափային տիպ վերդարձնող մեթոդները նույնպես կարող են ստանալ **ref**:
- out** –ի դեպքում փոփոխականները մեթոդից դուրս նախնական արժեքների վերագրման անհրաժեշտություն չունեն, սակայն պարտադիր է նախնական արժեքավորումը մեթոդում : C#7 –ում թույլատրվում է արգումենտում տիպի հայտարարում **out** հղման դեպքում, եթե փոփոխականի տիպի նախնական հայտարարությունը հանենք:
- in** –ի դեպքում մեթոդում պարամետրի փոփոխականի արժեքը **չի թույլատրվում փոփոխել**: Այլ խոսքով ստացվում է “readonly” ձև: Այն օգտակար է պարամետրի չափային տիպերի համար, որը արգելում է **տիպի լոկալ կոպիաների գեներացիան**` մանավանդ եթե պարամետրը մաշտաբային հիշողություն գրավող է: Այն օգուտ է նաև հղիչային պարամետրերի համար, որը լոկալ կոպիաներ չի ստեղծում, սակայն ամեն դեպքում, եթե անհրաժեշտ է արգելել արժեքի փոփոխություն: (C#7.2)
- ref** եւ **out** հղող պարամետրերի օգտագործման ժամանակ ավտոմատ ներքին ձևով կատարվում է boxing/unboxing:

❖ Example: ref

```
using System;
// static ref int ff(ref int a) //C# 7.0 (no vidio)
static void ff(ref int a)
{
    a = a * a * a;
    // return ref a;
}
int b = 3;
ff(ref b);
// b=ff(ref b);
Console.WriteLine(b);
```

❖ Example: out

```
using System;
static void ff(out int a)
{
    a = 4;
    a = a * a * a;
}
int b; //
ff(out b);
// ff(out int b); // int in argument C#7.0
Console.WriteLine(b);
```

❖ Discarding out Parameters (New 7.0) [Troelse -127] N

```
using System;
static void f(out int x, out bool y)
{
    x = 4;
    y = true;
}
int a;
bool b;
f(out a, out _);
Console.WriteLine("" + a);
```

❖ **Example: Square equation**

```

using System;
static void qarakusi_havasarum(int a, int b, int c, ref double x1, ref double x2)
{
    x1 = (-b + (Math.Sqrt(Math.Pow(b, 2) - 4 * a * c))) / (2 * a);
    x2 = (-b - (Math.Sqrt(Math.Pow(b, 2) - 4 * a * c))) / (2 * a);
}
double x1 = 0, x2 = 0;
int a=1, b=22, c=3;
if ((Math.Pow(b, 2) - 4 * a * c) >= 0)
    qarakusi_havasarum(a, b, c, ref x1, ref x2);
else
{
    Console.WriteLine("diskriminant < 0");
    return;
}
Console.WriteLine("x1= " + x1);
Console.WriteLine("x2= " + x2);

```

❖ **Using the in Modifier (New 7.2) [Troelse -128]**

```

using System;
static void f(in int b, in A ob, in string s)
{
    // b = 8; //error
    // s ="darb"; //error
    // ob = new A(); //error
    // ob.a = 9; // ok
    System.Console.WriteLine(b+ob.a+s); // ok
}
A ob = new A();
f(5, ob, "alb");

class A
{
    public int a=7;
}

```

❖ Միայն **կոնստրուկտորի** ներսից օգտագործվող մեթոդը կարող ունենալ պարամետրում **ref**, որը օգտվում է **readonly** դաշտից [Հեմ-63]:

- **ref** կարող է ունենալ նաև **լոկալ** փոփոխականը [C#7]: N

```

using System;
A ob = new A();
Console.WriteLine(ob.x);
int a = 99;
ref int b = ref a; // C#7 [Albahari-67] N (no video)
Console.WriteLine(b);
a = 22;
Console.WriteLine(b);

class A
{
    public readonly int x=886;
    public A()
    {
        x = 111;
        x = 999;
        ff(ref x);
    }
    void ff(ref int r)
    {
        r=555;
    }
}

```

❖ Constructors with **out** Parameters (New 7.3) [Troelsen-176] N (no video)

```
int b;
A ob = new A(out b);
Console.WriteLine(b);
class A
{
    public A(out int a)
    {
        a = 9;
    }
}
```

❖ **ref** - ? Ternary Operator (C#7.2) [Troelsen-101] N (no video)

```
int a = 4;
int b = 6;
ref int c = ref ((1 > 0) ? ref a: ref b); //
a = 2;
Console.WriteLine(c); // 2
// int c2 = (1 > 0) ? a : b;
// Console.WriteLine(c2); // 4
```

❖ Using **ref** Structs (New 7.2) [Troelsen-146] N

- Թույլատրվում է **ref** օգտագործել ստրուկտուրաների համար: Այն ստիպում է ստրուկտուրայից օգտվողներին **ստրուկտուրան պահել ստեկում**: Պարզ է այդ դեպքում հղիչային տիպ կլասը չի կարող ունենալ **ref** տիպի ստրուկտուրա կոմպոզիցիայի կամ ագրեգացիայի ձևով, որովհետև **ref**-վ արգելված է կույտից(heap) դիմել ստեկի տարածք:
- Որոշ սահմանափակումներ՝
 - ✓ Չեն կարող իրականացնել ինտերֆեյս՝ այսինքն ժառանգել ինտերֆեյս:
 - ✓ Չեն կարող վերագրվել **dynamic** կամ **object** տիպերին:
 - ✓ Չեն կարող օգտագործվել non-ref ստրուկտուրաներում:
 - ✓ Չեն կարող օգտագործվել ասինխրոն մեթոդներում (**կհ**), իտերատորներում(**կհ**), լամդաներում(**կհ**), լոկալ մեթոդներում:

```
S ob =new S(57);
Console.WriteLine(ob.x);
ref struct S
{
    public int x;
    public S(int a)
    {
        x = a;
    }
}
class A
{
    S ob = new S(); // error
}
```

❖ Առանց **ref, out** արժեքների մեկից ավել **return** վերադարձ օբյեկտի միջոցով:

```
A ob = new A();
int x = 3;
int y = 5;
ob=f(x, y);
Console.WriteLine(ob.a);
Console.WriteLine(ob.b);
A f(int x, int y)
{
    A ob = new A();
    ob.a=++x;
    ob.b=++y;
    return ob;
}
class A
{
    public int a;    public int b;    }
```

48. ref, out հղիչային տիպերի համար

[Шилдг-227; Троелсен-183]

- Թույլատրվում է օգտագործել **ref, out** հղիչային տիպերի համար: Այդ դեպքում ինքը հղումն ուղղարկվում է ըստ հղման: Կամ այլ խոսքով ասած, հնարավորություն է տրվում մեթոդում փոփոխել օբյեկտը, որին կար հղում:

- Սկզբից փորձենք օգտագործել **string** հղիչային տիպ, որը աշխատում է իր կոպիայի հետ՝

```
string s = "highcode";
//f(ref s);
f(s);
Console.WriteLine(s);
// void f( ref string s)
void f(string s)
{
    s += "barev";
}
Console.ReadKey();
```

- Փորձենք կիրառել կլասի համար **ref/out** եւ հասկանանք անհրաժեշտությունը.

```
A ob = new A(1);
f(ob); //
//f(ref ob);
Console.WriteLine(ob.a);
Console.ReadKey();
//void f(ref A ob2)
void f(A ob2) //
{
    ob2 = new A(2);
}
```

```
class A
{
    public int a;
    public A(int a)
    {
        this.a = a;
    }
}
```

- Որպես **ref** պարամետր կիրառենք զանգվածի համար

```
int a = 8;
//f(a, ref A.Ar);
f(a, A.Ar);
foreach (int v in A.Ar)
    Console.WriteLine(v);
void f(int n, int[] Ar) // Runtime error!
//void f(int n, ref int[] Ar)
{
    Ar = new int[n];
}
Console.ReadKey();
class A
{
    public static int[] Ar;
}
```

.....

49. Մեթոդի պարամետր - params

[Троелсен-163; Troelsen – 229; Шилдт-229]

- params պարամետրը հնարավորություն է տալիս մեթոդին աշխատել **փոփոխվող քանակ** ունեցող պարամետրերով: Այդ դեպքում պետք է նաև զանգվածի օգտագործում: Մեթոդը կարող է ունենալ այլ պարամետրեր, միայն params-ը պետք է գտնվի պարամետրերի վերջում:

```
using System;
class A
{
    // public int ff(int a,int b) // գերբեռնումը ծածկում է params -ին
    // {
    //     return a+b;
    // }
    public int ff(params int[] mm)
    {
        int k = mm[0];
        for (int i = 1; i < mm.Length; i++)
            if (mm[i] < k)
                k = mm[i];
        return k;
    }
}
class Program
{
    public static void Main()
    {
        A ob = new A();
        int min, a = 10, b = 20;
        min = ob.ff(a, b);
        Console.WriteLine(min);
        min = ob.ff(a, b, -1);
        Console.WriteLine(min);
        min = ob.ff(18, 28, 38, 8, 58);
        Console.WriteLine(min);
        int[] mm2 = { 55, 88, -3, 99, 44, 90, 234 };
        min = ob.ff(mm2);
        Console.WriteLine(min);
    }
}
```

- ❖ Մեթոդը կարող է ունենալ նաև այլ պարամետրեր

```
using System;
class A
{
    public int ff(string s,params int[] mm)
    {
        int k = mm[0];
        for (int i = 1; i < mm.Length; i++)
            if (mm[i] < k) k = mm[i];
        Console.Write(s);
        return k;
    }
}
class M
{
    public static void Main()
    {
        A ob = new A();
        min = ob.ff("min= ", 18, 28, 38, 8, 58);
        Console.WriteLine(min);
    }
}
```

50. Մեթոդի անվանական արգումենտներ և ոչ պարտադիր պարամետրեր

[Шилдт-247; Троелсен-165; Troelsen – 230; Нейгел-121]

- (Named Arguments.) Եթե մեթոդը ունի մեկից ավելի պարամետրեր, ապա անվանական արգումենտները հնարավորություն են տալիս խախտել ուղարկվող արգումենտների ներկայացման հաջորդականությունը: Կազմակերպման համար օգտագործվում է (:) վերջակետ սիմվոլը: Եթե անվանական արգումենտների հետ միասին օգտագործվում են նաև այլ արգումենտներ, ապա պարամետրերում եւ արգումենտներում անվանականները պետք է գտնվեն վերջում:
- C# 7.2 –ում թույլատրվում է անվանական արգումենտները լինեն սովորական արգումենտների հետ ներկայացված, սակայն նրանք պետք է դիրքով պահպանվեն պարամետրերում: [Troelsen – 231] N (no video)

```
using System;
f("alb", 99, 'A');
f(age:99, rank:'A', name:"alb"); // named // ok
f("alb", age:99, 'A'); // ok // C# 7.2
// f(name: "alb", 'A', 99); // error
// f(rank:'A', 99, name: "alb"); // error
static void f(string name, byte age, char rank)
{
    Console.WriteLine(name + age + rank);
}
```

❖ Մեթոդի ոչ պարտադիր պարամետրեր [Шилдт-252]

- (Optional Paramters). Պարամետրերը կարող են լինել նաև ոչ պարտադիր: Այդ դեպքում չնշված արգումենտները պարամետրում արժեքավորվում են եւ պարամետրերի ցուցակում գտնվում են վերջում: Ոչ պարտադիր պարամետրերը թույլատրված են մեթոդներում, կոնստրուկտորներում, ինդեքսատորներում, դելեգատներում: Օգտագործվում է վերագրման (=) օպերատոր:

```
class M
{
    static void f(string name, byte age=99)
    {
        System.Console.WriteLine(name+age);
    }
    static void Main(string[] args)
    {
        f("alb ");
    }
}
```

- Պարամետրերը կարող են լինել լիտերալ, հաստատուն՝

```
class M
{
    const byte tariq=101;
    //byte tariq = 101; error!!!!
    static void f(string name, byte age=tariq)
    {
        System.Console.WriteLine(name+age);
    }
    static void Main(string[] args)
    {
        f("alb ");
        f("darb ",100);
    }
}
```

- Անվանական եւ ոչ պարտադիր արգումենտներ խորհուրդ է տրվում օգտագործել միայն անհրաժեշտության դեպքում: Օրինակ՝ շատ քանակությամբ պարամետրերի առկայության դեպքում, երբ կա նաև քիչ կազմով արգումենտներով կանչ:

51. partial կլասներ, partial մեթոդներ

[Həш-91; Нейгел-133; Троелсен-240; Troelsen – 221]

- Կլասները կարող են լինել **մասնակի** (partial): Այն հնարավորություն է տալիս նույն անվանումով կլասը **տարանջատել** մասերի եւ տեղակայել տարբեր ֆայլերում կամ տվյալ ծրագրի տարբեր մասերում (C#2005 -ից սկսած): Բոլոր տեղակայված կլասների անդամները համարվում են մեկ կլասի անդամներ: Թարգմանությունից հետո այն **միավորվում** է մեկ կլասի: Նրանք հնարավորություն են տալիս խուսափել մեծ չափերի ֆայլերից եւ **հարմար խմբավորել** անդամները: (no Top-Level)

```
partial class A // File1.cs
```

```
{
    public void f()
    {
        s = "albdarb";
    }
}
```

```
partial class A // File2.cs
```

```
{
    string s;
}
class Program
{
    static void Main()
    {
        A ob = new A();
        ob.f();
    }
}
```

partial Մեթոդներ [Həш-92]

- Մասնակի կլասի օգնությամբ կարելի է ստանալ **մասնակի մեթոդներ** եւ այն բաժանել նկարագրության եւ կիրառական մասերի: Այն կարող է հայտարարվել մասնակի կլասի մի մասում, իսկ կոդը գրել նույն կլասի այլ հատվածում: Եթե մեթոդը հայտարարվել է, սակայն հետո չի իրականացվել, ապա վերջնական կոդում չի ընդգրկվում (գովելի առանձնահատկություն):
- Մասնակի մեթոդները նման են պայմանական կոմպիլացիային (**#if, #elif, #endif**) (**կհ**): Եթե ունենք մեթոդի նկարագրություն եւ ծրագրի այլ հատվածներից բազում կանչեր ու կա անհրաժեշտություն մեթոդը ժամանակավոր հանենք թարգմանությունից, ապա միաժամանակ պետք է հեռացնենք բոլոր նրան կանչող մեթոդները: Նշված դժվարությունից կարող ենք խուսափել մասնակի մեթոդներով, որը հնարավորություն է տալիս չփնտրել եւ չհեռացնել մեթոդին կանչող մասերը:
- Մասնակի մեթոդները ունեն **սահմանափակումներ`**
 - ✓ Կարող են հայտարարվել միայն մասնատված կլասներում: Վերադարձնում են **void**: Կարող են լինել **static**: Չեն կարող ընդունել **out** պարամետր: Թույլատրելի է **ref, params, in**, ինչպես նաև **this** որպես արգումենտ: Չունեն թույլատրության մոդիֆիկատոր, քանի որ (ոչ ակնհայտ) նրանք փակ են:
 - ✓ Չեն կարող լինել **virtual**: Չեն կարող լինել **external**: Չեն կարող լինել **unsafe**: Կարող են ընդհանրացվել (generics) մասնակիորեն: Չեն կարող կանչվել դիլեգատի միջոցով, որովհետև վերջնական թարգմանված կոդում նրանց լինելը երաշխավորված չէ: (**կհ**)

```
class Program
```

```
{
    static void Main()
    {
        A.f2();
    }
}
```

```
partial class A
```

```
{
    public static void f2()
    {
        f(); System.Console.WriteLine("alb");
    }
    static partial void f();
}
```

```
// partial class A
```

```
// { static partial void f()
// { System.Console.WriteLine("darb"); }
// }
```

52. var - ոչ հստակ տիպայնացում, Անանուն տիպեր

❖ **var** ոչ հստակ տիպայնացում [Троелсен-135; Troelsen-89; Хейгел-72]

- ✓ Եթե անհրաժեշտ է տիպը որոշել արտահայտության կոնտեքստից, ապա օգտագործվում է **var** տիպը: `int a=886;` արտահայտությունը կարելի է գրել նաև հետևյալ ձևով՝ `var a=886;`
- ✓ **var** տիպը անպայման պետք է արժեքավորվի **տեղում**: Կոմպիլիատորի կողմից ձևավորված **var** տիպը հնարավոր չէ **վերափոխել նոր տիպի**: Տիպը որոշվում է **թարգմանության** ժամանակ: **var** տիպը կարելի է կիրառել միայն լոկալ փոփոխականների համար: **var** տիպը չի կարող ստանալ **null** արժեք (չափային տիպերի համար): **var** տիպի ինիցիալիզացիան կարող է լինի արտահայտություն: **var**-ը դա ոչ հստակ տիպայնացում է, որը կողը դարձնում է ավելի կարդացվող եւ բարձրացում ծրագրի կարգավորվածությունը:

❖ **Անանուն տիպեր** [Шилдт-663; Троелсен-438; Troelsen – 436; Хейгел-130]

- Երբ անհրաժեշտ է ունենալ կլաս, որը **միայն տվյալներ** է պահում, ապա կարելի է օգտագործել **անանուն տիպեր**: Նրանք ներկայացնում են ժամանակավոր օբյեկտներ, որոնք տեսանելի են միայն լոկալ տեսանելիության տիրույթում եւ պարզ է: Անանուն տիպերի օբյեկտները միայն կարդալու համար են եւ նրանց արժեքները **վերագրումից հետո չի փոխվում**:
- Անանուն տիպերը հայտարարվում են **var** եւ **new** միջոցով եւ արժեքավորվում են տեղում:

```
class A
{
    public string name { get; set; }
    public int age;
}
```

- ✓ Կարելի է ստեղծել օբյեկտ եւ տվյալներին արժեքավորել հետևյալ ձևով՝

```
A ob = new A
{
    name = "Ani",
    age = 19
};
```

- Չօգտվելով **A** կլասից նմանատիպ նկարագրություն եւ արժեքավորում կարող ենք ստանալ անանուն տիպով՝

```
var ob= new
{
    name = "Nune",
    age = 17
};
```

- Անանուն տիպերի դեպքում **չկա կլասի անվան իդենտիֆիկատոր**, սակայն “կոմպիլիատորը” այն ստեղծում է իր կողմից ներքին ձևով: Լայն կիրառություն ունի **բազաների** հետ կատարվող աշխատանքի ժամանակ:

```
A ob = new A(); // կամ var ob = new A();
ob.name = "alb";
ob.age = 88;
Console.WriteLine(ob.name + ob.age);
A ob1 = new A
{
    name = "darb",
    age = 99
};
Console.WriteLine(ob1.name + ob1.age);
//Console.WriteLine(ob2.name + ob2.age); // error // gravity
var ob2 = new // անանուն տիպ
{
    name = "HomoDeus",
    age = 1000
};
Console.WriteLine(ob2.name + ob2.age);
Console.WriteLine(ob2); // Tuple //ob2.age = 101; //error
```

```
class A
{
    public string name { get; set; }
    public int age;
}
```

53. Tuples (C#6,7)

[Троелсен-189; Troelsen-161; Albahari-207; Гриффитс] No Video

- Tuple –ը տարբեր տիպի տվյալների հավաքածու է, որտեղ կարելի է տվյալներ գրանցել եւ կարդալ: Tuple ստանալու համար կլոր փակագծերում ներկայացվում են տարբեր տիպի արժեքներ, որոնք տարանջատված են ստորակետով: Օրինակ՝ ('a', 55, "barev"):
- Tuple –ի աշխատանքի ժամանակ օգտագործվում է **var** ոչ հստակ տիպայնացումը եւ ինչպես նրա անդամին կարելի է դիմել Item1, Item2, Item3 ... հատկանիշներով, այնպես էլ անվանական արգումենտի նման՝ **վերջակետով** (:): հայտարարել փոփոխականի անուն եւ վերագրել արժեք:
- Tuple –ը մեծ տարածում ունի մեթոդներում **վերադարձվող արժեք** ապահովելու համար: Ունենալով Tuple տարբեր տիպերի պարամետրերի խումբ կարելի է մեկից ավել արժեքի **return** ապահովել: **ref** եւ **out** պարամետրերը նույնպես այդ գործն են իրականացնում, սակայն Tuple –ը հենց նախատեսված է ծրագրավորման այդպիսի հնարքի համար:
- Tuple հավաքածուն պետք է պարունակի երկու կամ ավելի անդամներ:

```
(char c, int n, string s) v1 = ('a', 55, "barev");
Console.WriteLine(v1);
Console.WriteLine(v1.c);
Console.WriteLine("-----");
var v2 = ('a', 55, "barev");
v2.Item1 = 'A';
Console.WriteLine(v2.Item1);
Console.WriteLine(v2.Item2);
Console.WriteLine(v2.Item3);
Console.WriteLine("-----");
var v3 = (c:'a', n:55, s:"barev");
Console.WriteLine(v3.c);
Console.WriteLine(v3.n);
Console.WriteLine(v3.s);
Console.WriteLine("-----");
int a;
string s;
f(out a, out s);
Console.WriteLine(a+s);
Console.WriteLine("-----");
var v4 = fTup1();
Console.WriteLine(v4.Item1);
Console.WriteLine(v4.Item2);
static void f(out int a, out string s) //out
{
    a = 99;
    s = "albdarb";
}
static (int, string) fTup1() // Tuple Troelsen-164
{
    return (77, "aaa");
}
```

❖ Tuple Equality/Inequality (New 7.3) [Troelsen-163] N (No Video)

- C # 7.1 -ում ներկայացվում է Tuple –ների հավասարությունը (==) և անհավասարությունը (!=): Համեմատման ժամանակ կատարվում է implicit ձևով տիպերի վերափոխարկում և հաշվի չի առնվում NULL թույլատրությունը:

```
Console.WriteLine("> Tuples Equality/Inequality");
// lifted conversions
var left = (a: 5, b: 10);
(int? a, int? b) nullableMembers = (5, 10);
Console.WriteLine(left == nullableMembers); // true
// converted type of left is (long, long)
```

```
(long a, long b) longTuple = (5, 10);
Console.WriteLine(left == longTuple); // Also true
// comparisons performed on (long, long) tuples
(long a, int b) longFirst = (5, 10);
(int a, long b) longSecond = (5, 10);
Console.WriteLine(longFirst == longSecond); // Also true
Console.ReadKey();
```

- Tuples –ները, որոնք ընդգրկում են **այլ Tuples –ներ** նույնպես կարելի է համեմատել:
- Tuples –ները համեմատման ժամանակ պետք է ունենան **նույն** պարամետրերի քանակը (երբեմն պարամետրը անվանում են հատկանիշ):

❖ Deconstructing Tuples [Albahari – 102] N (No Video)

- Tuple տիպ կարող է վերադարձնել մեթոդը: Իսկ ինչ կարելի է ասել կլասի կամ ստրուկտուրայի օբյեկտների մասին: Պատասխան՝ կլասի կամ ստրուկտուրայի **դեկոնստրուկտորը** հնարավորություն է տալիս նրանց օբյեկտներին **վերադարձնել** Tuple տիպ:
- Օրինակում օգտագործվում է **out** պարամետրով դեկոնստրուկտոր, որը պետք է լինի **միայն** Deconstruct() անունով:

```
A ob = new A(4, 5);
(int x, int y) = ob; // Deconstruction
// (int x, int y) = ob.Deconstruct(5,6); // error
Console.WriteLine(x + " " + y); // 4 5
```

```
class A
{
    public int X, Y;
    public A(int x, int y)
    {
        X = x;
        Y = y;
    }
    //public void Deconstruct2(out int x, out int y) // error
    //public void Deconstruct3(ref int x, ref int y) // error
    public void Deconstruct( out int x, out int y)
    {
        x = X;
        y = Y;
    }
}
```

- Նույն հետադարձ կապը կարելի է իրականացնել ցանկացած մեթոդի **ref/out**-ով առանց Tuple -ի:

❖ switch expression with Tuples [Troelsen-109] N (No Video)

```
for (; ; )
{
    Console.Write("input = ");
    string s = Console.ReadLine();
    int n = new Random().Next(3);
    string result = (s,n) switch
    {
        ("aa",1) => "barev",
        //("aa", _) => "barev", //discard n
        ("bb",2) => "albdarb",
        (_,_) => "default",
    };
    Console.WriteLine(result+n);
}
Console.ReadKey();
```

54. record տիպ (New 9.0) N (No Video)

[Troelsen-222; Albahari – 214]

- record** տիպը **հղիչային** տիպ է և համարվում է class տիպի առանձնահատուկ ձև: Օրինակ՝ օբյեկտների **համեմատման** ժամանակ աշխատում է որպես չափային տիպ (հաջորդ օրինակ):
- Այն թույլատրում է անմիջական **մեկ տողով** հայտարարել անդամներ, որոնք ավտոմատ դառնում են “immutable”: Այն պարզեցնում է կոդը, ազատելով ծրագրավորողին **init-only** և **read-only** հասկանիշների օգտագործումից:

```
A ob1 = new A(44);
ob1.x = 66; // ok // public
Console.WriteLine(ob1.x);
A2 ob2 = new A2(55); // ok
Console.WriteLine(ob2.y);
// ob2.y = 77; //error // immutable
Console.ReadKey();

record B
{
}

record A:B // ok // as class // public
{
    public int x;
    public A(int xx)
    {
        x = xx;
    }
}

record A2(int y); // ok // immutable
// record class A2(int y); // ok
// class A2(int z); // error
```

❖ Understanding Equality with record Types [Troelsen-225] N (No Video)

- record** տիպը վերաորոշում է Equals, ==, և !=, հրամանները: Այն կլսաից տաբերվում է նրանով, որ օբյեկտների համեմատման ժամանակ իրեն պահում է **չափային** տիպի նման՝ այսինքն համեմատում է անդամների արժեքները, այլ ոչ թե հղումները:

```
A obA1 = new A(44);
A obA2 = new A(44);
Console.WriteLine(obA1==obA2); // True
B obB1 = new B(44);
B obB2 = new B(44);
Console.WriteLine(obB1==obB2); // False

record A
{
    public int x;
    public A(int xx)
    {
        x = xx;
    }
}

class B
{
    public int x;
    public B(int xx)
    {
        x = xx;
    }
}
```

❖ record struct [Price – 243; Albahari-212; Troelsen-241] (C#10) N (No Video)

- Ստրուկտուրայի 2 օբյեկտ հնարավոր չէ համեմատել (==) հրամանով: **record** ստրուկտուրան համեմատումը դառնում է հնարավոր:

```
S ob1 = new S();
S ob2 = new S();
Console.WriteLine(ob1==ob2);
Console.ReadKey();

record struct S // struct S (error)
{
    public int Name { get; init; }
}
```

55. Ընդլայնված մեթոդներ

[Троелсен-434; Troelsen – 429; Шилдт -678; Нэш-477]

- Ինչպես գիտենք երբ տիպը որոշված է թարգմանված է հավաքագրման բլոկում, այն հիմնականում **փոփոխման ենթակա չէ**: Միակ ձևը փոփոխման դա ավելացնել նոր անդամներ եւ վերաթարգմանել: **Ամեն դեպքում** այդպիսի փոփոխաություններ կարելի է կատարել օգտվելով System.Reflection.Emit անունի տարածքից՝ օպերատիվ հիշողությունում որոշ դինամիկ փոփոխություններ կատարելու համար:
- **Ընդլայնված մեթոդները** տալիս են նոր հնարավորություն՝ առանց վերաթարգմանման տիպում ստանալ նոր գործողություններ (մեթոդներ) տվյալ տիպի համար: Ընդլայնված մեթոդները պետք է հայտարարվեն **ստատիկ կլաստում**: Պետք է **առաջին** պարամետրը լինի **նշված this**–ով:
- Առանձնահատկություն՝
 - ✓ Եթե ընդլայնված մեթոդների ստատիկ կլասը **public** է, ապա օգտագործողի կլասը նույնպես պետք է լինի **public**: **/****
 - ✓ Եթե օգտագործվող կլաստում գոյություն ունի համեմատական մեթոդ ընդլայնված մեթոդի հետ, ապա օգտագործվող կլասի մեթոդը ունի առաջնայնություն:

```
using System;
// public static class A  /**
static class A
{
    public static void Armenia(this B ob)
    {
        Console.WriteLine("B class-um em, extention em");
    }
    public static void Armenia(this String ss)
    {
        Console.WriteLine("String-um em " + ss);
    }
    public static void Armenia(this Array ar)
    {
        Console.WriteLine("Array-um em ");
    }
}
// public class B  **
class B
{
    //public void Armenia() // ստուգել առաջնայնությունը
    //{
    //    Console.WriteLine("B class NO extention-um em");
    //}
}
class M
{
    static void Main()
    {
        B ob = new B();
        ob.Armenia();
        String s = "plus Es Haykakan em";
        s.Armenia();
        int[] ar = new int[10];
        ar.Armenia();
    }
}
```

56. Կանոնավոր արտահայտություններ

[Албыхари –963; Нейгел –277,277]

- Կանոնավոր արտահայտությունները նախատեսված են տողերը հատուկ ձևով մշակելու համար: Այլ խոսքով՝ տեքստային տվյալները կարելի է “ֆիլտրել”: Օրինակ՝
 - ✓ կարելի է նշել հետագա մաքրման համար կրկնվող սիմվոլներ:
 - ✓ դարձնել բոլոր բառերի առաջին տառը մեծատառ:
 - ✓ N տառից ավել բառերի սկզբնատառը դարձնել մեծատառ:
 - ✓ առանձնացնել ինտերնետի URI հասցեի տարբեր էլեմենտներ:
- Նույն արդյունքին կարելի է հասնել String եւ StringBuilder կլասներով, սակայն նրանց հնարավորությունները ավելի սահմանափակ են եւ ծրագրավորելը ավելի աշխատատար է:
- Սկզբունքորեն կանոնավոր արտահայտություններ ստանալու համար օգտվում ենք System.Text.RegularExpressions անունի տարածքի [Regex](#) կլասից, որի շաբլոնը հատուկ սիմվոլներով ծրագրավորում է տեքստի փնտրման **ֆիլտրը**:
- [Regex](#) կլասի Match() եւ Matches() մեթոդները (**համակնում**), որոնցից կօգտվենք, ընդունում են պարամետրեր՝ տեքստը, շաբլոնը եւ անհրաժեշտ հատկություններ [RegexOptions](#) թվարկումով: (Օրինակ՝ [RegexOptions.IgnoreCase](#) նշում է անտեսել մեծատառ/փոքրատառ):
- Անունի տարածք՝ System.Text.RegularExpressions;
- Match() մեթոդը հիշեցնում է [string](#) տիպի IndexOf() մեթոդին, այն տարբերությամբ, որ IndexOf() աշխատում է լիտերալի հետ, իսկ Regex.Match() աշխատում է շաբլոնի հետ: Match() մեթոդը վերադարձնում է առաջին պատահած արդյունքը: Եթե անհրաժեշտ է հաջորդը, ապա կարելի է կիրառել NextMatch() մեթոդը:

```
string s = "One color? There are two colours in my head! colo";
Match m1 = Regex.Match(s, "colo");
Console.WriteLine(m1.Index); // 4
Match m2 = m1.NextMatch();
Console.WriteLine(m2.Index); // 25
Match m3 = m2.NextMatch();
Console.WriteLine(m3.Index); // 45
Console.WriteLine("----- string ----");
Console.WriteLine(s.IndexOf("colo")); // 4
int n = s.IndexOf("colo");
Console.WriteLine(s.IndexOf("colo", n+1)); // 25
Console.WriteLine(s.LastIndexOf("colo")); // 45
```
- ❖ Կարելի է ստանալ արդյունքների “կուլեկցիա” օգտվելով Regex.Matches() մեթոդից:
- Matches() մեթոդը վերադարձնում է [MatchCollection](#) տիպի զանգված, որտեղ պահվում են արդյունքը եւ կարող ենք արտածել [foreach](#) ու:
- Նշենք մի քանի հատուկ սիմվոլներ, որոնք անհրաժեշտ են օրինակում.
 - ✓ @ նշանի կիրառումը շաբլոնում նշանակում է հրամանների առկայություն, որը իրականացվում է ծրագրավորման սիմվոլների միջոցով:
 - ✓ Եթե անհրաժեշտ է շաբլոնում հատուկ սիմվոլի կիրառություն, ապա օգտագործվում է թեք գիծ \: օրինակ աստղանիշ կարելի է ստանալ * հրամանով:
 - ✓ \b նշանակում է բառի վերջ եւ սկիզբը: \balb գտնում է alb տառերով սկսվող բառերը: alb\b դեպքում ստուգում է վերջը:
 - ✓ \B նշանակում է ցանկացած դիրք բացի բառի սահմանից:
 - ✓ \S նշանակում է ցանկացած սիմվոլ բացի “պրոբելից”:
 - ✓ \s նշանակում է ցանկացած “պրոբել” սիմվոլ:
 - ✓ Ինչպես գիտենք * հին ծրագրերում նշանակում է ցանկացած քանակով սիմվոլ:
S* նշանակում է ցանկացած քանակությամբ ցանկացած սիմվոլ բացի “պրոբելից”:


```
// Օգտվենք Matches() մեթոդից.
using System;
using System.Text.RegularExpressions;
class Program
{
    static void Main(string[] args)
    {
        string s = "aaion ioNcc ddiondd eeion ionkk aion";
        const string pattern = "ion"; // 2 7 15 24 28 35
        // string pattern = @"ion\b"; // 2 24 35
        // string pattern = @"\bion"; // 7 28
        // const string pattern = @"\ba\S*ion\b"; // 0 34
        MatchCollection mc = Regex.Matches(s, pattern, RegexOptions.IgnoreCase);
        foreach (Match m in mc)
            Console.WriteLine(m.Index + " ");
    }
}
```

• Հատուկ սիմվոլների ցուցակը:

? - Նախորդ սիմվոլը կարող է կրկնվել 0 կամ 1 անգամ, այլ խոսքով ? -ով նշում են, որ նախորդ սիմվոլը պարտադիր չէ:

```
Console.WriteLine(Regex.IsMatch("color", @"colou?r")); // True
Console.WriteLine(Regex.Match("color", @"colou?r").Success); // True
Console.WriteLine(Regex.Match("colour", @"colou?r").Success); // True
Console.WriteLine(Regex.Match("colouur", @"colou?r").Success); // False
//-----
Match m = Regex.Match("any colour you like", @"colou?r");
Console.WriteLine(m.Value); // colour
Console.WriteLine(m.Success); // True // հաջողություն
Console.WriteLine(m.Index); // 4
Console.WriteLine(m.Length); // 6
```

✓ **[]** - քառակուսի փակագծերը հնարավորություն են տալիս կատարել սիմվոլների ընտրություն, որոնք կարելի է համարել միացած են տրամաբանական կամով: `ma[np]` կամ `ma[n|p]` արտահայտությունը նշում է `man` եւ `map` բառերը: Եթե նշվում է զծիկ, ապա դա նշանակում է միջակայք: `[1-9]` կամ `[a-z]`:

```
Console.WriteLine(Regex.Matches("That is that.", "[Tt]hat").Count); // 2
^ (բացի) - ցանկացած սիմվոլ բացի [] փակագծերի նշվածները կապված “կամ” ով:
Console.WriteLine(Regex.Match("kk qaacc qddee", "q[ab]").Index); // 9
```

✓ **\d** - նշանակում է ցանկացած թիվ.

```
Console.WriteLine(Regex.Match("b1-c4", @"[a-h]\d-[a-h]\d").Success); // True
```

✓ **\D** - նշանակում է ոչ թվային սիմվոլ

```
Console.WriteLine(Regex.Match("alb-9darb", @"alb-\D").Success); // False
```

✓ **\w** - նշանակում է տառեր, թվեր եւ տակի զիծ.

```
Console.WriteLine(Regex.Match(@"alb-darb", @"lb\w").Success); // False
Console.WriteLine(Regex.Match(@"alb_darb", @"lb\w").Success); // True
```

✓ **^** - տեքստի սկիզբ.

```
Console.WriteLine(Regex.Match("abc deAb", "^[Aa]b")); // ab
```

✓ **\$** - տեքստի վեջը.

```
Console.WriteLine(Regex.Match("abc deAb", "[Aa]b$")); // Ab
```

✓ ***** - նախորդ սիմվոլը կարող է կրկնվել 0 կամ n անգամ.

```
Console.WriteLine(Regex.Matches("albbb ald alm", "alb*").Count); // 3
```

✓ **+** - նախորդ սիմվոլը կարող է կրկնվել 1 կամ n անգամ.

```
Console.WriteLine(Regex.Matches("albbb ald alm", "alb+").Count); // 1
```

✓ **.** (կետ) - ցանկացած միայնակ սիմվոլ բացի (\ n) տողադարձից.

```
Console.WriteLine(Regex.Matches("albbb ald alm", ".").Count); // 13
```

```
Console.WriteLine(Regex.Matches("albbb ald alm", "a.").Count); // 3
```


{n} - ստույգ n համընկնում:

{n,} - ամենաքիչը n համընկնում:

{n,m} - համընկնում {n,m} միջակայքում.

```
Regex pattern = new Regex(@"\d{3}");
Console.WriteLine(pattern.Match("12").Success); // False
Console.WriteLine(pattern.Match("123").Success); // True
Console.WriteLine(pattern.Match("123456").Value); // 123
pattern = new Regex(@"\d{3,}");
Console.WriteLine(pattern.Match("123456").Value); // 123456
pattern = new Regex(@"\d{3,5}");
Console.WriteLine(pattern.Match("123456").Value); // 12345
Console.WriteLine();
```

- Կանոնավոր արտահայտությունների կարևոր հատկություններից է նիշերի խմբավորումը:
() կլոր փակագծերով կարելի է խմբավորել ցանկացած հատված: Օրինակ. (an)+ հարամանի դեպքում + -ը ազդում է ոչ թե մեկ սիմվոլ դեպի նախորդ սիմվոլ, այլ 2 սիմվոլ an կոմբինացիա ձախ:
Console.WriteLine(Regex.Matches("slow slololow sw slooow", "s(lo)+w").Count); // 2
Console.WriteLine(Regex.IsMatch("Jenny", "Jen(ny|nifer)")); // True

❖ Օրինակ. URL (Uniform Resource Locator)

Ինտերնետում գործում է հետևյալ կառույցը. <protokol>://<adress>:<port>

- Ենթադրենք անհրաժեշտ է սետդել <http://www.albdarb.com:8080> տիպի URL ֆիլտր: Պարտադիր է “պրոտոկոլը”, պարտադիր է կայքը եւ պարտադիր է “պորտը”:
- Ներկայացնենք շաբլոն և տեսնենք արդյոք նա կայքի ֆիլտր կարող է հանդիսանալ?
"^(ht|f)tp(s)?://([^:]+)(:\d+)\$"
- ✓ առաջին խումբը ^ (ht|f)tp(s)?:// => թույլատրում է ունենալ տարբեր “պրոտոկոլներ”, (http://) կամ (ftp://) կամ (https://) կամ (ftps://)
- ✓ երկրորդ խումբը ([^:]+) => թույլատրում է www.albdarb.com եւ նշում է հաջորդ խումբը պարտադիր է եւ չի կարող սկսվել (:) վերջակերով:
- ✓ երրորդ խումբը (: \d+) => թույլատրում է :8080 պարտադրելով վերջակետ (:), հետո թիվ: Ֆիլտրը նշում է նաև, որ թվով պետք է ավարտվի կայքը (\$ հրաման):

```
using System;
using System.Text.RegularExpressions;
class Program
{
    static void Main(string[] args)
    {
        string s = "http://www.albdarb.com:8080";
        Console.WriteLine(Regex.IsMatch(s, @"^(ht|f)tp(s)?://([^: ]+)(:\d+)$")); // True
    }
}
```

.....

Օբյեկտ կողմնորոշվածություն

57. Ժառանգականություն, **protected**

[Troelsen-222,251; Troelsen-228, 238; Шилдт-329,336]

- Ժառանգականությունը (Inheritance) օբյեկտ կողմնորոշված ծրագրավորման գլխավոր հենքերից է: Այն հնարավորություն է տալիս կոդի **բազմակի օգտագործում** (code reuse):
- Ժառանգականությունը ապահովում է “**պատկանում է**” ենթակայության հարաբերություն (“**is-a**”): Նշենք, որ մյուս ենթակայության սկզբունքը դա “**ընդգրկում է**”, որը կատարվում է **կոմպոզիցիայի** միջոցով (“**has-a**”):
- Կոմպոզիցիա (ազդեցացիա)** “has a”(Containment/Delegation) [Troelsen-243]
- Ժառանգականության ժամանակ կարելի է օգտագործել մասնակի պաշտպանվածության ձև **protected** իդենտիֆիկատորի միջոցով, որը գործում է **միայն ժառանգության** սահմաններում: Տվյալ դեպքում ժառանգականության հնարավորությունները ավելին են կոմպոզիցիայի նկատմամբ և նման է ավտոհատկանիշների մեր ներկայացրած “**կիսահաղորդիչ**” գաղափարին:
- Ժառանգականության ժամանակ ավտոմատ կատարվում է **լռելիությամբ** կոնստրուկտորները և կոնստրուկտորների կատարման հաջորդականությունը կատարվում է **բազայինից** դեպի ժառանգ կլաս:
- Կլասը ժառանգում է օգտագործվող վերջակետ սիմվոլը: Ի տարբերություն C++ ի, շարվում **բազմաժառանգում** չի թույլատրվում (risk collision):

```
using System;
class A
{
    //public void fA()
    protected void fA()
    {
        Console.WriteLine("f A");
    }
}
class B : A
{
    public void fB()
    {
        fA();
        Console.WriteLine("f B");
    }
}
// class M:A,B // error
class M
{
    public static void Main()
    {
        A ob = new A();
        //ob.fA();//error!
        B ob2 = new B();
        ob2.fB();
        //ob2.fA();// error, if - protected void fA()
    }
}
//-----
class C:A //error ?? Ինչու ? // Object կլաս !!!
{
}
class B:C
{
}
class A:B
{
}
```

58. Ժառանգում **base, new**

[Шилдт-343; Нейгел-144; Троелсен-249, 266; Troelsen-236; 255]

- Կլասի անդամը կարող է համընկնել բազայինի հետ և ծածկում է բազայինին: Բազայինին դիմելու համար օգտվում ենք **base** - ից:
- new** –ն միջոց է **զգուշացման** դեմ, այսինքն տեղյակ ենք համընկման մասին:

```
using System;
class A
{
    protected int a = 9;
    public void ff()
    {
        Console.WriteLine("ffA " + a);
    }
}
class B : A
{
    new int a = 77;           // int a=88; //no error// warning
    public new void ff()
    {
        Console.WriteLine("ffB " + a);
        Console.WriteLine("ffB " + base.a);
    }
}
class M
{
    public static void Main()
    {
        A ob = new A();
        ob.ff();
        B ob2 = new B();
        ob2.ff();
    }
}
```

59. Բազային կոնստրուկտորի պարամետրերի փոխանցում

[Троелсен-249; Troelsen-236; Шилдт-339; Нейгел-149]

- Այսպիսի անհրաժեշտություն առաջանում է, երբ ժառանգ և բազային կլասի կոնստրուկտորները ունեն պարամետրեր և կատարվում է **պարամետրի փոխանցում**: Այն իրականացվում է **base** հրամանով:
- Սկզբից նշենք լռելիությամբ կոնստրուկտորների կանչման հաջորդականությունը, որը սկսում է բազայինից:

```
class A
{
    public A()
    {
        System.Console.WriteLine("aaaaaa");
    }
}
class B : A
{
    public B()
    {
        System.Console.WriteLine("bbbbbb");
    }
}
```

```

class C : B
{
    public C()           // public C(int a)
    {
        System.Console.WriteLine("cccccc");
    }
}
class Program
{
    static void Main(string[] args)
    {
        C ob = new C(); // C ob = new C(99);
    }
}
//.....

```

- Պարամետրով կոնստրուկտի առանձնահատկություն:

```

class A
{
    public A(int a)
    {
    }
    //public A()
    //{
}
class B : A
{
}
class Program
{
    static void Main(string[] args)
    {
        // error !!
    }
}
//.....

```

- Ցուցադրենք, թե ինչպես է պարամետր փոխանցվում բազային կոնստրուկտորին, որտեղ արդեն լռելիությամբ կոնստրուկտորը պարտադիր չէ:

```

class A_papa
{
    public A_papa(int a)
    {
        System.Console.WriteLine("papa" +a);
    }
    //public A_papa()
    //{
}
class B_erexa : A_papa
{
    public B_erexa(int a):base(a)
    {
        System.Console.WriteLine("erexa");
    }
}
class M
{
    public static void Main()
    {
        B_erexa ob = new B_erexa(55);
    }
}
//.....

```

60. Տիպերի վերափոխարկում, Կոնվերտացիաներ

[Троелсен-130,428; Troelsen-84,423; Нейгел-228] [Covaci – 122]

- Ներդրված տիպերը թույլ են տալիս կատարել վերափոխարկում, եթե նրանք նույն ընտանիքից են: Եթե տիպերը տարբեր են, ապա վերափոխարկում կարելի է կատարել **հատուկ** միջոցներով, օրինակ օգտվելով **Convert** կլաից:

```
int a = 123;
long b = a;    // Implicit conversion from int to long
int c = (int)b; // Explicit conversion from long to int
```

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

```
{
    int a = 99;
    uint b = 886;
    a = b; //error
    b = a; //error
    a = (int)b;
    b = (uint)a;
    bool b2 = true;
    int a2 = 99;
    a2 = Convert.ToInt32(b2);
    b2 = Convert.ToBoolean(a2);
}
```

❖ Կոնվերտացիաներ **Convert**, **Parse**, **TryParse**, **BitConverter** [Covaci – 122]

- Convert** **Parse**, **TryParse** կլասները հնարավորություն են տալիս բազային տիպերի միջև կատարել տիպերի վերափոխարկում: Եթե համեմատենք **Parse/TryParse** և **Convert** վերափոխարկումները, ապա կտեսնենք, որ **Convert**–ը **null** կարող է վերափոխարկել և չի առաջացնի **ArgumentNullException** բացառություն, ի տարբերություն **Parse/TryParse**–ի՝

```
int i = Convert.ToInt32(null);
Console.WriteLine(i); // 0
int j = int.Parse(null); // ArgumentNullException // error
```

- TryParse**–ը ունի **ստուգելու** առավելություն, որի միջոցով խուսափում ենք չթույլատրված վերափոխարկումներից:

- BitConverter** կլասը հնարավորություն է տալիս տվյալից ստանալ **բայթերի զանգված** և հակառակը: Օրինակ. **BitConverter.GetBytes()** մեթոդը տվյալից ստանում է բայթերի զանգված, իսկ **ToInt16(byte[] value, int startIndex)** մեթոդը բայթերի զանգվածը վերածում է **short** տիպի տվյալի, սկսած **startIndex** համարից:

```
class M
{
    static void Main()
    {
        string s = "true";
        bool b = bool.Parse(s);
        System.Console.WriteLine(b);
        // s = "false222";
        // bool a;
        // bool b = bool.TryParse(s, out a);
        int v=12345;
        byte[] valueBytes = System.BitConverter.GetBytes(v);
        foreach(byte x in valueBytes)
            System.Console.WriteLine(x);
        short v1;
        v1 = System.BitConverter.ToInt16(valueBytes, 0);
        // v1 = System.BitConverter.ToInt16(valueBytes, 1);
        // v1 = System.BitConverter.ToInt16(valueBytes, 3); // runtime error
        System.Console.WriteLine(v1);
    }
}
```

61. Հղում բազային կլասսով

[Шилдт-251; Троелсен-428; Troelsen-257, 424; Хейгел-233]

- **Նույն կլասի** տարբեր օբյեկտների ցուցիչներ կարելի է վերագրել, իսկ **տարբեր կլասների ոչ**: Եթե նրանք ժառանգականությամբ են կապված, ապա կա բացառություն: **Բազային ցուցիչին կարելի է** վերագրել ժառանգի ցուցիչ: Այդ դեպքում նա հղում է ժառանգ կլասին: Այդ ժառանգ ցուցիչով կարող է դիմել բազային կլասի անդամներին, իսկ ժառանգի անդամներին **միննույնն է ոչ**:

```
A obA = new A();
B obB = new B();
obA = obB;
Console.WriteLine(obA.ToString()); // B
//obB=obA; // Compile error //obB=(B)obA; // Exeption
obA.a = 44;
//obA.b=55;//error
```

```
class A
{
    public int a;
}
class B : A
{
    public int b;
}
```

❖ Ժառանգական կապվածությամբ տիպերի վերափոխարկում

- Տարբեր տիպեր, որոնք ստեղծվում են օգտագործողի կողմից (կլասներ, ստրուկտուրաներ), չի թույլատրվում կատարել վերագրում: Մակայն կարելի կատարել վերագրում, եթե այդ տիպերը կապված են ժառանգականությամբ: Վերագրել այդ դեպքում կարելի է “սովորական” (**Implicit**) ձևով կամ “ակնհայտ” (**Explicit**) ձևով օգտագործելով կլոր փակագծեր:

```
A obA = new A();
B obB = new B();
obA=obB; //ok
//obB = obA; // error
//obB = (B)obA; // error, եթե հանենք obA=obB; հրամանը
A ob1 = new B(); // Implicit cast
A ob2 = (A)new B(); // Explicitly cast
//B ob3 = (B)new A(); // error
```

```
class A
{ }
class B : A
{ }
```

62. Վիրտուալ մեթոդներ, Պոլիմորֆիզմ (Polymorphism)

[Шилдт-251] [Шилдт-335, 43; Троелсен-256,217; Troelsen-246]

- Եթե անհրաժեշտ է **բազային ցուցիչով դիմել ժառանգ կլասի մեթոդներին**, ապա մեթոդները պետք է լինեն և բազայինում և ժառանգում, ունենան նույն անունը, պարամետրերը և բազայինը լինի **վիրտուալ**:
- Առանց վիրտուալ մեթոդի բազային ցուցիչին կարելի է վերագրել ածանցյալ կլասի ցուցիչ և բազային ցուցիչով կարելի է դիմել **միայն** բազային անդամներին: Եթե նույն անունով մեթոդ կա, ապա այդ մեթոդը վիրտուալացնելով կարելի է հնարավորությունը մեծացնել - դիմել ածանցյալ կլասին:
- Այսպիսով - բազային ցուցիչով կարելի է դիմել ածանցյալ կլասի նույն անունով մեթոդներին:
- **virtual** – բազային մեթոդի սպեցիֆիկատոր
- **override** – ժառանգ մեթոդի սպեցիֆիկատոր

```
A ob = new A();
```

```

ob.ff();
B ob2 = new B();
ob = ob2;
ob.ff();
D ob3 = new D();
ob = ob3;
ob.ff();

```

```

class A
{
    public virtual void ff()
    {
        Console.WriteLine("ffA ");
    }
}
class B : A
{
    public override void ff()
    {
        Console.WriteLine("ffB ");
    }
}
class D : B
{
    public override void ff()
    {
        Console.WriteLine("ffD ");
    }
}

```

❖ **virtual** lı **Covariant return** unıhuı (C# 9) [Albahari - 119] (no video)

```

V ob = new A();
ob.f();
Console.ReadKey();
class papa
{
}
class B : papa
{
}

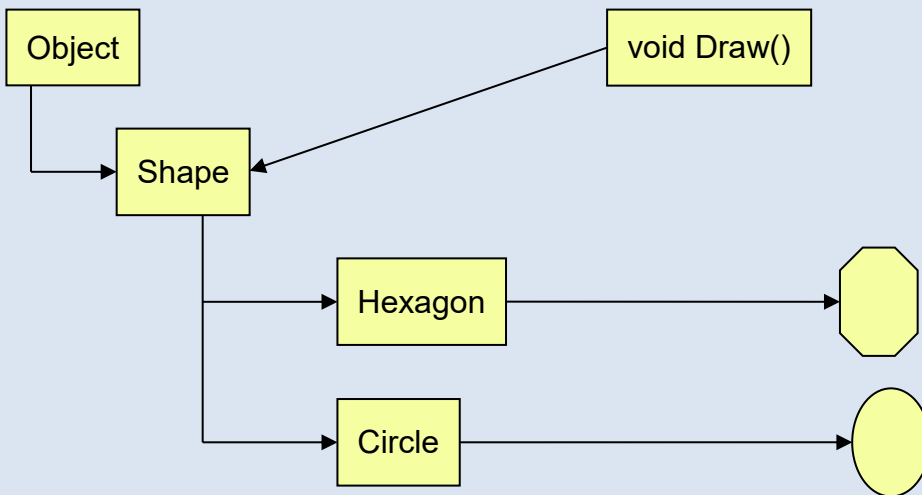
class V
{
    public virtual papa f()
    {
        Console.WriteLine("papa");
        papa ob = new();
        return ob;
    }
}

class A : V
{
    public override B f()
    {
        Console.WriteLine("B");
        B ob = new();
        return ob;
    }
}

```

❖ Պոլիմորֆիզմ (Polymorphism) [Троелсен-217; Troelsen-247; Шилдт-43]

- **Բազմա - ձևություն:** Գաղափարը վերցրված է “բիոլոգիայից”:
- Պոլիմորֆիզմը, օբյեկտ կոդմանրոշվածության հիմնային գաղափարներից է: Այն հնարավորություն է տալիս միննույն գործողության կառուցվածքին հանդես գալ տարբեր ձևերով: Վերցնենք կլասի մեթոդ: Նրա վերադարձնող արժեքի տիպը, մեթոդի անունը, պարամետրերի տիպը համարենք գործողության կառուցվածք, իսկ նեքին հրամանները համարենք մեթոդի հանդես գալու ձևը:
- Թարգմանման ժամանակ պոլիմորֆիզմ (**ստատիկ պոլիմորֆիզմ**) – իրագործվում է **գերբեռնված** մեթոդների միջոցով, որտեղ միայն մեթոդի անունն է համարվում գործողության կառուցվածք:
- Կատարման ժամանակ պոլիմորֆիզմ (**դինամիկ պոլիմորֆիզմ**) - իրագործվում է վիրտուալ մեթոդների օգնությամբ: **Այն հնարավորություն է տալիս բազային ցուցիչով աշխատել ժառանգ մեթոդների հետ:**



❖ NVI պատերն (Non-Virtual Interface) [Nash-423]

- Նախագծման ժամանակ ենթադրենք մենք ստեղծում ենք Base կլասը և այդ կլասից օգտվում են մեծ քանակով այլ նախագծողներ: Ենթադրենք անհրաժեշտ է կատարել բազային կլասում վիրտուալ մեթոդի ֆունկցիոնալ փոփոխություն: Պարզ է այդ փոփոխությունը վերաթարգմանման միջոցով պետք է կատարեն բոլոր նրանք, ովքեր օգտվում են բազային ցուցիչից: **Վերաթարգմանելը** ոչ ցանկալի պրոցես է: Նշված թերությունը հաղթահարվում է NVI “ոչ վիրտուալ ինտերֆեյս” պատրենի միջոցով: Հայտարարվում է բոլորին հասանելի ոչ վիրտուալ “ինտերֆեյս” բազային կլասում, իսկ վերավորոշումը տեղափոխվում է **պաշտպանված** (protected) մեթոդ:
- Այս շաբլոնը նման է GOF – ի Template Method պատերնին և լայն կիրառություն ունի NET Framework գրադարանում:

```

Base b = new Derived(); //pattern NVI
b.f();
class Base
{
    public void f()
    {
        vf();
    }
    protected virtual void vf()
    {
        Console.WriteLine("Base.DoWork()");
    }
}
class Derived : Base
{
    protected override void vf()
    {
        Console.WriteLine("Derived.DoWork()");
    }
}
  
```


63. Ժառանգականություն և **is** / **as** պատկանելիություն

❖ **as** keyword [Troelsen-259]

- Հիշենք, որ explicit casting –ը (տիպերի վերաթիսարկումը) իրականացվում է ծրագրի կատարման ժամանակ: C# -ում գոյություն ունի **as** keyword, որը հնարավորություն է տալիս ծրագրի կատարման ժամանակ որոշել տիպի պատկանելությունը այլ տիպերի: Եթե պատկանելությունը տեղի ունի, ապա կարելի է ստանալ այդ տիպի հղումը և հակառակ դեպքում վերադաձվում է **null** արժեք:
- ✓ [Be aware that explicit casting is evaluated at runtime, not compile time. C# provides the **as** keyword to quickly determine at runtime whether a given type is compatible with another. When you use the **as** keyword, you are able to determine compatibility by checking against a null return value.]

```
A obA = new A(); //
var ob=obA as A; //
// A obA = new B();
// var ob = obA as B; // բազային հղում չէ, որովհետև նոր տիպ է ձևավորվում և կարտածի fB
// A ob = obA as B; // այս դեպքում կարտածի fA
if (ob!=null)
{
    Console.WriteLine(ob);
    ob.f();
}
else
{
    Console.WriteLine("null");
}
Console.ReadKey();
class A
{
    public void f()
    {
        Console.WriteLine("fA");
    }
}
class B : A
{
    public void f()
    {
        Console.WriteLine("fB");
    }
}
```

❖ **is** keyword (Updated 7.0, 9.0) [Troelsen-260]

- C# -ում գոյություն ունի նաև **is** keyword, որը նույնպես ստուգում է երկու տիպերի նույն ընտանիքի լինելիությունը: Ի տարբերություն **as** –ի այն չի ստուգում հղում, այլ ստուգում է բուլյան արժեք:
- ✓ [In addition to the **as** keyword, the C# language provides the **is** keyword to determine whether two items are compatible. Unlike the **as** keyword, however, the **is** keyword returns false, rather than a null reference, if the types are incompatible.]
- C# 7.0 –ում լուծվում է նաև “**double-cast**” կրկնակի տիպերի վերափոխման խնդիրը (ուղղակի չի կատարվում), եթե տիպի պատկանելիությունը հաստատվում է:
- ✓ [New in C# 7.0, the **is** keyword can also assign the converted type to a variable if the cast works. This cleans up the preceding method by preventing the “**double-cast**” problem.]

```
A obA = new B();
//A obA = new A();
//if(obA is not B) // C# 9.0
if (obA is B b) // B b - իրենից ներկայացնում է "double-cast" problem
{
    Console.WriteLine("if 1");
    Console.WriteLine(obA);
    Console.WriteLine(b);
}
else if (obA is A a) // A a - իրենից ներկայացնում է "double-cast" problem
{
    Console.WriteLine("if 2");
}
```

```

    Console.WriteLine(obA);
    Console.WriteLine(a);
}
Console.ReadKey();
class A
{
}
class B : A
{
}

```

❖ Discards –ի օգտագործում **is** keyword -ով (New 7.0) [Troelsen-261]

- **is** keyword –ը կարելի է օգտագործել նաև discard (**_**) անստեղծ հրամանի հետ միասին:
- ✓ [The **is** keyword can also be used in conjunction with the **discard variable placeholder**.]

```

A obA = new B();
//A obA = new A();
if (obA is B)
{
    Console.WriteLine(obA);
}
else if (obA is var _)
{
    Console.WriteLine("discard variable placeholder");
}
Console.ReadKey();
class A
{
}
class B : A
{
}

```

❖ Pattern Matching (New 7.0) [Troelsen-262]

- **if.. else** կարելի է փոխարինել **switch** օպերատորով, որտեղ կարելի է նաև օգտագործել **when** օպերատորը իր պայմանով:
- discard –ը օգտագործվում է, երբ գոյություն ունի ընդհանուր տիպ, որը կբավարարի բոլոր դեպքերին: Հնարավոր է նաև պայմանի բավարարումով էլ ավելի նեղ ճանապարհ: Նման դեպքերում discard –ը դրվում է ամենավերջում, որովհետև ցանկացած այլ տեղում այն կչեզոքացնի մյուսներին: (նման է Exception կլասին)

```

A obA = new B(); //A obA = new A();
for (; ; )
{
    int a = Convert.ToInt32(Console.ReadLine());
    switch (obA)
    {
        case B ob when ob.p > 0: // C# 7.0
            Console.WriteLine(ob); // Console.WriteLine(obA.pp); // error
            break;
        case A ob when a > 4:
            Console.WriteLine(ob);
            break;
        case A _:
            Console.WriteLine("Discards can also be used in switch statements");
            break;
    }
}
Console.ReadKey();
class A
{
}
class B : A
{
    public int p { get; init; }
}

```

64. Տիպերի վերափոխարկում **implicit**, **explicit**

[Троелсен-429; Troelsen-427; Нейгел-230]

- Եթե անհրաժեշտ է տարբեր տիպերի օբյեկտներ վերագրել, որոնք կապված չեն ժառանգականությամբ, ապա պետք է օգտագործել օպերատորների գերբեռնում: **implicit** –ի միջոցով կատարվում է տիպերի թաքնված (ենթադրյալ) վերափոխարկում: **explicit** –ի միջոցով կատարվում է տիպերի բացահայտ վերափոխարկում () փակագծի օգտագործումով:

```
using System; // implicit class A, class B
class A
{
    public int n;
    public A(int n)
    {
        this.n = n;
    }
    public static implicit operator B(A oA)
    {
        B oB = new B();
        oB.m = oA.n;
        return oB;
    }
}
class B
{
    public int m;
}
class Program
{
    static void Main(string[] args)
    {
        A obA = new A(55);
        B obB = new B();
        Console.WriteLine(obB.m);
        obB = obA; // !!!!
        Console.WriteLine(obB.m); // հարց, կոպիա է թե ոչ ?
        obA.n = 88;
        Console.WriteLine(obB.m); // 55 or 88 ?
    }
}
//-----
using System; // explicit class A, class B
class A
{
    public int n;
    public A(int n)
    {
        this.n = n;
    }
    public static explicit operator B(A oA)
    {
        B oB = new B();
        oB.m = oA.n;
        return oB;
    }
}
class B
{
    public int m;
}
class Program
{
    static void Main(string[] args)
    {
        A obA = new A(55);
        B obB = new B();
        Console.WriteLine(obB.m);
```

```

        obB = (B)obA; // !!!!
        Console.WriteLine(obB.m);
    }
}
//-----
namespace ConsoleAppImplicit // implicit float, class A
{
    class A
    {
        public uint dollar;
        public ushort cent;
        public A(uint d, ushort c)
        {
            dollar = d;
            cent = c;
        }
        public static implicit operator float(A ob) // **
        { return ob.dollar + ob.cent/100f; } //
    }
    class Program
    {
        static void Main(string[] args)
        {
            A ob = new A(55, 99);
            float f = ob; // ** error!!
            System.Console.WriteLine(f);
        }
    }
}
//-----
using System; //explicit float, class A
namespace ConsoleAppExplicit
{
    class A
    {
        public uint dollar;
        public ushort cent;
        public A(uint d, ushort c)
        {
            dollar = d;
            cent = c;
        }
        public static explicit operator A(float f) // **
        {
            uint dd= (uint)f; //
            ushort cc = (ushort)((f-dd)*100); //
            return new A(dd,cc); //
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            float f=33.77f;
            A ob = (A)f; // ** error!!
            Console.WriteLine(ob.dollar+" " + ob.cent);
        }
    }
}

```

65. Մեկուսացված կլասներ և մեթոդներ, sealed

[Троелсен-245,259; Troelsen-232, 249; Шилдт-367; Нейгел-146]

• Մեկուսացված կլասներ

- ✓ Մեկուսացված կլասները դրանք անժառանգ կլասներ են, որոնց արգելված է դարձնել բազային կլաս և կատարել նրանցից ժառանգում:
- ✓ Իրականացվում է **sealed** – ի միջոցով:

```
sealed class A
```

```
{
```

```
}
```

```
class B:A // Error
```

```
{
```

```
}
```

- Երբեմն անհրաժեշտ է լինում կլասի ժառանգականությունը արգելել ոչ թե ամբողջ կլասի համար, այլ որոշ անդամների համար: Դա հանարավոր է, եթե անդամները վիրտուալ են և օգտագործում են **sealed** սահմանափակումը:

```
class A
```

```
{
```

```
    protected virtual void f()
```

```
    {
```

```
    }
```

```
}
```

```
class B : A
```

```
{
```

```
    protected override sealed void f()
```

```
    {
```

```
    }
```

```
}
```

```
class D : B
```

```
{
```

```
    protected override void f() //error !!
```

```
    {
```

```
    }
```

```
}
```

```
class M
```

```
{
```

```
    static void Main()
```

```
    {
```

```
    }
```

```
}
```

.....

66. Ներդրված կլասներ

[Троелсен-254; Troelsen-244; Liberty-101; Либерти-141]

- Թույլատրվում է կլասը ընդգրկի այլ կլասի հայտարարություն, որը կոչվում է ներդրված կլաս: Եթե ներդրված կլասի անդամը փակ է, ապա չի կարող օգտվել ոչ արտաքին կլասը, ոչ այլ կլասներ: Ներդրված կլասը կարող է օգտագործել արտաքին կլասի փակ անդամները: Եթե ներդրված կլասն է փակ, ապա արտաքին կլասը կարող օգտվել ներդրված կլասից, սակայն չեն կարող օգտվել այլ կլասներ:
- Հարց: **Կլասը ինչ մոդիֆիկատոր կարող է ընդունել:** Պատասխան: Բոլոր թույլատրության մոդիֆիկատորները, քանի որ, կլասը հնարավոր է լինի ներդրված: Իսկ ինչպես գիտենք, եթե կլասը ներդրված չէ, ապա միայն **public** և **internal**: (լռելիությամբ **internal**):

```
using System;
class A
{
    public void f()
    {
        Console.WriteLine("fA");
        B ob3 = new B();
        ob3.f2();
    }
    public class B
    {
        public void f2()
        {
            Console.WriteLine("fB");
            M ob5 = new M();
            Console.WriteLine(ob5.a);
        }
    }
}
class M
{
    public int a = new int();
    public static void Main()
    {
        A ob = new A();
        ob.f();
        A.B ob2 = new A.B();
        ob2.f2();
        Console.WriteLine(ob2); //A + B
    }
}
```

67. Աբստրակտ կլասներ

[Троелсен-260; Troelsen-250; Шилдт-363]

- Աբստրակտ կլասները հանդիսանում է օբյեկտ կողմնորոշվածության հաջորդ **կարարևոր հիմնային** մասը: Այն ավելացնում է ծրագրային **պարտականությունների տեխնոլոգիան**: Բացի պարտականությունից, աբստրակտ կլասը վիրտուալ մեթոդների նման հնարավորություն է տալիս **բազային ցուցիչով** աշխատել **ժառանգ կլասների անդամների** հետ:
- Եթե կա անհրաժեշտություն ժառանգ կլասում պարտադիր իրականացնել բազային կլասի մեթոդները, ապա բազային կլասում մեթոդը հայտարարվում է աբստրակտ: Այդ դեպքում կլասը ավտոմատ դառնում է աբստրակտ:
- Աբստրակտ մեթոդը բազային կլասում չի կարող ունենալ իրականացում:
- Չի թույլատրվում** ստեղծել աբստրակտ **կլասի օբյեկտ** և այն նույնանման է **փակ կոնստրուկտորի** կիրառմանը:

```
abstract class A
```

```
{  
    public abstract void f();  
}
```

- Ժառանգ կլասում մեթոդի համար օգտագործվում է **override** սպեցիֆիկատորը:

```
using System;
```

```
abstract class A
```

```
{  
    abstract public void f();  
}
```

```
class B : A
```

```
{  
    public override void f()  
    {  
        Console.WriteLine("ok!");  
    }  
}
```

```
class M
```

```
{  
    static void Main()  
    {  
        B ob = new B();  
        //A ob = new A();// error !  
        A ob2 = new B(); // OK !  
        ob2.f(); // OK !  
    }  
}
```

- ❖ Աբստրակտ կլասում թույլատրվում է հետևյալ անդամները.

```
delegate void dd();
```

```
abstract class A
```

```
{  
    abstract public void f();  
    abstract public int p { get; set; }  
    abstract public event dd ee;  
    abstract public int this [int i] { get;set; }  
    // abstract public delegate void dd2();// error  
}
```

- ❖ **Աբստրակտ պոլիմորֆիզմ** [Troelsen-252]

- Ի տարբերություն վիրտուալ պոլիմորֆիզմի, աբստրակտ պոլիմորֆիզմը ստիպում է ժառանգին իրականացնել և վերաորոշել պոլիմորֆ մեթոդ՝ այս ձևով դառնալով **“կոնտրակտ”** ծրագրավորման էլեմենտ:

68. Template Method պատերն

[Фримен-310; Шевчук -270; GOF-373]

- Աբստրակտ կլասի լավագույն կիրառություններից է Template Method փաթեթը: Template Method պատերնը որոշում է ալգորիթմների բնույթ և թույլատրում է ենթակլասներին վերաորոշել ալգորիթմների որոշ մասեր, չփոփոխելով ընդհանուր ստրուկտուրան:

```
using System; //ավելցուկով կոդ
```

```
class Coffee
{
    public void prepareRecipe()
    {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
    void boilWater()
    {
        Console.WriteLine("Boiling Water");    }
    void brewCoffeeGrinds()
    {
        Console.WriteLine("Dripping Coffee through filter");    }
    void pourInCup()
    {
        Console.WriteLine("Pouring into cup");    }
    void addSugarAndMilk()
    {
        Console.WriteLine("Adding Sugar and Milk ");    }
}

class Tea
{
    public void prepareRecipe()
    {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
    void boilWater()
    {
        Console.WriteLine("Boiling Water");    }
    void steepTeaBag()
    {
        Console.WriteLine("Steeping the Tea");    }
    void pourInCup()
    {
        Console.WriteLine("Pouring into cup");    }
    void addLemon()
    {
        Console.WriteLine("Adding Lemon");    }
}

class Program
{
    static void Main(string[] args)
    {
        Coffee ob = new Coffee();
        ob.prepareRecipe();
        Tea ob2 = new Tea();
        ob2.prepareRecipe();
    }
}

// ժարանգությամբ կոդի կրկնությունից խուսափում
using System;
class CaffeineBeverage
{
    protected void boilWater()
    {
```



```

        Console.WriteLine("Boiling Water");
    }
    protected void pourInCup()
    {
        Console.WriteLine("Pouring into cup");
    }
}
class Coffee : CaffeinBeverage
{
    public void prepareRecipe()
    {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
    void brewCoffeeGrinds()
    {
        Console.WriteLine("Dripping Coffee through filter");
    }
    void addSugarAndMilk()
    {
        Console.WriteLine("Adding Sugar and Milk ");
    }
}
class Tea : CaffeinBeverage
{
    public void prepareRecipe()
    {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
    void steepTeaBag()
    {
        Console.WriteLine("Steeping the Tea");
    }
    void addLemon()
    {
        Console.WriteLine("Adding Lemon");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Coffee ob = new Coffee();
        ob.prepareRecipe();
        Tea ob2 = new Tea();
        ob2.prepareRecipe();
    }
}

```

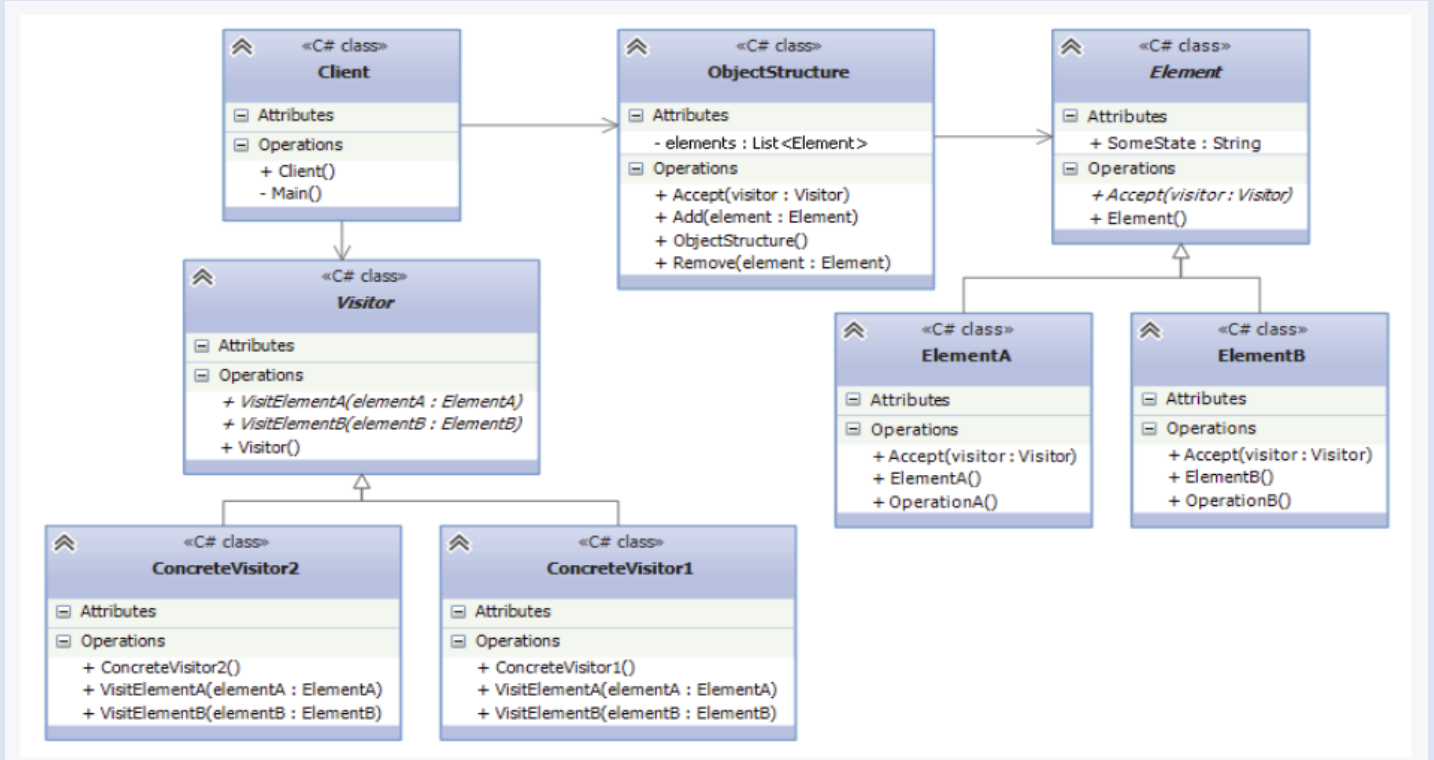
// prepareRecipe() մեթոդի կոդի կրկնությունն խուսափում: Միաժամանակ prepareRecipe() դառնում է Template Method

```
using System;
abstract class CaffeinBeverage
{
    public void prepareRecipe()
    {
        boilWater();
        brew();
        pourInCup();
        add();
    }
    protected void boilWater()
    {
        Console.WriteLine("Boiling Water");
    }
    protected void pourInCup()
    {
        Console.WriteLine("Pouring into cup");
    }
    public abstract void brew();
    public abstract void add();
}
class Coffee : CaffeinBeverage
{
    public override void brew()
    {
        Console.WriteLine("Dripping Coffee through filter");
    }
    public override void add()
    {
        Console.WriteLine("Adding Sugar and Milk ");
    }
}
class Tea : CaffeinBeverage
{
    public override void brew()
    {
        Console.WriteLine("Steeping the Tea");
    }
    public override void add()
    {
        Console.WriteLine("Adding Lemon");
    }
}
class Program
{
    static void Main(string[] args)
    {
        CaffeinBeverage ob = new Coffee();
        //Coffee ob = new Coffee();
        ob.prepareRecipe();
        Console.WriteLine("-----");
        ob = new Tea();
        //Tea ob2 = new Tea(); //ob2.prepareRecipe();
        ob.prepareRecipe();
    }
}
```

69. Visitor պատերն

[Шевчук – 276; GOF-379; Нэш -498]

- Visitor պատերնը գործողության պատերն է: Վիզիտորը հնարավորություն է տալիս առանց կլասների կառույցի փոփոխության ստանալ նրանց օբյեկտների նոր հնարավորություններ: Visitor պատերնի օգտագործման ժամանակ որոշվում է երկու խումբ կլասների հիերարխիա: Մեկը այն կլասն է որի համար պետք է ավելացնել հնարավորությունները: Երկրորդ հիերարխիան վիզիտորների համար է, որը այնտեղ որոշում է նոր հնարավորությունները:
- UML



- Visitor արստրակտ կլաս – վիզիտորի ինտերֆեյս:
- ConcreteVisitor – իրականացնում է վիզիտորի ինտերֆեյսը:
- Element արստրակտ կլաս – որոշում է Accept(), որը որպես պարամետր ստանում է Visitor օբյեկտ:
- ElementA, ElementB – իրականացնում է Accept() մեթոդը:
- ObjectStructure – որոշակի կառույց է, որը պահում է Element տիպի օբյեկտներ եւ նրանց հասանելիությունը: Նա կրող է լինել հասարակ ցուցակ կամ ավելի բարդ ծառատիպ կառույցներ:

❖ Visitor պատերնը ներդրված է C# -ի բազային կառույցում եւ իրականացվում է ընդլայնված մեթոդների միջոցով: Որպես օրինակ, ենթադրենք անհրաժեշտ է ավելացնել A կլասի օբյեկտներին Armenia() ընդլայնված մեթոդով:

```
A ob = new A();
ob.Armenia();
Console.ReadKey();
public static class Alb
{
    public static void Armenia(this A ob)
    {
        Console.WriteLine("A classum em");
    }
}
public class A
{
}
```

- ❖ Նախորդ օրինակը ներկայացնենք GOF -ի “վիզիտոր” պատերնով: Arm կլասը կդիտարկենք որպես “վիզիտոր” եւ նրա Armenia() մեթոդը կդառնա A կլասի անդամ: A կլասը օրինակում անվանենք ElementA կլաս:

```
using System;
using System.Collections;
    var structure = new ObjectStructure();
    structure.Adding(new ElementA());
    structure.accept(new ConcretVisitor1());
    structure.accept(new ConcretVisitor2());
Console.ReadKey();

abstract class Visitor
{
    public abstract void Armenia(ElementA eb);
}
class ConcretVisitor1 : Visitor
{
    public override void Armenia(ElementA eb)
    {
        Console.WriteLine("ElementA classum avelatsav Armenia1() method");
    }
}
class ConcretVisitor2 : Visitor
{
    public override void Armenia(ElementA eb)
    {
        Console.WriteLine("ElementA classum avelatsav Armenia2() method");
    }
}
abstract class Element
{
    public abstract void Accept(Visitor v);
}
class ElementA : Element
{
    public override void Accept(Visitor v)
    {
        v.Armenia(this);
    }
}
class ObjectStructure
{
    private ArrayList elements = new ArrayList();

    public void Adding(Element e)
    {
        elements.Add(e);
    }
    public void accept(Visitor visitor)
    {
        foreach (Element v in elements)
        {
            v.Accept(visitor);
        }
    }
}
```

Օբյեկտ տիպ

70. System.Object կլասս և նրա անդամները

[Троелсен-273; Troelsen – 263; Нэш-114; Нейгел-135]

- Բոլոր տիպերը առաջացել են System.Object կլասսից, նույնիսկ չափային տիպերը:
- Նկարագրենք Object կլասսի անդամները.

namespace System

```
{    public class Object
    {
        public virtual Boolean Equals(Object obj);
        public virtual Int32 GetHashCode();
        public Type GetType();
        public virtual String ToString();
        ~ Object(); // կամ // protected virtual void Finalize();
        protected Object MemberwiseClone();
        public static bool Equals(object objA, object objB);
        public static bool ReferenceEquals(object objA, object objB);
    }
}
```

- virtual Equals()** - Վերադարձնում է **true**, եթե երկու **հղումներ** նույն օբյեկտն են նշում: Օբյեկտների **պարունակությունը** համեմատելու համար անհրաժեշտ է վերաորոշել:
- virtual GetHashCode()** – Վերադարձնում է ամբողջ թիվ, որը ցույց է տալիս օբյեկտի **իդենտիֆիկացիոն** համարը: Այդ համարը վերագրվում է օբյեկտին նրա ստեղծման ժամանակ և ուղեկցվում է մինչև նրա վերանալը: Համարակալումը սկսում է 2 – ից:
- GetType()** – Վերադարձնում է օբյեկտի տիպը իր անունի տարածքով:
- virtual ToString()** - Լռելիությամբ վերադարձնում է օբյեկտի անունը իր անունի տարածքով:
- protected virtual Finalize()** - Ազատում է ռեսուրսները: Կանչվում է դետարուկտրի կողմից:
- protected MemberwiseClone()** - Ստեղծում է օբյեկտի **մակերեսային** կոպիան:
- static Equals()** - Ստուգում է երկու հղումները:
- static ReferenceEquals()** - Ստուգում է երկու հղումները:

using System;

class A //struct A

```
{
    public int a;
    public string name;
    public A(string s1, int a1)
    {
        a = a1;
        name = s1;
    }
}
```

// class M:A,Object //error

class M

```
{    static void Main()
    {
        A ob1 = new A("ani", 10);
        A ob2 = new A("nune", 10);
        Console.WriteLine("Type:{0}", ob2.GetType());
        Console.WriteLine("ToString:{0}", ob1.ToString());
        Console.WriteLine("hashcode ob1: " + ob1.GetHashCode());
        Console.WriteLine("hashcode ob2: " + ob2.GetHashCode());
        // ob1 = ob2;
        if(ob1.Equals(ob2))
            Console.WriteLine("yes");
        else
            Console.WriteLine("no");
    }
}
```

71. Object կլասի անդամների վերաորոշումը

❖ ToString() [Троелсен-276; Troelsen – 266; Хейгел-137]

- ❖ Կարելի է վերավորոշել ToString() - ը, քանի որ այն **virtual** է: Դա կատարվում է այն դեպքում, երբ օգտագործողը ցանկանում է տեսնել օբյեկտի տեքստային ներկայացումը իր ձևով: Ուղղակի պետք չէ քմահաճորեն այն ներկայացնել: ToString() - ը պետք է արտապատկերի ավելի բնորոշիչ ձևով:

```
using System;
    A ob = new A("T. Petrosyan");
    Console.WriteLine(ob.ToString());

class A
{
    string s;
    public A(string ss)
    {
        s = ss;
    }
    public override string ToString() //
    {
        return s.ToString();        }
}
```

❖ Equals() մեթոդների վերաորոշումը

[Троелсен-276; Troelsen – 266; Нэш-444,447; Рихтер-172]

- Equals() - ը համեմատում է երկու օբյեկտների հղումները: Եթե անհրաժեշտ է համեմատել օբյեկտի պարունակությունը, ապա այն պետք է վերավորոշել, քանի որ **virtual** է:

```
A ob = new A(99);
A ob2 = new A(99);
Console.WriteLine(ob.Equals(ob2)); // True
Console.WriteLine(ob==ob2); // False

class A
{
    private int a;
    public A(int aa)
    {
        a = aa;
    }
    public override bool Equals(object obj)
    {
        A temp;
        temp = (A)obj; //if ((A)obj).a == this.a)
        if (temp.a == this.a)
            return true;
        else
            return false;
    }
}
```

- Հարց: ինչու է թարգմանիչը տալիս “warning” Rebuild Solution թարգմանությունով: Որովհետև, եթե վերավորոշվել է Equals() վիրտուալ ֆունկցիան, ապա ցանկալի է վերաորոշել նաև GetHashCode() վիրտուալ ֆունկցիան: Հակառակ դեպքում կոմպիլիատորը տալիս է **զգուշացում**:

❖ GetHashCode() մեթոդի վերաորոշումը [Троелсен-277; Troelsen – 267; Нэш-448; Рихтер-175]

- Օբյեկտը իր ստեղծման ժամանակ ստանում է նաև **Hash կոդ**, որպեսզի տարբերվի մյուս օբյեկտներից: Կոդի գեներացման ժամանակ համակարգը օգտվում է օբյեկտի ընթացիկ զբաղեցրած հասցեից և ստանում թիվ: Գոյություն ունեն տարբեր ալգորիթմներ, որոնք կարող են վերատադրել Hash կոդ: Պարզագույն ձևերից է օրինակ ToString().GetHashCode() հրամանը:
- Օրինակը ցուցադրում է, թե ինչպես միևնույն **int a** արժեքի դեպքում Hash կոդը նույնն է ստացվում:

```
A ob = new A(99);
A ob2 = new A(99);
Console.WriteLine(ob.Equals(ob2));
Console.WriteLine(ob.GetHashCode());
Console.WriteLine(ob2.GetHashCode());
```

```

class A
{
    public int a;
    public A(int aa)
    {
        a = aa;
    }
    //public override int GetHashCode()
    //{
    //    return a.ToString().GetHashCode();
    //}
    public override bool Equals(object obj)
    {
        A temp = (A)obj;
        if (temp.a == this.a)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

- **GetHashCode()** -ի վերադարձումը միայն **Warning** ի դեմ չէ: Այն ավելի կարևոր և օգտակար կիրառում ունի: Օրինակ, Hashtable կոլեկցիայում (System.Collections անունի տարածք), տվյալներ գրելու համար անհրաժեշտ է երկուսը, և Equals և GetHashCode վերադարձել, որովհետև Hashtable -ը ճիշտ օբյեկտ ընտրելու համար կանչում է Equals() և GetHashCode() մեթոդները: Այլ խոսքով, եթե անհրաժեշտ է A կլասի օբյեկտները տեղադրել Hashtable կոլեկցիայում, ապա այդ դեպքում պարտադիր է երկու վերադարձումները:

❖ Անանուն տիպեր և GetHashCode.

```

var ob = new[]
{
    new
    {
        name = "Ani",
        age = 19
    },
    new
    {
        name = "Ani",
        age = 19
    },
    new
    {
        name = "Nune",
        age = 19
    }
};
Console.WriteLine(ob[0].GetHashCode());
Console.WriteLine(ob[1].GetHashCode());
Console.WriteLine(ob[2].GetHashCode());
Console.WriteLine(ob[0] == ob[1]);           // False
Console.WriteLine(ob[0] == ob[2]);           // False
Console.WriteLine(ob[0].Equals(ob[1]));      // True  // **
Console.WriteLine(ob[0].Equals(ob[2]));      // False // **
//string s1 = "a";
//string s2 = "a";
//Console.WriteLine(s1.Equals(s2));          // True  // **
//Console.WriteLine(s1 == s2);               // True

```

72. Object կլասի static անդամները

[Троелсен-279; Troelsen – 269; Нейгел-218]

- ```
public static bool Equals(object objA, object objB);
public static bool ReferenceEquals(object objA, object objB);
```
- Ստատիկ **ReferenceEquals()** մեթոդը համեմատում է միայն հղումները:
  - Եթե համեմատվող օբյեկտների տիպը հղիչային է, ապա ստատիկ **Equals()** և **ReferenceEquals()** մեթոդները տալիս են նույն արդյունքը, այսինքն համեմատում են հղումները:
  - Եթե համեմատվող օբյեկտների տիպը չափային է (ասենք ստրուկտուրա է), ապա **Equals()** և **ReferenceEquals()** մեթոդները տալիս են տարբեր արդյունքներ, այսինքն **Equals()** համեմատում է օբյեկտները իսկ **ReferenceEquals()** -ը համեմատում է հղումները:
  - Ստատիկ **Equals()** մեթոդը նման է օբյեկտի **Equals()** մեթոդին, ուղղակի այն հնարավոր չէ վերաորոշել և ավելի պարադոքս միտք ավելացնենք – օբյեկտի **Equals()** վերորոշման ժամանակ նույնպես ստատիկը վերաորոշվում է: Նրանք բացարձակ նույնն են նաև չափային տիպերի դեպքում:
  - Օբյեկտի **Equals** -ը և ստատիկ **Equals** -ը և կլասի և ստրուկտուրայի ժամանակ տալիս են նույն արդյունքները:
  - Օրինակը ցույց է տալիս, որ երեք **Equals** -ները նախատեսված են հղիչային տիպ, չափային տիպ և վերաորոշման կիրառման տարբեր կոմբինացիաների համար:

```
using System;
```

```
//struct A //struct - value type
```

```
class A
```

```
{
 public int a;
 public string name;
 public A(string s1, int a1)
 {
 a = a1;
 name = s1;
 }
 // public override bool Equals(object obj)
 // {
 // A temp = (A)obj;
 // if (temp.a == this.a)
 // {
 // return true;
 // }
 // else
 // {
 // return false;
 // }
 // }
```

```
}
class M
```

```
{
 static void Main()
 {
 A ob1 = new A("Ani", 10);
 A ob2 = new A("Ani", 10);
 //Console.WriteLine("Working with value types"); // struct
 Console.WriteLine("Working with reference types"); //class
 Console.WriteLine(ob1.Equals(ob2)); //false
 Console.WriteLine(Equals(ob1, ob2)); //false
 Console.WriteLine(ReferenceEquals(ob1, ob2)); //false
 }
} //Troelsen A.(2020 C#8 Core3) pg.254 sxa1 Object.Equals
```



## 73. Object.MemberwiseClone()

[Համ-115, 427]

- **MemberwiseClone()** մեթոդը հնարավորություն է տալիս կատարել օբյեկտների **մակերեսային** կոպիա: Դա նշանակում է, որ չափային տիպերը կրկնօրինակվում են կատարելով բիթ առ բիթ կոպիա, իսկ օբյեկտի կոմպոզիցիաները (հղիչային օբյեկտները) հղում են նույն օբյեկտին, այսինքն չկա օբյեկտի կոպիա:
- Եթե այսպիսի կոպիան չի բավարարում, ապա կա հնարավորություն կատարել ավելի լիարժեք կոպիա, օգտվելով **ICloneable** ինտերֆեյսից կատարելով նրա բոլոր պահանջները, մասնավորապես **Clone()** մեթոդի իրականացումը, որը կանցնենք ավելի ուշ:
- **protected** - ը ստիպում է սահմանափակվել այն հնարավորությունով, որ միայն կոպիա կատարող կլասը կարող է կատարել **MemberwiseClone()** հրամանը չօգտվելով **new** – ից և կոնստրուկտորի կանչից: **public** – ի դեպքում իրավիճակը ավելի կբարդանար այլ կլասների միջամտման պատճառով:

```
class A
{
 public void f()
 {
 A ob = new A();
 A ob2 = new A();
 ob2 = (A)ob.MemberwiseClone(); //ok!
 }
}
class M
{
 static void Main(string[] args)
 {
 A ob = new A(); // M ob = new M();
 A ob2 = new A(); // M ob2 = new M();
 ob2 = (A)ob.MemberwiseClone(); // error ? //why?
 // ob2 = (M)ob.MemberwiseClone();
 }
}
//-----
```

- ❖ Հաջորդ օրինակը ցույց է տալիս, որ **B** կլասի **N** –ին, եթե վերագրում ենք **1990**, ապա երկու օբյեկտների համար այն նույնն է կամ այլ խոսքով հղում են նույն արժեքին, քանի որ կա մեկ կոպիա, դրա համար էլ կոչվում է մակերեսային կոպիա: **A** կլասի **name** դաշտը օգտագործվում է ուղղակի առանց կոմպոզիցիայի, դրա համար նրանից կա երկու օրինակ, “**Ani**” և “**Nune**” տարբեր արժեքներ տարբեր օբյեկտներին համար:

```
using System;
//struct B // deep Clone
class B
{
 public int N;
 public B(int n)
 {
 N = n;
 }
}
class A
{
 public string name;
 public B b;
}
```

```

class M : A
{
 public static void Main()
 {
 M ob1 = new M();
 ob1.name = "Ani";
 ob1.b = new B(2999);
 M ob2 = (M)ob1.MemberwiseClone();
 Console.WriteLine(ob1.name + ob1.b.N);
 Console.WriteLine(ob2.name + ob2.b.N);
 Console.WriteLine("-----");
 ob1.name = "Nune";
 ob1.b.N = 3000;
 Console.WriteLine(ob1.name + ob1.b.N);
 Console.WriteLine(ob2.name + ob2.b.N);
 }
}
//-----

```

- Եթե, կոմպոզիցիան փոխենք ժառանգականությամբ, ապա կկատարվի ամբողջական կլոնավորում:

```

using System;
class B
{
 public int N;
 public B()
 {
 }
 public B(int n)
 {
 N = n;
 }
}
class A:B
{
 public string name;
}
class M : A
{
 public static void Main()
 {
 M ob1 = new M();
 ob1.name = "Ani";
 ob1.N = 55;
 M ob2 = (M)ob1.MemberwiseClone();
 Console.WriteLine(ob1.name + ob1.N);
 Console.WriteLine(ob2.name + ob2.N);
 Console.WriteLine("-----");
 ob1.name = "Nune";
 ob1.N = 1990;
 Console.WriteLine(ob1.name + ob1.N);
 Console.WriteLine(ob2.name + ob2.N);
 }
}

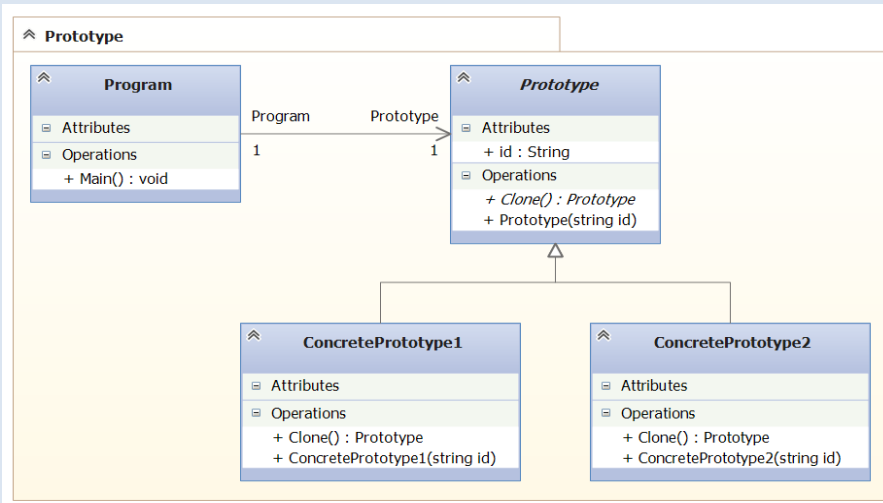
```

---

## 74. Prototype պատերն

[Шевчук – 72; GOF-146]

- Օբյեկտ ստեղծող պատերն: Հնարավորություն է տալիս ստեղծել նոր օբյեկտ-կլոն օրիգինալ պրոտոտիպ օբյեկտից:
- UML



- Prototype աբստրակտ կլաս, որը ստիպում է իրականացնել Clone() մեթոդը և հանդիսանում է կապող ինտերֆեյս: Այն պահպանում է օբյեկտի դաշտերը կամ այլ խոսքով կլոնի տվյալները:
- ConcretePrototype –ը իրականացնում է ինքնակլոնավորումը ըստ պարտականության: Կլոնավորումը կատարվում է բազային ցուցիչով ժառանգին դիմելով, որը բազային կոնստրուկտորին պարամետրեր ուղղարկելով: (C# -ում :base հրամանով):
- Client կամ Program կլասը սեղծում է պրոտոտիպի օրինակը, բազային ցուցիչով ժառանգին դիմելով և կատարում է կլոնավորում Clone() մեթոդով:

```
using System;
abstract class Prototype
{
 public int Id { get; set; }
 public Prototype(int id)
 {
 this.Id = id;
 }
 public abstract Prototype Clone();
}
class ConcretePrototype : Prototype
{
 public ConcretePrototype(int id) : base(id) { }
 public override Prototype Clone()
 {
 return new ConcretePrototype(Id);
 }
}
class Program
{
 {
 Prototype prototype = new ConcretePrototype(99);
 Prototype clone1 = prototype.Clone();
 Prototype clone2 = prototype.Clone();
 Console.WriteLine(prototype.Id);
 Console.WriteLine(clone1.Id);
 Console.WriteLine(clone2.Id);
 Console.WriteLine("Change clone1");
 clone1.Id = 88;
 Console.WriteLine("prototype = " + prototype.Id);
 Console.WriteLine("clone1 = " + clone1.Id);
 Console.WriteLine("clone2 = " + clone2.Id);
 }
}
```

## 75. dynamic տիպ

[Троелсен-600; Troelsen-668; Нэш-557; Шилдт-703; Нейгел-359]

- **object, var, dynamic** – ոչ ակնհայտ տիպային հայտարարում:
- **object** և **var** տիպիզացումը կատարվում է ծրագրի թարգմանման ժամանակ:
- **var** – փոփոխականի տիպը որշվում է առաջին վերագրման ժամանակ և փոփոխման ենթակա չէ:
- **object** – փոփոխականի տիպը կարող է փոխվել:
- **dynamic** – Համարժեք է **object** տիպին, սակայն տիպայնացումը կատարվում է կատարման ժամանակ և չկա թարգմանման հսկում:
- Ի տարբերություն **var** – ի դինամիկ տիպը կարող է լինել կլասի դաշտ, ֆունկցիայի պարամետր, վերադարձվող արժեքի տիպ:
- **dynamic** կարող են լինել նաև լոկալ փոփոխականները:

```
class A
```

```
{
 dynamic a = 999;
 dynamic p { get; set; }
 public dynamic f(dynamic k)
 {
 return k;
 }
}
```

### Սահմանափակումներ

- ✓ Դինամիկ տիպը չի կարող ընդունել **լամդա** արտահայտություն կամ անանուն մեթոդ: (ավելի ուշ)

```
dynamic a = GetDynamicObject();
```

```
a.Method(arg => Console.WriteLine(arg));
```

- ✓ Չի կարող հասկանալ ընդլայնված մեթոդներ և այդ պատճառով **LINQ-ում** չի կիրառվում: (ուշ)

```
dynamic a = GetDynamicObject();
```

```
var data = from d in a select d;
```

- ✓ Դինամիկ տիպը չի կարող օգտագործվել որպես ինդեքստորի ինդեքս: Հնարավոր չէ **object** և **dynamic** տիպերը համատեղ օգտագործել որպես գերբեռնված մեթոդի պարամետր:
- Դինամիկ տիպի օգտագործումը ստիպում է կատարել ընտրություն տիպի **անապահովության և օպտիմալ կոդի** միջև: Այն **կիրառելի** է ռեֆլեքսիայում ուշացած կապվածության համար: Կամ ասենք COM գրադարաններից օգտվելու համար (office փաթեթի տվյալներ):
- dotNET – ում օգտագործվում է **Dynamic Language Runtime (DLR)**. DLR -ը նորույթ չէ և օրինակ Smalltalk, LISP, Ruby, Python լեզուները աշխատում են դինամիկ միջավայրում:
- Կողը ստանում է մեծ ճկունություն: Կարելի է հանգիստ կատարել կոդում փոփոխություն, չդիմելով տիպերի բարդ վերափոխման մեխանիզմին:
- Փոխհարաբերության պարզեցում տարբեր պլատֆորմների և լեզուների միջև:
- Կատարման ժամանակ հիշողությունում տիպերի անդամների ավելացում կամ փոփոխում:
- Համագործակցում **IronPython** և **IronRuby** լեզուների հետ, որոնք աշխատում են դինամիկ միջավայրում և ունեն NET գրադարաններից օգտվելու հնարավորություն:

```
using System;
```

```
class M //Троелсен-360
```

```
{
 static void Main()
 {
 // var v = 3;
 // v = 'a'; //error!!!
 dynamic a = 555; // Run Time error!!
 // object a = 555; // Compiler error!!
 bool b = true;
 //a = b; //ok
 b = a;
 dynamic t = "Hello";
 Console.WriteLine(t.ToUpper());
 }
}
```

```

 Console.WriteLine(t.toupper());
 a += 5; // dynamic ok, object error!!
 }
}

//----- class ExpandoObject
using System;
class M // նման javascript-ին // Dynamic.ExpandoObject()
{
 static void Main()
 {
 dynamic viewbag = new System.Dynamic.ExpandoObject();
 viewbag.Name = "Tom";
 viewbag.Age = 46;
 Console.WriteLine(viewbag.Name+ " "+viewbag.Age);
 }
}

```

#### ❖ Performing **switch** Statement Pattern Matching (New 7.0, Updated 9.0) [Troelsen-105] N no video

- Ընդհանրապես **switch** օպերատորը թույլատրում է **object** կամ **dynamic** տիպ օգտագործել որպես պարամետր: Այն փոխակերպվում է **int**, **string**, **decimal** կամ այլ տիպերի և աշխատում է որպես **case** -ի պարամետր պահպանելով նույն տիպայնությունը: C#7.0 -ից սկսած **case** -ի պարամետրը չի համարվում հաստատուն տիպային շաբլոնով աշխատող, այլ **case** -ի պարամետրերը կարող են ստանալ տարբեր տիպերի արժեքներ: Կա սահմանափակում՝ **goto** չի աշխատում տվյալ շաբլոնի դեպքում:

```

for (; ;)
{
 object obj; // dynamic obj;
 Console.Write("input = ");
 string str=Console.ReadLine();
 if (int.TryParse(str, out int a))
 obj = a;
 else if (decimal.TryParse(str, out decimal dd))
 obj = dd;
 else if (str is string)
 obj = str;
 else
 obj = null;
 switch (obj)
 {
 case int i:
 Console.WriteLine("Your choice is an integer {0}.", i);
 break;
 case decimal d:
 Console.WriteLine("Your choice is a decimal. {0}", d);
 break;
 case string s: //
 Console.WriteLine("Your choice is a string. {0}", s); //
 break; //
 //case string s when s == "aa":
 // Console.WriteLine("Your choice is a string. {0}", s);
 // break;
 default:
 Console.WriteLine("Default");
 break;
 }
}

```

## 76. IronPython (պրակտիկ)

[albdarb]

- Դիմումիկ ծրագրավորման լեզվի հրամանների օգտագործում C# -ից:
- References->Manage NuGet Packages...->Browse->DLR-> DynamicLanguageRuntime->**Install**, որը հնարավորություն կտա `using Microsoft.Scripting`; անունի տարածքի կիրառությունը:
- Այժմ փնտրենք -> IronPython -> **Install**, որը հնարավորություն կտա ավելացնել անունի տարածքներ.  
`using IronPython.Hosting;`  
`using Microsoft.Scripting.Hosting;`

- Ստեղծենք Notepad -ով barev.py ֆայլը, որը ընդգրկի Python հրամաններ.

```
print 'Barev PAYTHON fileits'
x = 10
y = 20
z = x + y
print z
def fact(n):
 result = 1
 for i in xrange(2, n + 1):
 result *= i
 return result
```

- Օգտագործենք Python

```
//using Microsoft.Scripting;
using System;
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
class Program
{
 static void Main(string[] args)
 {
 ScriptEngine e = Python.CreateEngine();
 e.Execute("print 'barev Pythonov'");
 ScriptScope scope = e.CreateScope();
 e.ExecuteFile("C://1//barev.py", scope);
 dynamic xx = scope.GetVariable("x");
 dynamic yy = scope.GetVariable("y");
 dynamic zz = scope.GetVariable("z");
 Console.WriteLine($"Sum {xx} and {yy} = {zz}");
 Console.WriteLine("mutq tiv = ");
 int x = Int32.Parse(Console.ReadLine());
 dynamic f = scope.GetVariable("fact");
 dynamic result = f(x); // call function from Python
 Console.WriteLine(result);
 }
}
```

.....

# Ինտերֆեյս

## 77. Ինտերֆեյս

[Шилдг-375; Троелсен- 306; Troelsen – 297]

- Ինտերֆեյսը տրամաբանական կառույց է: Նրան օգտագործելու համար պետք է ժառանգել:
- Ինտերֆեյսը որոշում է անդամներ, որոնք պետք է **պարտադիր իրականացնեն** կլասները կամ ստրուկտուրաները: Ինքը ինտերֆեյսը **չի իրականացնում** այդ անդամները (**Framework**):
- Ինչպես գիտենք արտոնական կլասները ունենալով արտոնական մեթոդներ ստիպում են ժառանգման ժամանակ օգտագործող կլասներին անպայման իրականացնել այդ մեթոդները (արտոնական կլասներից չի կարելի օբյեկտ ստանալ): Այս ձևով արտոնական կլասը դառնում է ինտերֆեյս մյուս կլասների համար: C# -ում այս միտքը զարգացվել է առանձին ձևով – ինտերֆեյսներով:
- Հիշենք, որ արտոնական կլասները կարող են ունենալ ոչ արտոնական անդամներ և նրանց կարելի է իրականացնել ժառանգ կլասների միջոցով:
- Ինտերֆեյսում ոչ մի մեթոդ չի կարող ունենալ որոշում՝ կոդ (**եթե Framework** -ով ենք աշխատում), արտոնակում – ոչ արտոնական մեթոդները կարող են: Այլ խոսքով ինտերֆեյսում նշվում է ինչ ձևով պետք է իրականացվի մեթոդը, բայց կոնկրետ չի նշվում ինչպես: Եթե կլասը օգտագործում է ինտերֆեյս, ապա նա պարտավոր է ինտերֆեյսի բոլոր անդամները որոշել՝ այսինքն կիրառել:

Գրման ձևը՝

```
public interface I2
{
 bool f();
 int p {get;set;}
 string this [int x]{get; set;}
 event d e;
}
```

Օգտագործումը՝

- ```
class A : I2
{
    //.....
}
```
- Ինտերֆեյսը կարող է հայարարել մեթոդ, հատկանիշ, ինդեքսատոր, իրադարձություն:
 - Ինտերֆեյսը չի կարող ունենալ **սովյալների դաշտ**: Չեն կարող որոշել կոնստրուկտոր, դեստրուկտոր: Նրա ոչ մի անդամ չի կարող լինել **static** (**եթե Framework** է): C#9 –ի Core –ում թույլատրվում է ստատիկ անդամներ և նունիսկ ոչ ստատիկ մեթոդներ: Իմաստ չունի գրել անդամներին թույլատրելիության մոդիֆիկատոր **public**, որովհետև նա միշտ բաց է (**եթե Framework** է): Այլ խոսքով C#8 -ից սկսած ինտերֆեյսի անդամները կարող են ստանալ թույլատրության մոդիֆիկատոր:
 - C#9 –ի Core –ում թույլատրվում է **նաև** պայմանագրային մեթոդը նշել **abstract**:
 - Ժառանգման ժամանակ նույնպես անհրաժեշտ է թույլատրել **public** –ով:

```
using System;
A ob = new A();
Console.WriteLine(ob.f());
interface I2
{
    bool f();
}
class A : I2
{
    public bool f()
    {
        return true;
    }
}
```

.....

78. Ինտերֆեյսի հղում

[Шилдт-381; Троелсен-313]

- Ինտերֆեյսի հղումով կարելի է կանչել միայն ինտերֆեյսում հայտարարված գործողությունները:

```
interface I2
{
    void f();
}
class A: I2
{
    public void f()
    {
        Console.WriteLine("barev");
    }
    public void f2()
    {
        Console.WriteLine("barev 2");
    }
}
```

- Կարելի է կլասի անդամին, որը ժառանգել է ինտերֆեյս, դիմել հետևյալ ձևով
A ob = new A();
ob.f();
- Կարելի է նաև ստեղծել ինտերֆեյսի հղում և դիմել կլասի անդամին **ինտերֆեյսի հղման** միջոցով:
A ob = new A();
I2 iob = (I2)ob; // explicit կամ implicit
iob.f();
կամ մեկ տողով. I2 iob = (I2)new A();
iob.f2() // error
- Այսպիսով կարելի է հայտարարել հղիչային փոփոխական ինտերֆեյսային տիպի: Այդպիսի փոփոխականը կարող է հղել ցանկացած օբյեկտ, որը իրականացնում է ինտերֆեյսը: Երբ ինտերֆեյսը հղիչի միջոցով է կանչում մեթոդը, կատարվում է մեթոդի այն տարբերակը, որը իրականացնում է այդ օբյեկտը: Դա նման է բազային հղիչով ածանցյալ կլասի օբյեկտներին դիմելուն, կամ այլ խոսքով **պոլիմորֆ** օգտագործմանը:

```
A ob1 = new A();
B ob2 = new B();
I2 iob;
iob = ob1;
iob.f();
iob = ob2;
iob.f();
public interface I2
{
    void f();
}
class A : I2
{
    public void f()
    {
        Console.WriteLine("fA");
    }
}
class B : I2
{
    public void f()
    {
        Console.WriteLine("fB");
    }
}
```

❖ Interface-Default Implementations (New 8.0) [Troelsen-306] N

- While this is a trivial example, it does demonstrate one of the problems with default interfaces. Before using this feature in your code, make sure you measure the implications of the calling code having to know where the implementation exists.

```
I obi = new A();
//I obi =new A() { x=5,y=7};
Console.WriteLine(obi.z);
interface I
{
    int x { get; set; }
```



```

int y { get; set; }
int z => x * y;
}
class A : I
{
    public int x{ get; set; }
    public int y{ get; set; }
}

```

79. Ինտերֆեյսի ստատիկ անդամներ

[Troelsen – 308] N (no video)

- C#8 –ից սկսած .Net Core համակարգով կարելի է ինտերֆեյսներում **օգտագործել ստատիկ անդամներ**: Այս դեպքում ինտերֆեյսը որոշ հատկություններ է ձեռք բերում, որը **յուրահատուկ է աբստրակտ** կլասներին: Մասնավորապես կարող է ունենալ **ստատիկ դաշտ**, ստատիկ մեթոդ, ստատիկ հատկանիշ, ինչպես նաև **ստատիկ կոնստրուկտոր** և նույնիսկ **ստատիկ ստվորական ոչ ստատիկ մեթոդ**: Ոչ ստատիկ մեթոդին կարելի է դիմել միայն ինտերֆեյսի հղումով, որը կցված է (իմպլեմենտացիա) որևէ կլասի օբյեկտի, սակայն նրա նկարագրությունը որպես գերբեռնված անուն արդեն արգելվում է (**օրինակում /****):
- Ստատիկ անդամները **լռելիության բաց** են և կարելի է կիրառել թույլատրության (**public, private, protected, internal**) մոդիֆիկատորներ:
- Այսպիսով, ինտերֆեյսը **չի կարող** ունենալ ոչ ստատիկ դաշտ, ոչ ստատիկ կոնստրուկտոր, ստատիկ ինդեքսատոր: Ինտերֆեյսը **չի կարող** լինել ստատիկ:

```

I.fs();
A ob = new A();
// ob.f();// error
I obi = ob;
obi.f();
interface I
{
    //double d; //error
    static int x = 8;
    // int p { get; set; } // անհրաժեշտ է իրականացում (implementation)
    static void fs()
    {
        Console.WriteLine("static " + x);
    }
    void f()
    {
        //fs();
        Console.WriteLine("no static " + x);
    }
    static I() // static constructor // ok
    {
        x = 9;
        Console.WriteLine("static constructor");
    }
    // public I() // constructor // error
    // {
    // }
    void ff()
    {
        Console.WriteLine("ffI");
    }
    // public void ff(); // error /**
}
class A:I
{
    public void ff()
    {
        Console.WriteLine("ffA");
    }
}

```

80. Ինտերֆեյսի **is** և **as** պատկանելիության ձևեր

[Троелсен-314; Troelsen – 305]

- ❖ Շատ դեպքերում պարզ չի լինում թե արդյոք կլասը անհրաժեշտ ինտերֆեյս ապահովում է, թե ոչ: Պատկանելիությունը ստուգելու համար կա երեք ձև.
- ✓ առաջինը - **is** - այս դեպքում համեմատվում է օբյեկտի հետ և անհամապատասխանության դեպքում վերադարձնում է **false**:
- ✓ երկրորդը – **as**, այս դեպքում եթե հղումը ինտերֆեյս չի հայտնաբերում, ապա ստանում ենք **null** հղում:
- ✓ երրորդը – արտակարգ իրավիճակի մշակումով **try...catch**, որը տալիս է **InvalidCastException** իրավիճակ և որը պետք է մշակել (**կհ, ավելի ուշ կանցնենք**):
- **is** կարելի է օգտագործել **ցանկացած տիպի** համար, որը կստուգի տիպի պատկանելությունը: **as** -ը կարելի է օգտագործել միայն հղիչային տիպերի համար:

```
using System;
int a = 22;
if(a is int)
    Console.WriteLine("yes");
else
    Console.WriteLine("no");
string s="aaa";
//string s3 = s as string;
//if (s3==null) //
if(s as string!=null) //
    Console.WriteLine("Ayo");
else
    Console.WriteLine("Voch");
Console.ReadKey();
```

- Այժմ բերենք ինտերֆեյսի օրինակ`

```
❖ interface is
using System;
A ob = new A();
if (ob is I2 obi) // C#7
{
    Console.WriteLine(ob.PP);
    // or
    Console.WriteLine(obi.PP);
}
else
    Console.WriteLine("don't");
Console.ReadKey();
interface I2
{
    byte PP
    { get; }
}
class A : I2
{
    public byte PP
    {
        get
        {
            return 44;
        }
    }
}
```

```
❖ interface as
using System;
interface I2
{
    byte PP
    {
        get;
    }
}
class A : I2
{
    public byte PP
    {
        get
        {
            return 44;
        }
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        I2 iob = ob as I2;
        if (iob != null)
            Console.WriteLine(iob.PP);
        else
            Console.WriteLine("don't");
    }
}
```

```
❖ interface try..catch
using System;
    A ob = new A();
    I2 iob;
    try
    {
        iob = (I2)ob;
        Console.WriteLine(iob.PP);
    }
    catch (InvalidCastException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```
interface I2
{
    byte PP { get; }
}
class A : I2
{
    public byte PP
    {
        get { return 4; }
    }
}
```

81. Ինտերֆեյսը որպես պարամետր և վերադարձվող արժեք

[Troelsen-315; Troelsen-308, 310]

- Ինտերֆեյսը կարող է օգտագործվել ինչպես մեթոդի պարամետր, այնպես էլ մեթոդի վերադարձվող արժեք:

```
interface I2
{
    void ff();
}
class A : I2
{
    public void ff()
    {
        System.Console.WriteLine("interface - parameter");
    }
}
class M
{
    static void ffM(I2 iob)
    {
        iob.ff();
    }
    static void Main()
    {
        A ob = new A();
        I2 iob2 = ob;
        ffM(iob2); // ffc((I2)ob);
    }
}
```

// Ինտերֆեյսը որպես վերադարձվող արժեք

```
interface I2
{
    void f();
}
class A : I2
{
    public void f()
    {
        System.Console.WriteLine("interface - return");
    }
}
class M
{
    static I2 fm(object ob)
    {
        return (I2)ob;
    }
    static void Main()
    {
        A ob = new A ();
        I2 iob = fm(ob);
        iob.f();
    }
}
```

82. Ինտերֆեյսների ժառանգականություն, բազմաժառանգում

[Шилдг -387; Троелсен-322; Troelsen-317; 320]

- Ինչպես կլաստում ստեղծվում է ժառանգականություն, այնպես էլ ինտեֆեյսը կարող են կազմվել ենթակայության սկզբունքով, կազմելով հիերարխիկ ծառ:
- C# - ում կլասին չի թույլատրվում ժառանգել մեկից ավելի կլասներ:
- Թույլատրվում է մեկից ավելի ինտերֆեյսների ժառանգության կիրառությունը:

```
class C : A, I1, I2
{
}
```

- Ինտերֆեյսում նույնպես թույլատրելի է բազմաժառանգականությունը և “**risk collision**”:

```
interface I1 : I2, I3
{
    void ff();
}
```

```
interface I1
```

```
{
    void ff();
}
```

```
interface I2 : I1
```

```
{
    void ff2();
}
```

```
class A : I2
```

```
{
    public void ff()
    {
        System.Console.WriteLine("interface 1");
    }
    public void ff2()
    {
        System.Console.WriteLine("interface 2");
    }
}
```

```
class M
```

```
{
    public static void Main()
    {
        A ob = new A();
        I2 iob2 = (I2)ob;
        iob2.ff();
        iob2.ff2();
    }
}
```

83. Ինտերֆեյսների բացահայտ իրականացում

[Шилдг -388; Троелсен- 320; Troelsen-314]

- Ինտերֆեյսները կարելի է օգտագործել **նաև բացահայտ** իրականացումով:
- Բացահայտ իրականացման դեպքում անդամը միշտ ոչ ակնհայտ **փակ** է:
- Բացահայտ իրականացման դեպքում անդամը **չի կարող** ընդունել մոդիֆիկատորներ **abstract, virtual, override, new**:
- Բացահայտ իրականացումը օբյեկտով չի կարող աշխատել: Օգտագործման **միակ ձևը** ինտերֆեյսի հղումով իրականացումն է:
- Եթե անհրաժեշտ է օբյեկտով կանչ, ապա պետք է գրված լինի ոչ բացահայտ իրականացում, որը նույնպես **կարելի է** կանչել ինտերֆեյսի հղումով:

```
interface I
{
    void ff();
}

class A:I
{
    void I.ff()
    {
        System.Console.WriteLine("Welcome");
    }
}

class M
{
    static void Main()
    {
        I iob = new A();
        iob.ff();
    }
}

//-----
```

- Հնարավոր է դեպք, երբ կլասը ունի բազմաժառանգում ինտերֆեյսներով և մեթոդների անունները նշված ինտերֆեյսներում **համընկնում** են: Նշված դեպքում կարելի է օգտագործել **բացահայտ իրականացում**, եթե անհրաժեշտ է իրականացնել անհրաժեշտ ինտերֆեյսով: (**risk collision** –ի դեմ):

```
interface I1
{
    void ff();
}

interface I2
{
    void ff();
}

class A : I1, I2
{
    void I1.ff()
    {
        System.Console.WriteLine("A.I1.ff called");
    }
    void I2.ff()
    {
        System.Console.WriteLine("A.I2.ff called");
    }
    public void ff()
    {
        System.Console.WriteLine("A ff() called");
    }
}

class M
{
    static void Main()
    {
        A ob = new A();
        ob.ff();
        ((I1)ob).ff();
        ((I2)ob).ff();
    }
}
```

84. Ինտերֆեյս ICloneable

[Троелсен- 331; Troelsen-329; Шилдт-779]

- Ինչպես նշել ենք Object կլասը ունի մակերեսային կոպիայի (կլոնավորման) հնարավորություն MemberwiseClone() մեթոդով: Եթե անհրաժեշտ է **խորը** (Deep) կլոնավորում, այսինքն կոմպոզիցան նույնպես կլոնավորել, այլ ոչ թե հղել, ապա օգտագործվում է **ICloneable** ինտերֆեյսը: Ժառանգելով այն **պարտավորվում** ենք իրականացնել **public object Clone()** մեթոդը:
- Հնարավոր է մակերեսային է կլոնավորում օգտագործելով MemberwiseClone() մեթոդը Clone() մեթոդում:
- Օրինակում կատարվում է խորը կլոնավորում, Clone() մեթոդում գրելով կոմպոզիցիայի անդամների վերագրումը: Այս հնարավորությունը նման է կոպիայի կոնստրուկտորին:

```
using System; // Deep Copy // խորը կլոնավորում
class B
{
    public int x;
}
class A : ICloneable
{
    public string name;
    public B obB = new B();
    public A(int x, string s)
    {
        this.name = s;
        obB.x = x;
    }
    public override string ToString()
    {
        return string.Format("{0}; {1}", name, obB.x);
    }
    public object Clone()
    {
        A temp = (A)this.MemberwiseClone();
        B ob = new B();
        ob.x = this.obB.x;
        temp.obB = ob;
        return temp;
        //return this.MemberwiseClone(); // shallow copy
    }
}
class M
{
    static void Main()
    {
        A ob1 = new A(100, "ani");
        A ob2 = (A)ob1.Clone();
        Console.WriteLine(ob1);
        Console.WriteLine("clone.....");
        Console.WriteLine(ob2);
        Console.WriteLine();
        Console.WriteLine("changed.....");
        ob2.name = "nune";
        ob2.obB.x = 999;
        Console.WriteLine(ob1);
        Console.WriteLine("clone.....");
        Console.WriteLine(ob2);
    }
}
```

85. Ինտերֆեյս `IEnumerable`

[Троелсен- 326; Troelsen-322; Шилдт - 1001]

- Ինչպես գիտենք օբյեկտին կարելի է դիմել ինդեքսացիայով, եթե կիրառենք **ինդեքսատոր**: `for()` ցիկլով կարելի է օգտագործել ինդեքսատոր, սակայն `foreach()` ցիկլով այն **հնարավոր չէ**: Որպեսզի կարողանանք `foreach()` ցիկլով կատարել կլասի օբյեկտի ինդեքսացիա, ապա պետք է ընդունել պայմանագրային մոտեցում և օգտագործել `IEnumerable` ինտերֆեյսը:
- `IEnumerable` ինտերֆեյսի ժառանգման (իմպլիմենտացիայի) ժամանակ կլասը պարտավորվում է իրականացնել `GetEnumerator()` մեթոդ, որը ապահովում է տվյալ կլասի օբյեկտի ինդեքսավորված ընթերցումը:
- Օրինակ՝ `foreach` հրամանը կարողանում է ցանկացած զանգվածի անդամներ կարդալ ըստ հերթականության, որովհետև `Array` կլասը իրականացնում է `IEnumerable` ինտերֆեյսը: Այսինքն, ցանկացած տիպ կարող է ունենալ նույն պահվածքը՝ կարդալ ըստ հերթականության, եթե այն ժառանգում է `IEnumerable` ինտերֆեյսը և իրականացնում է `GetEnumerator()` մեթոդը: `GetEnumerator()` մեթոդը ֆորմալ ձևով հայտարարված է `IEnumerable` ինտերֆեյսում, որը գտնվում է `System.Collections` անունի տարածքում և վերադարձնում է `IEnumerator` ինտերֆեյս:
- Ինտերֆեյսների կիրառման առանձնահատկություններից է, որ նույնիսկ, եթե չկա ժառանգում ինտերֆեյսից, սակայն տվյալ ինտերֆեյսի կոնտրակտային մեթոդը (կամ այլ իմպլեմենտացիաներ) կա, ապա կոնտրակտը կկատարվի: Միայն այն դեպքում է պետք ինտերֆեյսի ժառանգումը, երբ որպես կլիենտ այլ կլասներ են օգտվում տվյալ կլասից և օգտվում են կոնտրակտային մեթոդից:

```
//-----
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
//-----

using System;
using System.Collections;
// class A //ok!
class A: IEnumerable //
{
    public char[] Z = { 'a', 'b', 'c', 'd' };
    public IEnumerator GetEnumerator()
    {
        return Z.GetEnumerator();
    }
    // Indexer
    // public char this[int i]
    // {
    //     get
    //     {
        return Z[i];
    }
}
class M
{
    static void Main(string[] args)
    {
        A ob = new A();
        foreach(char v in ob) // չի կարդում ինդեքսատորով
            Console.Write(v);
        // Console.WriteLine();
        // for(int i=0;i<4;i++)
        //     Console.Write(ob[i]);
    }
}
```


86. Ինտերֆեյս `IEnumerator`, `foreach`

[Троелсен- 326; Troelsen-322; Шилдт – 1001]

- `IEnumerator` ինտերֆեյսի ժառանգման դեպքում պարտադիր պետք է իրականացնել նրա անդամները: Ծրագրավորողը կարող է **ինքնուրույն** որոշել ներքին անդամների գործողությունները, որը ավելացնում է ծրագրավորողի աշխատանքը:

```
public interface IEnumerator
{
    bool MoveNext(); // Կուրսորի տեղաշարժ
    object Current { get; } // Ընթացիչ էլեմենտի ստացում
    void Reset(); // Տեղաշարժվող կուրսորի տեղաշարժի “reset”
}
//-----
using System.Collections; // Шилдт - 1001
class A : IEnumerator, IEnumerable
{
    char[] Z = { 'a', 'b', 'c', 'd' };
    int i = -1; // որովհետև foreach սկսում է MoveNext() հրամանով և i++ ու
    //int i = 0;
    public IEnumerator GetEnumerator()
    {
        return this;
    }
    public object Current
    {
        get { return Z[i]; }
    }
    public bool MoveNext()
    {
        if (i == Z.Length - 1)
        {
            Reset();
            return false;
        }
        i++; //i+=2;
        return true;
    }
    public void Reset()
    {
        i = -1; } // i=1 կազմի, եթե foreach աշխատի երկրորդ անգամ
}
class M
{
    static void Main()
    {
        A ob = new A();
        foreach (char v in ob)
            System.Console.Write(v + " ");
    }
}
```

❖ `foreach` և `IEnumerator` [Нейгел-197]

- `foreach` - ի մասին: Թարգմանության ժամանակ այն չի վերածվում միջանկյալ լեզվի օպերատորի: Այն վերածվում է `IEnumerator` ինտերֆեյսի մեթոդների և հատկանիշների:

```
foreach (var v in ob)
{
    Console.WriteLine(v);
}
```

Վերածվում է

```
IEnumerator<ob> ie = ob.GetEnumerator();
while (ie.MoveNext())
{
    ob v = (ob)ie.Current;
    Console.WriteLine(v);
}
```

- Սկզբից կանչվում է `GetEnumerator()` մեթոդը զանգվածի հաշվիչ ստանալու համար: `while` ցիկլը կատարվում է այնքան ժամանակ, քանի դեռ `MoveNext()` մեթոդը վերադարձնում է `true`: Զանգվածի էլեմենտները հասանելի են դառնում `Current` հատկանիշի միջոցով:

87. Իտերատորներ, Օպերատոր **yield**

[Троелсен- 328; Troelsen-325; Шилдт-1003]

- Իտերատորը տիպի համար “ալտերնատիվ” միջոց է, որը կարող է օգտվել **foreach** ցիկլից: Սահմանենք իտերատորը, դա **կողերի բլոկ** է, որը հնարավորություն է տալիս ամբողջ տվյալները կարդալ **ըստ հերթականության**: Որպես կողի բլոկ կարող է հանդես գալ մեթոդը, հատկանիշը կամ ցանկացած ֆունկցիոնալ գործողություն կատարող ծրագրային մաս:
- yield** օպերատորը նախատեսված է հեշտ համարակալում ապահովելու համար: **yield return** հրամանի միջոցով վերադարձնում է կոլեկցիայի մեկ էլեմենտ և **փոխում է ընթացիք** կարդացվող դիրքը հաջորդ էլեմենտի վրա:
- yield** օպերատորը օգտագործվում է, երբ անհրաժեշտ է լինում **foreach** - ի միջոցով աշխատացնել սեփական կլասները որպես կոլեկցիաներ (համարակալվող համակարգեր) **չօգտվելով** **IEnumerable** ինտերֆեյսի ժառանգումից: (Ինչպես գիտենք ինտերֆեյսի **ժառանգումը պարտադիր** է, եթե տիպը օգտագործվում է այլ միջավայրից, որի պահանջների մեջ է մտնում համարակալումը: Իտերատորը լուծում է այդ հարցը):

```
using System;
using System.Collections;
class A
{
    char[] Z = { 'a', 'b', 'c', 'd' };
    public IEnumerator GetEnumerator() // իտերատոր, որի անունը պահանջում է foreach ցիկլը
    {
        yield return Z[0]; // Միաժամանակ մեկից ավել return -ներ
        yield return Z[3]; //
        yield return Z[2]; //
        yield return Z[1]; //
        //-----կամ
        // for (int i = 0; i < Z.Length; i++)
        //     yield return Z[i];
        //----- կամ
        // foreach (char v in Z)
        //     yield return v;
    }
}
class M
{
    static void Main(string[] args)
    {
        A ob = new A();
        foreach(char v in ob)
            Console.WriteLine(v);
    }
}
```

- Իտերատորում պարտադիր չէ կուտակիչի առկայություն և կարելի է ընդհատել իտերատորի ցիկլը **yield break** հրամանով;

```
using System; //Шилдт-1004;
using System.Collections;
class A
{
    char ch = 'A';
    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < 26; i++)
        {
            if (i == 10)
                yield break;
        }
    }
}
```

```

        yield return (char)(ch + i);
        Console.WriteLine("barev");
    }
}
}
class M
{
    static void Main()
    {
        A ob = new A();
        foreach (char ch in ob)
            Console.Write(ch + " ");
        Console.WriteLine();
    }
}

```

88. Անվանական իտերատորներ

[Троелсен- 330; Troelsen-327; Шилдт-1007]

- Եթե անհրաժեշտ է, որպեսզի իտերատոր դառնա **ցանկացած անունով** մեթոդ, ապա իտերատորում GetEnumerator() մեթոդը **IEnumerator** ինտերֆեյսի փոխարեն պետք է վերադարձնի **IEnumerable** ինտերֆեյս և **foreach** ցիկլի օբյեկտ պետք է դառնա նշված **մեթոդը**: Այդ դեպքում իտերատորը կոչվում է **անվանական** իտերատոր:

```

using System;
using System.Collections;
class M
{
    //static char[] Z = { 'a', 'b', 'c' };
    static IEnumerable f()
    {
        yield return 'a';
        yield return 'b';
        yield return 'c';
        // for (int i = 0; i < Z.Length; i++)
        //     yield return Z[i];
    }
    static void Main(string[] args)
    {
        foreach(char v in f()) // f() method
            Console.WriteLine(v);
        // foreach (char v in Z)
        //     Console.WriteLine(v);
    }
}
// -----
using System; //example 2
using System.Collections;
class A
{
    string[] s = { "aaa", "bbb", "ddd", "ccc" };
    public IEnumerator GetEnumerator() // iterator
    {
        for (int i = 0; i < s.Length; i++)
            yield return s[i];
    }
    public IEnumerable f_revers() // named iterator
    {
        for (int i = 3; i >= 0; i--)
            yield return s[i];
    }
}

```

```

    }
}
class Program
{
    static void Main(string[] args)
    {
        var ob = new A();
        foreach (var v in ob)
            Console.WriteLine(v);
        Console.WriteLine();
        Console.WriteLine("revers.....");
        foreach (var v in ob.f_revers())
            Console.WriteLine(v);
    }
}
// -----
• Անվանական իտերատորի առավելություններից է պարամետրի փոխանցուման հնարավորությունը:
using System; // Qrakusi = um // yield return // ref
using System.Collections;
class M
{
    static IEnumerable f(int a, int b, int c)
    {
        double x1,x2;
        x1 = (-b + (Math.Sqrt(Math.Pow(b, 2) - 4 * a * c))) / (2 * a);
        x2 = (-b - (Math.Sqrt(Math.Pow(b, 2) - 4 * a * c))) / (2 * a);
        yield return x1;
        yield return x2;
    }
    static void Main()
    {
        int a = 1, b = 22, c = 3;
        foreach (double v in f(a,b,c))
            Console.WriteLine(v);
    }
}
//-----
using System; //Шилдт-1007;
using System.Collections;
class A
{
    char ch = 'A';
    public IEnumerable ff(int begin, int end)
    {
        for (int i = begin; i < end; i++)
            yield return (char)(ch + i);
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        foreach (char ch in ob.ff(5, 12))
            Console.Write(ch + " ");
    }
}

```

89. Ինտերֆեյս IComparable

[Троелсен- 335; Troelsen-334; Шилдт-778]

- Հնարավորություն է տալիս համեմատել երկու օբյեկտներ ստանալով 3 արդյունք (0, -1, 1):

```
interface IComparable
{
    int CompareTo(object ob);
}
```

- Ներկայացված կառուցվածքի մեջ `int CompareTo(object ob)` մեթոդ կանչող օբյեկտը համեմատվում է այն օբյեկտի հետ, որը որոշվում է պարամետրով: Եթե կանչող օբյեկտի պարունակությունը մեծ է պարամետրում նշվածից, ապա վերադարձվում է “1” արժեք: Եթե կանչող օբյեկտի պարունակությունը փոքր է պարամետրում նշվածից, ապա վերադարձվում է “-1” արժեք: Եթե երկու օբյեկտների պարունակությունը նույնն է, ապա վերադարձվում է “0” արժեք:
- `IComparable` ինտերֆեյսի օգտագործման տեղերից մեկը `System.Array` կլասն է: `Array` կլասը ունի `Sort()` մեթոդը, որը կարող է վերադասավորել ներդրված տիպերով անդամները (`int`, `string` և այլն): Այդ անդամները կարելի է վերադասավորել թվային կամ սիմվոլային հաջորդականությամբ, քանի որ նշված տիպի համար իրականացված է `IStructuralComparable` ինտերֆեյսը:
- Ինչպես գիտենք երկու օբյեկտ կարելի է նաև համեմատել **օպերատորների գերբեռնումով**, որը կվերադարձնի բուլյան արդյունք: `IComparable` ինտերֆեյսը ստիպում է 3 արդյունքի տարբերակը (0,-1,1) և այն կոնստրակտային է ինտերֆեյսի պարտականության պատճառով:

```
using System;
// class A //ok!
class A : IComparable
{
    public int n;
    //public static bool operator >(A ob1, A ob2) // < //
    //{
    //    if (ob1.n > ob2.n)
    //        return true;
    //    else    return false;
    //}
    public int CompareTo(object ob)
    {
        A temp = (A)ob; //A temp = ob as A;
        if (this.n > temp.n)
            return 1;
        if (this.n < temp.n)
            return -1;
        else
            return 0;
    }
}

class Program
{
    static void Main(string[] args)
    {
        A ob1 = new A();
        A ob2 = new A();
        ob1.n = 77;
        ob2.n = 55;
        Console.WriteLine(ob1.CompareTo(ob2));
    }
}

❖ Փոքր ավելի բարդ օրինակ: Վերցնենք հետևյալ կլասը.
class Car
{
    public string CarName;
    public int CarID { get; set; }
}
```

- Ինչ կարող է լինել, եթե օրինակ օգտագործենք ասենք `Car` տիպի զանգված և կիրառենք `Array.Sort(obCar)` հրամանը:
- Կառաջանա արտակազ իրավիճակ, քանի որ `Car` տիպի համար `IComparable` ինտերֆեյս իրականացված չէ: Որպեսզի հնարավոր լինի կիրառել `Array.Sort(obCar)` հրամանը, ապա անհրաժեշտ է իրականացնել `IComparable` ինտերֆեյսի `int CompareTo(object o)` մեթոդը: Անհրաժեշտ է ընտրել նաև `Car` կլասի ըստ որ անդամի է կատարվում վերադասավորման կարգավորումը: Օրինակ կարելի է վերցնել ըստ `CarID` դաշտի:

```
using System;
// class Car // error
class Car : IComparable //
{
    public string CarName;
    public int CarID { get; set; }
    public Car(string name,int id)
    {
        CarName = name;
        CarID = id;
    }
    public int CompareTo(object ob)
    {
        Car temp = ob as Car;
        if (this.CarID > temp.CarID)
            return 1;
        if (this.CarID < temp.CarID)
            return -1;
        else
            return 0;
        //..... string
        //if (this.CarName.CompareTo(temp.CarName) > 0)
        //    return 1;
        //if (this.CarName.CompareTo(temp.CarName) < 0)
        //    return -1;
        //else
        //    return 0;
        // ..... 2nd way
        //Car temp = ob as Car;
        //if (temp != null)
        //    return CarID.CompareTo(temp.CarID);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car[] ob=new Car[5];
        ob[0] = new Car("Toyota",55);
        ob[1] = new Car("Mercedes", 33);
        ob[2] = new Car("Lexus", 9);
        ob[3] = new Car("BMW", 44);
        ob[4] = new Car("Nissan", 5);
        foreach(Car c in ob)
            Console.WriteLine(c.CarID+ "\t"+ c.CarName);
        Array.Sort(ob);
        Console.WriteLine();
        foreach (Car c in ob)
            Console.WriteLine(c.CarID + "\t" + c.CarName);
    }
}
```

Դելեգատներ

90. Դելեգատներ, Խմբային դելեգատներ

[Шилдт-473, 476; Троелсен- 379,389; Troelsen-452; Нейгел-241, 251]

- Դելեգատը **տիպ** է, որի օբյեկտի **հղումը** կարող է **ստանալ մեթոդ**: Ստեղծելով դելեգատի օբյեկտ կարելի է նրա միջոցով կանչել մեթոդ: Նա նման է C++ -ի ցուցիչի, որը դիմում է ֆունկցիային: Սակայն դելեգատը համարվում է դեկլարվող օբյեկտ, և դա նշանակում է կատարման ժամանակ օգտվել որոշ առավելություններից (չթույլատվող հասցեներից խուսափում, հիշողության խախտման խուսափում...):
- Ինչպես գիտենք կարելի է հղել միայն օբյեկտին: Մեթոդը օբյեկտ չէ, սակայն նա նույնպես գտնվում է ֆիզիկական հասցեում և այդ հասցեով է կանչվում մեթոդը: Ահա այդ հասցեն վերագրվում է դելեգատին:
- Որտեղ են օգտագործվում դելեգատները:
 - ✓ առաջինը - դելեգատները ապահովում են **իրադարձությունների** աշխատանքը:
 - ✓ երկրորդ - նրանք թույլ են **տալիս ծրագրի օգտագործման ժամանակ կատարել մեթոդ**, որը հայտնի չէ թարգմանման ժամանակ:
 - ✓ երրորդ - **(callback)** - հետադարձ կանչի ֆունկցիա, որի աշխատանքը հիմնված է ֆունկցիայի ցուցիչի վրա: Երբ ֆունկցիան վերջացնում է իր աշխատանքը, նա տեղեկացնում է դրա մասին: Օգտագործվում է զուգահեռ ծրագրավորման ժամանակ: (Նախկինում նրանք օգտագործվում էին Windows API ֆունկցիաներից օգտվելու համար):
- Օրինակ **public delegate int dd(int x, int y);** Այս հայտարարումը թարգմանիչը վերածում է հետևյալ կլասի:

```
sealed class dd: MulticastDelegate
```

```
{  
    public dd(uint functionAddress);  
    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() and EndInvoke() մեթոդները ապահովում է **ասինխրոն** աշխատանք (**C#2**), որը նշանակում է ծրագիրը կարող է շարունակվել ֆունային հոսքում **զուգահեռ** ձևով: Նույն արդյունքին կարելի է հասնել հոսքերի միջոցով, սակայն նշված ձևը օգտագործման տեսակետից ավելի պարզ է:

Դելեգատների օգտագործումը [Шилдт-474; Нейгел-243]

- Դելեգատի հայտարարման ժամանակ որոշվում է ֆունկցիայի վերադարձնող արժեքը, պարամետրերի տեսակները և քանակը:
delegate int dd(int a1, int a2);
- Այնուհետև պետք է հայտարարել դելեգատի օբյեկտ: Որից հետո ստեղծում ենք մեթոդ, որը մշակում է դելեգատը: Նա պետք է ունենա նույն սիգնատուրան (վերադարձվող արժեք, պարամետրերի տեսակ և քանակի նույնություն), որը ունի դելեգատը:

```
class A
```

```
{  
    dd ddoject; // հայտարարել դելեգատի օբյեկտ  
    public int ff(int a1, int a2) // մշակում է դելեգատը  
    {  
        return a1 + a2;  
    }  
}
```

- Պետք է միացնել մշակողը դեկլատին: Ասենք Main() ծրագրում:

```
A ob = new A();
ob.ddobject = new dd(ob.ff);
int result = ddobject (5,9); // result = 14
using System;
delegate string dd(string ss);
class M
{
    static string replaceSpaces(string a)
    {
        Console.WriteLine("Replacement");
        return a.Replace(' ', '-');
    }
    static string removeSpaces(string a)
    {
        string temp = "";
        Console.WriteLine("removal");
        for (int i = 0; i < a.Length; i++)
            if (a[i] != ' ')
                temp += a[i];
        return temp;
    }
    static string reverse(string a)
    {
        string temp = "";
        Console.WriteLine("revers");
        for (int i = a.Length - 1; i >= 0; i--)
            temp += a[i];
        return temp;
    }
    public static void Main()
    {
        dd ddb = new dd(replaceSpaces);
        Console.WriteLine(ddb("It is the simple test"));
        ddb = new dd(removeSpaces);
        Console.WriteLine(ddb("It is the simple test "));
        ddb = new dd(reverse);
        Console.WriteLine(ddb("It is the simple test "));
    }
}
```

❖ Խմբային դեկլատներ [Шилдт-476; Троелсен- 389; Troelsen-462; Нейсел-251]

- Խմբային դեկլատներ - դա միայնակ դեկլատ է , որը փոխարինում է երկու և ավելի դեկլատների: Նա ձևավորվում է + = նշանով մեկ դեկլատը միացնելով մյուսին հերթի զկգրունքով: (- =) – ով կարելի է դեկլատը հանել խմբից՝ ցանկացած տեղից: Կանչելով դեկլատը կատարվում է դեկլատների խումբը:
- Նրանք պետք է ունենան **հիմնականում void** վերադարձվող արժեք: (Либерти-317) – կարելի է առանց **void** – ի, այդ դեպքում վերադարձնում է վերջին դեկլատի վերադարձրած արժեքը:
- Կանչելու հերթականությունը պաշտպանվում են նույն ձևով, ինչ ձևով ձևավորվել է դեկլատների խումբը:

```
using System;
public delegate void dd(int a1, int a2);
class M
{
    dd dd1; dd dd2; dd dd3;
    public void ffAdd(int a1, int a2)
```



```

{
    Console.WriteLine("add " + (a1 + a2));
}
public void ffSub(int a1, int a2)
{
    Console.WriteLine("sub " + (a1 - a2));
}
public void ffMul(int a1, int a2)
{
    Console.WriteLine("mul " + (a1 * a2));
}
public static void Main()
{
    M ob = new M();
    ob.dd1 = new dd(ob.ffAdd);
    ob.dd2 = new dd(ob.ffSub);
    ob.dd3 = new dd(ob.ffMul);
    dd Multidd = ob.dd1;
    Multidd += ob.dd2;
    Multidd += ob.dd3;
    Multidd(7, 5);
    Multidd -= ob.dd3;
    Multidd(4, 4);
}
}

```

91. Delegate և MulticastDelegate բազային կլասներ

[Троелсен- 383; Troelsen-454; Հәмш -288]

- System.MulticastDelegate կլասի որոշ անդամները

```

public abstract class MulticastDelegate : Delegate
{
    public sealed override Delegate[] GetInvocationList();
    public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);
}

```

- System.Delegate կլասի որոշ անդամները

```

public abstract class Delegate : ICloneable, ISerializable
{
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public MethodInfo Method { get; }
    public object Target { get; }
}

```

- ✓ Method – հատկանիշ, վերադարձնում է մեթոդի անունը իր տիպերով, որին նշում է դելեգատը:
- ✓ Target - հատկանիշ, եթե վերադարձնում է **null**, ապա դելեգատը նշում է ստատիկ մեթոդ: Եթե դելեգատը նշում է սովորական մեթոդ, ապա հատկանիշը վերադարձնում է կլասի անունը:
- ✓ Combine() – **static** մեթոդը օգտագործվում է այնպիսի դելեգատի ստեղծման համար, որը նշում է մի քանի ֆունկցիաներ: Օգտագործվում է ավելացման համար:
- ✓ Remove() - **static** մեթոդը ջնջում է դելեգատը:
- ✓ GetInvocationList() – վերադարձնում է դելեգատների օբյեկտների **ցուցակի զանգված**:
Հնարավորություն է տրվում խմբային դելեգատի հերթի փոփոխություն:

```

using System;
public delegate void dd(int a1, int a2);
class M
{

```

```

static dd dd1;
dd dd2;
static dd dd3;
public static void ffAdd(int a1, int a2)
{
    Console.WriteLine("add " + (a1 + a2));
}
public void ffSub(int a1, int a2)
{
    Console.WriteLine("sub " + (a1 - a2));
}
public static void ffMul(int a1, int a2)
{
    Console.WriteLine("mul " + (a1 * a2));
}
public static void Main()
{
    M ob = new M();
    dd1 = new dd(ffAdd);
    ob.dd2 = new dd(ob.ffSub);
    dd3 = new dd(ffMul);
    Console.WriteLine(dd1.Method);
    Console.WriteLine(ob.dd2.Method);
    Console.WriteLine(dd3.Method);
    if(dd1.Target==null)
        Console.WriteLine("static");
    if (ob.dd2.Target != null)
    {
        Console.WriteLine("no static");
        Console.WriteLine(ob.dd2.Target);
    }
    dd Multidd = dd1;
    Multidd += ob.dd2;
    Multidd += dd3;
    Console.WriteLine("Multidelegate");
    Multidd(7, 5);
    Console.WriteLine("Combining dd2");
    dd combinedDelegate = (dd)Delegate.Combine(Multidd, ob.dd2);
    combinedDelegate(7, 5);
    Console.WriteLine("Removing dd1");
    dd remDelegate = (dd)Delegate.Remove(Multidd, dd1);
    remDelegate(7, 5);
    // GetInvocationList()
    Console.WriteLine("Calling diferend delegate from Multidd");
    Delegate[] delegateList = Multidd.GetInvocationList();
    ((dd)delegateList[2])(7, 5);
    ((dd)delegateList[0])(7, 5);
    ((dd)delegateList[1])(7, 5);
    Console.WriteLine("Calling delegateList with foreach");
    foreach (var del in delegateList)
    {
        ((dd)del)(7, 5);
    }
}
}

```

92. Անանուն մեթոդներ

[Шилдг-484; Нэш-295; Троелсен- 408; Troelsen-478; Хейгел-254,254]

- Ինչպես գիտենք դեկլատ օգտագործելու համար անհրաժեշտ է համապատասխանող մեթոդ:
Սակայն կարելի է օգտագործել դեկլատ, որի կատարման մեթոդը գոյություն չունի առանձին, այլ անմիջապես և առանց անունի հաջորդում է դեկլատին: Դրանց անվանում են անանուն մեթոդներ:
- Անանուն մեթոդների օգտակարությունը կայանում է նրանում, որ կոդը լինում է **ավելի պարզ** և քիչ հրամաններ է գրվում: Բայց միևնույն է, թարգմանության ժամանակ տեղադրվում է մեթոդի անուն և **արագությանը չի ավելանում**: Առավելությունը զգացվում է, երբ օգտագործվում է իրադարձություններում, երբ երկու իրադարձություն օգտվում են միևնույն դեկլատից:
- Չի թույլատրվում օգտագործել անանուն մեթոդներում **goto, break, continue** անցումներ անանուն մեթոդի մարմնից դուրս: (Хейгел-255)
- Չի թույլատրվում նույն անունով լոկալ փոփոխականի կրկնություն այն մեթոդում, որտեղ ընդգրկված է անանուն մեթոդը:
- unsafe** կոդ նույնպես չի թույլատրվում օգտագործել անանուն մեթոդներում: (կանցնենք հետո C#2):

```
using System;
delegate string dd(string ss);
class M
{
    public static void Main()
    {
        string s = "It is the simple test";
        Console.WriteLine(s);
        dd ddob = delegate(string ss)
        {
            return ss.Replace(' ', '-');
        };
        Console.WriteLine(ddob(s));
    }
}
```

- Չի թույլատրվում նաև **ref, out** պարամետրերի օգտագործում անանուն մեթոդի մարմնում, եթե այդ պարամետրերը առաջարկվում են անանուն մեթոդ ընդգրկող մեթոդի պարամետրում: (Хейгел-255, Троелсен-407)
- Թույլատրվում է **ref, out** հետևյալ դեպքում.

```
void f(ref int m)
{
    dd dob = delegate (ref int n)
    {
        // m++; //error
        return n++;
    };
    int nn = 1;
    Console.WriteLine(dob(ref nn)); // 1
    Console.WriteLine(nn); // 2
}
int c = 9;
f(ref c);
delegate int dd(ref int n);
.....
```

93. Lambda արտահայտություն

[Шилдг-488; Нэш-503; Нейгел-255; Троелсен- 408,415; Troelsen-482]

- Լյամբդա արտահայտությունները ֆունկցիոնալ ծրագրավորման համակարգ է: Այն փոխարինում են անանուն մեթոդներին լինելով ավելի նոր արտահայտման ձև: Օգտագործվում է դելեգատների, իրադարձությունների և LINQ երի հետ:
- Օգտագործվում է => լյամբդա օպերատորը, որը արտահայտությունը բաժանում է երկու մասի: Ձախ մասում պարամետրերն են, իսկ աջ մասում արտահայտության մարմինը (ծրագրային կոդը):
- ✓ `c=>c+2` ավելացնում ենք 2 ով
- ✓ `n=>n%2==0` վերադարձնում է տրամաբանական արժեք
- Սկզբից հայտարարվում է **դելեգատ**, որը համատեղելի է լյամդա արտահայտությանը (դելեգատը պետք է համապատասխանի լյամդայի պարամետրերի և վերադարձվող արժեքի տիպերին), հետո դելեգատի օբյեկտ, որին վերագրվում է լյամդա արտահայտությունը:
- Եթե պարամետրերի քանակը **մեկից ավել** է, ապա նրանք ընդգրկվում են կլոր փակագծերի մեջ:
- Լյամդայի մարմինը ընդգրկում են ձևավոր փակագծի մեջ, եթե մեկից ավելի օպերատորներ են մասնակցում:

```
using System; //delegate
delegate int dd(string s);
class M
{
    static int f(string s)
    {
        return s.Length;
    }
    static void Main(string[] args)
    {
        dd ob = new dd(f);
        Console.WriteLine(ob("albert"));
    }
}

//.....

using System; // anonimus
delegate int dd(string s);
class M
{
    static void Main(string[] args)
    {
        dd ob = delegate(string c)
        {
            return c.Length;
        };
        Console.WriteLine(ob("albert"));
    }
}

//.....

using System; //lambda
delegate int dd(string s);
class M
{
    static void Main(string[] args)
    {
        dd ob = t => t.Length;
        Console.WriteLine(ob("albert"));
    }
}

//.....
MyDel del = delegate (int x) { return x + 1; }; // Anonymous method
MyDel lel = (x) => { return x + 1; }; // Lambda expression
//.....
```

```
using System;
    dd incr = count => count + 2;
    int x = 9;
    x = incr(x);
    Console.Write(x);
    Console.WriteLine();
    dd fact = n =>
    {
        int r = 1;
        for (int i = 1; i <= n; i++)
            r = i * r;
        return r;
    };
    Console.WriteLine("The factorial of 5 is " + fact(5));
delegate int dd(int v);
```

❖ Single-line method (C#7.0) [Troelsen-415; Troelsen-490] N (no video)

- Կարելի է լյամբդա արտահայտություն օգտագործել կլասի անդամ, չօգտվելով դելեգատի առկայությունից, եթե այն ներկայացվի մեկ տողով առանց ձևավոր փակագծերի:

```
// with delegate
using System;
A ob = new A();
dd odd = new dd (ob.Add);
Console.WriteLine(odd(6, 8));
Console.ReadKey();
delegate int dd(int x, int y);
class A
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
// with Lambda without Delegate
using System;
A ob = new A();
Console.WriteLine(ob.Add(6, 8));
Console.ReadKey();
class A
{
    public int Add(int a, int b)=>a + b;
}
```

• Constructors As Expression-Bodied Members (New 7.0) [Troelsen-176] N (no video)

```
using System;
A ob = new A(55);
Console.WriteLine(ob.x);
Console.ReadKey();
class A
{
    public int x;
    public A(int a) => x = a;
}
```

❖ Natural types for lambdas (C#10) N (no video)

- ✓ Լամբդա արտահայտությունը կարող է օգտագործվել նաև մեթոդի հրամանների տարածքում չունենալով դելեգատի հայտարարություն:

```
var v1 = (string s) => int.Parse(s);
```

```
int x = v1("12345");
Console.WriteLine(x);
// -----
var v2 = (string s1, string s2) => s1+s2;
string y=v2("11", "aa");
Console.WriteLine(y);
Console.ReadKey();
```

❖ Deconstructing Tuples with "Single-line" lambda [Troelsen-165] N (No Video)

- **Troelsen** - Լամբդա արտահայտության single-line մեթոդի կիրառման դեպքում Tuple -ների օգտագործման դեկոնստրուկտորը կարող է ունենալ **ցանկացած անուն** և վերագրվում Tuple -ների փոփոխականներին **միայն մեթոդի կանչով**: (Ընդհանրապես դեկանստրուկտորները հնարավոր չէ կանչել Deconstruct անունով: Նրանք պարունակում են **out** պարամետր հղելով կլասի դաշտին):
- [If you use the Lambda Expressions single-line method, the **Deconstruct (??)** for using Tuples can have any name.]
- **??** Ավելի ճիշտ կլինի ասել՝ ուղղակի սովորական **Single-line** լամբդա մեթոդի կանչ է և կապ չունի “դեկոնստրուկ” գաղափարի հետ, որովհետև մեթոդը կանչվում է օբյեկտի միջոցով, որը թույլատրվում է նաև սովորական մեթոդների կանչի ձևով:

```
using System;
class A
{
    public int X, Y;
    public A(int x, int y)
    {
        X = x;
        Y = y;
    }
    public (int x, int y) Deconstruct() => (X, Y); // lambda Deconstruction
    public (int x, int y) f() => (X, Y); // ok // lambda Deconstruction
}
class M
{
    public static void Main()
    {
        A ob = new A(6,7);
        // (int x, int y) = ob; // error // Deconstruction
        (int x, int y) = ob.Deconstruct(); // ok
        (int xx, int yy) = ob.f(); // ok
        Console.WriteLine(x + " " + y); // 6 7
        Console.WriteLine(xx + " " + yy); // 6 7
        Console.ReadKey();
    }
}
```

❖ Using switch Expressions (New 8.0) [Troelsen-108] N (no video)

- There is a lot to unpack in that example, from the lambda (=>) statements to the discard (_).

```
using System;
int x = new Random().Next(4);
string result = x switch
{
    0 => "zero",
    1 => "one",
    _ => "more than one"
};
Console.WriteLine(x + " " + result);
Console.ReadKey();
.....
```

94. Covariance և Contravariance դեկլարանք

[Шилдг-481]

- Մեթոդը, որին փոխարինում է դեկլարացիայի պետք է **համակնի** և վերադարձնող արժեքով և սիգնատուրայով (պարամետրերի տիպերով): Սակայն այս օրենքը այդքան էլ **խիստ չէ**, եթե կա ածանցիալ կլասների (այսինքն **Ժառանգականությամբ** կապված) հարաբերություն և աշխատում է “կոմպայանտություն” և “կոնտրավարյանտություն” հարաբերությունը:
- Սահմանում՝ **կոմպայանտություն** է համարվում, երբ տիպերը կապված են ժառանգականությամբ և բազային ցուցիչին վերագրվում է ժառանգի ցուցիչ: **Կոնտրավարյանտություն** է համարվում, երբ հակառակ կոնստրուկցիան է կիրառվում՝ այսինքն, ժառանգի ցուցիչին վերագրվում է բազային ցուցիչ:
- Covariance** – հնարավորություն է տալիս դեկլարացիայի մեթոդի հետ, որի **վերադարձնող** արժեքի տիպը լինի դեկլարացիայի (papa) տիպին ածանցիալ կլասի տիպ:
- Contravariance** - հնարավորություն է տալիս դեկլարացիայի մեթոդի հետ, որի **պարամետրեր** լինի բազային տիպ, դեկլարացիայի **պարամետրի** տիպի նկատմամբ:

```
using System; // Covariance
class papa
{ }
class A : papa
{ }
delegate papa dd();
class M
{
    public static A ff()
    {
        A temp = new A();
        return temp;
    }
    static void Main(string[] args)
    {
        papa ob = new papa();
        dd obdd = ff;
        ob = obdd();
    }
}
//-----
```

```
using System; // Contravariance
class papa
{ }
class A : papa
{ }
delegate void dd(A ob);
class Program
{
    public static void ff(papa ob)
    {
        ob = new papa();
    }
    static void Main(string[] args)
    {
        A ob = new A();
        dd obdd = ff;
        obdd(ob);
    }
}
```

- ❖ Մեթոդի և պարամետրերը և վերադարձվող արժեքը կարող են ձևավորվել կովարյանտ ձևով:
- ✓ Կովարյանտ պարամետրում, նշանակում է մեթոդի ստացող տիպը կարող է լինել բազային, իսկ մեթոդին արգումենտով ուղղարկվողը, ժառանգ (բազային ցուցիչին վերագրել ժառանգի օբյեկտ):
- ✓ Կովարյանտ վերադարձվող արժեքում, նշանակում է մեթոդի վերադարձվող տիպը կարող է լինել բազային, իսկ մեթոդի return ուղղարկվող օբյեկտը, ժառանգ:

```
using System; // Covariance method, return, params
class papa
{
}
class A : papa
{
}
class M
{
    public static papa f()
    {
        A ob = new A();
        return ob;
    }
    public static void f2(papa ob)
    {
    }
    static void Main(string[] args)
    {
        A ob = new A();
        f2(ob);
    }
}
//-----
```

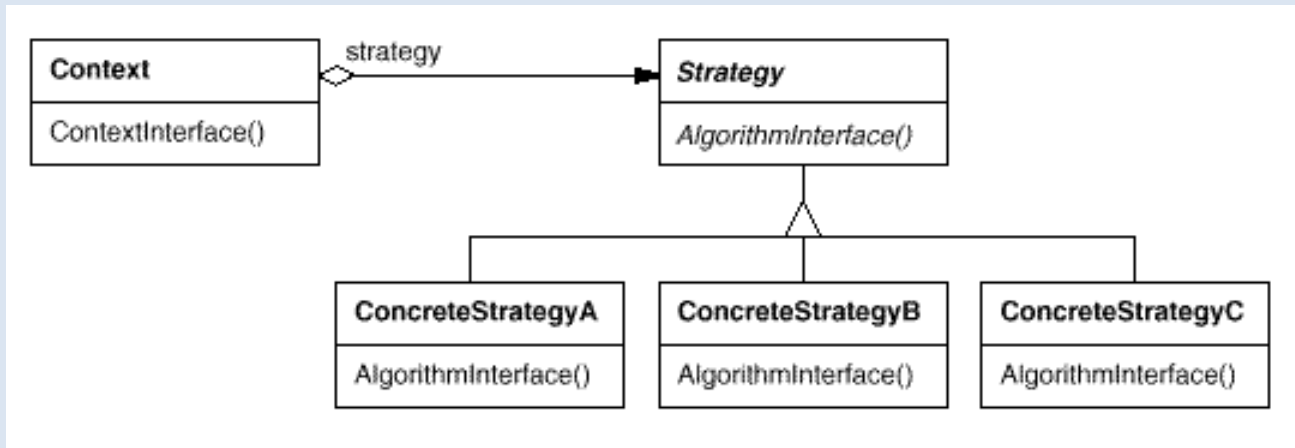
- Կոնտրավարյանտ ձևը մեթոդը չի պաշտպանում: (մեծ տիպից ուղղարկում փոքր տիպ):

```
using System; // Contravariance method, return, params
class papa
{
}
class A : papa
{
}
class M
{
    public static A f() //error
    {
        papa ob = new papa();
        return ob;
    }
    public static void f2(A ob)
    {
    }
    static void Main(string[] args)
    {
        papa ob = new papa();
        f2(ob); //error
    }
}
.....
```


95. Strategy պատերն

[Нэш-306; Шевчук - 265; GOF-362]

- Ծրագրավորման ժամանակ լինում են դեպքեր, երբ անհրաժեշտ է լինում տեղում որոշել, թե **միևնույն տվյալների** մշակման համար, որ ալգորիթմն է **ավելի հարմար**: Strategy փաթեթնը հնարավորություն է տալիս լուծել խնդիրը կիրառելով ալգորիթմների **դինամիկ օգտագործումը**: Օրինակ, կա վերադասավորման արագ և դանդաղ ալգորիթմներ իրենց իրականացումով: Շատ դեպքերում անհրաժեշտ է լինում արագը, որը մեծ ռեսուրսներ է կլանում, փոխել դանդաղ ալգորիթմների, որոնք ավելի քիչ պանաջներ ունեն ռեսուրսների նկատմամբ:
- UML



- Պատերնը կարելի է իրականացնել **կլասիկ ձևով** - կոնտրակտային ծրագրավորումով (GOF – ի պատերն), որը կատարվում է արստրակտ կլասների, որպես ինտերֆեյս ծառայման ձևով:
- Այն կարելի է իրականացնել նաև ավելի ժամանակակից մեթոդով - **դելեգատների** միջոցով:

```
using System; //fast and slow sort with delegate
public delegate Array ddStrategy(int[] A);
public class Client
{
    int[] Ar = { 3, 5, 1, 2, 4 };
    ddStrategy s;
    public Client(ddStrategy concretStrategy)
    {
        this.s = concretStrategy;
    }
    public void DoSomeWork()
    {
        s(Ar);
        for (int i = 0; i < Ar.Length; i++)
            Console.Write(Ar[i] + " ");
        Console.WriteLine();
    }
}
class M
{
    static void Main()
    {
        Client ob;
        ddStrategy ddfast = new ddStrategy (SortAlgorithms.SortFast);
        ob = new Client(ddfast);
        ob.DoSomeWork();
        ddStrategy ddslow = new ddStrategy (SortAlgorithms.SortSlow);
        ob = new Client(ddslow);
        ob.DoSomeWork();
    }
}
```

```

public class SortAlgorithms
{
    public static Array SortFast(int[] array)
    {
        Console.WriteLine("fast - SelectionSort");
        for (int i = 0; i < array.Length - 1; i++)
        {
            int k = i;
            for (int j = i + 1; j < array.Length; j++)
                if (array[k] > array[j]) k = j;
            if (k != i)
            {
                int temp = array[k];
                array[k] = array[i];
                array[i] = temp;
            }
        }
        return array;
    }
    public static Array SortSlow(int[] Ar)
    {
        Console.WriteLine("slow - BubbleSort");
        for (int i = 0; i < Ar.Length; i++)
        {
            for (int j = Ar.Length - 1; j > i; j--)
            {
                if (Ar[j] < Ar[j - 1])
                {
                    int temp = Ar[j];
                    Ar[j] = Ar[j - 1];
                    Ar[j - 1] = temp;
                }
            }
        }
        return Ar;
    }
}

```

Իրադարձություն

96. Իրադարձություն - event

[Троелсен-396; Troelsen-467; Шилдт-494]

- **event**, դա **ավտոմատ տեղեկացում** է որոշ գործողության կատարման մասին: Ում որ պետք է դա նա իր մոտ գրանցում է **իրադարձության մշակող** – event handler: Երբ իրադարձություն է տեղի ունենում, կանչվում են բոլոր գրանցված **իրադարձության մշակողները**:
- DOS և Windows օպերացիոն համակարգերը ունեն աշխատանքային հիմնարար տարբերություն, երբ տեղի է ունենում հայցեր արտաքին ազդակներից: DOS–ը ժամանակ առ ժամանակ հետևում է սարքերին, որոնց հետ ինֆորմացիա պետք է փոխանակի – այն կոչվում է հաջորդական հայցով (**sreial polling**): Windows–ում այն կազմակերպվում է իրադարձությունների ձևով – (**event driven**): Դա միջոց է, երբ Windows–ը տալիս է նշան, որ ինչ որ բան է տեղի ունեցել (օրինակ մկնիկից քիք է կատարվել) և միայն այդ դեպքում է կատարվում համապատասխան աշխատանքներ: (**Petzold,61rus**):
- C# -ում **ցանկացած օբյեկտ** կարող է **հրատարակել իրադարձություններ**, իսկ այլ կլասներ – **գրանցվեն** նրան: Երբ հրատարակող կլասը իրադարձություն է իրականացնում, բոլոր գրանցված կլասները իմանում են դրա մասին: Հրատարակիչը **որոշում է դելեգատ**, որի միջոցով է իրագործվում գրանցված կլասների մշակումը:
- Օրինակ – ստեղծիր իրենից ներկայացնում է հրատարակիչ, որը հրատարակում է քիք իրադարձություն: Մյուս կլասները կարող են դառնալ գրանցվողներ մաուս քիքը օգտագործելու համար:
- **Իրադարձությունը իրականացվում է** հետևյալ ձևով՝ հայտարարվում է որևէ կլասում **public event dd ee;** // որտեղ dd - դելեգատի անունը է:
- Պետք է հայտարարված լինի նաև **դելեգատ**, ասենք կլասից դուրս **delegate void dd();**
- Նրանք **void** են վերադարձնում, որովհետև հիմնականում օգտագործվում են խմբակային ձևով (+=):
- Ստեղծվում է **իրադարձության մշակող**, որը կատարում է իրադարձության իրականացումը: **void Onff() { }**
- Կատարվում է “**ռեգիստրացիա**” **ob.ee += new dd(Onff);**
- Իրադարձության գեներացիա է, օրինակ մկնիկի քիկը, ժամանակի ավարտ և այլն: (Օրինակում արհեստական է): Որպես **հետևանք կանչվում է** բոլոր **գրանցված** իրադարձությունները:

```
A ob = new A();
ob.ee += new dd(Onff);
ob.arhestakan();
void Onff()
{
    System.Console.WriteLine("iradartsutium");
}
delegate void dd();
class A
{
    public event dd ee;
    public void arhestakan()
    {
        ee();
    }
}
```

❖ **Իրադարձության իրականացում լամբդա արտահայտությամբ և անանուն մեթոդով [Шилдт -504]**

- Կարելի է իրականացնել իրադարձություն և լամբդայով և անանուն մեթոդով: Այս դեպքում կազմակերպման սինտաքսիսը չի փոխվում:
- Օրինակում կա առանձնահատկություն, երբ լամբդան գրանցված է և շարունակում ենք անանունով, += տեխնոլոգիան ասում է որ ավելացվել է եղած մշակողներին, դրա պատճառով **խմբակային է կանչում արդեն** և լամբդան և անանուն մեթոդը: Նույնը չի կարելի **սովորական մեթոդի** դեպքում: **A ob = new ();**
ob.ee += (n) => Console.WriteLine("event arajatsav lambda n= " + n);
ob.arhestakan(11);

```

ob.arhestakan(22);
ob.ee += delegate (int n)
{
    Console.WriteLine("event arajatsav ananimus n= " + n);
};
ob.arhestakan(33); // և լամբդա և անանուն մեթոդով
ob.arhestakan(44);
delegate void dd(int a);
class A
{
    public event dd ee;
    public void arhestakan(int a)
    {
        if (ee != null)
            ee(a);
    }
}

```

97. Multicast Event

[Шилдг-496]

- Ինչպես դելեգատները, իրադարձությունները նույնպես կարող են աշխատել խմբակային կանչով:
Դրա համար օգտագործվում է += և -= օպերացիաները:

```

MyEvent evt = new();
X xOb = new ();
Y yOb = new ();
Console.WriteLine("Add handlers to the event list.");
evt.SomeEvent += Handler;
evt.SomeEvent += xOb.Xhandler;
evt.SomeEvent += yOb.Yhandler;
Console.WriteLine("Raise the event.");
evt.OnSomeEvent();
Console.WriteLine();
Console.WriteLine("Remove a handler.");
evt.SomeEvent -= xOb.Xhandler;
Console.WriteLine("Raise the event.");
evt.OnSomeEvent();
static void Handler()
{
    Console.WriteLine("Event received by EventDemo"); }
delegate void MyEventHandler();
class MyEvent
{
    public event MyEventHandler SomeEvent;
    public void OnSomeEvent()
    {
        if (SomeEvent != null)
            SomeEvent();
    }
}
class X
{
    public void Xhandler()
    {
        Console.WriteLine("Event received by X object"); }
}
class Y
{
    public void Yhandler()
    {
        Console.WriteLine("Event received by Y object"); }
}

```

❖ Իրադարձության add և remove աքսեսորներ

[Шилдг-500; Гриффитс-510]

- Ինչպես գիտենք խմբակային իրադարձությունները ձևավորվում են ավտոմատ՝ թարգմանիչի կողմից: Սակայն կարելի է նրանց ձևավորումը կազմակերպել նաև ըստ ցանկության: Ինչպես դեկլարացիաներում դա կատարվում է [MulticastDelegate](#) բազային կլաստով, այնպես էլ այստեղ դա կատարվում է հատուկ add և remove աքսեսորներով, և այդ դեպքում add -ը օգտվում է value արժեքի օգտագործման հնարավորությունից:

```
MyEvent evt = new ();
X xOb = new ();
Y yOb = new ();
Z zOb = new ();
Console.WriteLine("Adding events.");
evt.e += xOb.Xhandler;
evt.e += yOb.Yhandler;
evt.e += zOb.Zhandler;
evt.OnSomeEvent();
Console.WriteLine("Remove xOb.Xhandler.");
evt.e -= xOb.Xhandler;
evt.OnSomeEvent();
Console.WriteLine();
Console.WriteLine("Add zOb.Zhandler.");
evt.e += zOb.Zhandler;
evt.OnSomeEvent();
delegate void MyEventHandler();
class MyEvent
{
    MyEventHandler[] evnt = new MyEventHandler[2];
    public event MyEventHandler e
    {
        add
        {
            for (int i = 0; i < 2; i++)
                if (evnt[i] == null)
                {
                    evnt[i] = value; break;
                }
        }
        remove
        {
            for (int i = 0; i < 2; i++)
                if (evnt[i] == value)
                {
                    evnt[i] = null; break;
                }
        }
    }
    public void OnSomeEvent()
    {
        for (int i = 0; i < 2; i++)
            if (evnt[i] != null)
                evnt[i]();
    }
}
class X
{
    public void Xhandler()
    {
        Console.WriteLine("Event received by X object");
    }
}
class Y
{
    public void Yhandler()
    {
        Console.WriteLine("Event received by Y object");
    }
}
class Z
{
    public void Zhandler()
    {
        Console.WriteLine("Event received by Z object");
    }
}
```

98. BCL գրադարանի իրադարձություններ, **EventHandler** - դեկլարատ

[Шилдт-506; Троелсен-403; Troelsen-476; Liberty-249; Либерти-321]

- Ընդունված կարգ է .NET գրադարանում իրադարձության մշակողները վերադաձնում են **void** և ընդունում են երկու պարամետրեր: Առաջին պարամետրը հղում է իրադարձության **աղբյուրի** օբյեկտին - այսինքն հրատարակչին: Երկրորդը – պարամետր է, որը հանդիսանում է EventArgs կլասի անդամ կամ նրան ժառանգած անդամ, պարունակում է տվյալներ, որոնք նախատեսված են մշակողի համար: Խորհուրդ է տրվում բոլոր ծրագրավորողներին այս կանոնը ընդունել:

```
using System;
class A : EventArgs
{
    public int c;
}
delegate void dd(object sender, A e);
class B
{
    public event dd ee;
    public void arhestakan()
    {
        A ob = new A();
        ob.c++;
        ee(this, ob); // արհեստական իրադարձություն
    }
}
class X
{
    public void Onff(object sender, A e)
    {
        Console.WriteLine(sender.ToString() + e.c);
    }
}
class M
{
    public static void Main()
    {
        X ob1 = new X();
        B ob = new B();
        ob.ee += new dd(ob1.Onff);
        ob.arhestakan();
    }
}
```

❖ **EventHandler** – դեկլարատ [Шилдт-508; Троелсен-405; Troelsen-478]

- BCL գրադարանը ունի ներդրված **EventHandler delegate**, որը պարզեցնում է աշխատանքը: Նրան կարելի է օգտագործել իրադարձության մշակման կանչի համար, որը չի պահանջում ժառանգում EventArgs կլասից:

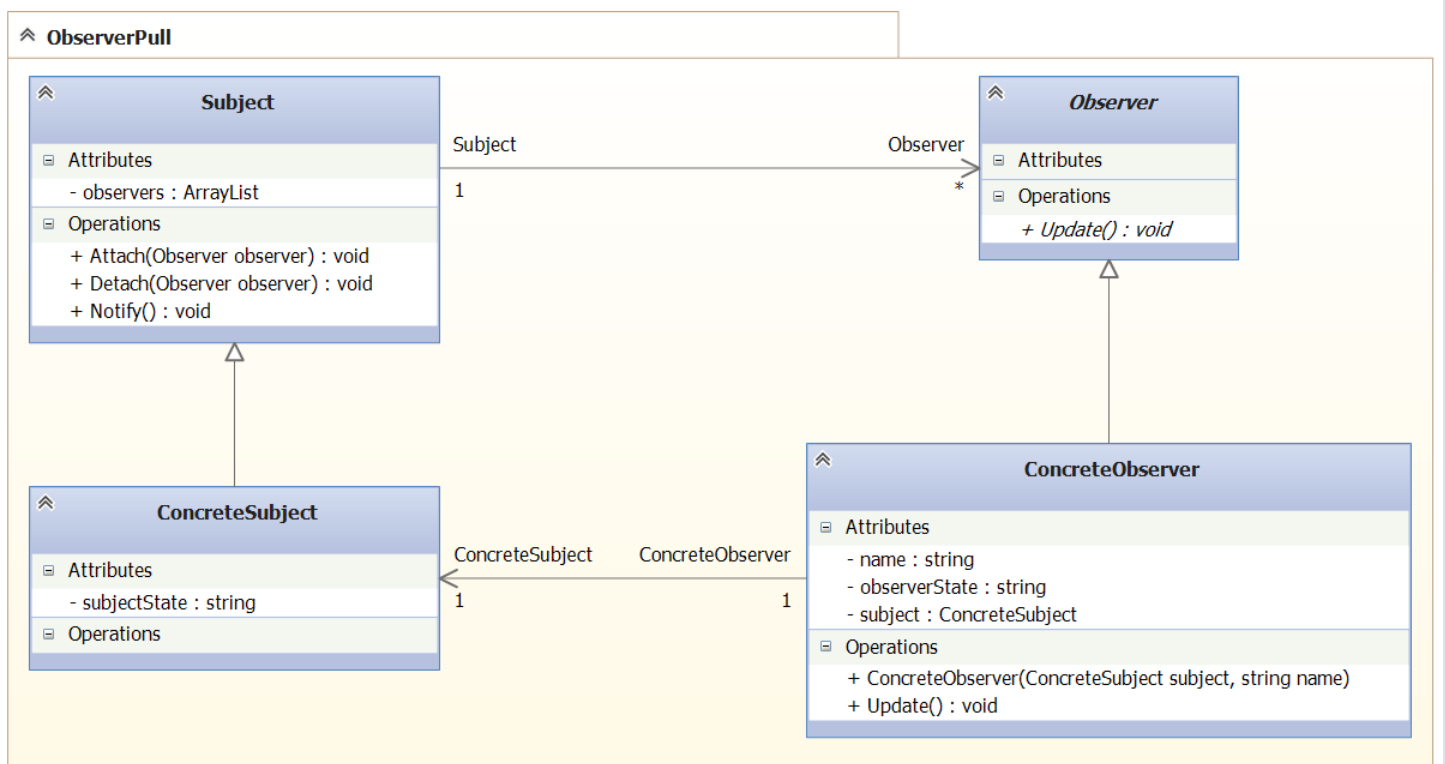
```
using System;
class A
{
    public event EventHandler ee;
    public void arhestakan()
    {
        ee(this, EventArgs.Empty);
    }
}
class M
{
    static void Onff(object sender, EventArgs e)
    {
        Console.WriteLine("iradartsutin " + sender);
    }
    public static void Main()
    {
        A ob = new A();
        ob.ee += Onff;
        ob.arhestakan(); // արհեստական իրադարձություն
    }
}
```

99. Observer պատերն

[Шевчук –228 ; Фримен -71; Sarcar-269; GOF-339]

- Գործողության պատերն:
- Պատերնը որոշում է մեկը շատին հարաբերություն այնպես, որ եթե մեկի մոտ կատարվում է փոփոխություն, մյուսները դա իմանում են և փոփոխվում: Նման է **հրատարակչություն/գրանցվող** համակարգին: Այլ խոսքով գոյություն ունի սուբյեկտ (Subject), որը հանդես է գալիս որպես հրատարակիչ, որում կարող է տեղի ունենալ օբյեկտի փոփոխություն: Երբ տեղի է ունենում օբյեկտի փոփոխման **իրադարձությունը** (event), ապա դրա մասին տեղեկացվում են **գրանցվածները** (Observer) և իրենց մոտ կատարում են փոփոխություն նոր իրավիճակի տեսքով:
- Subject –ի տեսակետից կատարվում է “**չանհանգստացնել մեզ**” սկզբունքը՝ անհրաժեշտության դեպքում մենք ձեզ հետ կկապվենք (**Հովիվուդի օրենք**): Observer պատերնը գործում է օբյեկտների **թույլ կապվածության** սկզբունքով, որը նշանակում է Subject –ը չի տիրապետում և խառնվում Observer –ի ներքին կառուցվածքին: Այն ինչ որ տեղ հանդիսանում է կիբերնետիկական “սև արկղ” և կապը իրականացվում է միայն **ինտերֆեյսի** միջոցով պահելով OOP **պարփակվածության** սկզբունքը:

❖ **Pull model** – Երբ գրանցվողը իրազեկվում է հրապարակման մասին և ինքն է գնում ստանալու հրատարակչությունը (փոփոխությունը):



```
using System;
using System.Collections;
class Program
{
    static void Main()
    {
        ConcreteSubject subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver1(subject));
        subject.Attach(new ConcreteObserver2(subject));
        subject.State = "Some State ...";
        subject.Notify();
    }
}
abstract class Observer
{
    public abstract void Update();
}
```

```

}
abstract class Subject
{
    ArrayList observers = new ArrayList();
    public void Attach(Observer observer)
    {
        observers.Add(observer);
    }
    public void Detach(Observer observer)
    {
        observers.Remove(observer);
    }
    public void Notify()
    {
        foreach (Observer observer in observers)
            observer.Update();
    }
}
class ConcreteSubject : Subject
{
    public string State { get; set; }
}
class ConcreteObserver1 : Observer
{
    string observerState;
    ConcreteSubject subject;
    public ConcreteObserver1(ConcreteSubject subject)
    {
        this.subject = subject;
    }
    public override void Update()
    {
        observerState = subject.State;
        Console.WriteLine(observerState + "Observer1");
    }
}
class ConcreteObserver2 : Observer
{
    string observerState;
    ConcreteSubject subject;
    public ConcreteObserver2(ConcreteSubject subject)
    {
        this.subject = subject;
    }
    public override void Update()
    {
        observerState = subject.State; // գնալ վերցնել հրապարակիչից
        Console.WriteLine(observerState + "Observer2");
    }
}

```

❖ **IObservable<T>** and **IObserver<T>** [Шилдт - 781; Schildt – 689]

- NET Framework 4.0 ում ավելացվել է նշված ինտերֆեյսները, որոնք օգտագործում են **Observer** պատերնի սկզբունքները: **Observable** շաբլոնում նշվում է հրապարակողը, որին հետևում են: **Observer** շաբլոնում նշվում են հետևողները, որոնք սպասում են հրապարակման իրադարձությանը: Observable հրապարակողի **Subscribe()** մեթոդով ռեգիստրացվում են հետևողները: Հրապարակման տեղեկացումը ուղղարկվում է հետևողներին **OnNext()**, **OnError()**, **OnCompleted()** մեթոդներով, որոնք որոշվում են Observer շաբլոնում:

Բացառիկ իրավիճակ (Exception)

100. Բացառիկ իրավիճակներ

[Шилдт-403; Троелсен- 280; Troelsen-271; Нейгел-419]

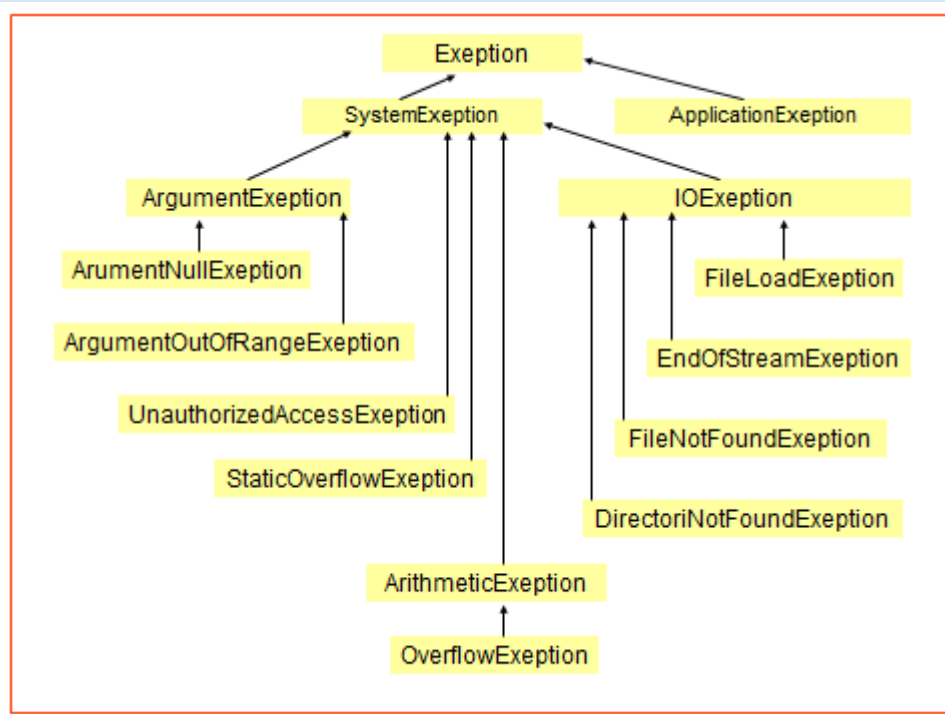
- Եթե ծրագրում կարող է տեղի ունենալ չնախատեսված անցումներ, ապա նախատեսվում է մի միջոց, որը կրող է ծրագրի հատվածը դնել հսկման տակ: Այդ միջոցը կոչվում է բացառիկ իրավիճակներ:
- Բացառիկ իրավիճակները առաջանում են սխալների տեսակներից: Սխալի տեսակներ են համարվում.
- ✓ Ծրագրային սխալներ – օրինակ ստեկի տարածքի լրացում, կամ զանգվածի սահմանի խախտում, գրոյի բաժանում:
- ✓ Օգտագործողի սխալներ - թվի փոխարեն տեքստի մուտք...
- ✓ Արտակարգ իրավիճակներ – չնախատեսված տարբեր վիճակներ, կապի խզում, չթույլատրված ֆայլից օգտվել ...
- Օգտագործվող հանգույցային բառերն են.
`try{...} catch{...} finally{...} throw`
- Թվային ինֆորմացիայի փոխարեն սխալի գրուշացումը կատարվում է տեքստային հայտարարությունների ձևով:
- Բոլոր բացառիկ իրավիճակները որոշված են կամ ժառանգված են `System.Exception` կլասից:
- Ծրագրի կասկածելի մասը դրվում է հսկման տակ `try` - ի միջոցով: Եթե անցանկալի երևույթ է տեղի ունենում, ապա ծրագիրը չի ընդհատվում և կատարվում է `catch` -ի պարունակությունը: `finally` – ն պարտադիր չէ, սակայն եթե այն կա, ապա վերջում կատարվում է նրա պարունակությունը:
- Եթե կա `try`, ապա անպայման պետք է լինի կամ `catch{...}` կամ `finally{...}` կամ երկուսը միասին:
- Առանձնահատկություն: `finally` – ի բլոկում չի թույլտրվում `return` հրամանը:

```
// try..catch
using System;
class M
{
    static void Main()
    {
        int a = 8;
        int b = 0;
        try
        {
            Console.WriteLine(a / b); // CPU
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(e.TargetSite);
            Console.WriteLine(e.Source);
        }
        finally
        {
            Console.WriteLine("!!!!!!");
            Console.WriteLine("Prkel");
        }
    }
}
```

- `double` և `float` տիպերը արտակարգ իրավիճակ չեն առաջացնում, եթե բաժանում է կատարվում 0–ի, այլ տալիս է ? անորոշություն: Ինչու? `decimal` տիպի դեպքում առաջանում է ատրակարգ իրավիճակ:

101. Exception կլաս

[Шилдт-418; Троелсен- 282,288; Troelsen-273, 279; Нейгел-427]



• Exception

• Exception կլասը ունի հատկություններ Message, StackTrace և TargetSite.

✓ Message - սիմվոլային տող է, որը նշում է սխալի բնույթը:

✓ StackTrace կամ ToString() – տող է, որը նշում է սխալ առաջացնող տողի համարը և ամբողջ ճանապարհը:

✓ TargetSite – նշում է մեթոդը, որտեղ առաջանում է արտակարգ իրավիճակը:

✓ Source – նշում է ֆայլը:

• Exception կլասի կոնստրուկտորները:

✓ public Exception() – լռելիությամբ կոնստրուկտոր:

✓ public Exception(string message) – ստանում է Message :

✓ public Exception(string message, Exception innerException) – ներքին արտակարգ իրավիճակ է, երբ մի իրավիճակ առաջացնում է մի այլ արտակարգ վիճակ:

✓ protected Exception(SerializationInfo info, StreamingContext context) – նշված պարամետրերը գտնվում են System.Runtime.Serialization անունի տարածքում: Այն գործում է, երբ սխալի մշակումը կատարվում է հեռահար և դրա համար անհրաժեշտ է օբյեկտի սերիալիզացիա և դեսերիալիզացիա: (կանցնենք):

• SystemException -ից ժառանգվող և հաճախ օգտագործվող արտակարգ իրավիճակներ.

✓ ArrayTypeMismatchException – տիպը չի համընկնում զանգվածի տիպի հետ:

✓ DivideByZeroException – զրոյի բաժանման փորձ:

✓ IndexOutOfRangeException – ինդեքսը անցել է զանգվածի չափից:

✓ InvalidCastException – սխալ տիպերի վերափոխարկում:

✓ OutOfMemoryException – անբավարար հիշողության պահանջ, որը կարող է առաջանալ new -ի կիրառումից:

✓ OverflowException – կարող է առաջանալ մաթեմատիկական հագեցումից,(переполнение):

✓ NullReferenceException – փորձարկում օգտագործել դատարկ հղում:

✓ ArithmeticException - կարող է առաջանալ մաթեմատիկական օպերացիաներից:

102. Արհեստական բացառիկ իրավիճակ

[Шилдг -414; Троелсен-285; Troelsen-277]

- Բացառիկ իրավիճակ առաջանում է ավտոմատ ձևով, որը մշակվում է համապատասխան հրամաններով: Սակայն գոյություն ունի **արհեստական** ձևով բացառիկ իրավիճակ ստեղծելու հնարավորություն, որը կարելի է առաջացնել **throw** հրամանի միջոցով: **throw** հրամանի պարամետր կարող է լինել `System.Exception` կլասի կամ նրան ժառանգող կլասի օբյեկտ: Պարզ է, որ օբյեկտի ստեղծման համար անհրաժեշտ է **new** հրամանի օգտագործում, որը նաև նշանակում է կոնստրուկտորի աշխատանք:

```
using System;
    try
    {
        throw new DivideByZeroException();
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("mshakum");
    }
//throw - finally
using System;
    int a = 8;
    int b = 0;
    try
    {
        throw new Exception();
        Console.WriteLine(a / b);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
        Console.WriteLine(e.HelpLink);
        // Exception ex =new Exception();
        // ex.HelpLink="http://www.albdarb.com";
        // throw ex;
    }
    finally
    {
        // return; error !
        Console.WriteLine("!!!!!!");
    }
```

❖ Throw As Expression - Anywhere (New 7.0) [Troelsen-280] N

- throw** հրամանը կարող է իրականացվել նաև ոչ **try...catch** -ի սահմաններում՝

```
using System;
A.f();
Console.ReadKey();
class A
{
    public static void f()
    {
        throw new Exception();
    }
}
```

.....

103. Արտակարգ իրավիճակի Rethrowing

[Шилдг -415; Троелсен-300; Troelsen-292]

- Որևէ `catch` -ի բլոկում մշակվող արտակարգ իրավիճակը կարող է ստեղծել նաև կրկնակի արտակարգ իրավիճակ, որը պետք է մշակվի **արտաքին բլոկում**: Դա տեղի ունի այն դեպքում, երբ անհաժեշտ է ունենալ մի քանի մշակողներ միևնույն արտակարգ իրավիճակի տարբեր “ասպեկտների” համար: Այսպիսի հնարավորության կարելի է հասնել օգտագործելով `throw` հրամանը **առանց օբյեկտի**.
`throw`;
- Չպետք է մոռանալ, որ `throw` հանդիպման դեպքում մշակումը չի կատարվում տվյալ `catch` -ի բլոկում, այլ հետ է ուղղարկվում արտաքին `catch` -ի բլոկ: (Microsoft -ը շատ է սիրում այս գաղափարը):
- Օրինակում զրոյի բաժանումը կատարվում է `GenException()` մեթոդում, սակայն, երբ զանգվածի սահմանի անցում է կատարվում, ապա այն մշակվում է կրկնակի ձևով: Այլ խոսքով `IndexOutOfRangeException` -ը մշակվում է `Main` ում:

```
using System;
class A
{
    public static void ff()
    {
        int[] n = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] m = { 2, 0, 4, 4, 0, 8 };
        for (int i = 0; i < n.Length; i++)
        {
            try
            {
                Console.WriteLine(n[i] / m[i]);
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("Can't divide by Zero!");
            }
            catch (IndexOutOfRangeException)
            {
                throw; // rethrow the exception
                Console.WriteLine("No matching element found.");
            }
        }
    }
}
class M
{
    static void Main()
    {
        try
        {
            A.ff();
        }
        catch (IndexOutOfRangeException)
        {
            // recatch exception
            Console.WriteLine("Error catching in Main");
        }
    }
}
```

.....

104. Բազմակի catch – ի օգտագործում

[Троелсен- 297; Troelsen-289; Нейсел-423]

- Բազմակի catch թույլարտվում է, սակայն ներկայացման հերթականությունը ունի նշանակություն և հիերարխիայի ավելի բարձրը ծածակում է ենթականերին, որը իր հերթին կարող է բերել կոնֆլիկտի:
- Բազմակի catch –ի դեպքում մշակվում է միայն մեկը, մյուսները անտեսվում են:
- Եթե catch –ում չկա պարամեր, ապա այն կոչվում է “ունիվերսալ” ցանկացած խափանում անցնում է այստեղով:

- Եթե կա բազմակի catch, ցանկալի է ավելի մասնակի դեպքերը գրել ավելի վերև, իսկ “ունիվերսալ” catch – ը վերջում: Հակառակ դեպքում մյուս catch –երը իմաստ չունեն և տալիս է թարգմանման սխալ:

```
using System;
class M
{
    static void Main()
    {
        string s;
        int[] Ar = new int[1];
        while (true)
        {
            try
            {
                Console.WriteLine("Enter number 0, 1 or other");
                s = Console.ReadLine();
                int c = Convert.ToInt16(s);
                switch(c)
                {
                    case 0:
                        int a = 5 / c;
                        break;
                    case 1:
                        Ar[c] = 0;
                        break;
                    default: throw new Exception("Exception catch for " + c);
                }
            }
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("zangvatsi sahman - "+e.Message);
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("zroi bajanum - " + e.Message);
            }
            catch (ArithmeticException e)
            {
                Console.WriteLine("tvabanakan xaghtum - "+e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("exception mshakum - "+ e.Message);
            }
            catch
            {
                Console.WriteLine("chogtagortsogh ");
            }
        }
    }
}
```

105. Custom բացառիկ իրավիճակներ (օգտագործողի կողմից ստեղծվող բացառություններ)

[Шилдг-422; Троелсен-293; Troelsen-285]

- Չնայած նշված բացառությունները ծածկում են մեծ քանակով խնդիրների լուծում, C# -ի առավելություններից է նաև այն, որ հնարավոր է ստեղծել օգտագործողի կողմից բացառիկ իրավիճակներ: Դրա համար ուղղակի անհրաժեշտ է ժառանգել `Exception` կլասը, օգտվել նրա անդամներից և նույնիսկ վերաորոշել: Կարելի է օգտվել `Exception` կլասի կոնստրուկտորներից կիրառելով `base` հրամանը:
- Բացառիկ իրավիճակ մշակող օգտագործողի կլասները կարող են կապված լինել նաև ժառանգականությամբ: Ինչպես գիտենք գրադարանում գտնվող ժառանգականությամբ կապված բացառությունները բազայինը ծածկում է ժառանգին: Դրա համար ավելի թույլ բացառությունը պետք է գրված լինի ավելի առաջ `catch` մշակողների ցուցակում: Նույն մոտեցումը կիրառվում է նաև օգտագործողի կողմից ստեղծվող բացառությունների համար, որոնք կապված են ժառանգականությամբ:

```
using System;
class A : Exception
{
    public A()
    {
        Console.WriteLine("throw anhatakan constuctor A");
    }
}
class M
{
    static void Main()
    {
        for (int x = 0; x < 2; x++)
        {
            try
            {
                if (x == 0)
                    throw new A();
                else
                    throw new Exception();
            }
            catch (A e)
            {
                Console.WriteLine(e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

.....

106. checked, unchecked

[Шилдг -428; Троелсен- 133; Troelsen-86; Нейгел-209]

- Ներդրված տիպը կարող է ծրագրի աշխատանքի ժամանակ ստանալ արժեք, որը իր հնարավոր սահմաններից դուրս էն: Այդ դեպքում տեղի է ունենում տվյալի կորուստ: Որպեսզի կարողանանք խուսափել դրանից օգտվում ենք **checked** հրամանից: Հրամանից հետո բլոկի մեջ գրվում է կասկածելի ծրագրի հատվածը և եթե տեղի է ունենում ինֆորմացիայի կորուստ, ապա կատարվում է ծրագրի ընդհատում և `OverflowException` բացառիկ իրավիճակ:
- **unchecked** հնարավորություն է տալիս բլոկի ներսում հսկման հնարավորությունից հանել որոշ հրամաններ:
- Ծրագրի հրամանները լռելիությամբ **unchecked** հրամանի տակ են:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        byte b = 254;
        //b++;
        //Console.WriteLine(b);
        byte c = 255;
        try
        {
            checked
            {
                b++;
                unchecked
                {
                    c++;
                }
            }
        }
        catch (OverflowException e)
        {
            Console.WriteLine(e.Message);
        }
        Console.WriteLine(c);
        Console.WriteLine(b);
    }
}
```

.....

Ընդհանրացումներ (Generics)

107. Կլասների ընդհանրացումներ

[Нэш-309; Шилдт-575; Нейгел-161; Рихтер-302; Троелсен-372; Troelsen-403]

- Օբյեկտ-կողմնորոշված ծրագրավորման հիմքերից է հանդիսանում **կոդի կրկնակի օգտագործումը**, կամ այլ խոսքով՝ ավելցուկային կոդից խուսափելը: Նշվածին կարելի է հասնել ժառանգականության, կոմպոզիցիայի կամ ներդրված կլասների շնորհիվ: “Ջեներիքները” կոդի կրկնակի օգտագործման նոր հնարավորություն են: Օրինակ՝ ստեղծվում է սորտավորման, փնտրման, համեմատման և այլ ալգորիթմներ և չի նշվում թե **ինչ տիպերի** հետ պետք է աշխատել: Տիպը որոշվում է **դիման ժամանակ (Рихтер)**:
- Բացի այդ, ընդհանրացումները մեծ դեր ունեն, կապված լեզվի **տիպիզացման խստության** բարձրացման հետ, որը բերում է **հսկաման** և սխալների հայտնաբերման բարձրացմանը:
- Եթե կլինեն կոդի հատվածներ, որոնք կարող են աշխատել “implicit” կամ “explicit” ձևով, ապա հնարավոր է ընդհանրացումով չօգտվել **տիպերի վրափոխարկումից**, որը ինչպես գիտենք համակարգչի աշխատանքի ռեսուրս են վատնում:
- Ընդհանրացումները նման են C++ -ի **շաբլոններին**, սակայն շաբլոնները ստատիկ են, այսինքն **թարգմանման** ժամանակ է կոդը որոշվում: Այստեղ գլխավոր թերությունը, որ այդ շաբլոնները հնարավոր չէ պահել DLL գրադարաններում (նրանք պահվում է ֆայլերում) և նրանց սիմվոլային կոդը տեսանելի է և խախտվում է **հեղինակային** իրավունքները:
- “Ջեներիքները” դիմամիկ են, դա նշանակում է կոդը ստացվում է **կատարման ժամանակ**, և պարզ է այլ օգտագործողների կողմից կարելի է տրամադրել գրադարանների միջոցով, սակայն ավելի **պաշտպանված (Нэш)**:
- Եթե նկարագրենք ջեներիքները, ապա նա **պարամետրերով տիպ** է: Ջեներիքներ կարող են լինել **կլասը, ստրուկտուրան, ինտերֆեյսը, դեկլարացիան, մեթոդը**:
- Կարելի է նշել, որ ընդհանրացում կարելի է կատարել նաև առանց ջեներիքի, օգտագործելով **Object կլասը**, քանի որ նա բազային է բոլոր տիպերի համար և կարելի է նրան հղել, որպես ընդհանուր տիպ: Խնդիրը կայանում է նրանում, որ այս դեպքում չկա **տիպերի անվտանգություն** և կատարվում տիպերի աշխատատար **վերափոխարկում** (օրինակ boxing/unboxing և այլն):

❖ **Կլասի** ընդհանրացում:

// Օրինակ առանց ընդհանրացման.

```
using System;
class A
{
    public int a;
}
class B
{
    public bool a;
}
class Program
{
    static void Main(string[] args)
    {
        A ob = new A();
        ob.a = 11;
        Console.WriteLine(ob.a);
        B ob2 = new B();
        ob2.a = true;
        Console.WriteLine(ob2.a);
    }
}
```


- կիրառենք **object** տիպը

```
using System;
class A
{
    public object a;
    // public dynamic a; //ok
}
class Program
{
    static void Main(string[] args)
    {
        A ob = new A();
        ob.a = 11;
        Console.WriteLine(ob.a);
        ob.a = true;
        Console.WriteLine(ob.a);
    }
}
```

- կիրառենք ընդհանրացում, որտեղ <T> ում կարող է օգտագործել ցանկացած տիպ

```
class A<T>
{
    public T a;
}
class M
{
    static void Main()
    {
        A<int> ob = new A<int>();
        ob.a = 5;
        System.Console.WriteLine(ob.a);
        A<bool> ob2 = new A<bool>();
        ob2.a = true;
        System.Console.WriteLine(ob2.a);
    }
}
```

//.....

- Տիպերը կարող են լինել նաև մեկից ավելին, բաժանվելով այն ստորակետներով <T,K> և եթե կլասը ժառանգում է ընդհանրացումով կլասից, ապա նա իր մոտ նույնպես պետք է ընդհանրացում ներկանացնի: Օրինակում, A կլասը պարտավոր է B կլասի ընդհանրացումը նշի:

```
class A<T,K>:B<T,K>
{
    public T a; // պարտադիր չէ
    public K b; // պարտադիր չէ
}
```

❖ Կրկնենք առավելությունները:

- ✓ Նախնական կողը **պաշտպանված** է և տեսանելի չէ օգտվողներին: Նույնը չի կարելի ասել C+ և Java լեզուների մասին:
- ✓ Տիպերը գտնվում են **անվտանգության** մեջ: Դա **հսկվում** է հիմնականում **թարգմանման** ժամանակ և սխալը հայտնաբերվում է շուտ: Հակառակ դեպքում, երբ գործը հասնում է կատարման, ապա սկսում է աշխատել արտակարգ իրավիճակների մեխանիզմը:
- ✓ **Արտադրողականության** բարձրացում և **արագ աշխատող** կող: Անհրաժեշտ չէ կիրառել տարբեր տիպերի **վերափոխարկումներ**: Object կլասի հետ աշխատանքի կարիք չկա և **boxing/unboxing** մեխանիզմի օգտագործում:
- ✓ Կողի **կրկնակի** օգտագործում կամ կողի ավելցուկից խուսափում:
- Ընդհանրացումները թույլատրում են ընդունել ինչպես ցանկացած տիպ, այնպես էլ կարող կատարել տիպերի օգտագործման որոշակի **սահմանափակումներ**, օգտագործելով **where** օպերատորը:

108. Ընդհանրացում և **where** սահմանափակում

[Шилдг-585; Троелсен-374; Troelsen-406]

- Կլաս ընդհանրացումը կարող է ձևավորել որոշ սահմանափակումներ կամ **թույլատրություններ**, որը կատարվում է **where** հրամանով: Գրման ձևը.

```
class class_name<T> where T: constraints
{
    // ...
}
```

- ✓ **constraints** – նշանակում է սահմանափակում, որտեղ սահմանափակումները տարանջատվում են ստորակետով:

- **Թույլատրվում** են հետևյալ տիպի սահմանափակումներ.

- ✓ Բազային կլասի սահմանափակում:

- ✓ Ինտերֆեյսի սահմանափակում:

- ✓ Հղիչային տիպի սահմանափակում, որը օգտագործում է **struct** հրամանը

- ✓ Չափային տիպի սահմանափակում, որը օգտագործում է **class** հրամանը

- ✓ Կոնստրուկտորի **սահմանափակման թույլատրություն**: Օգտագործվում է **new** հրամանը, որը թույլատրում է ջեներիք կլասի ներասում ջեներիք տիպի օբյեկտ ստեղծել:

- Թույլատրվում է մի քանի սահմանափակումներ միասին, սակայն որոշ կարգով: Օրինակ.

```
class A <T> where T : MyClass, IMyInterface, class, new()
{
}
```

- ✓ Միաժամանակ չի թույլատրվում **class** և **struct**

- ❖ **Հղիչային և չափային տիպերի սահմանափակում:** [Шилдг-599]

- Այն կատարվում է **class** և **struct** հրամանների թույլատրման միջոցով:

```
Test<A> ob = new Test<A>();
// Test<S> ob2 = new Test<S>(); // error
// Test<int> ob3 = new Test<int>(); // error !
Console.ReadKey();
```

```
class A
{ }
struct S
{ }
//class Test<T> where T : struct
class Test<T> where T : class
{
    T ? ob;
    public Test()
    {
        ob = null;
    }
}
```

- ❖ **Բազային կլասի սահմանափակում:**

- Օրինակ՝ եթե **<T>** ում նշված է տիպ, որ չի օգտագործում նշված բազային կլասը, ապա կատարվում է արգելափակում:

- Ը կլասի համար չի թույլատրվում ստեղծել օբյեկտ, որովհետև այն չի ժառանգված A կլասից:

```
A a = new A();
B b = new B();
C c = new C();
Test<A> t1 = new Test<A>(a);
t1.f();
Test<B> t2 = new Test<B>(b);
t2.f();
// Test<C> t3 = new Test<C>(c); // error!
// t3.f(); // error!
```

```

class A
{
    public void Barev()
    {
        Console.WriteLine("Barev");
    }
}
class B : A
{
}
// class C:A // ok
// class C:B //ok
class C //
{
}
class Test<T> where T : A
{
    T ob;
    public Test(T o)
    {
        ob = o;
    }
    public void f()
    {
        ob.Barev();
    }
}

```

❖ **Ընդհանրացում և կոնստրուկտորի կիրառման սահմանափակում** (կամ ավելի ճիշտ **new** -ի կիրառման թույլատրություն) [Шилдт-598]

- (**new**) սահմանափակումը թույլատրում է ստեղծել ընդհանրացման օբյեկտ, որը **հնարավոր չէ ընդհանուր դեպքում**:
- ✓ (**new**) -ն կարելի է օգտագործել մյուս սահմանափակումների հետ, բայց **միայն վերջում**:
- ✓ (**new**) -ն հնարավոր չէ օգտագործել **struct** սահմանափակման հետ, որը նշում է **չափային տիպ**:

```

Test<A> ob = new Test<A>();
class A
{
}
class Test<T>
{
    T ob = new T(); //error !
    public void f()
    {
        T ob = new T();
    }
}
//-----

```

```

Test<A> ob = new Test<A>();
class A
{
}
class Test<T> where T : new()
{
    T ob = new T();
    public void f()
    {
        T ob = new T();
    }
}

```

109. Generics – մեթոդներ

[Шилдг-607; Троелсен- 369; Troelsen-400; Нэш-316]

- Եթե մեթոդը գտնվում է ընդհանրացված կլասում, ապա ընդհանրացված տիպը կարող է լինել նաև մեթոդում և ավտոմատ դառնում է ընդհանրացված: Սակայն կարելի է նաև ստանալ ընդհանրացված մեթոդ, որը գտնվում է ոչ ընդհանրացված կլասում:
- Ընդհանրացված մեթոդները ավելի ժամանակակից են համարվում **գերբեռնված** մեթոդների հետ համեմատած: Չնայած մեթոդների գերբեռնումը ունի մեծ կիրառություն, շատ դեպքերում նրանք կատարում են նույն ծրագիրը տարբեր տիպերի համար:

❖ Ընդհանրացված կլասով.

```
using System;
class A<T,K>
{
    public void f(T a, K b)
    {
        Console.WriteLine(""+a+b);
    }
}
class M
{
    static void Main()
    {
        A<int,int> ob = new A<int,int>();
        ob.f(1, 2);
        A<string,string> ob2 = new A<string,string>();
        ob2.f("ab", "cd");
        A<bool,bool> ob3 = new A<bool,bool>();
        ob3.f(true,false);
    }
}
//-----
```

❖ Ընդհանրացված մեթոդով

```
using System;
class A
{
    public void f<T>(T a, T b)
    {
        Console.WriteLine(""+a+b);
    }
}
class M
{
    static void Main()
    {
        A ob = new A();
        ob.f<int>(1, 2);
        A ob2 = new A();
        ob2.f("ab", "cd");
        A ob3 = new A();
        ob3.f(true,false);
    }
}
```

- ❖ Ընդհանրացված մեթոդները նույնպես (ինչպես կլասները) կարող են ընդունել սահմանափակումներ: [Միտք-610]
- Նշված օրինակը չի աշխատի, որովհետև `where T : class` հրամանը չի թույլատրում չափային տիպի օգտագործում:

```
using System; //albdarb
class M
{
    static void Swap<T>(ref T a, ref T b) where T : class
    {
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
    static void Main()
    {
        // Swap 2 ints.
        int a = 10, b = 90;
        Console.WriteLine("Before swap: {0}, {1}", a, b);
        Swap<int>(ref a, ref b); // error !!!
        Console.WriteLine("After swap: {0}, {1}", a, b);
    }
}
```

- ❖ Օրինակ, որը ցուցադրում է մեթոդի ընդհանրացված վերադարձվող արժեք, որը պարտադիր չէ արգումենտում **լինել վերջինը** (Funk<> դեկլարատում դա պարտադիր է) և թույլատրության `new()` կիրառություն, որը նախատեսված է `N ob = new N();` հրամանի համար:

```
using System;
class A
{
    public N f<N,T,K>( T a, K b) where N : new()
    {
        N ob = new N();
        return ob;
    }
}
class M
{
    static void Main()
    {
        A ob = new A ();
        char c=ob.f<char, int, int>(55, 33);
        //char c = ob.f<char, int, int ?>(55, null);
        Console.WriteLine(c);
    }
}
```

❖ Կոմպարյանտություն [albdarb] N

- Ընդհանրացված մեթոդները (ինչպես սովորական մեթոդները) թույլատրում են միայն կոմպարյանտություն և պարամետրում և վերադարձնող արժեքում՝ այսինքն բազային ցուցիչին կարող է վերագրել ժառանգի օբյեկտ: Օրինակում նույնպես կօգտագործվի `where` սահմանափակում օբյեկտի `new` թույլատրությունը ստանալու համար:

```
using System;
class B
{
}

class C:B
{
}

class A
{
    public void f<T>(T a)
    {
        Console.WriteLine(a);
    }
    public T f2<T>() where T : new()
    {
        T ob =new T();
        return ob;
    }
}

class M
{
    static void Main()
    {
        B obB = new B();
        C obC = new C();
        A ob = new A();
        ob.f<B>(obC);           // ok           // covar
        //ob.f<C>(obB);        // error        // contravar
        obB = ob.f2<C>();       // ok           // covar
        //obC= ob.f2<B>();      // error        // contravar
        Console.WriteLine(obB);
    }
}
```

.....

110. Generics – դելեգատներ

[Шилдг-610; Нэш-318; Троелсен-393; Troelsen-466]

- Ինչպես մեթոդները, դելեգատները նույնպես կարող են օգտվել ընդհանրացումներից:
- Ընդհանրացումներ դելեգատները օժտված են **կոմպարյանտությամբ/ կոնտրավարյանտությամբ** և կարող են ընդունել **where** սահմանափակումներ:

```
dd<string> obdd1 = new dd<string>(f1);
obdd1("albdarb.com");
dd<int> obdd2 = f2; // կարճ գրման ձև
obdd2(99);
dd<string> ob = (a) => //lambda
{
    Console.WriteLine(a.ToUpper());
};
ob("Albert Darbinyan");
static void f1(string arg)
{
    Console.WriteLine(arg.ToUpper());
}
static void f2(int arg)
{
    Console.WriteLine(++arg);
}
// public delegate void dd<T>(T arg) where T:class;
public delegate void dd<T>(T arg);
//class Base
//{ }
//class A : Base
//{ }
```

❖ Կոմպարյանտություն/ կոնտրավարյանտություն

```
dd_params<A> ob1 = new dd_params<A>(f_contravarians_params); // ok
//dd_params<B> ob2 = new dd_params<B>(f_covarians_params); // error
//dd_return<A> ob3 = new dd_return<A>(f_contravarians_return); // error
dd_return<B> ob4 = new dd_return<B>(f_covarians_return); // ok
static void f_contravarians_params(B ob)
{ }
static void f_covarians_params(A ob)
{ }
static B f_contravarians_return()
{
    B ob = new B();
    return ob;
}
static A f_covarians_return()
{
    A ob = new A();
    return ob;
}
delegate void dd_params<T>(T ob);
delegate T dd_return<T>();
class B
{ }
class A : B
{ }
```

.....

111. Action<> և Func<> դելեգատներ

[Троелсен- 394; Troelsen-467; Нейсел- 247; Covaci - 216]

❖ Դելեգատի օգտագործման ժամանակ հաճախ կարելի է օգտվել Action<> և Func<> ներդրված դելեգատներից: Նրանք ներդրված է System անունի տարածքում և կարող են նշել (հղել) մինչև 16 արգումենտներով մեթոդների: Action<> չի վերադարձնում արժեք վերադարձնում է, այլ խոսքով void է վերադարձնում, իսկ Func<> կարող է վերադարձնել արժեք:

❖ Action<> դելեգատի օրինակ՝

```
using System;
// delegate void dd(string s, ConsoleColor o, int x); // old
class Program
{
    static void f(string s, ConsoleColor txtColor, int n)
    {
        Console.ForegroundColor = txtColor;
        for (int i = 0; i < n; i++)
            Console.WriteLine(s);
    }
    static void Main(string[] args)
    {
        //dd ob = new dd(f);
        //ob("barev", ConsoleColor.Yellow, 5);
        Action<string, ConsoleColor, int> d = new Action<string, ConsoleColor, int>(f);
        d("barev", ConsoleColor.Yellow, 5);
    }
}
```

❖ Func<> դելեգատի օրինակ, որի վերադարձնող պարամետրը գտնվում է պարամետրերի վերջում:

```
using System;
class Program
{
    static string f(int x, int y)
    {
        return x.ToString() + y.ToString();
    }
    static void Main(string[] args)
    {
        Func<int, int, string> dob = new Func<int, int, string>(f);
        string s = dob(10, 10);
        Console.WriteLine(s);
    }
}
```

❖ Func<> դելեգատի օրինակ լամբդա արտահայտության միջոցով.

```
using System; //alb
class Program
{
    static void Main(string[] args)
    {
        Func<string, int, int> ob;
        ob = (s, n) =>
        {
            Console.WriteLine(s);
            n++;
            return n;
        };
        Console.WriteLine(ob("alb= ", 50));
    }
}
```

❖ Action<> դելեգատը ունի նաև ոչ ընդհանրացված տարբերակ, իսկ Func<>-ը, ոչ:

```
Action ob;
ob = () => Console.WriteLine("barev");
ob();
```


112. Generics – ինտերֆեյսներ, Covariance և Contravariance

[Шилдг-612, 626; Нейгел -171]

- ❖ Ինտերֆեյսների ընդհանրացումը նման է կլասներին: Օրինակում ինտերֆեյսը որոշում է տիպ, որը պարտադրվում է օգտագործել մեթոդի պարամետրում և օգտագործվում է ինտերֆեյսի հղում:

```
using System;
interface IA<T>
{
    void f(T a);
}
class A<T> : IA<T>
{
    T a;
    public void f(T aa)
    {
        a = aa;
        Console.WriteLine(a);
    }
}
class Program
{
    static void Main(string[] args)
    {
        A<int> ob1 = new A<int>();
        ob1.f(999);
        A<char> ob2 = new A<char>();
        ob2.f('A');
        IA<int> iob1 = ob1;
        iob1.f(777);
        IA<char> iob2 = ob2;
        iob2.f('B');
    }
}
//.....
```

- ❖ Ինտերֆեյսի Covariance և Contravariance ընդհանրացումներ (NET 4)

[Шилдг - 626; Нейгел-171]

- Եթե մեթոդներում կոմպարյանտությունը **ներդրված** է, իսկ կոնտրավարյանտությունը **արգելված**, ապա ընդհանրացումներում արգելված են երկուսը միասին:
- Բոլոր ընդհանրացումները **ինվարիանտ** են: Դա նշանակում է խիստ տիպային հսկում և անվտանգություն: Նույնիսկ չի թույլատրվում տիպերի վերափոխարկում: Սակայն NET4 միջավայրում կատարվում է հնարավորությունների ընդլայնում, որը կիրառվում է դելեգատ ընդհանրացումների և ինտերֆեյս ընդհանրացումների համար: Այդ դեպքում կատարվում է ինվարիանտությունից անցում կոմպարիանտության և կոնտրավարյանտության:
- Եթե անհրաժեշտ է ապահովել Covariance և Contravariance հնարավորությունները, անհրաժեշտ է գրել **out** կոմպարիանտության համար, որպես վերադարձնող արժեք և ավելացնել **in** կոնտրավարյանտության համար, որպես պարամետր:
- Covariance և Contravariance հնարավորությունները դա **պոլիմորֆիզմի** կիրառման ձև է:

❖ Covariance ընդհանրացում

```
using System;
interface ICov<out T>
{
    T ff();
}
class Shape
{
}
class Rectangle:Shape
{
}
class ACov<T> : ICov<T>
{
    T t;
    public T ff()
    {
        return t;
    }
}
class Program
{
    static void Main(string[] args)
    {
        ICov<Shape> i1 = new ACov<Shape>(); // invariant
        ICov<Shape> i2 = new ACov<Rectangle>(); //covariant
        //ICov<Rectangle> i3 = new ACov<Shape>(); //contravariant //error!!!!
    }
}
// .....
```

❖ Contravariance ընդհանրացում [Шилдт -630; Нейгел-174]

```
using System;
interface IContra<in T>
{
    void ff2(T o);
}
class Shape
{
}
class Rectangle:Shape
{
}
class AContr<T> : IContra<T>
{
    public void ff2(T ob)
    {
        T ob2 = ob;
    }
}
class Program
{
    static void Main(string[] args)
    {
        IContra<Rectangle> i1 = new AContr<Rectangle>(); // invariant
        //IContra<Shape> i2 = new AContr<Rectangle>(); //covariant //error!!!!
        IContra<Rectangle> i3 = new AContr<Shape>(); //contravariant
    }
}
```

113. Կլասի հետաձգված ինիցիալիզացիա - Lazy<>

[Троелсен-504; Troelsen-364; Гриффитс-873]

- Կլասների նախագծման ժամանակ լինում են դաշտեր, որոնք հնարավոր է չօգտագործվեն մեթոդների կոդմիջ: Դա առանձնապես վնաս չէ, եթե դաշտի մեծությունը համեմատաբար փոքր է: Սակայն դաշտի մեծ չափերի դեպքում ցանկալի է կիրառել մոտեցումներ, որոնք թույլ կտան դաշտերը ստեղծել միայն օգտագործման ժամանակ: Այդպիսի ձևերից է օրինակ է **Factory Method** պատերնը:
- C# 2010 –ում գոյություն ունի Lazy <> կլաս, որը նույնպես լուծում է վերը նշված խնդիրը: Այն գտնվում է System անունի տարածքում:
- Lazy <> կլասը բացի լռելիությամբ կոնստրուկտորից ունի նաև պարամետրով կոնստրուկտոր, որի պարամետրը կարող է ընդունել ընդհանրացված դելեգատ իր System.Func<> տիպով: Այն կարող է ունենալ մինչև 16 արգումենտ: Func <> -ի աշխատանքը պարզեցնելու համար ցանկալի է օգտագործել լրամբղա արտահայտություն:
- Օրինակում GC.GetTotalMemory(false) մեծ հիշողության օգտագործում է ցույց տալիս: Օրինակում ավելի քիչ է հիշողությունը է զբաղեցնում Lazy<> կլասի օգտագործումը :
- ob.Get() հրամանն է միայն ավելացնում հիշողություն:

```
using System; // Lazy <> // albdarb
class A
{
    int[] a = new int[100000];
}
class C
{
    Lazy<A> obL= new Lazy<A>();
    // A obA = new A();
    public A Get()
    {
        return obL.Value;
        //return obA;
    }
}
class M
{
    static void Main(string[] args)
    {
        C ob = new C();
        //ob.Get();
        Console.WriteLine(GC.GetTotalMemory(false));
        Console.ReadKey();
    }
}
```

- ❖ Եթե անհրաժեշտ է օգտագործել լրամբղա արտահայտությանը, ապա այն կարելի է կիրառել հետևյալ ձևով.

```
Lazy<A> oL =
new Lazy<A>
(
    () =>
    {
        return new A();
    }
);
```

- ❖ Lazy <> կլասի նույն էֆեկտին կարելի է հասնել օգտագործելով կլասի դաշտը որպես **ազրեգացիա**, որը չի համարվում կլասի տվյալների մաս:

```
using System;
class A
{
    int[] a = new int[100000];
}
class C
{
    A obA; // ազրեգացիա
    public A Get()
    {
        obA = new A();
        return obA;
    }
}
class M
{
    static void Main(string[] args)
    {
        C ob = new C();
        ob.Get();
        Console.WriteLine(GC.GetTotalMemory(false));
        Console.ReadKey();
    }
}
```

- ❖ **Բիլ Գեյթս**ն ասել է, որ նա միշտ **ամենաձույլ** (Lazy) մարդկանց տալիս է **ամենադժվար** աշխատանքը, քանի որ նրանք կարողանում են գտնել **ամենահեշտ** ձևը այն անելու: Նա նախընտրում է աշխատել ոչ թե **տքնաջան**, այլ **խելամտորեն**:
 - ❖ **albdarb** -ը ասում է (Մուրը նույնպես): Աշխարհը զարգանում է **էքսպոնենցիալ** տեմպով: Ճիշտ արձագանքեք իրավիճակին:
 - ❖ Դասավանդման **կատեգորիալ** մոտեցումը մասնագետներ է պատրաստում **արհեստական բանականության** պայքարի մենամարտում: Ապագայի միակ շանսը:
-

END C#sharp 1

Թեսթի Օրինակներ.

❖ Ծրագիրը կարտածի (1 բալ)

```
using System;
class Program
{
    static void Main()
    {
        short a = 1;
        switch (a)
        {
            case 1: Console.Write("1");
            case 2: Console.Write("2");
                    break;
            default: Console.WriteLine("d");
                    break;
        }
    }
}
```

- a) 1
 - b) 12
 - c) 12d
 - d) d
 - e) ոչ, կտա թարգմանման սխալ
-

❖ Ինչ արդյունք կարտածի ծրագիրը (2 բալ)

```
using System;
class M
{
    static void Main()
    {
        string i = "1";
        string j = "1";
        Console.WriteLine(i == j);
        Console.WriteLine(Equals(i, j));
        Console.WriteLine(i.Equals(j));
        Console.WriteLine(ReferenceEquals(i, j));
    }
}
```

- a) TrueTrueTrueFalse
 - b) TrueFalseTrueFalse
 - c) TrueTrueFalseTrue
 - d) TrueTrueTrueTrue
-

❖ `int @int = new int();` (1 բալ)

- a) ճիշտ հրամանն չէ
- b) `int` փոփոխականին վերագրում է 0 արժեք
- c) կատարվում է `int` հասցեի վերագրում
- d) հայտարարվում է նոր տիպ
- e) նշվածներից ոչ մեկը ճիշտ չէ

❖ Main – ում ինտերֆեյսը օգտագործվում է (3 բալլ)

```
interface I<in T>
{
    void ff(T o);
}
class Shape
{
}
class Rectangle : Shape
{
}
class A<T> : I<T>
{
    public void ff(T o)
    {
        T oo = o;
    }
}
class Program
{
    static void Main(string[] args)
    {
        I< Shape > i3 = new A< Rectangle >();
    }
}
```

- a) կովարյանտ ձևով
 - b) կոնտրավարյանտ ձևով
 - c) տալիս է թարգմանման սխալ
 - d) նշվածներից կա երկու ճիշտ պատասխան
-

❖ Ծրագիրը (2 բալլ)

```
using System;
class AA
{
    public int a = 9;
}
class BB : AA
{
    public int b;
}
class M
{
    public static void Main()
    {
        AA obAA = new AA();
        BB obBB = new BB();
        obAA = obBB;
        obAA.a = 44;
        Console.WriteLine(obAA.a);
        obAA.b=55;
        Console.WriteLine(obAA.b);
    }
}
```

- a) կարտածի 44,55 հաջորդականությունը
 - b) կտա թարգմանության սխալ
 - c) կտա կատարման սխալ
 - d) կտա զգուշացում
-