مەنبە: www.udmish.cn

Microsoft

Based on
May 2007
Pre-release
Code—
with Updates
Online!

INTRODUCING MICROSOFT® LINQ

ھەققىدە ساۋات

كەمتۈك نۇسخىسى v1.0

Paolo Pialorsi
Marco Russo

ئۆزلەشتۈرگۈچى: مەردان ھوشۇر (ئۇدمىش)
كوررېكتور: ئەلى ئەركىن (سارۋان)

مۇندەرىجە

ئاتالغۇلار ئىزاھاتى 5
ئىككىنچى باب C# تىلىنىڭ خۇسۇسىيەتلىرى 10
C# 2.0 گە قايتا نەزەر 10
كۆپماسلىق 10
مۇۋەققەتلەر 13
نامسىز مېتود 15
Yield ۋە Enumerators 17
C# 3.0 نىڭ خۇسۇسىيەتلىرى 22
var خاس سۆزى 22
كېڭەيتىلمە مېتود 23
ئوبيېكتلارنى دەسلەپلەشتۈرۈش ئىپادىسى 28
نامسىز تىپ 31
Query ئىپادىسى (سۈرۈشتۈرۈك ئىپادىسى) 33
ئۈچىنچى باب Linq گىراماتىكىسىدىن ئاساس 35
LINQ سۈرۈشتۈرۈكلىرى (LINQ Queries) 35
سۈرۈشتۈرۈك گىراماتىكىسى 35
تولۇق سۈرۈشتۈرۈك ئىپادىسى 38
سۈرۈشتۈرۈك مەشغۇلاتچىلىرى 40
Where مەشغۇلاتچىسى 40
ئەمەللەشتۈرۈش مەشغۇلاتچىلىرى (Projection Operators) 42
SelectMany مەشغۇلاتچىسى 43
Ordering مەشغۇلاتچىلىرى (سورتلاش مەشغۇلاتچىلىرى) 45
OrderBy بىلەن OrderByDescending مەشغۇلاتچىسى 45
ThenBy بىلەن ThenByDescending مەشغۇلاتچىسى 46
Reverce مەشغۇلاتچىسى (كۆمتۈرۈش) 48
Grouping مەشغۇلاتچىلىرى (گۇرپىلاش) 48
Join مەشغۇلاتچىلىرى (ھەمدەم) 51

Join (ھەمدەم) 51
GroupJoin مەشغۇلاتچىسى 53
Set مەشغۇلاتچىلىرى (توپلام) 55
Distinct مەشغۇلاتچىسى (تەكرارنى تازىلاش) 55
Union, Intersect, and Except 56
Aggregate مەشغۇلاتچىلىرى (جەملەش مەشغۇلاتچىلىرى) 59
Count ۋە LongCount مەشغۇلاتچىسى 59
Sum مەشغۇلاتچىسى (يىغىندا) 60
Min and Max 63
Average مەشغۇلاتچىسى 65
Generation مەشغۇلاتچىلىرى (قۇرغۇچ) 67
Range (دائىرە مەشغۇلاتچىسى) 67
Repeat (تەكرار مەشغۇلاتچىسى) 68
Empty (قۇرۇق مەشغۇلاتچىسى) 69
Quantifiers مەشغۇلاتچىلىرى (مىقدار مەشغۇلاتچىلىرى) 69
Any مەشغۇلاتچىسى 69
All (ھەممە مەشغۇلاتچىسى) 70
Contains (بارمۇ مەشغۇلاتچىسى) 70
Partitioning مەشغۇلاتچىلىرى (پارچىلاش) 71
Take (نى- ئېلىش مەشغۇلاتچىسى) 72
TakeWhile (چاغدا- ئېلىش مەشغۇلاتچىسى) 73
Skip بىلەن SkipWhile 74
ئېلېمېنت مەشغۇلاتچىلىرى 75
First (تۇنجى مەشغۇلاتىچىسى) 75
FirstOrDefault 75
Last بىلەن LastOrDefault 76
Single 76
SingleOrDefault 77
ElementAt بىلەن ElementAtOrDefault 77

DefaultIfEmpty 78
باشقا مەشغۇلاتچىلار 79
Concat (ئۇلاش مەشغۇلاتچىسى) 79
SequenceEqual 80
كىچىكتۈرۈلمە سۈرۈشتۈرۈكنىڭ قىممەتلىنىشى ۋە كېڭەيتىلمە مېتود چارىسى 81
كىچىكتۈرۈلمە سۈرۈشتۈرۈكنىڭ قىممەتلىنىشى 81
كېڭەيتىلمە مېتود چارىسى 83
ئالماشتۇرۇش مەشغۇلاتچىلىرى 84
ToArray بىلەن ToList 85
ToDictionary 87
ToLookup 88
OfType بىلەن Cast 90

ئاتالغۇلار ئىزاھاتى

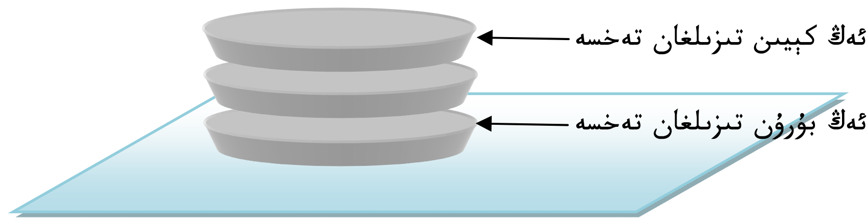
مەزكۇر كىتابنى ئوقۇش جەريانىدا پات-پات كۆلۈپ تۇرىدىغانلىقىڭىزغا ئىشەنچىم كامىل. ئەسلىدە ئاتالغۇلار ئىزاھاتى مەن يازىدىغان مەزمۇن ئەمەس ئىدى، لېكىن كومپيۇتېر ساھەسىدىكى كەسپىي ئاتالغۇلارنى ئاتالغۇ ئىزاھات لۇغەتلىرىمىزدىن تولۇق تاپالايدىغان بولۇشىمىزغا يەنە بىر مەزگىل كېتىدىغان بولغاچقا، بىر قىسىم كەسپىي ئاتالغۇلارنى جۆيلۈشىمىزگە توغرا كەلدى. شۇڭا كىتاب مەزمۇنىدا تۆۋەندىكى سۆزلەرنى ئۇچراتقىنىڭىزدا كۆلگەچ توغرا چۈشىنەرسىز. ئاتالغۇ تەرجىمە خىزمىتىدە قىممەتلىك پىكىرلەرنى بەرگەن ئۆمەرجان ئابدۇراخمان (ئۇچقۇن) ئەپەندىگە كۆپ تەشەككۇر.

كۆپماس \ كۆپماسلىق - ئىنگلىزچىسى: generic، خەنزۇچىسى: 泛型 بولۇپ، ئورتاق تىپلىپ، ھەممىگە ماس كېلىدىغان دېگەن مەنىلەرنى بېرىدۇ. پروگرامما نۇختىسىدىن چۈشەنگەندە، generic تىپ بولسا كونكرېت بىر تىپنى كۆرسەتمەي بەلكى تىپى مەشغۇلاتقا قاتناشقۇچى كونكرېت تىپ تەرىپىدىن بەلگىلىنىدىغان ئېنىقسىز، ھەممىگە ماس كېلىدىغان تىپ بولغاچقا، ئۇنىڭ ھەممىگە ماس كېلىدىغان ئالاھىدىلىكىنى گەۋدىلەندۈرۈش يۈزىسىدىن ئۇنىڭ سۈپەت شەكلى «كۆپماس»، ئىسىم شەكلى «كۆپماسلىق» دەپ ئېلىندى.

كود قاپچۇقى - ئىنگلىزچىسى: Code Container، خەنزۇچىسى: 代码容器 بولۇپ، مەلۇم كودلارنى ئۆزىدە ئېلىپ يۈرەلەيدىغان كود بۆلىكىنى كۆرسىتىدۇ. Container يەنى 容器 پروگراممىا ساھەسىدە كۆپ ئىشلىتىلىدىغان ئاتالغۇ بولۇپ، بىر تۈركۈم نەرسىلەرنى ئۆزىدە تۇتۇپ تۇرالايدىغان ئوبيېكتلارنى كۆپۈرەك مۇشۇ سۆز بىلەن ئاتايدۇ. مەسىلەن، Control Container (كونترول قاپچۇقى) دېگەندەك. بۇ خىل ئالاھىدىلىكنى ئىپادىلەش ئۈچۈن «كود ساندۇقى»، «كود ئېلىپ يۈرگۈچ» دېگەندەك بىر قىسىم سۆز بىرىكمىلىرىنى ئويلاپ باقتىم، لېكىن «كود قاپچۇقى» دەپ ئاتاش ئەڭ مۇۋاپىق تۇيۇلدى.

چاقىرغۇ - ئىنگلىزچىسى: callback، خەنزۇچىسى: 回调 بولۇپ، چاقىرىش دېگەن مەنىدە. پروگراممىدا بىر دانە callback مەلۇم كود بۆلىكىگە توغۇرلانغان ئىسترېلكىغا ئۇلانغان بولىدۇ. callback نى ئىجرا قىلىش ئارقىلىق ئۇ توغۇرلانغان كود بۆلىكىنى ئىجرا قىلغىلى بولىدۇ. دېمەك ئۇ ئەشۇ كود بۆلىكىنىڭ چاقىرغۇچىسى، شۇڭا ئۇ «چاقىرغۇ». بۇ مۇشۇنداق ئاتاشقا يېتەرلىك ئاساس بىلىندى.

دەستە - ئىنگلىزچىسى: stack، خەنزۇچىسى: 栈 بولۇپ، شەيئىلەرنىڭ ئۈستىمۇ-ئۈستى تىزىلغان دەستىسىنى كۆرسىتىدۇ. پروگراممىدىكى stack بولسا بىر خىل ئالاھىدە سانلىق مەلۇمات قۇرۇلمىسى بولۇپ، ئۇنىڭ قۇرۇلمىسى خۇددى تەخسە تىزغانغا ئوخشايدۇ. يەنى، تەخسىلەر بىرىنىڭ ئۈستىگە بىر قويۇلىدۇ. ئۇلارنى ئېلىش ئۈچۈن ئەڭ تۆپىدىن بىر-بىرلەپ ئېلىش كېرەك، دېمەك ئەڭ ئاخىردا تىزىلغىنى ئەڭ باشتا ئېلىنىدۇ، ئەڭ باشتا تىزىلغىنى ئەڭ ئاخىردا ئېلىنىدۇ. بۇ جەرياننى تۆۋەندىكى رەسىمدىكىدەك ئىپادىلەش مۇمكىن:



شۇڭا بۇ خىل سانلىق مەلۇمات قۇرۇلمىسى دەستە دەپ ئېلىندى.

ئۆمۈر - ئىنگلىزچىدا: lifetime، خەنزۇچىدا: 生命周期 بولۇپ، ئۆمۈرى دېگەن مەنىدە. ئۇ پروگراممىدا مەلۇم ئۆزگەرگۈچى مىقدارنىڭ ئۆزى تۇرغان كود بۆلىكىدە ئۈنۈملۈك بولالايدىغان بولالايدىغان ۋاقىت ئۇزۇنلۇقىنى كۆرسىتىدۇ. مەسىلەن، تۆۋەندىكى پروگراممىدا ئوخشىمىغان ئورۇندا ئېنىقلانغان ئۆزگەرگۈچى مىقدارلارنىڭ ئۈنۈملۈك مەزگىلى ئوخشاش بولمايدۇ.

```
public class MisalClass
{
    public int ozgerguchi1 = 1;

    public int misalFun()
    {
        int ozgerguchi2 = ozgerguchi1 + 1 ;
        return ozgerguchi2;
    }
}
```

يۇقىرىقى كودتا ozgerguchi1 نىڭ ئۈنۈملۈك مەزگىلى MisalClass نىڭكى بىلەن ئوخشاش بولۇپ، MisalClass مەۋجۇتلا بولىدىكەن ئۇمۇ مەۋجۇت بولۇپ تۇرالايدۇ. ozgerguchi2 نىڭ ئۈنۈملۈك مەزگىلى قىسقىراق بولۇپ، misalFun فۇنكسىيىسىنىڭ ئىجراسى باشلىنىشى بىلەنلا ئۈنۈم بولۇشقا باشلاپ، ئىجرا تاماملانغاندا ئۈنۈمسىزلىنىدۇ. ئۆزگەرگۈچى مىقدارلارنىڭ بۇ خىل خاسلىقىنى ئۇنىڭ «ئۆمرى» دەپ ئاتاش مۇۋاپىق بىلىندى.

ئۆتكۈنچى كود - ئىنگلىزچىسى IL(Intermediate Code)، خەنزۇچىسى 中间语言 بولۇپ، NET. ئائىلىسىدىكى تىللاردا يېزىلغان كودلار ئاخىرىدا ھەممىسىگە ئورتاق بولغان IL تىلىغا ئايلاندۇرۇلۇپ، ھەققىي ئىجرا بولغاندا ئۇمۇ يەنە ماشىنا تىلىغا ئايلىنىدۇ. ئەمەلىيەتتە، #C بىلەن VB.NET يېزىلغان كودلارنى ئارلاش ئىشلىتىشكە مۇمكىن بولىدىغانلىقىنىڭ سەۋەبىمۇ شۇ. ئۇ خاراكتېرىگە ئاساسەن «ئۆتكۈنچى كود» دەپ ئېلىندى.

تىللارغا ئورتاق ئىجرا سۇپىسى ـ ئىنگلىزچىسى Common Language Runtime، خەنزۇچىسى: 通用语言运行库 بولۇپ، NET. ئائىلىسىدىكى تىللاردا تۈزۈلگەن پروگراممىلار ئىجرا بولۇش ئۈچۈن زۆرۈر ئاساس بولىدىغان ئىجرا سۇپىسىدۇر. ئۇنىڭ خاراكتېرى چىقىش قىلىنىپ «تىللارغا ئورتاق ئىجرا سۇپىسى» دەپ ئېلىندى.

تىپ ـ ئىنگلىزچىسى: Type، خەنزۇچىسى:类型 بولۇپ، تۈرى، تىپى دېگەن مەنىلەردە. پروگراممىدىكى Type بولسا پروگراممىدىكى مەلۇم ئۆزگەرگۈچى مىقدارنىڭ ئۆزىدە سانلانغان ئۇچۇرنىڭ خاراكتېرگە ئاساسەن بۆلۈنگەن تىپىنى ئىپادىلەيدۇ. لېكىن ئۇ Class (类) دىن پەرقلىنىدۇ.

تۈر ـ ئىنگلىزچىسى: Class[klɑ:s]، خەنزۇچىسى: 类 بولۇپ. تىپ، تۈر دېگەن مەنىدە. لېكىن بۇ تىپ Type(类型) بىلەن دوقال كېلىپ قالغانلىقى ئۈچۈن تۈر دەپ ئېلىندى. ئۇنىڭدىن باشقا 项目 project, نىمۇ تۈر دەپ ئېلىش مۇۋاپىق، شۇڭا بۇ كىتاپتا «تۈر» ھەم class ھەم project دېگەن مەنىلەرنى بىرىدۇ. زادى قايسى مەنىدە كەلگەنلىكىدى ئۆزى تۇرۇشلۇق مەزمۇندىن كەلتۈرۈۋېلىدۇ.

مېتود ـ ئىنگلىزچىسى: Method['meθəd]، خەنزۇچىسى: 方法, 函数 بولۇپ، پروگراممىدا بىر بۆلەك ۋەزىپىنى مۇستەقىل ئورۇنلايدىغان كود بۆلىكىنى كۆرسىتىدۇ. سۆز مەنىسىدىن ئالغاندا «ئۇسۇل» دېگەن مەنا بىرىدۇ. لېكىن پروگراممىدىكى خاراكتېرىدىن ئالغاندا ئۇنى «ئۇسۇل» دەپ ئاتاش مۇۋاپىق بىلىنمىگەچكە، بۇ كىتابتا مېتود دەپ ئېلىندى.

كود ـ تەرجىمە ـ ئىنگلىزچىسى: [kəm'pail] Compile، خەنزۇچىسى: 编译 بولۇپ، كومپيۇتېر تىلىدا يېزىلغان كودنى باشقا بىر خىل كومپيۇتېر تىلىغا تەرجىمە قىلىش جەريانىنى كۆرسىتىدۇ (بۇ مەشغۇلاتنى كۆپىنچە شۇ كودنى يازغان IDE ئورۇنلايدۇ). بۇ جەريانىنى 编辑 يەنى تەھرىرلەش دىيىشكە بولمايدۇ. مەشغۇلاتنىڭ جەريان خاراكتېرى چىقىش قىلىنىپ كود ـ تەرجىمە دەپ ئېلىندى. كود تەرجىمە مەشغۇلاتنى ئېلىپ بارغۇچى بولسا «كود ـ تەرجىمان» دەپ ئېلىندى.

مۇۋەققەت ـ ئىنگلىزچىسى delegate، خەنزۇچىسى 委托 بولۇپ، ۋەكىل، ھاۋالە، مۇۋەققەت دېگەن مەنىلەردە. پروگراممىدا delegate خاسلىقى بىرىلگەن تىپلار ئەمەلىيەتتە باشقا كودلارنىڭ ئادرېسىنى ساقلايدىغان ۋەكىل تىپلاردۇر. شۇڭا «مۇۋەققەت» دەپ ئېلىندى.

خۇلق ـ ئىنگلىزچىسى behavior بولۇپ، پروگراممىدا مەلۇم كودنىڭ ئىجرا بولۇش ئالاھىدىلىكىنى ۋە خىزمەت پرىنسىپىنى كۆرسىتىدۇ. دېمەك ئۇ شۇ پروگراممىنىڭ خۇلقى.

خاس سۆز ـ ئىنگلىزچىسى: Key Word، خەنزۇچىسى: 关键词, 保留词 بولۇپ، ھەربىر پروگراممىرلار تەرىپىدىن ئىشلىتىشكە بولمايدىغان بىر قىسىم سۆزلەرنى كۆرسىتىدۇ. مەسىلەن، C# تىلىدا ...static, public, delegate, float, int قاتارلىق سۆزلەر خاس سۆزلەرگە تەۋە بولۇپ ئۇلارنى ئۆزگەرگۈچى مىقدارلارنىڭ ئىسمى سۈپىتىدە قوللىنىشىمىزغا يول قويۇلمايدۇ. بۇنداق سۆزلەر ھەربىر تىلنىڭ ئۆزى ئۈچۈن خاس بولغانلىقى ئۈچۈن ئۇلار «خاس سۆز» دەپ ئېلىنىدى. لېكىن ئاچقۇچلۇق سۆز، يەنى 关键词 خاس سۆزدىن پەرقلىنىدىغان بولۇپ، ئۇنىڭ ئىشلىتىلىش دائىرىسى كەڭرەك. ئۇ مەلۇم نەتىجىگە ئېرىشىش ئۈچۈن تەمىنلىگەن ئاچقۇچلۇق ئۇچۇرنى كۆرسىتىدۇ.

بىنورماللىق ـ ئىنگلىزچىسى: Exceprion، خەنزۇچىسى: 异常 بولۇپ، تاسادىبىلىق، بىنورماللىق دېگەن مەنىلەرنى بېرىدۇ. پروگراممىدا ئۇ پروگرامما ئىجرا جەريانىدا خاتالىق كۆرۈلسە ھاسىل بولىدىغان خاتالىق تىپىنىڭ ئورتاق نامى. مەسىلەن، int i=1/0; بۇ جۈملە IDE تەرىپىدىن خاتالىق يوق دەپ قارىلىدۇ. ئەمما ھەرقانداق ساننى 0 گە بۆلسە مەنىسىز بولىدىغانلىقى ئۈچۈن پروگرامما ئىجرا بولۇپ مۇشۇ جۈملىگە كەلگەندە DevidedByZeroException تىپلىق خاتالىق ئوبيېكتى ھاسىل قىلىدۇ. بۇ خىل ھادىسە «بىنورماللىق» دەپ ئېلىنىدى. مەسىلەن، باياقى جەريانىدا DevidedByZeroException تىپلىق بىنورماللىق چىقىرىلىدۇ.

تىزما ـ ئىنگلىزچىسى: Sequance، خەنزۇچىسى: 串 بولۇپ، بىرقانچە ئېلېمېنتلارنىڭ رەتلىك قاتارىنى كۆرسىتىدۇ. پروگراممىدا، Sequance ئىچىدىكى ئېلېمېنتلار تەرتىپلىك تىزىلغان بولۇپ، ئۇلارغا نۆلدىن باشلانغان تەرتىپ نومۇرى قويۇلغان بولىدۇ. ئۇلار تەرتىپلىك تىزىلغان ئالاھىدىلىكى كۆزدە تۇتۇپ ئۇلارنى «توپلام» دەپ ئېلىشنىڭ ئورنىغا «تىزما» دەپ ئېلىنىدى. گەرچە Collection(集合) مۇ ئېلېمېنتلارنىڭ توپى بولسىمۇ، ئۇ سەل ئابستراكتراق بولۇپ، ئۇنىڭ ئىچىدىكى ئېلېمېنتلار تەرتىپلىك بولۇشىمۇ مۇمكىن، تەرتىپسىز بولۇشىمۇ مۇمكىن. مەسىلەن، Hashtable دىكى ئېلېمېنتلارنى تەرتىپ نومۇرى بىلەن زىيارەت قىلغىلى بولمايدۇ.

ئەزا ـ ئىنگلىزچىسى: Item، خەنزۇچىسى: 项 بولۇپ، مەلۇم تىزما ئىچىدىكى بىر دانە ئېلېمېنتنى كۆرسىتىدۇ. مەسىلەن int[] sanlar = new int[]{1, 2, 3, 4} دەپ ئېنىقلانسا، 1، 2، 3 ۋە 4 لەر sanlar نىڭ ئەزالىرىدۇر.

سۈرۈشتۈرۈك ـ ئىنگلىزچىسى: Query، خەنزۇچىسى: 查询 بولۇپ، ئىزدەش، سۈرۈشتە قىلىش دېگەن مەنىلەردە. كومپيۇتېر پروگراممىچىلىقى ساھەسىدە بۇ ئاتالغۇ ساندان(数据库) مەشغۇلاتىدا ئىنتايىن كۆپ ئىشلىتىلىدۇ. بولۇپمۇ بارلىق Sql جۈملىلىرى بىردەك Query دەپ ئاتىلىدۇ. بۇ خىل ئالاھىدىلىكنى ئۇيغۇر تىلىدا «سۈرۈشتۈرگۈچى»، «ئىزدەش جۈملىسى»، «ئىزدىگۈچى» دېگەندەكلار بىلەن ئىپادىلەش مۇمكىن. لېكىن Query جەريانىنىڭ ئۆزىنى ئىشلەتكۈچى (سۈرۈشتۈرگۈچى) دىن پەرقلىنىدىغان مۇستەقىل جەريان بولغانلىقى كۆزدە تۇتۇلۇپ «سۈرۈشتۈر» سۆزىگە «ۈك» سۆز ياسىغۇچى قوشۇمچىسىنى قوشۇش ئارقىلىق «سۈرۈشتۈرۈك» دەپ ئېلىنىدى.

كىتاب مەزمۇنىدا، SQL سۈرۈشتۈرۈكى (SQL Query)، LINQ سۈرۈشتۈرۈك ئىپادىسى (LINQ Query Expression)، سۈرۈشتۈرۈك ئىپادىسى(Query Expression) قاتارلىق «سۈرۈشتۈرۈك» نى ئۆز ئىچىگە ئالغان ئاتالغۇلارنى ئۇچرىتىپ تۇرىسىز. ئۇلارغا ئايرىم ئىزاھات بېرىلمىدى.

تۆۋەندىكى ئاتالغۇلارنىڭ تەرجىمىسىلا بېرىلدى

ئۇيغۇرچىسى	ئىنگلىزچىسى	خەنزۇچىسى
تۇرغۇن قەۋەت	Static Layer	静态层
كۆرسەتمە	Predicate	谓语
قاتتىق تىپلىق تىل	Strongly Typed Language	强类型语言
تىپ كەلتۈرۈلمىسى	Type Inference	类型推导
ئەمەلىيەشتۈرۈش	Projection	
چارلاش	Enumerate	
قىممەتلىك تىپ	Value Type	值类型
چاقىرىلما تىپ	Reference Type	引用类型

ئەسكەرتىش: يۇقىرىقى ئاتالغۇلار بىردەك بىر قىسىم كەسپداشلار بىلەن پىكىرلىشىش ئاساسىدا تەرجىمە قىلىنغان، مۇشۇ كىتابتىن باشقا ھەرقانداق ماتېرىياللاردىكى ئوخشاش ئاتالغۇلارغا ئاساس بولالمايدۇ. ھەم مەزكۇر تەرجىمىلەر سەۋەبلىك كېلىپ چىققان ھەرقانداق مەسئۇلىيەتنى ئۈستىمىزگە ئالمايمىز.

ئىككىنچى باب C# تىلىنىڭ خۇسۇسىيەتلىرى

تىلغا باغلانغان سۈرۈشتۈرۈك(LINQ) نى ئىشلىتىش ئۈچۈن C#3.0 دىكى بارلىق يېڭىلىقلارنى ئىگىلەش ھاجەتسىز. مەسىلەن، ھېچبىر يېڭىلىق «تىللارغا ئورتاق ئىجرا سۇپىسى»(CLR) نىڭ ئۆزگىرىشىنى تەلەپ قىلمايدۇ. LINQ بولسا يېڭى كود-تەرجىمانلار(C# 3.0 ياكى Microsoft Visual Basic 9.0) غا بېقىندىغان بولۇپ، بۇ كود-تەرجىمانلار .NET2.0 Microsoft دىمۇ نورمال ئىشلەيدىغان ئۆتكۈنچى كود ھاسىل قىلالايدۇ.
قانداقلا بولمىسۇن، بۇ بابتا C# تىلىنىڭ (C# 1.0 دىن C# 3.0 غىچە) خۇسۇسىيەتلىرىنى قىسقىچە تونۇشتۇرۇپ ئۆتۈش مۇۋاپىق بىلىندى. شۇندىلا C# تىلى ئاساسىڭىزنىڭ ئاجىز بولۇشى سەۋەبلىك LINQ بىلىملىرىنى ئاڭقىرالماسلىقىڭىزنىڭ ئالدىنى ئالغىلى بولۇشى مۇمكىن. ئەگەر مەزكۇر بابنى ئاتلاپ ئۆتۈپ كەتسىڭىز، LINQ گىرامماتېكىسىنىڭ ھەققىي ماھىيىتىنى بىلگىڭىز كەلگەندە قايتا كۆرۈپ باقارسىز.

C# 2.0 گە قايتا نەزەر

C# 2.0 دە ئەسلىدىكى C# تىلى ئاساسىدا كۆپ ئىلگىرلەشلەر بولدى. مەسىلەن،كۆپماسلىق ئۇقۇمىنىڭ قوشۇلۇشى پروگراممېرلارنى بىردىن ئارتۇق تىپ پارامېترىنى ئۆز ئىچىگە ئالغان تۈر ۋە مېتودلارنى يېزىش ئىمكانىيىتىگە ئىگە قىلدى. ئەمەلىيەتتە، كۆپماس بولسا LINQ نىڭ تايانچىسى.
مەزكۇر پاراگرافتا، «كۆپماسلىق»، «نامسىز مېتود»(C# 3.0 دىكى lambda ئىپادىلىرىنىڭ ئاساسى)، yield خاس سۆزى ۋە IEnumerable ئېغىزى قاتارلىق LINQ ئۈچۈن ئىنتايىن مۇھىم بولغان C# 2.0 نىڭ خۇسۇسىيەتلىرى تونۇشتۇرۇلىدۇ. LINQ نى ھەقىقىي چۈشىنىش ئۈچۈن بۇ ئۇقۇملارنى بىلىش زۆرۈردۇر.

كۆپماسلىق

نۇرغۇن پروگرامما تىللىرىدا ئۆزگەرگۈچى مىقدار ۋە ئوبيېكتلارنى كونترول قىلىش ئۈچۈن ئېنىق تىپ ۋە تىپلارنى ئالماشتۇرۇشقا كەسكىن ئالماشتۇرۇش قائىدىلىرى بېكىتىلگەن. ئومۇملاشتۇرۇش نۇختىسىدىن ئېيتقاندا، قاتتىق تىپلىق تىللاردا يېزىلغان كودلاردا بەزى نوقسانلارنىڭ ساقلانغانلىقىنى بايقايمىز. تۆۋەندىكى كودقا قاراپلى:

```
int Min( int a, int b ) {
    if (a < b) return a;
    else return b;
}
```

يۇقىرىقى كودتىكى Min مېتودى پەقەت تىپى int بولغان پارامېترلارغىلا مەشغۇلات بىجىرەلەيدۇ. ئەگەر ئۇنى باشقا تىپلىق پارامېترلارغا قوللانماقچى بولساق، چوقۇم شۇ تىپقا خاس مېتود

```
float Min( float a, float b ) {
    if (a < b) return a;
    else return b;
}
```

object تىپى بارلىق تىپلارنىڭ ئاتىسى بولغاچقا، بالا تىپلارنى ئۇنىڭغا ئالماشتۇرغىلى ۋە ئەسلىگە قايتۇرغىلى بولىدۇ، شۇڭا **object** تىپىنى ئورتاق تىپ قىلىپ ئىشلىتىشكە ئادەتلەنگەن پروگراممېرلار بەلكىم تۆۋەندىكىدەك كود يېزىپ يۇقىرىقى ئاۋارىچىلىقتىن قۇتۇلماقچى بولۇشى مۇمكىن:

```
object Min( object a, object b ) {
    if (a < b) return a;
    else return b;
}
```

تولىمۇ ئەپسۇس، ئومۇملاشقان **object** تىپىغا نىسبەتەن «دى كىچىك»(<) مەشغۇلاتچىسىنى ئىشلىتىش ئۇنۇمسىزدۇر. شۇڭا يەنىلا ئورتاق ئېغىزنى ئىشلىتىشكە توغرا كېلىدۇ:

```
IComparable Min( IComparable a, IComparable b ) {
    if (a.CompareTo( b ) < 0) return a;
    else return b;
}
```

گەرچە مەسىلىنى دەماللىق ھەل قىلغان بولساقمۇ، لېكىن چوڭ پېشكەلدىن بىرنى تېرىدۇق: **Min** فۇنكسىيىسى ئىجرا جەريانىدا ئۆتكۈنچى تىپ ھاسىل قىلىپ قويىدى. يەنى، مەسىلەن، **Min** نى ئىشلەتكۈچىنىڭ ئۇنىڭغا ئىككى دانە **int** تىپلىق پۈتۈن سان يوللىشى **int** دىن **IComparable** غا ئالماشتۇرۇش مەشغۇلاتىنى كەلتۈرۈپ چىقىرىدۇ، لېكىن بۇ جەريان ئېنىقلا ئارتۇقتىن-ئارتۇق **CPU** چىقىمى تەلەپ قىلىدۇ. ھەتتا بەزىدە بىنورماللىق(**Exception**) چىقىرىشىمۇ مۇمكىن.

```
int a = 5, b = 10;
int c = (int) Min( a, b );
```

2.0 C# دا بۇ مەسىلە كۆپماسلىق ئارقىلىق ھەل قىلىندى. كۆپماسلىقنىڭ ئاساسى پىرىنسىپى شۇكى، **C#** كود-تەرجىمانىنىڭ تىپ رەتلەش خىزمىتى ھازىر-جاۋاب كود-تەرجىمانغا ئۆتكۈزۈپ بېرىلدى. تۆۋەندىكىسى **Min** فۇنكسىيىسىنىڭ كۆپماس نۇسخىسى:

```
T Min<T>( T a, T b ) where T : IComparable<T> {
    if (a.CompareTo( b ) < 0) return a;
    else return b;
}
```

ئەسكەرتىش: ھازىر-جاۋاب كود-تەرجىمان(Jitter) دېگىنىمىز NET. ئىجرا سۇپىسىنىڭ بىر قىسمى بولۇپ، ئۇ ئۆتكۈنچە كود IL نى ماشىنا كودىغا ئايلاندۇرىدۇ. NET. مەنبە كودىنى كود-تەرجىمە قىلغىنىڭىزدا، ئىجرا بولالايدىغان ۋە IL كودىنى ئۆز ئىچىگە ئالغان ھاسىلە ياسايدۇ. بۇ ھاسىلە پەقەت تۇنجى قېتىم ئىجرا بولغاندىلا ھازىر-جاۋاب كود-تەرجىمان تەرىپىدىن ماشىنا كودىغا ئايلاندۇرۇلىدۇ.

تىپ رەتلەش ۋەزىپىسىنى ھازىر جاۋاب كود-تەرجىمانغا تاپشۇرۇش ياخشى قارار. چۈنكى: ئۇ ئوخشاش كودنىڭ ئوخشىمىغان نۇسخىلىرىنى ھاسىل قىلالايدۇ، شۇڭا ئوخشىمىغان نۇسخىدىكى كودنى ئوخشىمىغان تىپلارغا ئىشلەتكىلى بولىدۇ. بۇ ئۇسۇل ماكرولۇق كېڭەيتىشكە ئوخشاپ قالىدۇ. ئوخشىمايدىغان يېرى، بۇ ئۇسۇلغا كودنىڭ شىددەت بىلەن كۆپىيىشىنىڭ ئالدىنى ئېلىش ئۈچۈن ياخشىلىنىش ئېلىپ بېرىلغان. يەنى، چاقىرىلما تىپلىق پارامېتىرنى كۆپماس تىپى ئورنىدا ئىشلىتىدىغان كۆپماس مېتودلار ئۈچۈن بىرلا نۇسخىدا IL كودى ھاسىل قىلىپ قويىدىغان ئەھۋال مەۋجۇت ئىدى.
كۆپ ماسلىق ئارقىلىق

```
int a = 5, b = 10;
int c = (int) Min( a, b );
```

نىڭ ئورنىغا

```
int a = 5, b = 10;
int c = Min<int>( a, b );
```

دەك يازالايمىز.
ئەمەدى، بۇرۇن تىپ ئالماشتۇرۇشقا سەرپ قىلىپ يۈرگەن CPU چىقىملىرىمىز تېجىلىپ، پروگراممىمىز تېخىمۇ تېز ئىجرا بولىدىغان بولدىدە.... چۈنكى بۇيەردە ھېچقانداق تىپ ئالماشتۇرۇش مەشغۇلاتى يۈز بەرمەيدۇ. ئۇنىڭ ئۈستىگە كود-تەرجىمان بېرىلگەن پارامېتىرنىڭ قىممىتىگە ئاساسەن كۆپماس تىپ T نىڭ ئېنىق تىپىنى پەرەز قىلالايدۇ(تىپ كەلتۈرۈلمىسى دەپ ئاتىلىدۇ). شۇڭا يۇقىرىقى كودنى تېخىمۇ ئاددىيلاشتۇرۇپ تۆۋەندىكىدەكمۇ يازالايمىز:

```
int a = 5, b = 10;
int c = Min( a, b );
```

تىپ كەلتۈرۈلمىسى: تىپ كەلتۈرۈلمىسى ئىنتايىن مۇھىم ئىقتىدار بولۇپ، ئۇ كود-تەرجىماننى تېپقا ئالاقىدار تەپسىلىي ئىشلارغا بۆيرۇپ، سىزنى تېخىمۇ ئابستراكت كود يېزىش ئىمكانىيىتىگە ئىگە قىلىدۇ.

كۆپماسلىقتىن پايدىلىنىپ كۆپماس مېتود ئېنىقلىغىلىلا بولۇپ قالماي، يەنە كۆپماس تۈر، ئېغىزلارنىمۇ ئېنىقلىغىلى بولىدۇ. كۆپماسلىق ئۈستىدە توختىلىش بۇ كىتابنىڭ ئاساسلىق مەقسىتى بولمىغاچقا تەپسىلىي توختالماي. لېكىن، شۇنى يەنە دېگىم كەلدى: كۆپماسلىقنى چۈشەنمەي تۇرۇپ LINQ كۆرسىڭىز ئۆزىڭىزنى راھەت ھېس قىلالمايسىز.

مۇۋەققەتلەر

مۇۋەققەت دېگىنىمىز بىر ياكى بىردىن ئارتۇق مېتودنىڭ قاپلانما تۈرىدىن ئىبارەت. ئىچكى قىسمىدا، بىر دانە مۇۋەققەتتە مۇشۇ مۇۋەققەت قاپلىغان مېتودلارغا قارىتىلغان ئىستىرېلكىلار تىزمىسى ساقلىنىدۇ. ھەر بىر ئىستىرېلكا ئۆزى توغۇرلانغان مېتودنى ئۆز ئىچىگە ئالغان تۈرنىڭ چاقىرىلمىسىغا ماس كېلىدۇ.
ھەربىر مۇۋەققەت بىر-قانچە مېتودنى قاپلىيالايدۇ. لېكىن بۇ پاراگرافتا پەقەت بىرلا مېتودنى قاپلىغان مۇۋەققەتلەر كۆپرەك كۆڭۈل بۆلىنىدۇ. ئابستراكتراق نۇختىدىن ئېلىپ ئېيتساق، بۇ خىلدىكى مۇۋەققەتلەرنى «كود قاپچۇقى» غا ئوخشىتىش مۇمكىن. بۇ قاپچۇقتىكى كودنى ئۆزگەرتكىلى بولمايدۇ. لېكىن ئۇ چاقىرىلما دەستىلىرىگە ئەگىشىپ يۆتكىلەلەيدۇ ھەمدە تاكى ئۇنى ئىشلىتىش ئېھتىياجى قالمىغانغا قەدەر مەۋجۇت بولۇپ تۇرالايدۇ. ئۇنىڭدىن باشقا، مۇۋەققەت يەنە، ئۆزى قاپلىغان مېتودنى ئۆز ئىچىگە ئالغان تۈرنى ھېچ بولمىغاندا ئۆزىنىڭ ئۆمرى بىلەن تەڭ ياشىغۇزالايدۇ.

نامسىز مېتود ئەمەلىيەتتە مۇۋەققەتنىڭ گراماتىكىسىغا ئاساسەن ئۆزگەرتىلىپ قۇرۇلغان. ئۇنىڭغا ئائىت مەزمۇنلار كېيىنكى پاراگرافتا سۆزلىنىدۇ. مۇۋەققەتتىن بىرنى ئېنىقلاش ئەمەلىيەتتە شۇ مۇۋەققەتنىڭ ئۆزىنى قۇرىدىغان تىپتىن بىرنى ئېنىقلاش بىلەن باراۋەر. مۇۋەققەت ئېنىقلىغاندا چوقۇم ئۇنىڭ مېتود ئەندىزىسىنى تولۇق تەمىنلىشى كېرەك. كود 2.1 دا ئۈچ خىل ئوخشمايدىغان مۇۋەققەتنىڭ ئېنىقلىمىسى بېرىلدى. ئۇلار ئۆزلىرى بىلەن ئوخشاش ئەندىزىدىكى مېتودلارنى قاپلىيالايدۇ.

كود 2.1

```
delegate void SimpleDelegate();
delegate int ReturnValueDelegate();
delegate void TwoParamsDelegate( string name, int age );
```

مۇۋەققەت بۇرۇنقى C تىلىدىكى فۇنكسىيە ئىستىرېلكىسىغا قارىغاندا تېخىمۇ قېلىپلاشقان، تېخىمۇ بىخەتەر. C# 1.x دە ئاشكارە يوسۇندا ئوبيېكت قۇرۇش ئۇسۇلى ئارقىلىقلا مۇۋەققەت قۇرغىلى بولاتتى. مەسىلەن كود 2.2 دا كۆرسىتىلگەندەك:

كود 2.2

```
public class DemoDelegate {
    void MethodA() { … }
    int MethodB() { … }
    void MethodC( string x, int y ) { … }

    void CreateInstance() {
        SimpleDelegate a = new SimpleDelegate( MethodA );
        ReturnValueDelegate b = new ReturnValueDelegate ( MethodB );
        TwoParamsDelegate c = new TwoParamsDelegate( MethodC );
```

```
            // …
        }
}
```

1.x C# دىكى مۇۋەققەت ئىنقلاش گراماتىكىسدا نوقسان بارلىقنى بايقىدىڭىزمۇ؟. قائىدىسى بويىچە نىشان مېتودنىڭ ئەندىزىسى مۇۋەققەت ئەندىزىسى بىلەن بىردەك بولۇشى كېرەك. ئۇنداق بولمىغاندا كومپىلايدىن ئۆتمەيدۇ. شۇنداق تۇرۇقلۇق يەنە نېمە ئۈچۈن new خاس سۆزنى ئىشلىتىمىز، ئۇنى ئىشلىتىش ئىنقلا بىزدىن مۇۋەققەت ئىسمىنى بىلىشىمىزنى تەلەپ قىلىدۇ. ئەمەلىيەتتە مۇۋەققەت تىپىنى كودنىڭ باش-ئاخىرنى تەھلىل قىلىش ئارقىلىقمۇ كەلتۈرۈپ چىقىرىۋالغىلى بولىدۇ(كود-تەرجىمان شۇنداق قىلىشى كېرەك ئىدى).

2.0 C# دە بۇ مەسىلە بايقىلىپ گراماتىكىسى تېخىمۇ ئاددىيلاشتۇرۇلدى. ئالدىنقى مىسال كوددا قۇرغان مۇۋەققەت ئوبيېكتىنى ئەمدى new خاس سۆزنى ئىشلەتمەي تۇرۇپمۇ قۇرالايدىغان بولدۇق. پەقەت مېتود نامىنى يوللاپ بەرسەكلا بولىدۇ. كود-تەرجىمان مېتود ئەندىزىسىگە ئاساسەن مۇۋەققەت تىپىنى كەلتۈرۈپ چىقىرىپ، new خاس سۆزنى كود-تەرجىمە جەريانىدا ئاپتوماتىك قوشىدۇ. مەسىلەن، كود 2.3 دىكى 2.0 C# كودى كود-تەرجىمە قىلىنىش ئارقىلىق ھاسىل بولغان IL كودى، 1.x C# نىڭ مىسال كودىدا ھاسىل قىلىنغىنى بىلەن ئوپمۇ-ئوخشاش.

كود 2.3

```
public class DemoDelegate {
    void MethodA() { … }
    int MethodB() { … }
    void MethodC( string x, int y ) { … }

    void CreateInstance() {
        SimpleDelegate a = MethodA;
        ReturnValueDelegate b = MethodB;
        TwoParamsDelegate c = MethodC;
        // …
    }
    // …
}
```

ھەتتا كۆپماس مۇۋەققەتمۇ ئىنىقلىيالايسىز. بۇ، كۆپماس تۈرنىڭ ئىچىدە مۇۋەققەت ئىنقلاشتا بەكلا ئەسقاتىدۇ. ئۇنىڭ ئۈستىگە كۆپماس مۇۋەققەت LINQ دا ئىنتايىن مۇھىم ئورۇن تۇتىدۇ. مەۋجۇت مېتودنىڭ ئىچىگە ھەركەتچان ھالەتتە كود قىستۇرۇش بولسا مۇۋەققەتنىڭ ئەڭ ئومۇملاشقان قوللىنىشىدۇر. كود 2.4 دا Repeat10Times بولسا ئۆزگەرتكىمىز يوق مېتود:

كود 2.4

```
public class Writer {
    public string Text;
    public int Counter;
    public void Dump() {
        Console.WriteLine( Text );
        Counter++;
    }
}

public class DemoDelegate {
    void Repeat10Times( SimpleDelegate someWork ) {
        for (int i = 0; i < 10; i++) someWork();
    }

    void Run1() {
        Writer writer = new Writer();
        writer.Text = "C# chapter";
        this.Repeat10Times( writer.Dump );
        Console.WriteLine( writer.Counter );
    }
    // …
}
```

نۆۋەتتە **SimpleDlegate** تىپلىق چاقىرغۇ مەۋجۇت(**simpleWork** شۇ)، بىراق بىزنىڭ مەقستىمىز قىستۇرۇلغان مېتودقا ھەرپـبەلگە تىزمىسى يوللاپ بېرىش ۋە قىستۇلغان مېتودنىڭ قانچە قېتىم ئىجرا بولغانلىقىنى ساناش. شۇڭا، **Dump** مېتودى ئۈچۈن ئۈچۈر تەمىنلەپ بېرىدىغان **Writer** تۈرىنى قۇرىۋالدۇق. دېمەك، كود قىستۇرۇش ئۈچۈن ئىككى تۈر ئېنىقلاشقا توغرا كەلدى. ئەمەلىيەتتە بۇ خىل ئۇسۇلنى نامسىز مېتود ئارقىلىق تېخىمۇ ئاددىيلاشتۇرغىلى بولىدۇ.

نامسىز مېتود

ئالدىنقى بۆلەكتە، مۇۋەققەتنىڭ دائىملىق ئىشلىتىش ئۇسۇلى ئۈستىدە توختالدۇق. 2.0 #C دە نامسىز مېتودلارنى قوللىنىش ئارقىلىق كود 2.4 دىكىدەك كودلارنى يېزىشنىڭ تېخىمۇ ئاددىي يوللىرى تەمىنلەنگەن. مەسىلەن:

كود 2.5

```
public class DemoDelegate {
    void Repeat10Times( SimpleDelegate someWork ) {
        for (int i = 0; i < 10; i++) someWork();
    }

    void Run2() {
        int counter = 0;
        this.Repeat10Times( delegate {
            Console.WriteLine( "C# chapter" );
            counter++;
        } );
        Console.WriteLine( counter );
    }
    // …
}
```

بۇ كوددا **Writer** تۈرىنى ئىشلىتىشنىڭ ھاجىتى بولمىدى. چۈنكى كود ـ تەرجىمان **Writer** تۈرىنىڭ رولىنى ئۆتەيدىغان نامسىز تۈرنى يوشۇرۇن قۇرۇپ، ئىشلارنى ئۆز يولىدا ماڭغۇزالايدۇ. ئەمەلىيەتتە، بىز **Repeat10Times** نى چاقىرىش بىلەن بىر ۋاقىتتا مېتوددىن بىرنى ئېنىقلىدۇق. ئېنىقلانغان نامسىز مېتود **Repeat10Times** نىڭ پارامېترى **someWork** قا يوللىنىپ قىستۇرۇلما شەكلىدە ئىجرا بولۇۋېرىدۇ، ئۇنىڭ ئۈچۈن ئايرىم تۈر ئېنىقلاشنىڭ ئورنى قالمىدى. بۇ يەردە ئېنىقلانغان ئاشۇ نامى يوق مېتود «نامسىز مېتود» دەپ ئاتىلىدۇ.
نامسىز مېزود ئېنىقلىغاندا چوقۇم **delegate** خاس سۆزىدىن باشلىنىشى كېرەك.
شۇنىسى ئېسىڭىزدا تۇرسۇن: ئەمەلىيەتتە كود ئىچىگە كود قىستۇرغىلى بولمايدۇ، پەقەت مەلۇم كودقا قارىتىلغان ئىسترېلكىنىلا باشقا كود بۆلىكى ئىچىگە قىستۇرۇشقا بولىدۇ.
ئەگەر نامسىز مېتود يوللانماقچى بولغان ئورۇندىكى مۇۋەققەت تىپلىق پارامېتىر تەۋە بولغان مۇۋەققەت تىپىنىڭ ئەندىزىسىدە پارامېتىر بېكىتىلگەن بولسا، **delegate** خاس سۆزىنىڭ ئارقىسىغا تىرناق ئېچىپ پارامېتىرلىق شەكلىنى قوللىنىشقا بولىدۇ. تۆۋەندىكى كود 2.1 دىكى **TwoParamsDlegate** تىپىغا ماس نامسىز مېتود ئائىت مىسال بېرىلدى:

كود 2.6

```
public class DemoDelegate {

    void Repeat10Times( TwoParamsDelegate callback ) {
        for (int i = 0; i < 10; i++) callback( "Linq book", i );
    }

    void Run3() {
```

```
        Repeat10Times( delegate( string text, int age ) {
            Console.WriteLine( "{0} {1}", text, age );
        } );
    }
    // …
}
```

Yield ۋە Enumerators

1.x #C دە چــارلاش مەشــغۇلاتىنى قــوللىتىش ئۈچــۈن ئىككــى دانە ئېغىــز تەمىــنلەنگەن. System.Collections نام بوشلۇقىدا ئۇلارنىڭ ئېنىقلىمىسى بېرىلگەن:

كود 2.7

```
public interface IEnumerator {
      bool MoveNext();
      object Current { get; }
      void Reset();
}
public interface IEnumerable {
      IEnumerator GetEnumerator();
}
```

دېمەك، **IEnumerable** ئېغىزىنى ئەمەلگە ئاشۇرغان ئوبيېكتنى **IEnumerator** ئېغىزىنى ئەمەلگە ئاشۇرغان ئوبيېكت ئارقىلىق چارلىغلى بولىدۇ. چارلاش مەشغۇلاتى **MoveNext** مېتودىنى چاقىرىش ئارقىلىق تاكى **false** قىممەت قايتۇرۇلغىچە ئېلىپ بېرىلالايدۇ.

كــود 2.8 دە يــۇقىرىقى ئۇســۇل ئــارقىلىق چــارلىغلى بولــىدىغان تــۈر ئېنىقلانــدى. كــۆرۈپ تۇرغىنىڭىزدەك، **CountdownEnumerator** تۈرى ئىنتايىن مۇرەككەپ. ئۇنىڭدا، چارلىغۇچىنىڭ قايتۇرىــدىغىنى باشــقا نەرســە ئەمەس بەلكــى **Countdown** تۈرىــدە ئېنىقلانغــان مىقــدار **StartCountdown** دىن باشلىنىپ بىردىن كېمىيىپ بارىدىغان قىممەتدۇر.

كود 2.8

```
public class Countdown : IEnumerable {
    public int StartCountdown;

    public IEnumerator GetEnumerator() {
        return new CountdownEnumerator( this );
    }
}
```

```
public class CountdownEnumerator : IEnumerator {
    private int _counter;
    private Countdown _countdown;

    public CountdownEnumerator( Countdown countdown ) {
        _countdown = countdown;
        Reset();
    }

    public bool MoveNext() {
        if (_counter > 0) {
            _counter--;
            return true;
        }
        else {
            return false;
        }
    }

    public void Reset() {
        _counter = _countdown.StartCountdown;
    }

    public object Current {
        get {
            return _counter;
        }
    }
}
```

ھەققىي چارلاش مەشغۇلاتى پەقەت **CountdonwEnumerator** ئىشلىتىلگەندىلا ئېلىپ بېرىلىدۇ. مەسىلەن، تۆۋەندىكى ئىشلىتىلىشىگە قاراڭ:

كود 2.9

```
public class DemoEnumerator {
    public static void DemoCountdown() {
        Countdown countdown = new Countdown();
        countdown.StartCountdown = 5;
```

```
        IEnumerator i = countdown.GetEnumerator();
        while (i.MoveNext()) {
            int n = (int) i.Current;
            Console.WriteLine( n );
        }
        i.Reset();
        while (i.MoveNext()) {
            int n = (int) i.Current;
            Console.WriteLine( "{0} BIS", n );
        }
    }
    // …
}
```

GetEnumerator مېتودىنى چاقىرىش ئارقىلىق چارلىغۇچى ئوبيېكت تەمىنلىنىدۇ. كودىدا، بىز، Reset مېتودىنىڭ رولىنى نامايەن قىلىش ئۈچۈن ئىككى قېتىم ئايلاندۇرۇش(loop) ئېلىپ باردۇق. Current خاسلىقى ئارقىلىق ئېرىشكەن قىممەتنى چوقۇم int غا ئايلاندۇرۇشىمىز كېرەك. چۈنكى بىزنىڭ ئىشلەتكىنىمىز چارلاش ئېغىزلىرىنىڭ كۆپماس نۇسخىلىرى بولمىغاچقا، Current دىن قايتىدىغىنى object تىپلىق بولىدۇ.

ئەسكەرتىش: 2.0 #C دا يېڭىدىن قوشۇلغان كۆپماس ئۇقۇمىنى چارلاش ئېغىزلىرىغىمۇ قوللانغان. ماس ھالدا، System.Collections.Generic نام بوشلۇقىدا IEnumerable<T> ۋە IEnumerator<T> دىن ئىبارەت كۆپماس چارلاش ئېغىزلىرى كۆپەيتىلگەن. بۇ ئېغىزلار ئارقىلىق ئۆزىمىزنىڭ تىپلىرى بىلەن object تىپى ئارىسىدىكى ئالمىشىش جەريانىنى قىسقارتقىلى بولىدۇ. بۇ ئىقتىدار چارلانغۇچى قىممەتلىك تىپ بولغاندا تېخىمۇ ئۈنۈمىنى كۆرسىتىدۇ. چۈنكى CPU سەرپىياتى تەلەپ قىلىدىغان قاپلاش ۋە قاپتىن يېشىش مەشغۇلاتى ئېلىپ بېرىلمايدۇ.

1.x #C دىن تارتىپ، foreach جۈملىسىنى ئىشلىتىپ چارلاش مەشغۇلاتىنى تېخىمۇ ئاددىيلاشتۇرغىلى بولىدى. كود 2.10 نىڭ نەتىجىسى يۇقىرىقى مىسالنىڭكى بىلەن ئوخشاش.

كود 2.10

```
public class DemoEnumeration {
    public static void DemoCountdownForeach() {
        Countdown countdown = new Countdown();
        countdown.StartCountdown = 5;

        foreach (int n in countdown) {
            Console.WriteLine( n );
        }
```

```
        foreach (int n in countdown) {
            Console.WriteLine( "{0} BIS", n );
        }
    }
   // ...
}
```

foreach ئىشلىتىلگەندە، كود-تەرجىمان ئىچكى قىسمىدا چارلانغۇچىنىڭ GetEnumerator مېتودىنى چاقىرىپ IEnumerator ئېغىزنى(چارلانغۇچىنىڭ) ئېرىشكەندىن كېيىن ھەربىر ئايلانمىدىن بۇرۇن MoveNext مېتودىنى چاقىرىدۇ. لېكىن، (foreach ئىشلىتىلگەندە) Reset مېتودى ھەرگىز چاقىرىلمايدۇ. (ئاخىرقى چەكتىن باشقا قايتىش مەسىلىسىنى چارلىغۇچى تۈردىن ئىككىنى قۇرۇش ئارقىلىق ھەل قىلىدۇ)

2.0 C# دا يېڭى قوشۇلغان yield جۈملىسى ئارقىلىق كود-تەرجىماننى ئاپتوماتىك ھالدا IEnumerator ئېغىزىنى ھاسىل قىلدۇرغىلى بولىدۇ. yield جۈملىسىنى پەقەت return ياكى break جۈملىسىنىڭ ئالدىدىلا بىۋاسىتە ئىشلەتكىلى بولىدۇ. كود 2.11 دا ئىقتىدارى CountdownEnumerator تۈرى بىلەن ماس كېلىدىغان تۈر قۇرۇلدى.

كود 2.11

```
public class CountdownYield : IEnumerable {
    public int StartCountdown;

    public IEnumerator GetEnumerator() {
        for (int i = StartCountdown - 1; i >= 0; i--) {
            yield return i;
        }
    }
}
```

لوگىكا نۇختىسىدىن ئېلىپ ئېيتقاندا، yield return جۈملىسى ئىجرا مەشغۇلاتىنى ۋاقىتلىق توختىتىپ قويۇش بىلەن باراۋەر بولۇپ، MoveNext كېيىنكى قېتىم چاقىرىلغاندا ئاندىن داۋاملىشىدۇ. شۇنىسى ئېسىڭىزدە بولسۇن، پۈتكۈل چارلاش جەريانىدا GetEnumerator مېتودى پەقەت بىرلا قېتىم چاقىرىلىپ IEnumerator ئېغىزىنى ئەمەلگە ئاشۇرغان تۈر قايتۇرۇپ بېرىدۇ. پەقەت مۇشۇ تۈرلا yield جۈملىسىنى ئۆز ئىچىگە ئالغان مېتود خۇلقىنى ھەقىقىي ئەمەلگە ئاشۇرغان بولىدۇ.

yield جۈملىسىنى ئۆز ئىچىگە ئالغان مېتود «迭代器 ,interator» دەپ ئاتىلىدۇ (چارلىغۇچى، يەنى بىردىن-بىردىن ئېلىپ تەكشۈرگۈچى). بىردانە interator بىرقانچە yield جۈملىسىنى ئۆز ئىچىگە ئالالايدۇ. كود 2.12 دىكىدەك پىرىنسىپ قائىدىگە مۇتلەق ئۇيغۇن بولۇپ، ئىقتىدار جەھەتتە

كود 2.12

```
public class CountdownYieldMultiple : IEnumerable {
    public IEnumerator GetEnumerator() {
        yield return 4;
        yield return 3;
        yield return 2;
        yield return 1;
        yield return 0;
    }
}
```

IEnumerator نىڭ كۆپماس نۇسخىسىنى قوللىنىش ئارقىلىق CountdownYield تۈرىنىڭ كۈچلۈك تىپلانغان (强类型 ,strongly typed) نۇسخىسىنى ھۆججەتقا چىقارغىلى بولىدۇ. مەسىلەن:

كود 2.13

```
public class CountdownYieldTypeSafe : IEnumerable<int> {
    public int StartCountdown;

    IEnumerator IEnumerable.GetEnumerator() {
        return this.GetEnumerator();
    }
    public IEnumerator<int> GetEnumerator() {
        for (int i = StartCountdown - 1; i >= 0; i--) {
            yield return i;
        }
    }
}
```

كۈچلۈك تىپلانغان نۇسخىسىدا ئىككى دانە GetEnumerator مېتودى ئېنىقلانغان بولۇپ، بىرى كۆپ ماس بولمىغان كودلارغا (IEnumerable نى قايتۇرىدىغان) ماس كېلىدۇ، يەنە بىرى بولسا كۈچلۈك تىپلانغىنى.

C# 3.0 نىڭ خۇسۇسىيەتلىرى

var خاس سۆزى

يېڭىدىن قوشۇلغان var خاس سۆزى ئارقىلىق، تىپى ئېنىق بولمىغان ئۆزگەرگۈچى مىقدار ئېنىقلاشقا بولىدۇ. var بىلەن باشقا «ئېنىق تىپلار» ئارىسىدىكى ئالماشتۇرۇشنى .NET قۇرۇلمىسى ئاپتوماتىك بېجىرەلەيدۇ. شۇنى ئەسكەرتىش زۆرۈركى، var ئارقىلىق ئېنىقلاش object ئارقىلىق ئېنىقلاشقا باراۋەر ئەمەس. تۆۋەندىكى مىسال بۇنىڭغا ئىسپات بولالايدۇ:

```
var a = 2;          // پۈتۈن سان تىپلىق قىلىپ ئېنىقلاندى
object b = 2;       //   قىممەتلىك تىپ چاقىرىلما تىپقا ئالماشتۇرۇلدى، سەرپىياتى يۇقىرى
int c = a;          // ھېچ قانداق ئالماشتۇرۇش يۈز بەرمەيدۇ، تېز
int d = (int) b; //  چاقىرىلما تىپ قىممەتلىك تىپقا مەجبۇرى ئالماشتۇرۇلدۇ، سەرپىيات
```

varنىڭ ھەقىقىي تىپىنىڭ قانداق بولىشى ئەمەلىي ئەھۋالغا قاراپ بېكىتىلىدۇ

```
int a = 5;
var b = a;
بۇ ئىككى خىل ئېنىقلاش باراۋەر
int a = 5;
int b = a;
```

C# تىلىدا تىپلارنىڭ تۈرلىرى شۇنچىلىك تولۇق تۇرۇقلۇق يەنە نېمىشقا پروگراممىنىڭ ئوقۇشچانلىقىنى تۆۋەنلىتىۋېتىدىغان var خاس سۆزىنى ئىشلىتىمىز؟ قارىماققا var ھورۇن پروگراممېرلارنىڭ ئېھتىياجى ئۈچۈن لايىھىلەنگەندەك تۇرسىمۇ، ئەمەلىيەتتە ئۇ «نامسىز ئۆزگەرگۈچى مىقدار»(كېيىن سۆزلىنىدۇ) ئېنىقلاشنىڭ بىردىن- بىر يولى.

بۇ يەردىكى var تىپ- بىخەتەرلىكىنى ئەمەلگە ئاشۇرغانلىقى ئۈچۈن Basic Vsiaul دىكى varخاس سۆزىدىن پەرقلىنىدۇ(كۆپ كۈچلۈك).

Var تىپىنى پەقەت يەرلىك ئورۇندىلا ئىشلىتىشكە بولىدۇ. ئۇنى تۈرنىڭ مېتودلارنىڭ پارامېتىرى ياكى قايتۇرما قىممىتىنىڭ تىپى ئورنىدا ئىشلىتىشكە بولمايدۇ.

توغرا ئىشلىتىش ئۇسۇلى:

كود 2

```
public void ValidUse( decimal d ) {
    var x = 2.3;            // double
    var y = x;              // double
    var r = x / y;          // double
    var s = "sample";       // string
    var l = s.Length;       // int
    var w = d;              // decimal
```

```
    var p = default(string); // string
}
```

خاتا ئىشلىتىش ئۇسۇلى:

كود 3

```
class VarDemo {
    // تۈر ياكى ئېغىزلارنىڭ خاسلىقى سۈپىتىدە ئىشلىتىشكە بولمايدۇ
    var k =0;

    // پارامېتىردا ئېنىق تىپى بېكىتىلىشى كېرەك
    public void InvalidUseParameter( var x ){}

    // قايتۇرما قىممەت تىپى ئېنىق بولۇشى كېرەك
    public var InvalidUseResult() {
        return 2;
    }
    public void InvalidUseLocal() {
        var x;          // گراماتىكىلىق خاتالىق، تەڭلىك بەلگىسى بولۇشى كېرەك
        var y = null;   //'null' نىڭ قايسى تىپ ئىكەنلىكىنى بىلەلمەيدۇ
    }
   // …
}
```

كېڭەيتىلمە مېتود

C# بولسا ئوبىكتىپقا يۈزلەنگەن تىل بولۇپ، ئاتا تۈرگە بالا تۈرنى ۋارىسلىق قىلدۇرۇپ يېڭى مېتودلارنى قوشۇش ياكى ئەسلى بار بولغان مېتودلارنى قايتا يېڭىلاش ئارقىلىق ئاتا تۈرنىڭ ئىقتىدارنى ئاشۇرۇش مەقسىتىگە يەتكىلى بولىدۇ. بىراق «بىخەتەرلىك» بىلەن «جانلىق ئىشلىتىش» بۇ خىل مەسىلىلەرنى ھەل قىلىشتىكى ئىككى قارمۇ-قارشى ئامىل بولۇپ كېلىۋاتىدۇ.

C#3.0 دە نۆۋەتتە بار بولغان (.Net نىڭ ئۆزىدە بار بولغانلىرىمۇ شۇ) تۈر (تىپلارمۇ شۇ) لارغا يېڭى تۈرنى ۋارىسلىق قىلدۇرماي تۇرۇپ يېڭى ئىقتىدار قوشۇشقا يول قويۇلغان ۋە مۇناسىۋەتلىك ئەمەلگە ئاشۇرۇش قائىدىلىرى بەلگىلەنگەن. مەسىلەن پۈتۈن سان تىپى int نى مىسالغا ئالساق:

كود 4

```
int i =5;
Int j = i+1;      //j == 6
int t = i-1;      //t == 4
```

بۇ ئۈچ قۇر پروگراما ھەممىمىزگە چۈشۈنۈشلۈك. يۇقىرىقى جۈملىلەرنى تۆۋەندىكى يول ئارقىلىق ئەمەلگە ئاشۇرغىلى بولامدۇ دەپ پەرەز قىلىپ باقايلى:

كود 5

```
int i =5;
Int j = i.Increase();     // j == 6
int t = i.Decrease();     // t == 4
```

بۇنىڭ ئۈچۈن int تىپىنىڭ چوقۇم ئۆز قىممىتىنى بىر ئاشۇرۇپ ياكى بىر تۆۋەنلىتىپ قايتۇرۇپ بېرەلەيدىغان ()Increaseۋە ()Decrease ناملىق (ياكى باشقا ناملىق) مېتودلىرى بولۇشى كېرەك. لېكىن ئەسلى قۇرۇلمىدا intنىڭ بۇ ئىقتىدارلىرى تەمىنلەنمىگەن. بىز دەۋەتقان«كېڭەيتىلگەن مېتود» ئىقتىدارى دەل مۇشۇنىڭدەك مەسىلىلەرگە تاقابىل تۇرۇش مەقسىتىدە قوشۇلغان.

ئەمدى بىز intنى كېڭەيتىپ باقايلى. تۆۋەنتىكى تۈرىمىزگە(project) يېڭىدىن تۈر قوشۇپ ئۇنىڭ ئىسمىنى IntegerExtensionدەپ قويايلى.(خالىغانچە قويسىڭىز بولىدۇ). ئۇنىڭ مەزمۇنى تۆۋەندىكىدەك بولسۇن:

كود 6

```
namespace ConsoleApplication5
{
    static class IntegerExtension
    {
        public static int Increase(this int i)
        {
            return i + 1;
        }
        public static int Decrease(this int i)
        {
            return i - 1;
        }
    }
```

ئەمدى تۆۋەندىكى رەسىمگە قاراڭ:

Program.cs* | IntegerExtension.cs | Start Page

ConsoleApplication5.Program

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Data.OleDb;
5  using System.IO;
6  namespace ConsoleApplication5
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             int i = 5;
13             int j = i.
14
15         }
16
17
18
19
20
```

CompareTo
Decrease
Equals
GetHashCode
GetType
GetTypeCode
Increase
ToString

(extension) int int.Increase()

دېمەك، ()Increaseبىلەن ()Decrease ئەمدى int نىڭ «بولۇپ كەتتى».

يېزىش قائىدىسىنى تېخىمۇ تەپسىلىيرەك چۈشىنىش ئۈچۈن ()Increase مېتودىنى مىسالغا ئالايلى. كېڭەيتىش ئۇسۇلىنىڭ ئورتاق قۇرۇلمىسى تۆۋەندىكىدەك بولىدۇ:

```
static class IntegerExtension
{
    public static int Increase(this int i)
    {
        return i + 1;
    }
}
```

يۇقىرىقى قۇرۇلمىدىكى ئاستىغا قىزىل سىزىق سىزىلغانلىرىنى ئۆز پېتى يېزىشىڭىز كېرەك. يېشىل سىزىق سىزىلغانلىرى بولسا ئېھتىياجىغا قاراپ ئۆزگىرىدۇ. دېمەك:

- كېڭەيتىش مېتودى ۋە ئۇنى ئۆز ئىچىگە ئالغان تىپ چوقۇم تۇراقلىق بولۇشى كېرەك(static).
- كېڭەيتىش مېتودىنىڭ پارامېتىرى ئالدىغا this خاس سۆزىنى قوشۇش ئارقىلىق كېڭەيتىلمەكچى بولغان تىپنى بەلگىلىشىمىز كېرەك. مەسىلەن: يۇقارقى مىسالدا int تىپىنى كېڭەيتىدۇ.
- كېڭەيتىش مېتودىنىڭ قايتۇرما قىممەت تىپى خالىغانچە بولسا بولىدۇ. مەسىلەن:

كود 7

```
        public static float Devide(this int i)
        {
            return (float)i / 2;
        }
```

بۇ مېتود **int** تىپىغا ئۆز قىممىتىنىڭ يېرىمى قايتۇرۇش ئىقتىدارىنى قوشىدۇ. ئەلۋەتتە، پۈتۈن ساننى ئىككىگە بۆلسەك كەسىر سانمۇ چىقىدۇ. شۇڭا قايما قىممەت تىپىنى **float** قىلدۇق.

- بىردانە كېڭەيتىش تۈرى ئىچىگە بىرقانچە تىپنىڭ كېڭەيتىش مېتودلىرىنى ئارلاش يازغىلى بولىدۇ. مەسىلەن:

كود 8

```
    static class MixedExtension
    {
        public static int Increase(this int i)
        {
            return i + 1;
        }
        public static float DoubleIt(this float f)
        {
            return f * f;
        }
    }
```

intتىپىنىڭ كېڭەيتىش مېتودى **Increase()** بىلەن **float** تىپىنىڭ كېڭەيتىش مېتودى **DoubleIt()** بىر تۈر ئىچىگە يېزىلدى. كېيىنكىسى بولسا **float** تىپىغا ئۆز قىممىتىنىڭ كۋادراتىنى قايتۇرۇش ئىقتىدارىنى قوشىدۇ.

- كېڭەيتىش مېتودى كۆپ پارامېتىرنى قوللايدۇ.

ئەمەلىيەتتە يۇقىرىقى مىساللارنىڭ ھەممىسىدىكى بىز قوشقان كېڭەيتىش مېتودلىرىدا ئەمەلىي پاتامېتىرلار يوق. مەسىلەن **int j = i.Increase()** . بۇ قۇردا **Increase()** مېتودى بىر ئىشنى قىلىدۇ ئەمما ئۇنىڭغا پارامېتىر يوللاپ بەرمىدۇق. **Increase()** مېتودى ئەسلى قىممىتىگە بىرنى قوشۇپ قاتۇرۇپ بېرىدۇ. ئەگەر بىز مەزكۇر مېتودتىن پايدىلىنىپ ئەسلى قىممەتكە ئۇچنى قوشقۇزماقچى بولساق، مۇنداق كود يېزىشىمىز مۇمكىن:

كود 9

```
int i = 5;
int j = i.Increase().Increase().Increase();    //j==8
```

ئەگەر ئاشۇرماقچى بولغان قىممەتنى پاتامېتىر ئارقىلىق يوللاپ بەرسەكچۇ؟ ئەلۋەتتە بولىدۇ. بۇنىڭ ئۈچۈن تۆۋەندىكىدەك كود يازىمىز:

كود 10

```
public static int Increase(this int i, int degree)
{
return i + degree;
}
```

بىرىنچى پارامېتىرى «مۇشۇ تىپلىق ئۆزۈم» دېگەن مەنىدە ، ئىككى پارامېتىر بىر دانە پۈتۈن سان تىپلىق (بۇ يەردىكى پاتامېتىر تىپىغا چەك يوق) قىممەت. قايتۇرما قىممەت ئەسلى قىممەت بىلەن ئىككىچى پاتامېتىردا بېرىلگەن قىممەتنىڭ يىغىندىسى. ئەمدى تۆۋەندىكىدەك كود يازالايمىز.

كود 11

```
int i = 5;
int j = i.Increase(3);          //j ==  5+3 == 8
```

يەنە مەسلەن:

كود 12

```
static class MixedExtension
{
    public static string ExtendedTrim(this string s, char c)
    {
        return s.Trim(new char[] { c });
    }
}
```

مۇناسىپ ئىشلىتىش:

كود 13

```
string str1 = "مەن............كىردىم ياشقا";
string str2 = str1.ExtendedTrim('.');    // str2 == "كىردىم مەنياشقا"
```

يىغىنچاقلىغاندا تۈر(ياكى تىپ)لار ئارىسىدىكى مېتود كېڭەيتىشنى تۆۋەندىكى قېلىپلاشتۇرۇش مۇمكىن:

كود 2.25

```
public class A {
    public virtual void X() {}
}
public class B : A {
    public override void X() {}
```

```
    public void Y() {}
}

static public class E {
    static void X( this A a ) {}
    static void Y( this A b ) {}

    public static void Demo() {
        A a = new A();
        B b = new B();
        A c = new B();

        a.X(); // A.X نى چاقىرش
        b.X(); // B.X نى چاقىرش
        c.X(); // B.X نى چاقىرش

        a.Y(); // E.Y نى چاقىرش
        b.Y(); // B.Y نى چاقىرش
        c.Y(); // E.Y نى چاقىرش
    }
}
```

ئوبيېكتلارنى دەسلەپلەشتۈرۈش ئىپادىسى

C#1.x دە خاسلىقلارنى ياكى يەرلىك ئۆزگەرگۈچى مىقدارلارنى بىر جۈملە ئارقىلىقلا دەسلەپلەشتۈرۈشكە بولىدۇ.

```
int i = 3;
string name = 'Unknown';
Customer c = new Customer( "Tom", 32 );
```

بۇنداق دەسلەپلەشتۈرۈش ئۇسۇلىغا چاقىرىلما تىپقا(引用类型) قوللىنىلغاندا ئالدىن بەلگىلەنگەن ماس ھالدىكى قۇرغۇچى مېتودلىرىنى ئىجرا قىلىدۇ. يۇقىرىقى ئۈچىنچى قۇر ئۇنۇملۇك بولۇشى ئۈچۈن چوقۇم **Customer** تۈرى تۆۋەندىكىدەك يېزىلغان بولۇشى كېرەك.

```
public class Customer {
    public int Age;
    public string Name;
    public string Country;
```

```
    public Customer( string name, int age ) {
        this.Name = name;
        this.Age = age;
    }
    // …
}
```

دېمەك ئۇ (new Customer("Tom", 32 دەك ئىشلىتىلىشكە مۇناسىپ قۇرغۇچى مېتودىنى تەييارلىشى كېرەك. بىراق بۇ قۇرغۇچىنىڭ بىر ئاجىزلىقى بار يەنى: قۇرغان پەيتتە **name** بىلەن **age**نى چوقۇم يوللاپ بېرىشىمىز كېرەك. ئەگەر **Country** بىلەن **name** لا بولغان، **age** ئى قۇرۇق بولغىنىنى قۇرۇش ئۈچۈن ياكى تۈۋەندىكىدەك كود يېزىش ، ياكى مۇناسىپ قۇرغۇچى مېتودىنى تەمىنلەش كېرەك.

كود 2.27

```
Customer customer = new Customer();
customer.Name = "Marco";
customer.Country = "Italy";
```

C#3.0 دە يۈقىرىقىدەك مەشغۇلاتلارنى تېخىمۇ ئاددىي(قىسقا) جۈملىلەر ئارقىلىق ئەمەلگە ئاشۇرغىلى بولىدۇ. مەسىلەن:

كود 2.28

```
// ئوبيېكتنى دەسلەپلەشتۈرۈشتىن ئاۋال تۈرنىڭ كۆڭۈلدىكى قۇرغۇچى مېتودىنى يوشۇرۇن ھالدا چاقىرىدۇ
Customer customer = new Customer { Name = "Marco", Country = "Italy"
};
// ئوبيېكتنى دەسلەپلەشتۈرۈشتىن ئاۋال تۈرنىڭ كۆڭۈلدىكى قۇرغۇچى مېتودىنى ئاشكارە ھالدا چاقىرىدۇ
Customer customer = new Customer() { Name = "Marco", Country =
"Italy" };
```

لېكىن ئالدىنقى شەرت شۇكى تىرناق ئىچىدىكى خاسلىقلار چوقۇم **public** بولۇشى كېرەك.
ئەگەر تۈر مەلۇم پاتامېتىرلىق قۇرغۇچى مېتودلارنى تەمىنلىگەن بولسا، يۇقىرىقى جۈملىلەرنى قۇرغۇچى مېتودى ئاساسىدا يازغىلى بولىدۇ. مەسىلەن:

كود 2.29

```
// كۆڭۈلدىكى ئەمەس قۇرغۇچى مېتودىنى ئاشكارە ھالدا چاقىرىدۇ
Customer c2 = new Customer( "Paolo", 21 ) { Country = "Italy" };
```

c2نىڭ يۇقىرىقى ئىنىقلىمىسى ئەمەلىيەتتە تۆۋەنكى ئىككى جۈملە بىلەن باراۋەر

```
Customer c2 = new Customer( "Paolo", 21 );
c2.Country = "Italy";
```

بۇ قائىدە تۆۋەندىكىدەك مەسىلىلەردە تېخىمۇ كۆچىنى كۆرسىتىدۇ. مەسىلەن: تۆۋەندىكىدەك مۇناسىۋەتلىك ئىككى تۈر بار

كود 2.30

```
public class Point {
    int x, y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}

public class Rectangle {
    Point tl, br;
    public Point TL { get { return tl; } set { tl = value; } }
    public Point BR { get { return br; } set { br = value; } }
}
```

C#2.0 ۋە بۇرۇنقى نەشرلەردە **Rectangle** تۈردىن بىر ئېنىقلاش ۋە قىممەتلىرىنى تولدۇرۇش ئۈچۈن تۆۋەندىكىدەك جەريان كېتەتتى(ئەلۋەتتە، بۇمۇ بىر خىل يولى)

```
Rectangle rectangle2 = new Rectangle();
Point point1 = new Point();
point1.X = 0;
point1.Y = 1;
rectangle2.TL = point1;
Point point2 = new Point();
point2.X = 2;
point2.Y = 3;
rectangle2.BR = point2;
Rectangle rectangle1 = rectangle2;
```

ئەگەر C#3.0 نىڭ قائىدىسىنى قوللانساق، تۆۋەندىكى بىر-قانچە قۇر ئارقىلىقلا مەقسەتكە يېتەلەيمىز

كود 2.31

```
Rectangle r = new Rectangle {
    TL = new Point { X = 0, Y = 1 },
    BR = new Point { X = 2, Y = 3 }
};
```

ياكى

```
Rectangle r = new Rectangle {
    TL = { X = 0, Y = 1 },
```

```
    BR = { X = 2, Y = 3 }
};
```

ئەگەر قۇرماقچى بولغىنىڭىز مەلۇم تۈرنىڭ تىزىملىكى بولسا يۇقىرىقى قائىدە يەنىلا كۈچكە ئىگە:

كود 2.32

```
List<int> integers = new List<int> { 1, 3, 9, 18 };

List<Customer> list = new List<Customer> {
    new Customer( "Jack", 28 ) { Country = "USA"},
    new Customer { Name = "Paolo" },
    new Customer { Name = "Marco", Country = "Italy" },
};

ArrayList integers = new ArrayList() { 1, 3, 9, 18 };
ArrayList list = new ArrayList {
    new Customer( "Jack", 28 ) { Country = "USA"},
    new Customer { Name = "Paolo" },
    new Customer { Name = "Marco", Country = "Italy" },
};
```

يىغىنچاقلىغاندا يېڭىدىن تەمىنلەنگەن ئۇسۇللا ئوبيېكىتلارنى قۇرۇش ۋە دەسلەپلەشتۈرۈش مەشغۇلاتلىرىنى ئاددى بىر ياكى بىرقانچە فۇنكسىيە ئىچىگىلا مۇجەسسەملەشتۈرۈپ. شۇ ئارقىلىق كودىمىزنىڭ ئوقۇشچانلىقىنى زور دەرىجىدە يۇقىرى كۆتۈرگەن.

نامسىز تىپ

ئوبيېكتلارنى قۇرۇش ۋە دەسلەپلەشتۈرۈش ئۈچۈن ئەمدى مەزكۇر ئوبيېكتنىڭ قايسى تىپلىق ئىكەنلىكىنى بىلىشنىڭ ھاجىتى قالمىدى(زۆرۈر تېپىلسا). بۇ ئارقىلىق قۇرۇلغان ئوبيېكتنىڭ تىپى «نامسىز تىپ» دەپ ئاتىلىدۇ. مەسىلەن:

كود 2.33

```
Customer c1 = new Customer { Name = "Marco",  Age=34 };
var c2 = new Customer { Name = "Paolo",  Age=30 };
var c3 = new { Name = "Tom", Age = 31 };
var c4 = new { c2.Name, c2.Age };
var c5 = new { c1.Name, c1.Country };
var c6 = new { c1.Country, c1.Name };
```

بۇ يەردىكى c1 بىلەن c2 ئوبيېكتلارنىڭ تىپى بولسا «ناملىق تىپ» يەنى ئۇلارنىڭ تىپى Customer. بۇ ئىككىسىنىڭ ئارىسىدىكى c2نىڭ نىمىشقا Customer تىپ بولۇپ قالغانلىقىدىكى سەۋەب كود-تەرجىمانىنىڭ(编译器) جۈملىنىڭ باش-ئاخىرىغا ئاساسەن ئاپتوماتىك كەلتۈرۈپ چىقارغانلىقىدا. c3, c4, c5, c6 ئوبيېكتلارنىڭ تىپى بولسا «نامسىز تىپ» يەنى ئۇلارنىڭ تىپ يوق. (Customer مىكىن دەپ ئويلاپ قالماڭ). سەۋەبى ئۇلار ئورۇنلاشقان قۇرلاردا ئۇلارنىڭ Customer ئىكەنلىكىنى بىلگىلى بولىدىغان يېتەرلىك ئاساس يوق.

ئەمەسە بۇ نامسىز تىپلىق ئوبيېكتلارنىڭ خاسلىقلىرىنىڭ ئىسىملىرى قايسى؟ ئۇلارنىڭ قىممەتلىرىچۇ؟ ئۇلارنى قانداق زىيارەت قىلىمىز؟

ئوبيېكت	خاسلىقلىرى	خاسلىق نامىنىڭ	خاسلىق قىممەتلىرى	خاسلىق قىممىتىنىڭ
c3	Name بىلەن Age	ئۆزىمىز بەرگەن	"Tom" بىلەن 31	ئۆزىمىز بەرگەن
c4	Name بىلەن Age	c2 دىن تەقلىدلىۋالدى	c2نىڭ مۇناسىپ قىممەتلىرى بىلەن ئوخشاش	c2
c5	Name بىلەن Contry	c1دىن تەقلىدلىۋالدى	c1نىڭ مۇناسىپ قىممەتلىرى بىلەن ئوخشاش	c1
c6	Contry بىلەن Name	c1دىن تەقلىدلىۋالدى	c1نىڭ مۇناسىپ قىممەتلىرى بىلەن ئوخشاش	c1

ئەگەر تۆۋەندىكى پرگراما ئارقىلىق بۇ ئوبيېكتلارنىڭ تىپىنى كۆرسەتسەك نەتىجە ئاستىدىكى رەسىمدە كۆرسىتىلگەندەك بولىدۇ.

```
Console.WriteLine( "c1 is {0}", c1.GetType() );
Console.WriteLine( "c2 is {0}", c2.GetType() );
Console.WriteLine( "c3 is {0}", c3.GetType() );
Console.WriteLine( "c4 is {0}", c4.GetType() );
Console.WriteLine( "c5 is {0}", c5.GetType() );
Console.WriteLine( "c6 is {0}", c6.GetType() );
```

```
file:///F:/projects/ConsoleApplication5/ConsoleApplication5/bin/Debug/Conso...
c1 is ConsoleApplication5.Customer
c2 is ConsoleApplication5.Customer
c3 is <>f__AnonymousType0`2[System.String,System.Int32]
c4 is <>f__AnonymousType0`2[System.String,System.Int32]
c5 is <>f__AnonymousType1`2[System.String,System.String]
c6 is <>f__AnonymousType2`2[System.String,System.String]
```

دېمەك مۇنداق يەكۈنگە ئېرىشەلەيمىز:

بۇلارنىڭ ئارىسىدىكى c1 بىلەن c2 بىردەك Customer تىپلىق؛ c2 بىلەن c3 ھەر ئىككىلىسى بىر نامسىز تىپنىڭ ئوبيېكتلىرى. چۈنكى ئۇلار ئۇچۇن بېرىلگەن خاسلىق ناملىرى ئوپمۇ-ئوخشاش، شۇڭا ئايرىم تىپ قۇرۇشنىڭ ھاجىتى يوق.؛ گەرچە c5 بىلەن c6 نىڭ خاسلىق ناملىرىنى ئوخشاش بولسىمۇ پاتامېتىردىكى تەرتىپى ئوخشىمايدۇ، شۇڭا ئۇلار ئۇچۇن ئايرىم تىپ قۇرۇلغان.

«نامسىز تىپ»ئوبيېكتلىرنىڭ خاسلىقلىرىنى obj.property شەكلى ئارقىلىق زىيارەت قىلالايسىز، يەنى: string str = c4.Name;

Query ئىپادىسى (سۈرۈشتۈرۈك ئىپادىسى)

C#3.0 تەمىنلىگەن يەنە بىر مۇھىم ئىقتىدار شۇكى، گىراماتىكا جەھەتتە Sql جۈملىسىگە ئوخشىشىپ كېتىدىغان جۈملىلەر ئارقىلىق سان- سىفىر مەشغۇلاتى ئېلىپ بېرىش. بۇ گىراماتىكا ئارقىلىق يېزىلغان جۈملىلەر Linq تەمىنلىگەن مۇناسىۋەتلىك تۈر، ئېغىز ۋە مېزوتلار ئارقىلىق ئەڭ ئەڭ ئاخىرىدا C#3.0 نىڭ ئۆزلەشتۈرۈش جۈملىلىرىگە ئايلاندۇرۇلىدۇ.

Query ئىپادىسى توغۇرلىق ھازىر كۆپ توختىلىمايدۇ. بۇ ئىپادىگە ئائىت كۆپرەك مەزمۇنلار Linq غا ئالاقىدار مەزمۇنلاردا سۆزلىنىدۇ. ھازىر پەقەت تىپىك بولغان ئىشلىتىلىشى ۋە ئۇنىڭ ئەڭ ئاخىرىدا قانداق قىلىپ Linq تىپ مېتودلىرىغا ئايلىنىدىغانلىقى ئاددى چۈشەندۈرۈلىدۇ. تۆۋەندىكى كودنى Query ئىپادىسىنىڭ تىپىك مىسالى دېيىشكەن بولىدۇ:

```
var customers = new [] {
    new { Name = "Marco", Discount = 4.5 },
    new { Name = "Paolo", Discount = 3.0 },
    new { Name = "Tom", Discount = 3.5 }
};

 var query =
    from c in customers
    where c.Discount > 3
    orderby c.Discount
    select new { c.Name, Perc = c.Discount / 100 };

foreach( var x in query ) {
    Console.WriteLine( x );
}
```

ھەربىر سۈرۈشتۈرۈك ئىپادىسى from خاس سۆزىدىن باشلىنىپ(چوڭ ـ كىچىك ھەرپكە سەزگۈر) يا select يا group خاس سۆزى بىلەن ئاخىرلىشىدۇ. from خاس سۆزى Linq مەشغۇلاتى ئېلىپ بېرىلماقچى بولغان ھەمدە IEnumerable<T> ئېغىزىنى ئەمەلگە ئاشۇرغان ئوبيېكتنى بەلگىلەيدۇ. بۇ يەردىكى IEnumerable<T> ئېغىزىنى ئەمەلگە ئاشۇرۇش دېگەننى زاغرا تىل ئېيتقاندا ئەزالىرىنى بىر-بىرلەپ ئوقۇغىلى بولىدۇ، يەنى foreach نى ئىشلىتىشكە بولىدۇ دېگەنلىك.

يۇقىرىقى كود تۆۋەندىكى تۈپىنى بارلىققا كەلتۈرىدۇ

```
{ Name = Tom, Perc = 0.035 }
{ Name = Marco, Perc = 0.045 }
```

3.0 C# يۇقىرىقى Query ئىپادىسىنى كود-تەرجىمە مەزگىلىدە تۆۋەندىكىدەك يېزىلغان باراۋەر ھالەتتە ئالماشتۇرىدۇ:

```
var query = customers
            .Where( c => c.Discount > 3)
            .OrderBy( c => c.Discount )
            .Select( c=> new { c.Name, Perc = c.Discount / 100 } );
```

ھەربىر ئىپادە خاس سۆزى(مەسىلەن: select) مەلۇم كۆپماس مېتودقا(generic method) باراۋەر. بۇلاردىن شۇنى ھېس قىلالايمىزكى، بىز يۇقىرىدا سۆزلىگەن بارلىق يېڭى قائىدىلەر ئەڭ ئاخىرىدا Linq غا بېرىپ تاقىشىدۇ. يەنى var بولسا query نەتىجىسىنى ئېنىقلاشقا، نامسىز تىپ بولسا query نەتىجىسىنى ساقلاشقا. Select، from مەشغۇلاتچىلىرى بولسا مۇناسىپ Linq مىزوتلىرىنىڭ ئورنىغا ئىشلىتىلىدۇ.

تۆتىنچى باب Linq گىراماتىكىسىدىن ئاساس

زامانىۋى پروگراممىرلاش تىلى ۋە يۇمشاق دېتاللار ئاساسەن دېگۈدەك ئوبيېكىتقا يۈزلەنگەن قۇرۇلما تەرەپدارى بولۇۋاتىدۇ. نەتىجىدە بىزنىڭ كۆپ قىسىم مەشغۇلاتلىرىمىز جەدۋەل ۋە رېكورتلار بىلەن ئەمەس بەلكى ئوبيېكىت تىپلىرى ۋە ئۇلارنىڭ ئەزالىرى بىلەن ھەپىلىشىش بولىۋاتىدۇ. شۇ سەۋەبلىك پروگراممىرلار ئالدىدا بار مۇقىم سان مەنبەلىرىنى(مەسىلەن: ساندان) پروگراممىرلار ئايرىش يوللىرى ئۈستىدە ئىزدىنىۋاتىدۇ. تىلغا ئورنىتىلغان سۈرۈشتۈرۈك (Language Integrated Query) يەنى ئاتالمىش Linq پروگراممىرلارنى تارماقچىلار تىزمىسى (يەنى ئوبيېكىتلار objects، گەۋدىلەر entities، ساندان رېكورتلىرى database records، XML نۇختىلىرى قاتارلىقلار) غا نىسبەتەن ئۈنۈملۈك مەشغۇلات قىلىش چارىسى بىلەن تەمىنلەيدۇ. ئۇنىڭ ئەڭ ئۈتۈقلۇق يېرى شۇكى ئۇ تىزمىلارغا (ساندانمۇ شۇ) لارغا بولغان مەشغۇلاتىڭىزنى پروگرامما تىلى بىلەن يۈكسەك دەرىجىدە يېقىنلاشتۇرغان بولۇپ پروگرامما تىلىنىڭ ئاددى قائىدىلىرى ئارقىلىق مۇرەككەپ مەشغۇلاتلارنى(مەسلەن ساندان مەشغۇلاتلىرى) ئېلىپ بارالايسىز.

LINQ سۈرۈشتۈرۈكلىرى (LINQ Queries)

LINQ بىر قىسىم سۈرۈشتۈرۈك ئەمەللىرىنى ئاساس قىلىدىغان بولۇپ، ئاساسلىقى IEnumerable<T> ئېغىزىنى ئەمەلگە ئاشۇرغان ھەرقانداق تىپقا مەشغۇلاپ ئېلىپ بارالايدۇ. ئۇنىڭ ئۈستىگە Net.قۇرۇلمىسىدىكى مۆتلەق كۆپ قىسىم تىپلار تىپلىرى مەزكۇر ئېغىزنى ئەمەلگە ئاشۇرغان. ھەتتا ئۇنى ئۆزىڭىزنىڭ تىپلىرىدا ئەمەلگە ئاشۇرماقمۇ قىيىن ئەمەس. دېمەك Linq پروگراممىڭىزنىڭ كۆپ قىسىم بۆلەكلىرىدە نامايەن بولالايدۇ.

Linq بەلگىلىك كېڭەيتىشچانلىققا ئىگە بولۇپ، ئوخشىمىغان تۈردىكى سان مەشغۇلاتلىرىغا خاسلاشتۇرغىلى بولىدۇ. مەسىلەن Linq To Sql ئارقىلىق MsSQL ساندانىغا ، Linq To XML ئارلىق XML ئۈچۈرلىرىغا ئۈنۈملۈك مەشغۇلات ئېلىپ بېرىلىدۇ.

سۈرۈشتۈرۈك گىراماتىكىسى

گىراماتىكىنى سۆزلەشتىن بۇرۇن گەپنى ئاددى مىسال ئارقىلىق باشلاي. تۆۋەندىكەن تۆر بار دەپ پەرز قىلايلى

```
public class Developer {     //پروگراممېر تۈرى
    public string Name;
    public string Language;
    public int Age;
}
```

يۇقارقى تۆر تېپىنىك توپلىمىغا مەشغۇلات ئېلىپ بارماقچى بولۇق. **Linq To Objects** ئارقىلىق مەزكۇر توپتىكى #C تىلىنى ئىشلىتىدىغان پروگراممېرلارنىڭ نامىنى بېسىپ چىقىرىش ئۈچۈن تۆۋەندىكىدەك كود يېزىلىشى مۇمكىن

كود 4.1

```
using System;
using System.Linq;
using System.Collections.Generic;

class app {
    static void Main() {
        Developer[] developers = new Developer[] {
            new Developer {Name = "Paolo", Language = "C#"},
            new Developer {Name = "Marco", Language = "C#"},
            new Developer {Name = "Frank", Language = "VB.NET"}};

        IEnumerable<string> developersUsingCsharp =
            from   d in developers
            where  d.Language == "C#"
            select d.Name;

        foreach (string item in developersUsingCsharp) {
            Console.WriteLine(item);
        }
    }
}
```

مەزكۇر كود سۇزۇپ چىققان پروگراممېرلار **Paolo** بىلەن **Marco**.

يۇقىرىقى سۈرۈشتۈرۈك جۈملىلىرى قارىماققا **Sql** جۈملىسىگە ئىنتايىن ئوخشىشىپ كېتىدۇ. بىراق ئەمەس. ئۇنىڭغا مۇنداق ئېنىقلىما بېرىلگەن:

سۈرۈشتۈرۈك ئىپادىسى بولسا بىر خىل دەرەخسىمان ئىپادە بولۇپ بىر ياكى بىرقانچە ئۇچۇر مەنبەسىگە بىر ياكى بىرقانچە سۈرۈشتۈرۈك ئەمىلى ئارقىلىق مەشغۇلات ئېلىپ بارىدۇ.

ئادەتتە سۈرۈشتۈرۈك نەتىجىسى بىر توپلام قىممەتلەر تىزمىسىدىن ئىبارەت بولۇپ قىممەتلەرگە چېقىلىش يۈز بەرگەندىلا ئاندىن سۈرۈشتۈرۈك ھەقىقى ئىجرا بولىدۇ.

سۈرۈشتۈرۈك ئىپادىسى **select** يەنى تاللاش كوماندىسى ئارقىلىق بېكىتىلىپ **(select d.Name)** **from** كوماندىسى ئارقىلىق توپلامغا ئەمەللەشتۈرۈلىدۇ **(from d as developers)**. بۇ تاللاش مەشغۇلاتىنى **where** غا يانداشقان سۈزۈش شەرتىگە بوي سۇنىدۇ. يەنى

```
where d.Language == "C#"
```

ئەمەلىيەتتە كود ـ تەرجىمە قىلىش مەزگىلىدە where ئىپادىسى System.Linq نام بوشلۇقىدا ئېنىقلانغان Enumerable تۈرىنىڭ كېڭەيتىلمە مېتودى Where غا ئۆزلەشتۈرۈلىدۇ.
System.Linq نام بوشلۇقىدا IEnumerable<T> ئۈچۈن ئورۇنلانغان كېڭەيتمە مېتودلار تەمىنلەنگەن.
دېمەك يۇقاردىكى سۈرۈشتۈرۈك ئىپادىسى ماھىيەتتە تۆۋەندىكى ئاساسى مېتود قوللىنىشنىڭ باشقىچە ئىپادىلىنىشى خالاس.

```
IEnumerable<string> expr =
    developers
    .Where(d => d.Language == "C#")
    .Select(d => d.Name);
```

بۇنىڭدىكى Where مېتودى بىلەن Select مېتودىنىڭ ھەر ئىككىلىسى Lambda ئىپادىسىنى ئۆزگە پارامېتر قىلىدۇ [مەسىلەن: (d => d.Language == "C#")]. بۇ Lambda ئىپادىلىرى ئاخىرىدا System.Linq نام بوشلۇقىدا ئالدىن بەلگىلەنگەن كۆپماس مۇۋەققەت تىپلىرىغا تەرجىمە قىلىنىدۇ. تۆۋەندىكىلىرى بىكىتىلگەن كۆپماس مۇۋەققەت تىپلىرىنىڭ تولۇق تىزىملىكى:

```
public delegate T Func< T >();
public delegate T Func< A0, T >( A0 arg0 );
public delegate T Func< A0, A1, T > ( A0 arg0, A1 arg1 );
public delegate T Func< A0, A1, A2, T >( A0 arg0, A1 arg1, A2 arg2 );
public delegate T Func< A0, A1, A3, T > ( A0 arg0, A1 arg1, A2 arg2, A3
    arg3 );
```

Enumerable تۈرىنىڭ كۆپ قىسىم كېڭەيتمە مېتودلىرى يۇقىرىقى مۇۋەققەت تىپلارنى پارامېتىر سۈپىتىدە قوبۇل قىلالايدۇ. مەسىلەن: تۆۋەندىكى كوددىكىدەك قوللىنىش يۇقىرىقى بارلىق ئىپادىلەرنىڭ ئەڭ ئاخىرقى ئۆزلەشتۈرۈلمىسىدۇر.

كود 4.2

```
Func<Developer, bool> filteringPredicate = d => d.Language == "C#";
Func<Developer, string> selectionPredicate = d => d.Name;
IEnumerable<string> expr =
    developers
    .Where(filteringPredicate)
    .Select(selectionPredicate);
```

C#3.0 كود ـ تەرجىمانى يۇقىرىقى ھالەتكە قانداق كەلتۈرۈشنى بىلىدۇ. ئەلۋەتتە ئەگەر سىز Linq بىلىملىرىنى پىششىق بىلىپ بولغاندىن كېيىن بىۋاستە مۇشۇ خىل گىراماتىكىنى قوللانسىڭىزمۇ بولىدۇ. لېكىن تەشەببۇسۇم شۇكى زۆرۈرىيەت تۇغۇلمىسىلا سۈرۈشتۈرۈك ئىپادىسىنى ئىشلىتىڭ.

تولۇق سۈرۈشتۈرۈك ئىپادىسى

ئالدىنقى مەزمۇنلاردا سۈرۈشتۈرۈك ئىپادىسىنىڭ ئوبيېكتلار توپلىمى ئۈستىدىكى ئاددىي مەشغۇلاتلىرىنى كۆرۈپ ئۆتتۇق. تولۇق بولغان سۈرۈشتۈرۈك ئىپادىسى تېخىمۇ مۇكەممەل گراماتىكىلىق قۇرۇلمىغا ئىگە. ھەربىر ئىپادە from دىن باشلىنىپ يا select يا group دىن ئاخىرلىشىدۇ. Sql جۈملىسىگە ئوخشاش select دىن باشلىنىپ from دىن ئاخىرلاشماسلىقىدىكى سەۋەب كود يازغاندا مىكروسوفتنىڭ «ئەقلىي تەۋسىيە»ئىقتىدارى بىلەن تەمىنلەشكە قولايلىق يارىتىش ئۈچۈندۇر. select خاس سۆزى ئىپادە نەتىجىسىنى enumerable ئوبيېكتىغا ئورۇنلاشتۇرىدۇ. group خاس سۆزى بولسا ئىپادە نەتىجىسىنى گۇرۇپپىلاش شەرتىگە ئاساسەن ھەربىر گۇرۇپپا enumerable بولغان گۇرۇپپىلار توپلىمىغا بۆلىدۇ.
تۆۋەندىكىسى سۈرۈشتۈرۈك ئىپادىسىنىڭ تولۇق بولغان ئەندىزە(قېلىپ) كودى:

```
query-expression ::= from-clause query-body

query-body ::=
join-clause*
(from-clause join-clause* | let-clause | where-clause)*
orderby-clause?
(select-clause | groupby-clause)
    query-continuation?

from-clause ::= from itemName in srcExpr

select-clause ::= select selExpr

groupby-clause ::= group selExpr by keyExpr
```

تۇنجى from خاس سۆزى كەينىگە نۆل ياكى نەچچە from، let، ياكى where خاس سۆزلىرى ئەگىشىپ كېلەلەيدۇ. let مەشغۇلاتچىسى ئىپادە نەتىجىسىگە نام بېرەلەيدۇ. from مەشغۇلاتچىسىگە كۆپ دانە join مەشغۇلاتچىسى ئەگىشەلەيدۇ. ئەڭ ئاخىرقى select مەشغۇلاتچىسىنىڭ ئاخىرىغا orderby خاس سۆزنى ئەگەشتۈرۈش ئارقىلىق نەتىجىنى مەلۇم خاسلىقىغا ئاساسەن سورتلىغىلى بولىدۇ.

بۇندىن كېيىنكى ئۇقۇملارنى چۈشەندۈرۈشكە قولايلىق بولۇش ئۈچۈن مىساللارغا تولۇقراق بولغان تۈر قۇرۇلمىسى تۈزۈۋالايلى. بىز بۇندىن كېيىن دائىم خېرىدارلار توپلىمىغا مەشغۇلات ئېلىپ بارىمىز (class Customer) . ھەربىر خېرىدارنىڭ بۇيرۇتقان ماللىرى بار. ئۇلارنى ئىپادىلەش كودى تۆۋەندىكىچە:

مىساللار ئۈچۈن تۈر قۇرۇلمىسى

```
public enum Countries {
    USA,
    Italy,
}

public class Customer {
    public string Name;
    public string City;
    public Countries Country;
    public Order[] Orders;
}

public class Order {
    public int Quantity;
    public bool Shipped;
    public string Month;
    public int IdProduct;
}

public class Product {
    public int IdProduct;
    public decimal Price;
}

// -------------------------------------------------------------------
// خېرىدارلار توپلىمىنى دەسلەپلەشتۈرۈش
// -------------------------------------------------------------------
customers = new Customer[] {
  new Customer {Name = "Paolo", City = "Brescia", Country =
 Countries.Italy, Orders =
  new Order[] {
    new Order {Quantity = 3, IdProduct = 1 , Shipped = false, Month =
 "January"},
    new Order {Quantity = 5, IdProduct = 2 , Shipped = true, Month =
 "May"}}},
  new Customer {Name = "Marco", City = "Torino", Country = Countries.Italy,
 Orders =
```

```
  new Order[] {
    new Order {Quantity = 10, IdProduct = 1 , Shipped = false, Month =
"July"},
    new Order {Quantity = 20, IdProduct = 3 , Shipped = true, Month =
"December"}}},
  new Customer {Name = "James", City = "Dallas", Country = Countries.USA,
Orders =
  new Order[] {
    new Order {Quantity = 20, IdProduct = 3 , Shipped = true, Month =
"December"}}},
  new Customer {Name = "Frank", City = "Seattle", Country = Countries.USA,
Orders =
  new Order[] {
    new Order {Quantity = 20, IdProduct = 5 , Shipped = false, Month =
"July"}}}};

products = new Product[] {
    new Product {IdProduct = 1, Price = 10 },
    new Product {IdProduct = 2, Price = 20 },
    new Product {IdProduct = 3, Price = 30 },
    new Product {IdProduct = 4, Price = 40 },
    new Product {IdProduct = 5, Price = 50 },
    new Product {IdProduct = 6, Price = 60 }};
```

سۈرۈشتۈرۈك مەشغۇلاتچىلىرى

ھازىردىن باشلاپ *System.Linq* نام بولشلۇقىدا تەمىنلەنگەن ئاساسلىق مېزودلار ۋە كۆپماس مۇۋەققەت تىپلىرىنى چۈشەندۈرۈش ئارقىلىق Linq دىن پايدىلىنىپ سۈرۈشتۈرۈك ئېلىپ بېرىش تونۇشتۇرۇلىدۇ.

Where مەشغۇلاتچىسى

(يۇقىرىقى مىسال كودى ئاساسىدا) سىزگە دۆلەت تەۋەلىكى Italy(ئىتالىيە) بولغان خېرىدارلارنىڭ نام ۋە شەھەر تىزىملىكى لازىم. بۇنىڭ ئۈچۈن بارلىق خېرىدارلارنى چارلاش جەريانىدا where خاس سۆزى ئارقىلىق دۆلەت تەۋەلىك چەكلىمىسىنى بەرسىڭىزلا كۇپايە(بۇرۇن سۈرۈشتۈرۈك ئىپادىسىدىكى خاس سۆزلەرگە مۇناسىپ مەشغۇلاتچىلارنىڭ بارلىقىنى ئەسكەرتكەن، بۇ يەردىكى Where مەشغۇلاتچىسى where خاس سۆزىگە مۇناسىپ كېلىدۇ) . يەنى:

كود 4.3

```
var expr =
    from   c in customers
    where  c.Country == Countries.Italy
    select new { c.Name, c.City };
```

تۆۋەندىكىسى Where مەشغۇلاتچىسىنىڭ ئەندىزىلىرى:

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

قىلىپ كودىدىن شۇنى كۆرۈۋېلىشقا بولىدۇ، Where مەشغۇلاتچىسىنىڭ ئەمەلىيەتتە ئىككى خىل ئەندىزىسى بار بولۇپ، بىز يۇقىرىدا ئىشلەتكىنى بىرىنچى خىلى. يەنە بىر خىلىدا بولسا پۈتۈن سان تىپلىق پارامېتىر بىكىتەلەيمىز. ئۇ نۆۋەتتىكى ئوبيېكتنىڭ رەت تەرتىۋىنى بىلدۈرىدۇ. مەسىلەن:

كود 4.4

```
var expr =
    customers
    .Where((c, index) => (c.Country == Countries.Italy && index >= 1))
    .Select(c => c.Name);
```

بۇ ئىپادە تەرتىپ نومۇرى 0 بولغان خېرىدار (يەنى Paolo ئىسىملىك خېرىدار) index>=1 شەرتىنى قانائەتلەندۇرمىگەچكە نەتىجە توپلىمىغا كىرەلمەيدۇ.
ئەپىسۇسلىنارلىق يېرى شۇكى، ئىككىنچى خىل ئەندىزىنى سۈرۈشتۈرۈش ئىپادىسىدە ئىپادىلىگىلى بولمايدۇ (from.... where.... select... ئارقىلىق دېمەكچى).
شۇنىسى ئېسىڭىزدە تۇرسۇنكى، نورمال ھالەتتە تۇرغۇن قەۋەت (persistence layer)، مەسىلەن: ساندان) دىن كۆپ مىقداردىكى ئۇچۇرنى ئىچكى ساقلىغۇچقا ئوقۇپ كىرىشنى ياخشى مەشغۇلات دېگىلى بولمايدۇ. ئادەتتە، ئۇچۇرلارنى تۇرغۇن قەۋەت دەرىجىسىدىلا بەتلەرگە بۆلگەن(شەرتلەر ئارقىلىق) ياخشى. گەرچە بۇ خىل ئۇسۇل بارلىق ئۇچۇرنى ئىچكى ساقلىغۇچتا تۇرۇپ بەتلەرگە بۆلگەنگە قارىغان ئاستىراق(نىسپى) بولسىمۇ، يەنىلا چوقۇم كىرىش جەريانىغا كەتكەن ۋاقىتتىن ئۇتالايمىز.

ئەمەللەشتۈرۈش مەشغۇلاتچىلىرى (Projection Operators)

Select مەشغۇلاتچىسى

ئەمەللەشتۈرۈش مەشغۇلاتچىلىرى مەنبەدىن ئۇچۇرلارنى شەرتكە ئاساسەن يىغىپ نەتىجىگە ئورۇنلاشتۇرۇش رولىنى ئوينايدۇ. يۇقىرىدا كۆپ ئۇچراتقان Select مەشغۇلاتچىسى ئۇنىڭ تىپىك مىسالى. بۇنداق دېيىشىمىزدىكى سەۋەب ئۇ سۈرۈشتۈرۈك نەتىجىسىنى IEnumerable<T> ئېغىزنى ئەمەلگە ئاشۇرغان ئوبيېكت ھالىتىدە قايتۇرۇپ بېرەلەيدۇ.

Select مەشغۇلاتچىنىڭ ئەندىزىسى تۆۋەندىكىچىدەك:

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, int, S> selector);
```

خۇددى Where مەشغۇلاتچىسىغا ئوخشا Select مەشخۇلاچىسمۇ مەنبە تىزمىنى(توپلام ئوبيېكت) چارلايدۇ ۋە شەرتكە ئۇيغۇنلىرىنى نەتىجە تىزمىسىغا(نەتىجە توپلام ئوبيېكتى) ئورۇنلاشتۇرىدۇ.

مەسىلەن تۆۋەندىكى جۈملىگە قاراڭ:

```
var expr = customers.Select(c => c.Name);
```

مەزكۇر جۈملىنىڭ نەتىجىسى خېرىدارلارنىڭ ئىسمىنىڭ تىزمىسى بولىدۇ (*IEnumerable<string>*).

ئەمدى تۆۋەندىكىسىنى كۆرۈپ باقايلى:

```
var expr = customers.Select(c => new { c.Name, c.City });
```

بۇ جۈملىدىن قايتىدىغىنى Name ۋە City دىن ئىبارەت ئىككى دانە خاسلىقى بولغان نامسىز تىپلىق ئوبيېكتلار تىزمىسى بولۇپ، ئۇ خاسلىقلىرىنىڭ قىممىتى مەنبە خېرىدارلار تىزمىسىدىكى (customers) يەككە خېرىدار ئوبيېكتىدىن كېلىدۇ.

Select مەشغۇلاتچىسىنىڭ ئىككىنچى خىل ئەندىزىسىدە پۈتۈن سان تىپلىق پارامېترى ئىشلىتەلەيمىز. بۇ پارامېتر مەنبە ئوبيېكت تىزمىسىدىكى ئوبيېكتلارنىڭ نۆلدىن باشلانغان تەرتىپ نومۇرىنى كۆرسىتىدۇ.

SelectMany مەشغۇلاتچىسى

دۆلەت تەۋەلىكى Italy بولغان بارلىق خېرىدارلارنىڭ بارلىق زاكازلىرىغا ئېرىشمەكچى بولساڭىز. بۇنىڭ ئۈچۈن تۆۋەندىكىدەك كود يېزىشىڭىز مۇمكىن:

كود 4.6

```
var orders =
    customers
    .Where(c => c.Country == Countries.Italy)
    .Select(c => c.Orders);

foreach(var item in orders) { Console.WriteLine(item); }
```

Select مەشغۇلاتچىسىنىڭ نورمال خۇلقىغا ئاساسەن مەزكۇر سۈرۈشتۈرۈكنىڭ نەتىجىسى IEnumerable<Order[]> تىپلىق بولۇپ نەتىجە تىزمىسىنىڭ ھەربىر ئەزاسىمۇ ھەم مەلۇم خېرىدارنىڭ زاكاز تىزمىسى بولىدۇ(Orders[]).

پروگراممىنىڭ ئىجرا نەتىجىسى تۆۋەندىكىچە:

```
DevLeap.Linq.Operators.Order[]
DevLeap.Linq.Operators.Order[]
```

دېمەك بۇ بىز مەقسەت قىلغان IEnumerable<Order> گە ئېرىشىشنىڭ نەتىجىسى ئەمەس. تەكشى IEnumerable<Order> تىپلىق نەتىجىگە ئېرىشىش ئۈچۈن، SelectMany مەشغۇلاتچىسىنى ئىشلىتىشكە توغرا كېلىدۇ. ئۇنىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source,
    Func<T, IEnumerable<S>> selector);
public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source,
    Func<T, int, IEnumerable<S>> selector);
public static IEnumerable<S> SelectMany<T, C, S>(
    this IEnumerable<T> source,
    Func<T, IEnumerable<C>> collectionSelector,
    Func<T, C, S> resultSelector);
```

مەزكۇر مەشغۇلاتچى مەنبە تىزمىسىنى چارلاپ نەتىجە ئەزالىرىنى(items) برىلەشتۈرىدۇ ۋە ئۇلارنى چارلاشقا بولىدىغان(IEnumerable<T>نى ئەمەلگە ئاشۇرغان) بىر دانە تىزما شەكلىدە قايتۇرۇپ بېرەلەيدۇ. ئۇنىڭ ئىككىچى ئەندىزىسى مەشغۇلاتنى تەرتىپ نومۇرى بىلەن باغلىيالايدۇ.

بايىقى مەسىلىنى SelectMany بىلەن ھەل قىلىش چارىسى مۇنداق:

كود 4.7

```
IEnumerable<Order> orders =
    customers
    .Where(c => c.Country == Countries.Italy)
    .SelectMany(c => c.Orders);
```

كود 4.7 دىكى ئىپادە تۆۋەندىكى سۇرۇشتۇرۇك ئىپادىسى بىلەن باراۋەر:

كود 4.8

```
IEnumerable<Order> orders =
    from    c in customers
    where   c.Country == Countries.Italy
        from    o in c.Orders
        select o;
```

يۇقىرىقى سۇرۇشتۇرۇك ئىپادىسىدىكى **select** خاس سۆزى تۇنجى **from** خاس سۆزدىن باشقا **from** لار بىلەن بىرلىشىپ **SelectMany** مەشغۇلاتچىسىغا تەرجىمە قىلىنىدۇ. باشقىچە قىلىپ ئېيتقاندا، ئەگەر ئىپادىدە بىردىن ئارتۇق **from** خاس سۆزى بولسا كاللىگىزدا تۆۋەندىكىدەك قائىدىگە سىڭدۈرۈڭ: **select** بىلەن تۇنجى **from** نىڭ بىرلىشىشى **Select** مەشغۇلاتچىسىغا تەرجىمە قىلىنىدۇ، باشقا بىرلىشىشلەر بىردەك **SelectMany** مەشغۇلاتچىسىغا تەرجىمە قىلىنىدۇ. ئۈچىنچى خىل ئەندىزىسىنى تەپسىلىي سۆزلىمەيمەن. ئۇنىڭ ئۈچۈن بىر مىسال قالدۇراي:

كود 4.10

```
IEnumerable<Order> orders =
    from    c in customers
    where   c.Country == Countries.Italy
        from    o in c.Orders
        select new {o.Quantity, o.IdProduct};
```

Ordering مەشغۇلاتچىلىرى (سورتلاش مەشغۇلاتچىلىرى)

Ordering مەشغۇلاتچىلىرىنىڭ قوللىنىشچانلىقىمۇ بىرقەدەر يۇقىرى بولۇپ، نەتىجە تىزمىسىدىكى ئەزالارنىڭ تەرتىپىنى ۋە يۆلىنىشىنى بەلگىلەش ئىقتىدارىغا ئىگە.

OrderBy بىلەن OrderByDescending مەشغۇلاتچىسى

ئەگەر Sql ئارقىلىق ساندان مەشغۇلاتى ئېلىپ بېرىپ باققا بولسىڭىز نەتىجىلەرنى مەلۇم شەرتكە ئاساسەن سورتلاش(تىزىش) نىڭ مۇھىم نۇختا ئىكەنلىكىنى ئەسكەرتىشىڭىزنىڭ ھاجىتى يوق.
Linq بولسا ordering مەشغۇلاتچىلىرى ئارقىلىق سۈرۈشتۈرۈك نەتىجىسىگە ئاشما ياكى كېمەيتىلىك سورتلاش ئېلىپ بارالايدۇ. مەسىلەن: دۆلەت تەۋەلىكى Italy بولغان خېرىدارلارنىڭ نامى بىلەن شەھەر خاسلىقلىرىغا نامىنىڭ ئالفابىت جەدۋىلىدىكى(ئىنگلىزلارنىڭ ئېلىپبە تەرتىپى) كېمىيىپ بېرىش تەرتىپى بويىچە تىزىلغان ھالىتىدە ئېرىشىش ئۈچۈن تۆۋەندىكىدەك كود يېزىش مۇمكىن:

كود 4.11

```
var expr =
    from    c in customers
    where   c.Country == Countries.Italy
    orderby c.Name descending
    select  new { c.Name, c.City };
```

يۇقىرىقى سۈرۈشتۈرۈك ئىپادىسى تۆۋەندىكى مۇناسىۋەتلىك ئەندىزىلەردىكى ماس كېڭەيتمە مېتودلار ئارقىلىق كود 15 دىكى جۈملىلەرگە تەرجىمە قىلىنىدۇ.

```
public static IOrderedSequence<T> OrderBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
public static IOrderedSequence<T> OrderBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
public static IOrderedSequence<T> OrderByDescending<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
public static IOrderedSequence<T> OrderByDescending<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
```

كود 15

```
var expr =
    customers
    .Where(c => c.Country == Countries.Italy)
    .OrderByDescending(c => c.Name)
    .Select(c => new { c.Name, c.City } );
```

ThenBy بىلەن ThenByDescending مەشغۇلاتچىسى

ئەگەر نەتىجىنى بىردىن ئارتۇق شەرت بىلەن سورتلىماقچى بولسىڭىز بۇ ئىككى مەشغۇلاتچى ھاجىتىڭىزدىن چىقىدۇ. تۆۋەندىكىلەر ئۇلارنىڭ ئەندىزىلىرى:

```
public static IOrderedSequence<T> ThenBy<T, K>(
    this IOrderedSequence<T> source,
    Func<T, K> keySelector);
public static IOrderedSequence<T> ThenBy<T, K>(
    this IOrderedSequence<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
public static IOrderedSequence<T> ThenByDescending<T, K>(
    this IOrderedSequence<T> source,
    Func<T, K> keySelector);
public static IOrderedSequence<T> ThenByDescending<T, K>(
    this IOrderedSequence<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
```

ئۇلارنىڭ ئەندىزىلىرى **OrderBy** بىلەن **OrderByDescending** مەشغۇلاتچىلىرىنىڭكىگە ئوخشىشىپ كېتىدۇ. پەرقلىق يېرى شۇكى، **ThenBy** بىلەن **ThenByDescending** نى پەقەت **IOrderedSequence<T>** گىلا قوللىنىشقا بولىدۇ (**IEnumerable<T>** غا بولمايدۇ). شۇڭا بۇ ئىككىسىنى پەقەت **OrderBy** بىلەن **OrderByDescending** نىڭ كەينىگە ئۇلاپلا ئىشلەتكىلى بولىدۇ، مۇستەقىل ئىشلەتكىلى بولمايدۇ. چۈنكى **OrderBy** لاردىن كەلگەن نەتىجە **IOrderedSequence<T>** نى ئەمەلگە ئاشۇرغان.

تۆۋەندە بۇنىڭغا بىر مىسال كەلسۇن:

كود 4.12

```
var expr = customers
    .Where(c => c.Country == Countries.Italy)
    .OrderByDescending(c => c.Name)
    .ThenBy(c => c.City)
    .Select(c => new { c.Name, c.City } );
```

ئاۋۋال نامىنىڭ كېمەيمىسى بويىچە تىزىپ ئاندىن ئالدىنقى تىزىش نەتىجىسىنى شەھەرنىڭ (City) ئاشمىسى بويىچە تىزىدۇ.

يۇقىرىقى جۈملىلەرنى سۈرۈشتۈرۈك ئىپادىسى ئارقىلىق مۇنداق ئىپادىلەش مۇمكىن:

كود 4.12

```
var expr =
    from    c in customers
    where   c.Country == Countries.Italy
    orderby c.Name descending, c.City
    select  new { c.Name, c.City };
```

ئەگەر ئۆزىڭىز سېلىشتۇرۇلماقچى بولغان تىپىڭىزغا نىسبەتەن سېلىشتۇرۇش قائىدىسىنى بېكىتمىگەن بولسىڭىز، Linq سىستېمىنىڭ كۆڭۈلدىكى قىممەت ھالىتى بويىچە سېلىشتۇرۇش ئېلىپ بارىدۇ. ئەگەر ئالاھىدە سېلىشتۇرۇش زۆرۈرىيتى تۇغۇلسا، مەسىلەن خېرىدارلارنىڭ ئىسمى ئۇيغۇرچە يېزىلغان بولسا ئۇلارنى ئۇيغۇر تىلى ئېلىپبە تەرتىپى بويىچە تىزىش كېرەك. بۇنداق ئەھۋاللاردا ئەمەلىي ئەھۋالغا خاس بولغان سېلىشتۇرغۇچ ياساپ، سورتلاش مەشغۇلاتچىلىرىغا پارامېتىر ئارقىلىق يوللاپ بېرىشىمىز كېرەك. مەسىلەن: ئاۋۋال ئۇيغۇرچە خەتنى ئۆز ئىچىگە ئالغان ئىككى دانە *string* نى سېلىشتۇرۇش تۈرىنى تەمىنلەپ:

كود 16

```
using System.Globalization;
private class UyghurComparer: IComparer<string> {
    public int Compare(string x, string y) {
 return [قائىدىگە ئاساسەن سېلىشتۇرغاندىكى نەتىجە]
} }
```

ئاندىن ئۇنى تۆۋەندىكى ئۇسۇل بويىچە ئىشلىتىمىز:

كود 17

```
IEnumerable<Order> orders =
    customers
    .SelectMany(c => c.Orders)
    .OrderBy(o => o.Month, new MonthComparer());
```

Reverce مەشغۇلاتچىسى (كۆمتۈرۈش)

بەزىدە نەتىجە تىزمىسىنىڭ تەرتىپىنى ئەكسىگە ئۆرۈش ئېھتىياجى تۇغۇلىدۇ. Linq دا بۇنداق ئەھۋالغا نىسبەتەن Reverce ناملىق كېڭەيتىلمە مېتود تەمىنلەنگەن. يەنى:

```
public static IEnumerable<T> Reverse<T>(
    this IEnumerable<T> source);
```

تۆۋەندە Reverce نى ئىشلىتىشتىن ئاددى بىر مىسال:

كود 4.14

```
var expr =
    customers
    .Where(c => c.Country == Countries.Italy)
    .OrderByDescending(c => c.Name)
    .ThenBy(c => c.City)
    .Select(c => new { c.Name, c.City } )
    .Reverse();
```

بىراق Linq دا Reverce مەشغۇلاتچىسى ئۈچۈن مەخسۇس خاس نام بېكىتىلمىگەن (مەسىلەن Select مەشغۇلاتچىسى ئۈچۈن select خاس نامى بېكىتىلگەن)، شۇڭلاشقا سۈرۈشتۈرۈك ئىپادىسى مەزكۇر مەشغۇلاتچىنى ئىشلىتىش ئۈچۈن تۆۋەندىكى مىسالدا كۆرسىتىلگەندەك ئۇسۇل قوللىنىش كېرەك:

كود 4.15

```
var expr =
    (from    c in customers
    where   c.Country == Countries.Italy
    orderby c.Name descending, c.City
    select  new { c.Name, c.City }
    ).Reverse();
```

Grouping مەشغۇلاتچىلىرى (گۇرۇپپىلاش)

سان- سىفىر تىزمىلىرىنى (ئەزا تىزمىلىرى) قانداق تاللاش، سۈزۈش ۋە سورتلاش مەشغۇلاتلىرىنى كۆرۈپ ئۆتتۇق. بەزىدە يۇقىرىقى مەشغۇلاتلاردىن ئېرىشكەن ئەزالارنى بىرلا تۈپلامغا ئورۇنلاشتۇرماي، مەلۇم شەرتكە ئاساسەن ئوخشىمىغان گۇرۇپپىلارغا بۆلۈش زۆرۈرىيىتى تۇغۇلىدۇ. Linq بۇنداق ئەھۋاللارنى ئويلاشقان ۋە مۇناسىپ مەشغۇلاتچى GroupBy نى تەمىنلىگەن. ئۇنىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source, Func<T, K> keySelector);
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source, Func<T, K> keySelector,
    IEqualityComparer<K> comparer);
public static IEnumerable<IGrouping<K, E>> GroupBy<T, K, E>(
    this IEnumerable<T> source, Func<T, K> keySelector,
    Func<T, E> elementSelector);
public static IEnumerable<IGrouping<K, E>> GroupBy<T, K, E>(
    this IEnumerable<T> source, Func<T, K> keySelector,
    Func<T, E> elementSelector, IEqualityComparer<K> comparer);
```

بارلىق ئەندىزىلىرى **IEnumerable<IGrouping<K, T>>** لىق قىممەت قايتۇرۇدىغان بولۇپ، بۇنىڭدىكى **IGrouping<K, T>** بولسا **Enumerable<T>** نى ئەمەلگە ئاشۇرغان خاسلاشتۇرۇلغان كۆپماس ئېغىز.

مەسىلەن: **GroupBy** مەشغۇلاتچىسى ئارقىلىق خېرىدارلارنى دۆلەت تەۋەلىكى بويىچە گۇرپىلارغا بۆلۈش ۋە نەتىجىنى زىيارەت قىلىش ئۈچۈن تۆۋەندىكىدەك كود يازىمىز:

كود 4.16

```
var expr = customers.GroupBy(c => c.Country);

foreach(IGrouping<Countries, Customer> customerGroup in expr) {
    Console.WriteLine("Country: {0}", customerGroup.Key);
    foreach(var item in customerGroup) {
        Console.WriteLine(item);
    }
}
```

كود 4.16 دە كۆرستىلگەندەك ھەربىر گۇرپىدىكى ئەزالارنى(دۆلەت تەۋەلىكى ئوخشاش بولغان خېرىدارلار) چارلاشتىن ئاۋال چوقۇم گورۇپىلارنىڭ ئاچقۇچ سۆزىنى چارلاش كېرەك. ھەر بىر گۇرپا **IGrouping<Countries, Customer>** لىق ئوبيېكت. باشقىچە ئېيتقاندا، يۇقىرىقى مىسالغا نىسبەتەن، خېرىدارلار دۆلەت تەۋەلىكىگە ئاساسەن گۇرپىلارغا بۆلىنىدۇ، ھەربىر گۇرپىنىڭ بىر ئاچقۇچلۇق سۆزى بولىدۇ، بۇ ئاچقۇچلۇق سۆز دەل شۇ گۇرپىدىكى خېرىدار ئورتاق تەۋە بولغان دۆلەت ئىسمىدۇر.

سۇرۇشتۇرۇك ئىپادىسىدە **goup...by...** خاس سۆزلۈك گىراماتىكا ئارقىلىق **GroupBy** مەشغۇلاتچىسىنى ئىقتىدارىنى ئىشلىتەلەيسىز. مەسىلەن كود 4.17 دىكى مەسىلە ئۈچۈن تۆۋەندىكىمۇ ئۇنۇملۇك:

كود 4.17

```
var expr =
    from  c in customers
    group c by c.Country;

foreach(IGrouping<Countries, Customer> customerGroup in expr) {
    Console.WriteLine("Country: {0}", customerGroup.Key);
    foreach(var item in customerGroup) {
        Console.WriteLine(item);
    }
}
```

كود 4.18 دا گۇرپىلاشقا ئالاقىدار يەنە بىر خىل مىسال بېرىلدى. بۇ قېتىم **GroupBy** نىڭ ئەڭ ئاخىرىقى ئەندىزىسى ئىشلىتىلدى.

كود 4.18

```
var expr =
    customers
    .GroupBy(c => c.Country, c => c.Name);

foreach(IGrouping<Countries, string> customerGroup in expr) {
    Console.WriteLine("Country: {0}", customerGroup.Key);
    foreach(var item in customerGroup) {
        Console.WriteLine("  {0}", item);
    }
}
```

يۇقىرىقى پروگراممىنىڭ ئىجرا نەتىجىسى تۆۋەندىكىچە:

```
Country: Italy
  Paolo
  Marco
Country: USA
  James
  Frank
```

Join مەشغۇلاتچىلىرى (ھەمدەم)

Join مەشغۇلاتچىلىرى Linq سۈرۈشتۈرۈك ئىچىدىكى تىزملىرى ئارىسىدىكى مۇناسىۋەتنى بەلگىلەشكە ئىشلىتىلىدۇ. SQL ساندان مەشغۇلاتىنى ئىلىپ ئېيتساق ئاساسەن ھەرقانداق سۈرۈشتۈرۈك بىر ياكى بىرقانچە جەدۋەلنى ھەمدەملەشتۈرىدۇ. Linq دا، بىر قىسىم ھەمدەم مەشغۇلاتچىلىرى يۇقىرىقىدەك خۇلقنى ئۆزگە مۇجەسسەملەشتۈرگەن.

Join (ھەمدەم)

شەك-شۆبىسىزكى Join بولسا ھەمدەم مەشغۇلاتچىلىرى ئىچىدىكى تۈنجى مەشغۇلاتچى. ئۇنىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static IEnumerable<V> Join<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, K> outerKeySelector,
    Func<U, K> innerKeySelector,
    Func<T, U, V> resultSelector);
public static IEnumerable<V> Join<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, K> outerKeySelector,
    Func<U, K> innerKeySelector,
    Func<T, U, V> resultSelector,
    IEqualityComparer<K> comparer);
```

Join مەشغۇلاتچىسى پارامېتىردا تۆت دانە كۆپماس تىپنىڭ يوللىنىشىنى تەلەپ قىلىدۇ. T بولسا سىرتقى مەنبە تىزما تىپىغا ۋەكىللىك قىلىدۇ، U بولسا ئىچكى مەنبە تىزما تىپىنى ئىپادىلەيدۇ. كۆرسەتمە outerKeySelector بىلەن innerKeySelector لار بولسا سىرتقى ۋە ئىچكى مەنبە تىزمىلىرىدىكى ئەزالارنىڭ ئاچقۇچلۇق سۆزلىرىنى قانداق پەرق ئېتىشنى ئېيتىپ بېرىدۇ. بۇ ئاچقۇچلۇق سۆزلەرنى ھەر ئىككىلىسى K تىپلىق بولۇپ، Join ئۇلارنىڭ تەڭلىكىنى ئۆزگە شەرت قىلىدۇ. V بولسا ئاڭ ئاخىرقى كۆپماس تىپ بولۇپ ئۇ Join نىڭ نەتىجە تىزمىسىدىكى ھەربىر ئەزانىڭ تىپىنى ئىپادىلەيدۇ.

Join نىڭ ئىككى خىل ئەندىزىسىدە سېلىشتۇرغۇچقا ئورۇن بېرىلگەن بولۇپ، ھەمدەملەشمەكچى بولغان ئىككى تىزمىنىڭ ئاچقۇچلۇق سۆزلىرىنى سېلىشتۇرۇشقا ئىشلىتىلىدۇ. بۇنىڭ ئارقىلىق سېلىشتۇرۇق قائىدىسىنى خاسلاشتۇرالايسىز. ئەگەر مەزكۇر پارامېتىرغا Null بېرىلسە ياكى بىرنچى خىل ئەندىزىسى ئىشلىتىلسە سېلىشتۇرۇش ئۈچۈن كۆڭۈلدىكى قىممەتكە بەلگىلەپ قويۇلغان سېلىشتۇرغۇچى EqualityComparer<TKey>.Default ئىشلىتىلىدۇ.

ئەمدى **Join** نى مىسال ئارقىلىق چۈشۈنۈپ باقايلى. خېرىدارلارنى ئۇلارنىڭ زاكازلىرى ۋە مەھسۇلاتلار بىلەن باغلاپ ئويلاپ كۆرۈڭ. تۆۋەندىكى سۈرۈشتۈرۈڭ زاكازلارغا ماس مەھسۇلاتلارنى ھەممەلەشتۈرىدۇ:

كود 4.19

```
var expr =
    customers
    .SelectMany(c => c.Orders)
    .Join( products,
           o => o.IdProduct,
           p => p.IdProduct,
           (o, p) => new {o.Month, o.Shipped, p.IdProduct, p.Price });
```

يۇقىرىقى سۈرۈشتۈرۈك تۆۋەندىكىدەك نەتىجە چىقىرىدۇ:

```
{Month=January, Shipped=False, IdProduct=1, Price=10}
{Month=May, Shipped=True, IdProduct=2, Price=20}
{Month=July, Shipped=False, IdProduct=1, Price=10}
{Month=December, Shipped=True, IdProduct=3, Price=30}
{Month=January, Shipped=True, IdProduct=3, Price=30}
{Month=July, Shipped=False, IdProduct=4, Price=40}
```

بۇ مىسالدا **orders** سىرتقى تىزمىغا، **products** بولسا ئىچكى تىزمىغا ۋەكىللىك قىلىدۇ.**lambda** ئىپادىسىدىكى **o** بىلەن **p** لار ئايرىم-ئايرىم ھالدا **Order** ۋە **Product** تىپلىق.
ئەگەر يۇقىرىقى ئىپادىنى **Sql** جۈملىسى ئارقىلىق ئىپادىلىسەك، تۆۋەندىكىدەك يېزىش مۇمكىن:

```
SELECT     o.Month, o.Shipped, p.IdProduct, p.Price
FROM       Orders AS o
INNER JOIN Products AS p
      ON   o.IdProduct = p.IdProduct
```

ئەگەر ئىپادىنى سۈرۈشتۈرۈك ئىپادىسى ئارقىلىق يازساق تۆۋەندىكىدەك بولىدۇ:

كود 4.20

```
var expr =
    from c in customers
        from  o in c.Orders
        join  p in products
              on o.IdProduct equals p.IdProduct
        select new {o.Month, o.Shipped, p.IdProduct, p.Price };
```

يۇقىرىقى سۆرۈشتۈرۈك ئىپادىسىدە دىققەت قىلىشقا تېگىشلىك يېرى شۇكى

```
(o.IdProduct equals p.IdProduct)
```

بۇ ئىككىسىنىڭ تەرتىپىنى ئۆزگەرتىشكە بولمايدۇ. يەنى: سىرتقى تىزما باشتا ئىچكى تىزما ئاخىرىدا كېلىشى كېرەك.

GroupJoin مەشغۇلاتچىسى

Sql دىكى **LEFT OUTER JOIN** ياكى **RIGHT OUTER JOIN** غا ئوخشىشىپ كېتىدىغان ئىقتىدارنى ئەمەلگە ئەمەلگە ئاشۇرۇش ئۈچۈن. **GroupJoin** مەشغۇلاتچىسى ياخشى تاللاش. ئۇنىڭ ئەندىزىلىرى **Join** مەشغۇلاتچىسىنىڭكىگە ئىتايىن ئوخشىشىپ كېتىدۇ:

```
public static IEnumerable<V> GroupJoin<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, K> outerKeySelector,
    Func<U, K> innerKeySelector,
    Func<T, IEnumerable<U>, V> resultSelector);
public static IEnumerable<V> GroupJoin<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, TKey> outerKeySelector,
    Func<U, TKey> innerKeySelector,
    Func<T, IEnumerable<U>, V> resultSelector,
    IEqualityComparer<TKey> comparer);
```

ئەمەلىي ئىشلىتىلىشىنى تۆۋەندىكى مىسال ئارقىلىق ھېس قىلىۋېلىڭ، ھازىرچە كۆپ توختالمايمەن.

كود 4.21

```
var expr =
    products
    .GroupJoin(
        customers.SelectMany(c => c.Orders),
        p => p.IdProduct,
        o => o.IdProduct,
        (p, orders) => new { p.IdProduct, Orders = orders });
foreach(var item in expr) {
    Console.WriteLine("Product: {0}", item.IdProduct);
    foreach (var order in item.Orders) {
        Console.WriteLine("     {0}", order); }}
```

تۆۋەندىكىسى يۇقىرىقى جۈملىلەرنىڭ ئىجرا نەتىجىسى:

```
Product: 1
    3 - False - January - 1
    10 - False - July - 1
Product: 2
    5 - True - May - 2
Product: 3
    20 - True - December - 3
    10 - True - January - 3
Product: 4
Product: 5
    20 - False - July - 5
Product: 6
```

سۈرۈشتۈرۈك ئىپادىسىدە Join مەشغۇلاتچىسى ...joint...into خاس سۆزلىرى ئارقىلىق ئىپادىلىنىدۇ. كود 4.21 بىلەن بارەۋەر سۈرۈشتۈرۈك ئىپادىسى تۆۋەندىكىچە:

كود 4.23

```
var expr =
    from    p in products
    join    o in (
            from c in customers
                from    o in c.Orders
                select o
            ) on p.IdProduct equals o.IdProduct
            into orders
    select new { p.IdProduct, Orders = orders };
```

Set مەشغۇلاتچىلىرى (تويلام)

Linq سەپىرىمىز ئۇچۇر تىزمىلىرىغا ئىلىپ بىرىلىدىغان «بىرىكمىسى»، «كەسمىسى» ۋە «دىن باشقا» لاردەك ئەڭ ئاساسى مەشغۇلاتلارغا كىلىپ قالدى.

Distinct مەشغۇلاتچىسى (تەكرارنى تازىلاش)

بىرقانچە خىرىدار ئوخشاش بىر مەھسۇلاتنى زاكاز قىلىشى مۇمكىن. دىمەك سىز خىرىدارلار زاكازلىرىدىكى مەھسۇلاتلارنى ئوقۇپ چىقسىڭىز نەتىجە تىزمىسىدا تەكرار ئەزا كۆرۈلىشى مۇمكىن. مۇشۇ خىل تەكرارلىقنىڭ ئالدىنى ئىلىش ئۈچۈن قانداق قىلىش كىرەك؟ مۇشۇ خىل مەسىلە Sql جۈملىسىدە ئادەتتە Join سۈرۈشتۈرۈكى ئىچىدە DISTINCT خاس سۆزىنى ئىشلىتىش ئارقىلىق ھەل قىلىناتتى. Linq سۈرۈشتۈرۈكلىرىدە بولسا بۇنىڭ ئۈچۈن تويلام مەشغۇلاتچىلىرىنىڭ ئەزاسى بولغان Distinct مەشغۇلاتچىسىنى تەمىنلىگەن. ئۇنىڭ ئەندىزىسى ئىنتايىن ئاددى بولۇپ پەقەت مەنبە تىزمىنىلا قوبۇل قىلىپ ئۇنىڭ ئەزالىرى ئارىسىدىكى ئۆزگىچە ئەزالارنى قايتۇرۇپ بىرىدۇ.

Distinct ئۈچۈن بىر مىسال كود 4.24 دە بىرىلدى.

```
public static IEnumerable<T> Distinct<T>(
    this IEnumerable<T> source);
```

كود 4.24

```
var expr =
    customers
    .SelectMany(c => c.Orders)
    .Join(products,
          o => o.IdProduct,
          p => p.IdProduct,
          (o, p) => p)
    .Distinct();
```

گەرچە Distinct مەشغۇلاتچىسىغا ماس سۈرۈشتۈرۈك خاس سۆزى تەمىنلەنمىگەن بولسىمۇ، ئۇنى سۈرۈشتۈرۈك نەتىجىسى ئۈستىگە قوللىنالايمىز. مەسىلەن تۆۋەندىكىدەك:

كود 4.25

```
var expr =
    (from c in customers
         from   o in c.Orders
         join   p in products
```

```
                on o.IdProduct equals p.IdProduct
        select p
   ).Distinct();
```

نورمال ھالەتتە Distinct ئېلمېنتلارنى ئۇلارنىڭ GetHashCode ۋە Equals مېتودلىرى ئارقىلىق بىر-بىرى بىلەن سېلىشتۇرىدۇ ۋە ئۆزئارا پەرقلەندۈرىدۇ. ئەگەر زۆرۈر تېپىلسا Distinct نىڭ تۆۋەندىكى ئەندىزىسى ئارقىلىق ئۆزىمىزنىڭ سېلىشتۇرغۇچىسىنى يوللاپ سېلىشتۇرۇش خۇلقىنى كونترول قىلالايمىز.

Union, Intersect, and Except

Union مەشغۇلاتچىسى ئىككى تىزما ئېلېمېنتلىرىنى تەكرارلىقنى يوقاتقان ئاساستا بىرلەشتۈرۈپ چىقىدۇ. مەسىلەن:

كود 4.26

```
var expr = customers[1].Orders.Union(customers[2].Orders);
```

خۇددى Distinct قا ئوخشاش بۇلارمۇ ئېلېمېنتلىرنى سېلىشتۇرۇش ئۈچۈن GetHashCode ۋە Equals مېزودلارنى ئىشلىتىدۇ(بىرىنچى ئەندىزىسىدە). ئەلۋەتتە، خاسلاشتۇرۇلغان سېلىشتۇرغۇچنى ئىشلىتىشكە يول قويغان ئەندىزىلىمۇ بار. مىسالەن، كود 4.26 نىڭ نەتىجىسى تۆۋەندىكىدەك بولىدۇ.

```
10 - False - July  - 1
20 - True - December  - 3
20 - True - December - 3
```

نەتىجە ئويلىغان يېرىڭىزدىن چىقمىغاندەكمۇ؟ ئاخىرقى ئىككى قۇر ئويمۇ ئوخشاش تۇرىدىغۇ، ئەجبا Distinct نىڭ رولى بولمىغاندىمۇ؟

«مىساللار ئۈچۈن تۆر قۇرۇلمىسى» دىكى خېرىدار ئوبيېكتلىرىنى دەسلەپلەشتۈرۈش كودىغا(مەزكۇر ماقالىدىكى بارلىق كودلار نىڭ مىسال مەشغۇلات ئوبىكتى) نەزەر سالىدىغان بولساق، دەسلەپلەشتۈرۈلگەن ھەر بىر زاكاز ئەزاسى(order) بولسا Order ناملىق چاقىرىلما(引用) تىپىنىڭ ئوخشىمىغان ئوبيېكتلىرى. گەرچە ئىككىنچى خېرىدارنىڭ ئىككىنچى زاكازى بىلەن ئۈچىنچى خېرىدارنىڭ بىرىنچى زاكازى سان- سىفىر جەھەتتىن ئوخشاش بولغىنى بىلەن ئۇلارنىڭ Hash كودى ئوخشىمايدۇ.

نەتىجىدە ئۇلارنى سېلىشتۇرغاندا يەنىلا ئوخشىمىغان ئوبيېكت بولۇپ چىقىپ Distinct نىڭ تەكرارلارنى سۈزۈش ئەگلىكىدىن(ئەگلەك: ئائىلىلەردە ئۇن تاسقاشقا ئىشلىتىلىدىغان سۈزگۈچ) ئۆتۈپ كېتىدۇ.

قوشۇمچە ساۋات 1

مەلۇم چاقىرلما تىپنىڭ ئىككى ئوبيېكتىنى (مەسلەن بىزنىڭ مىسالىمىزدىكى (Customer سىستېما كۆڭۈلدىكى قىممەت ھالەتتە سېلىشتۇرغاندا ھەربىرىنىڭ Hash كودىنى سېلىشتۇرىدۇ. گەرچە ئۇ ئىككىسىنىڭ خاسلىقلىرىنىڭ قىممەتلىرى ئوخشاش بولسىمۇ لېكىن Hash كودى ئوخشاش چىقمايدۇ. چۈنكى ئۇلار بەربىر ئىككى ئوبيېكت. ھەرقانداق تىپلىق ئوبيېكتلارنىڭ Hash كودى Net. قۇرۇلمىسىدىكى بارلىق تىپلارنىڭ ئەجدادى بولغان Objectتىپتىن مىراس قالغان ()GetHashCode مېتودى ئارقىلىق ئېلىنىدۇ. شۇڭا بۇنداق ئەھۋال ئاستىدا تىپلارنىڭ مىراس ئالغان ()GetHashCode ۋە Equals مېتودلىرىنى قاپلاپ يېزىش (override, 重写) ئارقىلىق ئۇلارنىڭ سېلىشتۇرۇلۇش خۇلقلىرىنى ئۆزگەرتىشكە توغرا كېلىدۇ. ئەمما قىممەتلىك(值类型 ,value type) تىپلارنى(int, float, struct دېگەندەك) سېلىشتۇرۇشتا ئىش باشقىچەرەك بولىدۇ. ئەگەر ئىككىسىنىڭ قىممىتى ئوخشاشلا بولسا ئۇلار تەڭداش دەپ قارىلىدۇ.

يىغىنچاقلىغاندا Linq دىكى بارلىق سېلىشتۇرۇشلار يۇقىرىقى قائىدىگە چۈشىدۇ، شۇڭا نەتىجە ئويلىغان يېرىڭىزدىن چىقماي قالىدىغان ئىشلاردىن خالى بولالمايسىز. مۇشۇنداق مەسىلىلەرنىڭ ئالدىنى ئېلىش ئۈچۈن، ئۆزىڭىزنىڭ شەخسى چاقىرلما تىپلىرىغا (Order دەك) ئامال بار سېلىشتۇرۇش خۇلقلىرىنى قوشۇڭ. ياكى كۆپ قىسىم پروگراممىرلا تەشەببۇس قىلغاندەك class نىڭ ئورنىغا struct دەك قىممەتلىك تىپلىق ھاسىلات تىپلارنى ئىشلىتىڭ.

دېمەك قوشۇمچە ساۋات 1 ئېيتىپ ئۆتكەن ئۇسۇلىمىز بويىچە زاكاز(Order) تىپىنىڭ سېلىشتۇرۇش خۇلقىنى ئۆزگەرتىشكە توغرا كەلدى. ئۇسۇلى مۇنداق:

```
public class Order {
    public int Quantity;
    public bool Shipped;
    public string Month;
    public int IdProduct;
// ToString() مېتودىنى قاپلاپ يېزىش ئارقىلىق، مەزكۇر تىپنىڭ ئىسمىغا ۋەكىللىك قىلىدىغان //
// ھەرپ-بەلگە تىزمىسىنىڭ فورماتىنى ئۆزگەرتكىلى بولىدۇ//
    public override string ToString() {
        return String.Format("{0} - {1} - {2} - {3}",
        this.Quantity, this.Shipped, this.Month, this.IdProduct);
    }

    public override bool Equals(object obj) {
        if (!(obj is Order))
            return false;
```

```
        else {
            Order o = (Order)obj;
            return(o.IdProduct == this.IdProduct &&
                   o.Month == this.Month &&
                   o.Quantity == this.Quantity &&
                   o.Shipped == this.Shipped); }
    }
    public override int GetHashCode() {
        return String.Format("{0}|{1}|{2}|{3}", this.IdProduct,
          this.Month, this.Quantity, this.Shipped).GetHashCode();
    }
}
```

يۇقىرىقىدەك مەسىلىنى ھەل قىلىشنىڭ يەنە بىر چارىسى، Distinct مەشغۇلاتچىسىنىڭ ئىككىنچى خىل ئەندىزىسى بويىچە، ئۆزىمىز تۈزۈۋالغان سېلىشتۇرغۇچنى پارامېتىر ئارقىلىق يوللاپ بېرىش. ئۈچىنچى خىل ئۇسۇلى، يەنى ئەڭ ئاخىرقى ئۇسۇلى بولسا Order تىپىنى قىممەتلىك تىپقا ئالماشتۇرۇش. دېمەكچى، Order تىپىنى class ئەمەس struct تىپلىق قىلىش. مەسىلەن:

```
//struct بولسا قىممەت تىپلىق ھاسىلە تىپ
public struct Order {
    public int Quantity;
    public bool Shipped;
    public string Month;
    public int IdProduct;
}
```

شۇنىسى ئېسىڭىزدە تۇرسۇن، C#3.0 دىكى بارلىق نامسىز تىپلار بىردەك چاقىرلما تىپقا تەۋە.

تۆۋەندىكىسى Intersect بىلەن Except نى ئىشلىتىشتىن مىسال:

كود 4.27

```
var expr1 = customers[1].Orders.Intersect(customers[2].Orders);
var expr2 = customers[1].Orders.Except(customers[2].Orders);
```

Intersect مەشغۇلاتچىسى ئىككى تىزمىدىكى ئەزالارنىڭ كەسمىسىنى ئالىدۇ، يەنى، ھەر ئىككىلىسىدا بولغانلىرىنى.

Except (دىن باشقا) مەشغۇلاتچىسى بولسا، بىرىنچى تىزمىدىكى ئىككىنچى تىزمىدا ئۇچرىمايدىغان ئەزالارنى ئالىدۇ.

بۇلارغىمۇ تەڭداش بولغان سۈرۈشتۈرۈك خاس سۆزلىرىنىڭ يوقلۇقىنى دېگىم كەلمەيۋاتىدۇ. بىراق تۆۋەندىكىدەك ئارلاشتۇرۇپ ئىشلىتەلەيمىز:

كود 4.28

```
var expr =
    (from c in customers
         from    o in c.Orders
         where   c.Country == Countries.Italy
         select o
    ).Intersect(
        from c in customers
            from    o in c.Orders
            where   c.Country == Countries.USA
            select o);
```

Aggregate مەشغۇلاتچىلىرى (جەملەش مەشغۇلاتچىلىرى)

بەزىدە تىزمىدىكى ئەزالارغا نىسبەتەن ھېسابلاش ئېلىپ بېرىش توغرا كېلىدۇ. Linq بۇنىڭ ئۈچۈن **Average**, **Min**, **Sum**, **LongCount**, **Count** مەشغۇلاتچىلىرىدىن تەشكىل تاپقان جەملەش مەشغۇلاتچىلىرى ئائىلىسىنى تەمىنلىگەن. بۇلارنىڭ كۆپ قىسمىلىرىنىڭ قىلىدىغان ئىشى مۇرەككەپ بولمىغاچقا خۇلقلىرىنى(**behavior**) چۈشۈنۈش تەسكە توختىمايدۇ.

Count ۋە LongCount مەشغۇلاتچىسى

تۆۋەندىكىسى Count مەشغۇلاتچىسىنى زاكاز تىزمىسىغا قارىتا ئىشلىتىشتىن مىسال:

كود 4.29

```
var expr =
    from    c in customers
    select new {c.Name, c.City, c.Country, OrdersCount =
 c.Orders.Count() };

foreach (var v in query)
Console.WriteLine("{0}-{1}-{2}-{3}", v.Name, v.City, v.Country,
 v.OrderCount);
```

ئىجرا نەتىجىسى:

```
Paolo-Brescia-Italy-2
Marco-Torino-Italy-2
James-Dallas-USA-1
Frank-Seattle-USA-1
```

يۇقىرىقى پروگراممىدىن ھاسىل بولغان نامسىز تىپلىق ئوبيېكىتلارنىڭ OrdersCount ناملىق خاسلىقىنىڭ قىممىتى ماس خېرىدارنىڭ زاكاز سانىنى ئالىدۇ. دېمەك، Count مەشغۇلاتچىسى مەنبە تىزما ئىچىدىكى ئەزالارنىڭ سانىنى int شەكلىدە قايتۇرۇپ بېرىدۇ. Count مەشغۇلاتچىسىنىڭ يەنە بىر خىل ئەندىزىسى بولۇپ، ئۇ مەنبە تىزمىدىكى مەلۇم شەرتنى قانائەتلەندۈرگەن ئەزالار سانىنى تېپىپ بېرىدۇ. مەسىلەن: خېرىدارلار ئارىسىدا زاكازىنىڭ سانى ئىككىگە تەڭ بولغانلارنىڭ سانىنى تېپىپ باقايلى:

```
int equalTwo = customers.Count(c => c.Orders.Count() == 2);
Console.WrieteLine(equalTwo.ToString());
```

LongCount نىڭ ئىقتىدارى Count بىلەن ئوخشاش بولۇپ، قايتۇرىدىغان قىممىتى long تىپلىق، خالاس. (long تىپىنىڭ سىغىمى int دىن يۇقىرى)

Sum مەشغۇلاتچىسى (يىغىندا)

بۇ مەشغۇلاتچى سەل ئالاھىدىرەك. ئاۋال ئۇنىڭ ئەندىزىلىرىنى كۆرۈپ ئۆتەيلى:

```
public static Numeric Sum(
    this IEnumerable<Numeric> source);
public static Numeric Sum<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

ئۇنىڭ قايتما قىممىتى ياكى ئىككىنچى ئەندىزىدىكى پارامېتىرى بولسۇن ئۇلارنىڭ تىپى Numeric ئىكەن. بۇ يەردىكى Numeric بولسا «سانلىق» دېگەن مەنىدە بولۇپ، int, int?, long, long?, float, float?, double, double?, decimal, decimal? لارنىڭ خالىغان بىرىنى كۆرسىتىدۇ. int غۇ پۈتۈن سان تىپى، ئەمسە int? چۇ؟
C#2.0 دىن تارتىپ قىممەتلىك تىپلىق ئۆزگەرگۈچى مىقدارلارغا ھېچ قانداق قىممەت يوللانمايدىغان ھالەتنى ئىپادىلەش زۆرۈرىيىتىدىن(بولۇپمۇ ساندانىدىكى قۇرۇق int تىپىغا ماسلاشتۇرۇش) شۇ تىپ خاس سۆزىنىڭ ئارقىسىغا سوئال بەلگىسى ئارقىلىق (T? دەك) قۇرۇق بولالايدىغان تىپ ھالىتىنى ئىپادىلەيدۇ (ئۇنىڭ تىپى *Nullable<T>*). مەسىلەن int? ئارقىلىق Nullable<System.Int32> ئىپادىلىنىدۇ. int? i; i=null; Console.WriteLine(i). دەك يېزىش مۇتلەق ئىناۋەتلىك. لېكىن int i; Console.WriteLine(i); بۇ قۇرلار تەكشۈرۈشتىن ئۆتمەيدۇ.
Sum نىڭ بىرىنچى ئەندىزىسىدە مەنبە تىزمىدىكى ئەزالار سانلىق تىپلىق دەپ قارىلىپ ئۇلارغا بىۋاستە قوشۇش ئەملى بېجىرىپ يىغىندىنى قايتۇرىدۇ. ئەگەر مەنبە قۇرۇق بولسا نۆل قايتىدۇ. ئەگەر ئەزالار null بولالايدىغان تىپلىق بولۇپ ھەممىسىنىڭ قىممىتى null بولسا نەتىجىدە null قايتىدۇ. مەزكۇر ئەندىزە مەنبە تىزمىدىكى ئەزالارنى بىۋاستە قوشقىلى بولىدىغان ئەھۋال ئاستىدا ئىشلىتىلىدۇ. مەسىلەن: مەنبە تىزما سانلار گۇرۇپپىسى تىپلىق بولسا، تۆۋەندىكىدەك ئىشلىتەلەيمىز:

```
int[] values = { 1, 3, 9, 29 };
int   total  = values.Sum();
```

نەتىجە 1+3+9+29=42 بولىدۇ.
ئەگەر تىزما بىۋاستە قوشقىلى بولىدىغان سانلىق ئەزالاردىن تۈزۈلمىسە(مەسىلەن خېرىدارلار تىزمىسىدەك)، Sum نىڭ ئىككىنچى ئەندىزىسىنى قوللىنىپ ئەزانىڭ قايسى خاسلىقىنى قوشۇشنى بەلگىلەپ قويىمىز. مەسىلەن:

كود 4.30

```
var customersOrders =
    from c in customers
        from    o in c.Orders
        join    p in products
                on o.IdProduct equals p.IdProduct
        select new { c.Name, OrderAmount = o.Quantity * p.Price };

var expr =
    from    c in customers
    join    o in customersOrders
            on c.Name equals o.Name
            into customersWithOrders
    select new { c.Name,
                 TotalAmount = customersWithOrders.Sum(o =>
o.OrderAmount) };
```

ئۈستىدىكى مىسالدا customers تىزمىسىنى customersOrders تىزمىسى بىلەن ھەمدەملەپ، ھەربىر خېرىدارنىڭ زاكاز سانىغا ئېرىشىپ ئۇلارنى قوشۇش نەتىجىسىنى ئالدۇق.
ئادەتتە يۇقىرىقى كودنى سۈرۈشتۈرۈك ئىپادىسى ئارقىلىق تۆۋەندىكىدەك ئىپادىلەيمىز:

كود 4.31

```
var expr =
    from    c in customers
    join    o in (
            from c in customers
                from    o in c.Orders
                join    p in products
                        on o.IdProduct equals p.IdProduct
                select new { c.Name, OrderAmount = o.Quantity * p.Price }
            ) on c.Name equals o.Name
            into customersWithOrders
```

```
    select new { c.Name,
                 TotalAmount = customersWithOrders.Sum(o =>
o.OrderAmount) };
```

Linq سۈرۈشتۈرۈك گراماتىكىسى vs. SQL

يېزىپ مۇشۇ يەرگە كەلگەندە، ئىككىسىنى سېلىشتۇرۇپ بېقىشنى توغرا تاپتىم، چۈنكى ئۇلار ئوخشىشىپ كېتىدىغانلىقتىن تۈرسىمۇ ئارىسىدا ئىنتايىن مۇھىم پەرق بار. بۇ توغرىلىق توختىلىشنىڭ ھاجىتى بار دەپ ئويلىدىم.

تۆۋەندىكىسى كود 4.31 دىكى سۈرۈشتۈرۈك ئىپادىگە ئوخشاپ كېتىدىغان Sql ئىپادىسى. (خېرىدارلارنىڭ ئىسمى بىردىن-بىر دەپ پەرەز قىلىندى):

```
SELECT   c.Name, SUM(o.OrderAmount) AS OrderAmount
FROM     customers AS c
INNER JOIN (
    SELECT     c.Name, o.Quantity * p.Price AS OrderAmount
    FROM       customers AS c
    INNER JOIN orders AS o ON c.Name = o.Name
    INNER JOIN products AS p ON o.IdProduct = p.IdProduct
    ) AS o
ON       c.Name = o.Name
GROUP BY c.Name
```

يۇقىرىقى Sql جۈملىلىرىنىڭ نەقەدەر كېلەڭسىز ئىكەنلىكىنى ھېس قىلغانسىز!. ئەمەلىيەتتە ئۇنىدىن ئاددىراق Sql ئىپادىسى ئارقىلىق ئوخشاش نەتىجىگە ئېرىشەلەيمىز:

```
SELECT   c.Name, SUM(o.OrderAmount) AS OrderAmount
FROM     customers AS c
INNER JOIN (
    SELECT     o.Name, o.Quantity * p.Price AS OrderAmount
    FROM       orders AS o
    INNER JOIN products AS p ON o.IdProduct = p.IdProduct
    ) AS o
ON       c.Name = o.Name
GROUP BY c.Name
```

لېكىن تېخىمۇ قىسقا، تېخىمۇ ئاددىي ئىپادىلىيەلەيمىز:

```
SELECT     c.Name, SUM(o.Quantity * p.Price) AS OrderAmount
FROM       customers AS c
INNER JOIN orders AS o ON c.Name = o.Name
```

```
INNER JOIN products AS p ON o.IdProduct = p.IdProduct
GROUP BY    c.Name
```

ئەگەر بىز ئۈچىنچى خىل Sql چە ئىپادىلەش ئۇسۇلىنى Linq سۈرۈشتۈرۈك ئىپادىسىدە ئىپادىلىمەكچى بولساق بىر قىسىم قىيىنچىلىقلارغا دۇچ كېلىشىمىز مۇمكىن. سەۋەبى، Sql بولسا ئۇچۇرلارنى مۇناسىۋەتلىك ئارقىلىق سۈرۈشتۈرىدۇ، بارلىق ئۇچۇرلار تاكى سۈرۈشتۈرۈلگەنگە قەدەر تەكشى ھالەتتە(جەدۋەلدە) تۈرىدىغان بولۇپ ئۇلار ئارىسىدا دەرىجە، يەنى تەۋەلىك مۇناسىۋەتلىك ئىپادىلەنمەيدۇ. بىراق Linq بولسا خۇددى خېرىدار\زاكازلار\مەھسۇلاتلار غا ئوخشاش يەرلىك دەرىجە، يەنى تەۋەلىك مۇناسىۋەتلىك بولغان ئۇچۇرلارغا مەشغۇلات ئېلىپ بارىدۇ. بۇ پەرق ئىككى خىل ئۇسۇلنىڭ ئوخشىمىغان شارائىتتا ئۆزىگە خاس ئارتۇقچىلىقىنىڭ بارلىقىنى چۈشەندۈرىدۇ. يەنى ئارتۇقچىلىقى سىزنىڭ قانداق ئۇچۇرغا قانداق سۈرۈشتۈرۈش ئېلىپ بارىدىغانلىقىڭىز تەرىپىدىن بەلگىلىنىدۇ.
يۇقىرىقى سەۋەبلەر تۈپەيلى ئوخشاش مەنبەدىن ئوخشاش نەتىجىگە ئېرىشىدىغان خىل(ئەڭ ياخشىلانغىنى) سۈرۈشتۈرۈك ئىپادىسى Sql بىلەن Linq دا پەرقلىق يېزىلىدۇ.

Min and Max

جەملەش مەشغۇلاتچىلىرى ئىچىدىكى Max بىلەن Min مەشغۇلاتچىسى مەنبە تىزما ئىچىدىكى ئەڭ چوڭ ۋە ئەڭ كىچىك ئەزانى تېپىشقا ئىشلىتىلىدۇ. ئۇلارمۇ بىرقانچە خىل ئەندىزىلەنگەن:

```
public static Numeric Min/Max(
    this IEnumerable<Numeric> source);
public static T Min<T>/Max<T>(
    this IEnumerable<T> source);
public static Numeric Min<T>/Max<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
public static S Min<T, S>/Max<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
```

بىرىنچى ئەندىزىسىدە ئەزالارغا نىسبەتەن ئارخىمىتىكىلىق سېلىشتۇرۇش ئېلىپ بارىدۇ. شۇڭا مەزكۇر ئەندىزىنى ئەزالىرى سان بولغان تىزملارغا ئىشلىتىش مۇۋاپىق. مەسىلەن: تۆۋەندىكى مىسالدا بارلىق خېرىدارلارنىڭ بارلىق زاكازلىرىنىڭ مىقدارىغا نىسبەتەن ئەڭ كىچىكىنى تېپىش مەشغۇلاتى ئېلىپ بېرىلىدۇ:

كود 4.32

```
var expr =
    (from c in customers
        from    o in c.Orders
        select o.Quantity
    ).Min();
```

ئىككىنچى خىل ئەندىزىلەردە ئەزالارنىڭ تىپىنى ئېتىۋارغا ئالماي چوڭ ـ كىچىكىنى تاپىدۇ. (ئەلۋەتتە، چوڭ ـ كىچىكنى تېپىش ئۈچۈن سېلىشتۇرۇش ئېلىپ بېرىش كېرەك) ئەگەر مەنبە تىزمىدىكى ئەزا تىپى IComparable<T> نى (ياكى IComparable نى) قوللىسا مۇشۇنىڭ ئەمەللەشتۇرۇلۇشى ئارقىلىق سېلىشتۇرۇش قىلىدۇ. ئەگەر يۇقىرىقى ئىككىلا ئېغىزنى ئەمەلگە ئاشۇرمىغان بولسا ArgumentException تىپلىق بىنورماللىق(异常نى مۇشۇنداق ئاتاپ تۇردۇم) قويۇپ بېرىدۇ. بىنورماللىق خاتالىق ئۇچۇرىنىڭ ئاساسى مەزمۇنى مۇنداق: «كەم دېگەندە بىر دانە ئوبيېكت IComparable ئېغىزىنى ئەمەلگە ئاشۇرغان بولۇشى كېرەك.»

دەلىلى ئۈچۈن كود 4.33 دىكىدەك كود يېزىپ قەستەن بىنورماللىق چىقىرىلدى. مەزكۇر كوددىكى خاتالىق سەۋەبى شۇكى، گەرچە نەتىجە تىزمىسى زاكاز مىقدارىنىڭ تىپلىمى بولسىمۇ لېكىن كود 71 دىكى سۈرۈشتۈكتىن پەرقلىنىدۇ. يەنى ئالدىنقىسىدىكى چارلاش نەتىجىسىنىڭ تىپى سانلار گۇرپىسى. كىيىنكىسىنىڭ بولسا بىرلا سانلىق قىممەتلىك خاسلىقى بولغان نامسىز تىپلىق ئەزالار توپلىمى. نامسىز تىپلار بىردەك سېلىشتۇرۇش ئېغىزلىرىنى ئەمەلگە ئاشۇرمىغان بولىدۇ.

كود 4.33 زورلاپ تىپ خاتالىقى سەۋەبلىك بىنورماللىق چىقىرىش

```
var expr =
    (from c in customers
        from o in c.Orders
        select new { o.Quantity}
    ).Min();
```

مەنبە تىزمىسى قۇرۇق ياكى بارلىق ئەزالارنىڭ قىممىتى null بولۇپ قالغان ئەھۋال ئاستىدا، ئەگەر ئەندىزىدىكى تىپى Numeric تىپى قۇرۇق بولالايدىغان تىپ بولسا، نەتىجە null بولىدۇ. ئۇنداق بولمايدىكەن، *ArgumentNullException* تىپلىق بىنورماللىق قويۇپ بېرىلىدۇ. ئاخىرقى ئىككى خىل ئەنزىدىكى تاللاش كۆرسەتمىسى ئارقىلىق ئەزا ئىچىدىكى قايسى خاسلىقنى سېلىشتۇرۇشنى بەلگىلەپ بېرىپ يۇقىرىقىدەك بىنورماللىقنىڭ ئالدىنى ئالغىلى بولىدۇ. مەسىلەن:

كود 4.34 قىممەت تاللىغۇچ ئارقىلىق خېرىدار تىپلىرىغا نىسبەتەن Max مەشغۇلاتى ئېلىپ بېرىش

```
var expr =
    (from c in customers
        from o in c.Orders
        select new { o.Quantity}
).Min(o => o.Quantity);
```

Average مەشغۇلاتچىسى

مەزكۇر مەشغۇلاتچى ئەزا قىممەتلىرىنىڭ ئوتتۇرچە قىممىتىنى تاپىدۇ. ئۇ ئالدىدا سۆزلىگەن Max, Min, Sum لارغا ئوخشاش ئەزالارنىڭ قىممەتلىك تىپ بولغان ھالىتىگە مۇۋاپىق. ئۇنىداق بولمىغان ئەھۋال ئاستىدا باشقا ئەندىزىلىرى ئارقىلىق كۆپ خاسلىق چاقىرىلما تىپىنىڭ قىممەتلىك خاسلىقىنى بەلگىلەپ قويۇش كېرەك. شۇندىلا نورمال مەشغۇلات ئېلىپ بارالايدۇ. ئۇنىڭ تۆۋەندە كۆرسىتىلگەندەك ئەندىزىلىرى بار:

```
public static Result Average(
    this IEnumerable<Numeric> source);
public static Result Average<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

ئوتتۇرچە قىممىتى ئېلىنماقچى بولغان Numeric تىپ int, int?, long, long?, float, float?, double, double?, decimal, or decimal?. لارنىڭ بىرى بولۇشى كېرەك. شۇنداق بولغاندا نەتىجە مەنبە تىپقا ۋارىسلىق قىلىپلا قالماي، قۇرۇق بولالايدىغان ئالاھىدىلىكىنىمۇ ساقلاپ قالىدۇ. ئۇنىڭدىن باشقا، مەنبەدىكى ئەزا int ياكى long تىپلىق بولسا، نەتىجە double تىپلىق؛ مەنبەدىكى ئەزا int? ياكى long? تىپلىق بولسا، نەتىجە double? تىپلىق بولىدۇ.
بۇنداق بولۇشىنى چۈشىنىش تەس ئەمەس، يەنى، بىرقانچە پۈتۈن سانىنىڭ ئوتتۇرچە قىممىتى كەسىر سان چىقىشى مۇمكىن.
تۆۋەندىكىسى ئىككى خىل ئەندىزىگە بىردىن مىسال:

كود 4.35 Average نىڭ ئىككى خىل ئەندىزىسى مەسىلات باھالىرىغا مەشغۇلات ئېلىپ بارىدۇ

```
var expr =
    (from p in products
    select p.Price
    ).Average();
var expr =
    (from p in products
    select new { p.Price }
    ).Average(p => p.Price);
```

ئىككىنچى مىسالدىكى كۆپ رەڭگە بويالغان سۈرۈشتۈرۈك ئىپادىسىنىڭ نەتىجىسىگە Average نىڭ ئىككىنچى خىل ئەندىزىسىنى ئىشلىتىشتىكى سەۋەب، ئۇنىڭ نەتىجە تىزمىسىدىكى ئەزالار نامسىز تىپلىق بولغانلىقى ئۈچۈن ئۇلار ئارىسىدا بىۋاسىتە ئارفمېتىكلىق ئەمەل بىجىرگىلى بولمايدۇ. شۇڭا چوقۇم ئۇنىڭ قايسى خاسلىقىنىڭ ئوتتۇرچە قىممىتىنى lambda ئىپادىسى ئارقىلىق بەلگىلەپ بېرىشىمىز كېرەك.
تۆۋەندە بارلىق خېرىدارلار ۋە ئۇلارنىڭ زاكاز مىقدارلىرىنىڭ ئوتتۇرچە قىممىتىنى چىقىرىدىغان مىسال بېرىلدى:

كود 4.36

```
var expr =
    from    c in customers
    join    o in (
            from c in customers
                from    o in c.Orders
                join    p in products
                        on o.IdProduct equals p.IdProduct
                select new { c.Name, OrderAmount = o.Quantity * p.Price }
            ) on c.Name equals o.Name
            into customersWithOrders
    select new { c.Name,
                 AverageAmount = customersWithOrders.Average(o =>
o.OrderAmount) };
```

يۇقىرىقى مىسالنىڭ نەتىجىسى تۆۋەندىكىدەك بولۇشى مۇمكىن:

```
{Name=Paolo, AverageAmount=65}
{Name=Marco, AverageAmount=350}
{Name=James, AverageAmount=600}
{Name=Frank, AverageAmount=1000}
```

Generation مەشغۇلاتچىلىرى (قۇرغۇچ)

مەسىلەن 2000ـ يىلىدىن 2007ـ يىلىغىچە بولغان ئارىلىقتىكى زاكازلار تىزىملىكىنى چىقىرىش، ياكى ئوخشاش بىر ئۇچۇرغا ئوخشاش بىر مەشغۇلاتنى تەكرار ئېلىپ بېرىشتەك مەشغۇلاتلارنى قۇرغۇچ مەشغۇلاتچىلىرى ئىنتايىن ئەپچىللىك بىلەن ئېلىپ بارالايدۇ.

Range (دائىرە مەشغۇلاتچىسى)

ئۇ بەلگىلىك دائىرىدىكى قىممەتلەر تىزمىسى ھاسىل قىلىپ بېرىش رولىغا ئىگە بولۇپ، تۆۋەندىكى بىردىن- بىر ئەندىزىگە ئىگە:

```
public static IEnumerable<int> Range(
    int start,
    int count);
```

كود 4.40 دا 2005ـ يىلىدىن 2007ـ يىلىغىچە ئارىلىقتىكى زاكازلارنى سۈزۈپ ئېلىش مەشغۇلات كودى بېرىلدى.

ئەسكەرتىش: مەزكۇر مەسىلىگە نىسبەتەن where خاس سۆزى ئارقىلىق يىل چېكى قويۇش ئەقلىگە ئەڭ مۇۋاپىق ئۇسۇل. بۇ كود پەقەت Range نىڭ ئىشلىتىش ئۇسۇلىنى چۈشەندۈرۈش مەقسىتىدە مىسال ئۈچۈنلا بېرىلدى، لېكىن ياخشى ئۇسۇل بولۇشى ناتايىن.

كود 4.40 2005ـ يىلىدىن 2007ـ يىلىغىچە بولغان زاكازلارغا ئېرىشىش

```
var expr =
    Enumerable.Range(2005, 3)
    .SelectMany(x => (from  o in orders
                      where  o.Year == x
                      select new { o.Year, o.Amount }));
```

يۇقىرىقىلاردىن باشقا Range نى يەنە «كۆۋادىرات كۆتۈرۈش»، «ھەسسىلەش» ۋە فاكتورىيالنى ھېسابلاشتەك ئورۇنلارغىمۇ قوللىنىشقا بولىدۇ. مەسىلەن:

كود 4.41 number نىڭ پاكتېرىيالىنى Range دىن پايدىلىنىپ تېپىش

```
static int Factorial(int number) {
    return (Enumerable.Range(0, number + 1)
            .Aggregate(0, (s, t) => t == 0 ? 1 : s *= t)); }
```

Repeat (تەكرار مەشغۇلاتچىسى)

مەزكۇر مەشغۇلاتچى مەنبە تىزمىدىكى ئەزالارنى تەكرارلاپ كۆپەيتىدۇ. ئەگەر ئەزا چاقىرىلما تىپلىق بولسا، ھەر بىر ئەزانىڭ ئۆزى ئەمەس بەلكى مۇناسىپ چاقىرغۇچىسى(引用) تەكرارلىنىدۇ.
Repeat مەشغۇلاتچىسى كۆپەيتىلگەن تىزمىدىكى ئەزالارنى دەسلەپلەشتۈرۈش، ئوخشاش سۈرۈشتۈرۈكنى نەچچە قېتىم ئىجرا قىلىشتەك ئەھۋاللاردا ئىشلىتىلىدۇ. تۆۋەندىكى مىسالدا خېرىدارلارنىڭ ئىسىملىرىنى ئالىدىغان سۈرۈشتۈرۈكنى ئۈچ قېتىم تەكرار ئىجرا قىلىدۇ.

كود 4.42 سۈرۈشتۈرۈكنى تەكرار ئىجرا قىلىش

```
var expr =
    Enumerable.Repeat( (from   c in customers
                        select c.Name), 3)
.SelectMany(x => x);

foreach (var v in expr)
                Console.WriteLine(v);
```

يۇقىرىقى كودتا Repeat نىڭ نەتىجىسى تىزمىلارنىڭ تىزمىسى بولىدۇ. شۇڭا ئۇنىڭ ئۈستىدە يەنە SelectMany مەشغۇلاتچىسىنى قوللىنىپ بارلىق ئەزالارنى بىر تىزمىغا تەكشى رەتلەپ ئورۇنلاشتۇردۇق.
ئىجرا نەتىجىسى تۆۋەندىكىچە:

```
Paolo
Marco
James
Frank
Paolo
Marco
James
Frank
Paolo
Marco
James
Frank
```

Empty (قۇرۇق مەشغۇلاتچىسى)

مەزكۇر مەشغۇلاتچى مەلۇم تىپلىق ئەزانىڭ قۇرۇق تىزمىسىنى قايتۇرۇپ بېرەلەيدۇ. يەنى قۇرۇق تىزما ھاسىل قىلىشتا ئىشلىتىلىدۇ. مەسىلەن:

كود 4.43 Empty دىن پايدىلىنىپ خېرىدارلار تىپىنىڭ قۇرۇق تىزمىسىنى ھاسىل قىلىش

```
IEnumerable<Customer> customers = Enumerable.Empty<Customer>();
```

Quantifiers مەشغۇلاتچىلىرى (مىقدار مەشغۇلاتچىلىرى)

تىزما ئىچىدە مەلۇم شەرتنى قانائەتلەندۈرىدىغان ئەزانىڭ بار- يوقلۇقىنى تەكشۈرۈشمۇ ئېھتىياجىمىزنىڭ سىرتىدا ئەمەس. گەرچە بۇنداق مەسىلىلەرنى بىز ئالدىنقى مەزمۇنلاردا سۆزلەپ ئۆتكەن مەشغۇلاتچىلارنى بىرلەشتۈرۈپ ئىشلىتىش ئارقىلىق ئەمەلگە ئاشۇرغىلى بولىسىمۇ، لېكىن Linq مۇشۇنداق ئەھۋاللارغا خاس مەشغۇلاتچىلارنى تەمىنلىگەن.

Any مەشغۇلاتچىسى

تونۇشتۇرماقچى بولغان تۇنجى مەشغۇلاتچى Any بولۇپ، ئۇ بېرىلگەن راس- يالغان تىپلىق كۆرسەتمىگە ئاساسەن راس- يالغانلىق قىممەت قايتۇرىدۇ. تۆۋەندىكىلەر ئۇنىڭ ئەندىزىلىرى:

```
public static bool Any<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
public static bool Any<T>(
    this IEnumerable<T> source);
```

يۇقىرىقى مىتود ئەندىزىلىرىنى كۆرگىنىڭىزدە، كاللىڭىزغا «مەنبەدىن باشقا ھېچقانداق شەرت قوبۇل قىلمايدىغان مىتود شەكلى نېمىگە ئاساسەن راست- يالغان قىممەت قايتۇرىدۇ؟» دېگەن سۇئال كېلىشى مۇمكىن. مەزكۇر ئەندىزىدە، ئەگەر مەنبە تىزمىدا كەم دېگەندە بىر دانە ئەزا بولسا راستنى، ئۇنداق بولمىسا يالغاننى قايتۇرىدۇ. ھېسابتا، مەلۇم تىزمىنىڭ قۇرۇق ياكى ئەمەسلىكىنى تەكشۈرىدۇ.
ئىككى خىل ئەندىزىسىدە بولسا، كۆرسەتمە ئارقىلىق شەرت يوللىنىدۇ، ئەگەر تىزمىدا ئۇشبۇ شەرتنى قانائەتلەندۈرىدىغان ئەزادىن كېمىدە بىرسى بولسا راستنى، بولمىسا يالغاننى كەلتۈرىدۇ.
مەسىلەن: تۆۋەندىكى مىسالدا بارلىق خېرىدارلارنىڭ بارلىق زاكازلىرىنىڭ ئىچىدە مەھسۇلات تەرتىپ نومۇرى (idProduct) 1 گە تەڭ بولغان مەھسۇلاتنىڭ مەۋجۇت ياكى ئەمەسلىكى تەكشۈرۈلىدۇ.

كود 4.44

```
bool result =
    (from c in customers
        from   o in c.Orders
        select o)
    .Any(o => o.IdProduct == 1);
//تۆۋەندىكى قۇردا نەتىجىنى قەستەن يالغان چىقاردۇق، چۈنكى تىزما قۇرۇق//
result = Enumerable.Empty<Order>().Any(o => o.IdProduct == 1);//false
```

شۇنىسى ئېسىگىزدە بولسۇن، تەكشۈرۈش جەريانىدا **o.IdProduct == 1** قانائەتلەنگەن زامانلا ئومۇمىي ئىپادە ئاخىرلىشىدۇ.

All (ھەممە مەشغۇلاتچىسى)

All مەشغۇلاتچىسى تىزمىدىكى بارلىق ئەزالارنىڭ بېرىلگەن شەرتنى قانائەتلەندۈرىدىغان ياكى قانائەتلەندۈرمەيدىغانلىقىنى تەكشۈرۈشكە ئىشلىتىلىدۇ. ئەگەر ھەممىسى شەرتنى قانائەتلەندۈرسە راستىنى، ئۇنداق بولمىسا يالغانىنى قايتۇرىدۇ.

مەسىلەن: ھەربىر زاكاز مىقدارىنىڭ مۇسبەت سان بولۇش شەرتىنى قانائەتلەندۈرىدىغان ياكى قانائەتلەندۈرمەيدىغانلىقىنى تەكشۈرۈپ باقايلى:

كود 4.45

```
bool result =
    (from c in customers
        from o in c.Orders
        select o)
    .All(o => o.Quantity > 0);

result = Enumerable.Empty<Order>().All(o => o.Quantity > 0); //false
```

ئەسكەرتىش: ئەگەر All مەشغۇلاتچىسى قۇرۇق تىزمىغا ئىشلىتىلسە نەتىجىسى بىردەك راست بولىدۇ.

Contains (بارمۇ مەشغۇلاتچىسى)

بېرىلگەن ئەزانىڭ مەنبە تىزمىدا بار- يوقلۇقىنى تەكشۈرىدۇ. بار بولسا راستىنى، بولمىسا يالغانىنى قايتۇرىدۇ. ئۇنىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static bool Contains<T>(
    this IEnumerable<T> source,
    T value);
public static bool Contains<T>(
    this IEnumerable<T> source,
    T value,
    IEqualityComparer<T> comparer)
```

ئەگەر مەنبە تىزمىدىكى ئەزالار **ICollection<T>** ئېغىزىنى ئەمەلگە ئاشۇرغان بولسا، **Linq** ئامال بار ئۇشبۇ ئېغىزنى ئىشلىتىدۇ. چۈنكى بۇنداق بولغاندا سېلىشتۇرۇش بىرقەدەر تېز بولىدۇ.
ئەگەر ئەمەلگە ئاشۇرمىغان بولسا، نىشان ئەزا بىلەن مەنبە تىزمىدىكى ئەزالار بىرمۇ-بىر سېلىشتۇرىلىدۇ. ئەگەر ئىككىنچى ئەندىزە بويىچە شەخسى سېلىشتۇرغۇچنى يوللاپ بەرسە، شۇنى ئىشلىتىدۇ. ئۇنداق بولمىسا كۆڭۈلدىكى قىممەت بويىچە **EqualityComparer<T>.** نى ئىشلىتىدۇ.
تۆۋەندىكى مىسالدا تۇنجى خېرىدارنىڭ زاكازلىرى ئىچىدە بېرىلگەن زاكازنىڭ بار-يوقلىقى تەكشۈرۈلدى:

```
orderOfProductOne = new Order {Quantity = 3, IdProduct = 1 ,
    Shipped = false, Month = "January"};
bool result = customers[0].Orders.Contains(orderOfProductOne);
```

Partitioning مەشغۇلاتچىلىرى (پارچىلاش)

بەزىدە تىزمىنىڭ مەلۇم بۆلىكىدىكى ئەزالارغا نىسبەتەنلا مەشغۇلات ئېلىپ بېرىش توغرا كېلىدۇ. مەسىلەن: خېرىدارلار تىزمىسىدىكى ئالدىنقى N نەپەر خېرىدارنىڭ ئۇچۇرىغا ئېرىشىش دېگەندەك. ئەلۋەتتە بۇنداق مەسىلەرنى **Where** مەشغۇلاتچىسى بىلەن **Select** مەشغۇلاتچىسىغا نۆلدىن باشلىنىدىغان تەرتىپ نومۇرىنى كۆرسەتكۈچىنىڭ پارامېتىرى قىلىپ بېرىش ئارقىلىقمۇ ھەل قىلالايمىز. لېكىن ئۇنىڭ دائىم قولايلىق، بىۋاستە يول بولۇشى ناتايىن. شۇڭا **Linq** دىكى مۇشۇنداق مەسىلىلەرگە خاس مەشغۇلاتچىلارنى ئىشلىتىش ياخشى تاللاش.
بۇ تۆردىكى مەشغۇلاتچىلار ئارىسىدىكى **Take** بىلەن **TakeWhile** بولسا ئۆز يوللىرىدا ئايرىم-ئايرىم ھالدا ئالدىنقى N دانە ئەزا ياكى بېرىلگەن شەرتنى قانائەتلەندۈرگىچە ئالدىنقى مانچە ئەزانى ئالىدۇ.
Skip ۋە **SkipWhile** مەشغۇلاتچىسى **Take** بىلەن **TakeWhile** مەشغۇلاتچىسىنىڭ تولۇقلىمىسى بولۇپ، ئالدىنقى N ئەزانى ياكى بېرىلگەن شەرتنى قانائەتلەندۈرىدىغان ئالدىنقى مانچە ئەزانى ئاتلاپ ئۆتۈپ كېتىدۇ.

Take (نى ـ ئېلىش مەشغۇلاتچىسى)

تۆۋەندىكى ئەندىزىگە قاراڭ:

```
public static IEnumerable<T> Take<T>(
    this IEnumerable<T> source,
    int count);
```

دېمەك Take مەشغۇلاتچىسى مەنبە تىزمىدىكى ئالدىنقى count دانە ئەزانى قايتۇرۇپ بېرىدۇ. ئەگەر كۆزلىمە مىقدار نۆل بولسا قۇرۇق تىزمىنى، مەنبە تىزما ئۇزۇنلۇقىدىن چوڭ بولسا مەنبە تىزمىنىڭ ئۆزىنى قايتۇرىدۇ. بۇ مەشغۇلاتچى «خېرىدارلار ئارىسىدىن زاكاز پۇل مىقدارى ئەڭ چوڭ بولغان ئالدىنقى 2 خېرىدارنى تاللاش» دېگەندەك مەسىلىلەرگە تولىمۇ ماس كېلىدۇ. مەسىلەن:

كود 4.47

```
var topTwoCustomers =
    (from     c in customers
     join     o in (
              from c in customers
                  from    o in c.Orders
                  join    p in products
                          on o.IdProduct equals p.IdProduct
                  select new { c.Name, OrderAmount = o.Quantity * p.Price }
              ) on c.Name equals o.Name
              into customersWithOrders
     let      TotalAmount = customersWithOrders.Sum(o => o.OrderAmount)
     orderby TotalAmount descending
     select   new { c.Name, TotalAmount }
    ).Take(2);
```

كۆرگەنسىز! گەرچە ئومۇمى سۈرۈشتۈرۈك ئىپادىسى ئىنتايىن چىرماش بولغان بىلەن، Take نى ئورۇنلاشتۇرۇش ناھايىتى ئاددىي. يۇقىرىقى ئىپادىدە بۇرۇن سۆزلىگەن سۈرۈشتۈرۈك خاس سۆزلىرىنى ئىشلىتىش بىلەن بىر ۋاقىتتا يېڭى خاس سۆز let مۇ ئىشلىتىلدى. let خاس سۆزى ئىپادە ئىچىدە مەلۇم ھەرپ ـ بەلگە تىمسىنى مەلۇم قىممەتكە ياكى مەلۇم فورمىلا قىممىتىگە تەگداش قىلىش رولىنى ئوينايدۇ. زاغرا تىل بويىچە ئىپادە ئىچىدە يەرلىك ئۆزگەرگۈچى مىقدار ئېنىقلايدۇ دېسەكمۇ بولىدۇ(بۇ زاغرا تىل ئەمەسمۇ يا...🤔). مەسىلەن: يۇقىرىقى مىسالىمىزدا زاكازى بار خېرىدارنىڭ بارلىق زاكازلىرىنىڭ قىممەت يىغىندىسىنى TotalAmout دېگەنگە ساقلاپ تۇردۇق.

TakeWhile (چاغدا ـ ئېلىش مەشغۇلاتچىسى)

ئىشلىتىش ئۇسۇلى Take بىلەن ئوخشىشىپ كېتىدىغان بولۇپ، بۇنىڭدا ئالدىنقى مانچىسىنى ئەمەس، بەلكى ئالدىنقى ئەزالارنى تاكى شەرت قانائەتلەنمىگىچە ئالىدۇ. تۆۋەندىكىلەر ئۇنىڭ ئەندىزىلىرى:

```
public static IEnumerable<T> TakeWhile<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
public static IEnumerable<T> TakeWhile<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

بىرىنچى خىل ئەندىزىسىدە، مەنبە تىزمىنىڭ نۆلىنچى ئەزاسىدىن باشلاپ چارلاش ئېلىپ ئۇنىڭ كۆرسىتىكۇچتىكى شەرتكە چۈشىدىغانلىقى چۈشمەيدىغانلىقىنى تەكشۈرىدۇ، ئەگەر چۈشسە(كۆرسەتمە قىممىتى راست بولسا) ئۇنى ئۇ نەتىجە توپلىمىغا قوشۇپ بولۇپ كېيىنكى ئەزانى ئوخشاش ئۇسۇلدا تەكشۈرىدۇ. بۇ جەريان تاكى مەلۇم ئەزا شەرتكە چۈشمىگەنگە ياكى ئەڭ ئاخىرقى ئەزانى تەكشۈرۈش تاماملانغانغا قەدەر داۋاملىشىدۇ.
ئىككىنچى خىل ئەندىزىسىدە پارامېتىر ئارقىلىق بېرىدانە پۈتۈن سان يوللاپ بېرىمىز، چارلاش مەشغۇلاتى ئۇشبۇ سان تەرتىپلىك ئەزادىن باشلىنىدۇ.
ئۇنىڭدىن باشقا شەرت كۆرسەتمىسىنى مۇۋاپىق تۈزۈش ئارقىلىق تېخىمۇ مۇرەككەپ مەسىلىلەرنى ھەل قىلالايمىز. مەسىلەن: تۆۋەندىكى مىسالدا زاكاز مىقدارى قىممىتى ئومۇمى قىممىتىنىڭ 80% تىنى ئىگىلەيدىغان ئالدىنقى خېرىدارلارغا ئېرىشىدۇ:

كود 4.48

```
//
var limitAmount = globalAmount * 0.8m;
var aggregated = 0m;
var topCustomers =
    (from    c in customers
     join    o in (
             from c in customers
                 from   o in c.Orders
                 join   p in products
                        on o.IdProduct equals p.IdProduct
                 select new { c.Name, OrderAmount = o.Quantity *
p.Price }
             ) on c.Name equals o.Name
             into customersWithOrders
     let     TotalAmount = customersWithOrders.Sum(o => o.OrderAmount)
```

```
    orderby TotalAmount descending
    select  new { c.Name, TotalAmount }
   )
   .TakeWhile( X => {
                 bool result = aggregated < limitAmount;
                 aggregated += X.TotalAmount;
                 return result;
             } );
```

Skip بىلەن SkipWhile

Skip بىلەن SkipWhile نىڭ ئەندىزىلىرى Take بىلەن TakeWhile لەرنىڭكىگە ئىنتايىن ئوخشىشىپ كېتىدۇ. يەنى:

```
public static IEnumerable<T> Skip<T>(
    this IEnumerable<T> source,
    int count);
public static IEnumerable<T> SkipWhile<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
public static IEnumerable<T> SkipWhile<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

يايام تېخى بۇ مەشغۇلاتچىلار Take بىلەن TakeWhile نىڭ تولۇقلىغۇچىلىرى دەپ ئېيتقانىدۇق. ئەمەلىيەتتە ، تۆۋەندىكى كود خېرىدارلارنىڭ تولۇق تىزمىسىنى قايتۇرىدۇ:

```
var result = customers.Take(3).Union(customers.Skip(3));
var result = customers.TakeWhile(p).Union(customers.SkipWhile(p));
```

ئېلېمېنت مەشغۇلاتچىلىرى

ئېلېمېنت مەشغۇلاتچىلىرى تىزما ئىچىدىكى مەلۇم بىرلا ئەزاغا ئالاقىلدار مەشغۇلاتنى ئېلىپ بارىدىغان بولۇپ، مەلۇم ئورۇندىكى ياكى شەرتكە ئۇيغۇن بىرلا ئەزانى قايتۇرىدۇ. ئەگەر تاپالمىسا كۆڭۈلدىكى قىممەت ئەزاسىنى قايتۇرىدۇ.

First (تۇنجى مەشغۇلاتچىسى)

First مەشغۇلاتچىسى تىزمىدىكى تۇنجى ياكى (ئەگەر كۆرسەتمە شەرتى بېرىلسە) شەرتنى قانەئەتلەندۈرىدىغان ۋە ياكى ئورۇن قائىدىسىگە چۈشىدىغان تۇنجى ئەزانى قايتۇرىدۇ:

```
public static T First<T>(
    this IEnumerable<T> source);
public static T First<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

بىرىنچى خىل ئەندىزىسىدە مەنبە تىزمىنىڭ بىرىنچى ئېلېمېنتىنى قايتۇرىدۇ. ئىككىنچى خىلىدا بېرىلگەن شەرتكە چۈشىدىغان تۇنجى ئېلېمېنتنى قايتۇرىدۇ. ئەگەر شەرتكە چۈشىدىغان ئېلېمېنت تېپىلمىسا ياكى مەنبە تىزما قۇرۇق بولسا، مەشغۇلاتچى **InvalidOperationException** تىپلىق بىنورماللىق قويۇپ بېرىدۇ. تۆۋەندە بىرمىسال:

كود 4.49 دۆلەت تەۋەلىكى USA بولغان تۇنجى خېرىدارغا ئېرىشىش

```
var item = customers.First(c => c.Country == Countries.USA);
```

ئەلۋەتتە يۇقىرىقى مەقسەتنى تۆۋەنكى ئۇسۇل ئارقىلىقمۇ ئەمەلگە ئاشۇرالايمىز:

```
var item = customers.Where(c => c.Country == Countries.USA).Take(1);
```

بىراق First ئىپادە مەقسەتنى تېخىمۇ ئېنىق ئىپادىلىيەلەيدۇ.

FirstOrDefault

FirstOrDefault نى «تۇنجىسى بولمىسا كۆڭۈلدىكىنى» دەپ تەرجىمە قىلسام مۇۋاپىق دەپ ئويلىدىم. ئۇنىڭ مەشغۇلات پرىنسىپى First` بىلەن ئوخشاش بولۇپ، بىردىن- بىر پەرقى. ئەگەر شەرتكە ئۇيغۇن ھېچقانداق ئېلمېنت تېپىلمىسا مەنبە ئەزالىرىنى چاقىرىلما تىپلىق null نى، قىممەتلىك تىپلىق بولسا شۇ تىپنىڭ **nullable** ئىنى قايتۇرىدۇ. تۆۋەندە بىر مىسال:

كود 4.50

```
var item = customers.FirstOrDefault(c => c.City == "Las Vegas");
Console.WriteLine(item == null ? "null" : item.ToString()); // null

IEnumerable<Customer> emptyCustomers = Enumerable.Empty<Customer>();
item = emptyCustomers.FirstOrDefault(c => c.City == "Las Vegas");
Console.WriteLine(item == null ? "null" : item.ToString()); // null
```

Last بىلەن LastOrDefault

Last بىلەن LastOrDefault لار بولسا First بىلەن FirstOrDefault لارغا ئوخششىپ كېتىدىغان بولۇپ، First بىلەن FirstOrDefault تۈنجىسىنى ئالسا، بۇلار ئەڭ ئاخىرىدىكىنى ئالىدۇ. باشقا بارلىق قۇرۇلمىلىرى ئوپمۇ- ئوخشاش. تۆۋەندىكىلىرى ئۇلارنىڭ ئەندىزىلىرى:

```
public static T Last<T>(
    this IEnumerable<T> source);
public static T Last<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
public static T LastOrDefault<T>(
    this IEnumerable<T> source);
public static T LastOrDefault<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Single

ئەگەر تىزمىدىن بىرتال ئەزانى شۇنداقلا (مەسىلەن: تىزمىلارنىڭ ۋەكىلى سۈپىتىدە) ئالماقچى بولسىڭىز Single مەشغۇلاتچىسىنى ئىشلىتىڭ. ئۇنىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static T Single<T>(
    this IEnumerable<T> source);
public static T Single<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

ئەگەر كۆرسەتكۈچ شەرتى بېرىلمىسە، قايتىدىغىنى تىزمىدىكى بىرىنچى ئېلېمېنت بولىدۇ. ئۇنداق بولمىغاندا شەرتكە چۈشىدىغان بىرتال ئېلېمېنت بولىدۇ. ئەگەر كۆرسەتكۈچ بولمىسا ھەمدە تىزمىدا بىردىن ئارتۇق ئېلېمېنت بولسا InvalidOperationException تىپلىق بىنورماللىق قويۇپ بېرىلىدۇ.

ئەگەر كۆرسەتكۈچ بېرىلسە لېكىن تىزمىدا شەرتكە چۈشىدىغان بىرمۇ ئېلېمېنت بولمىسا ياكى تىزما قۇرۇق بولسا InvalidOperationException تىپلىق بىنورماللىق چىقىرىدۇ. تۆۋەندىكىلەر ئوخشىمىغان ئەھۋاللارغا ماس مىساللار:

كود 4.51

```
// نى قايتۇرىدۇ Product 1
var item = products.Single(p => p.IdProduct == 1);
Console.WriteLine(item == null ? "null" : item.ToString());

//پېشكىلى چىقىرىدۇ InvalidOperationException
item = products.Single();
Console.WriteLine(item == null ? "null" : item.ToString());

//پېشكىلى چىقىرىدۇ InvalidOperationException
IEnumerable<Product> emptyProducts = Enumerable.Empty<Product>();
item = emptyProducts.Single(p => p.IdProduct == 1);
Console.WriteLine(item == null ? "null" : item.ToString());
```

SingleOrDefault

SingleOrDefault مەشغۇلاتچىسى قۇرۇق ياكى شەرتكە ئۇيغۇن ئېلېمېنت بولمىغان تىزمىدىن كۆڭۈلدىكى قىممەتنى قايتۇرىدۇ. بۇ يەردىكى «كۆڭۈلدىكى قىممەت» FirstOrDefault مەشغۇلاتچىسىدا سۆزلەنگىنى بىلەن ئوخشاش بىلەن ئوخشاش.

ئەسكەرتىش: default قىممەت پەقەت شەرتكە ئۇيغۇن ئېلېمېنت تېپىلمىغاندىلا قايتىدۇ. ئەگەر تىزمىدا شەرتكە ئۇيغۇن بىردىن ئارتۇق ئېلېمېنت بولۇپ قالسا مەشغۇلاتچى InvalidOperationException تىپلىق بىنورماللىق چىقىرىدۇ.

ElementAt بىلەن ElementAtOrDefault

ElementsAt بىلەن ElementAtOrDefault تىزمىنىڭ بەلگىلەنگەن ئورۇندىكى ئېلېمېنتنى ئېلىش ئۈچۈن ئىشلىتىلىدۇ:

```
public static T ElementAt<T>(
    this IEnumerable<T> source,
    int index);
public static T ElementAtOrDefault<T>(
    this IEnumerable<T> source,
    int index);
```

ElementAt تە، پارامېتر ئارقىلىق قايسى ئورۇندىكى ئېلېمېنتنى ئېلىشنى كۆرسىتىپ بېرىش كېرەك. تەرتىپ نومۇرى نۆلدىن باشلىنىدۇ. دېمەك، ئىككىنچى يوللىسىڭىز ئۈچىنچى ئېلېمېنتىغا ئېرىشىسىز. ئەگەر بەرگەن سانىڭىز مەنفى بولسا ياكى تىزما سان چەكلىمىسىن ئېشىپ كەتسە، مەشغۇلات ArgumentOutOfRangeException تىپلىق بىنورماللىق چىقىرىدۇ.
ElementAtOrDefault تا بولسا، ElementAt تىكى بىنورماللىق قويۇپ بېرىدىغان ئەھۋاللاردا، بىنورماللىق قويۇپ بېرىشنىڭ ئورنىغا كۆڭۈلدىكى قىممەت قايتۇرىدۇ. قايتۇرۇش پرىنسىپى FirstOrDefault نىڭكى بىلەن ئوخشاش. تۆۋەندە ئۇلارنىڭ ئىشلىتىلىشىگە ئائىت مىسال بېرىلدى:

كود 4.52

```
// Product 2 نى قايتۇرىدۇ //
var item = products.ElementAt(2);
Console.WriteLine(item == null ? "null" : item.ToString());

// null قايتۇرىدۇ //
item = Enumerable.Empty<Product>().ElementAtOrDefault(6);
Console.WriteLine(item == null ? "null" : item.ToString());

// null قايتۇرىدۇ //
item = products.ElementAtOrDefault(6);
Console.WriteLine(item == null ? "null" : item.ToString());
```

DefaultIfEmpty

DefaultIfEmpty قۇرۇق تىزما ئۈچۈن كۆڭۈلدىكى قىممەتنى قايتۇرىدۇ:

```
public static IEnumerable<T> DefaultIfEmpty<T>(
    this IEnumerable<T> source);
public static IEnumerable<T> DefaultIfEmpty<T>(
    this IEnumerable<T> source,
    T defaultValue);
```

بەلگىلىمىسى بويىچە، ئۇ مەنبە تىزمىدىكى ئەزالار توپىنى قايتۇرىدۇ. ئەگەر مەنبە تىزما قۇرۇق بولسا، بىرىنچى ئەندىزىدە default(T) نى، ئىككىنچى ئەندىزىسىدە پارامېتر ئارقىلىق بېرىلگەن defaultValue نى قايتۇرىدۇ.
خاسلاشتۇرۇلغان كۆڭۈلدىكى قىممەت قۇرۇۋېلىشنىڭ پايدىسى كۆپ. مەسىلەن، ھېچقانداق ئۇچۇرى بولمىغان قۇرۇق خېرىدار لازىم بولغاندا Empty ناملىق خاسلىق ئارقىلىق ئېرىشكىلى بولىدىغان قىلساق مۇنداق يازىمىز:

```
public static Customer Empty {
    get {
        Customer empty = new Customer();
        empty.Name = String.Empty;
        empty.Country = Countries.Italy;
        empty.City = String.Empty;
        empty.Orders = (new
List<Order>(Enumerable.Empty<Order>())).ToArray();
       return(empty);
    }
}
```

بەزىدە بۇنداق قىلىش ياخشى ئىش، بولۇپمۇ بۆلەك سىنىقى ئېلىپ بارغاندا ئىنتايىن ئەس قاتىدۇ. ئۇنىڭدىن باشقا، سۈرۈشتۈرۈك **GroupJoin** نى ئىشلىتىپ سول- سىرتقى ھەمدەمنى بايقىغاندا، **Null** بولۇش ئېھتىماللىقى بولغان نەتىجىنى كۆڭۈلدىكى قىممەتكە ئايلاندۇرۇۋېتىش ياخشى ئادەت.

تۆۋەندە **DefaultEmpty** نى ئىشلىتىشتىن بىر مىسال:

كود 4.53

```
var expr = customers.DefaultIfEmpty();

var customers = Enumerable.Empty<Customer>(); // Empty array
IEnumerable<Customer> customersEmpty =
    customers.DefaultIfEmpty(Customer.Empty);
```

باشقا مەشغۇلاتچىلار

Concat بىلەن **SequanceEqual** مەزكۇر پاراگرافتىكى ئەڭ ئاخىرقى مەشغۇلاتچىلار.

Concat (ئۇلاش مەشغۇلاتچىسى)

Concat بىر تىزمىنى يەنە بىر تىزمىغا ئۇلاش ئېلىپ بارىدىغان بولۇپ ئۇنىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static IEnumerable<T> Concat<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

كۆرىۋېلىشقا بولىدۇ، Concat ئارقىلىق **IEnumerable<T>** تىپلىق خالىغا ئىككى دانە تىزمىنى بىر بىرىگە ئۇلىيالايمىز.

تۆۋەندىكى مىسالدا، دۆلەت تەۋەلىكى **Italy** بولغان خېرىدارلار تىزمىسىنى دۆلەت تەۋەلىكى **USA** بولغان خېرىدارلار تىزمىسغا ئۇلاش ئېلىپ بارىدۇ:

كود 4.54

```
var italianCustomers =
    from    c in customers
    where   c.Country == Countries.Italy
    select c;

var americanCustomers =
    from    c in customers
    where   c.Country == Countries.USA
    select c;

var expr = italianCustomers.Concat(americanCustomers);
```

SequenceEqual

سېلىشتۇرۇش مەشغۇلاتچىسى يەنە بىر قوللىنىشچان مەشغۇلاتچى بولۇپ، **SequenceEqual** ئارقىلىق ئەمەلگە ئاشۇرۇلغان:

```
public static bool SequenceEqual<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
public static bool SequenceEqual<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second,
    IEqualityComparer<T> comparer);
```

سېلىشتۇرۇش جەريانىدا مەنبە تىزمىدىكى ھەربىر ئەزا نىشان تىزمىدىكى ئۆزى بىلەن ئوخشاش ئورۇندىكى ئەزا بىلەن سېلىشتۇرۇلۇپ چىقىدۇ. ئەگەر ئىككى تىزمىنىڭ بارلىق ئەزالىرى ئورۇن جەھەتتىن ۋە سان جەھەتتىن ئوخشاش چىقسا ئاندىن راست قايتىدۇ، خالىغان بىرسى قانائەتلەنمىسە يالغان قايتىدۇ. ئەگەر تىزمىدىكى ئەزالار چاقىرىلما تىپلىق بولسا، ئۇنىڭ سېلىشتۇرۇلۇق خۇلقىنى قول ئارقىلىق ئۆزگەرتىشىڭىز كېرەك. ياكى ئىككىنچى خىل ئەندىزە ئارقىلىق خاسلاشتۇرۇلغان سېلىشتۇرغۇچ بىلەن تەمىنلىسىڭىزمۇ بولىدۇ. (بۇ توغرىلىق ئالدىنقى مەزمۇنلاردا توختالغان)

كېچىكتۈرۈلمە سۈرۈشتۈرۈكنىڭ قىممەتلىنىشى ۋە كېڭەيتىلمە مېتود چارىسى

[Deferred Query Evaluation and Extension Methods Resolution, 延迟的查询赋值与扩展方法]

ئالماشتۇرۇش مەشغۇلاتچىلىرى (مۇشۇ پاراگرافنىڭ ئاخىرىدا سۆزلىنىدۇ) سۈرۈشتۈرۈك نەتىجىسىنى ئۇخشىمىغان شەكىللەردە ئالماشتۇرالايدۇ. بۇ مەشغۇلاتچىلارغا بولغان ئېھتىياجنى چۈشەندۈرۈش ئۈچۈن Linq نىڭ «كېچىكتۈرۈلمە سۈرۈشتۈرۈكنىڭ قىممەتلىنىشى» ۋە «كېڭەيتىلمە مېتود چارىسى» لاردىن ئىبارەت Linq نىڭ بارلىق ئەمەلگە ئاشۇرۇلمىلىرى ئۈچۈن ئىنتايىن مۇھىم بولغان ئىككى خۇلقىنى بايان قىلىش زۆرۈر. شۇڭا مەزكۇر پاراگرافنى تېخىمۇ ئەستايىدىللىق بىلەن كۆرۈشىڭىزنى ئۈمىد قىلىمەن.

كېچىكتۈرۈلمە سۈرۈشتۈرۈكنىڭ قىممەتلىنىشى

Linq سۈرۈشتۈرۈكلىرى ئېنىقلانغاندا ئەمەس بەلكى ئىشلىتىلگەندىلا ئاندىن ئىجرا بولىدۇ. تۆۋەندىكى سۈرۈشتۈرۈك ئىپادىسىگە قاراڭ:

```
List<Customer> customersList = new List<Customer>(customers);

var expr =
    from    c in customersList
    where   c.Country == Countries.Italy
    select c;
```

يۇقىرىقى كود دۆلەت تەۋەلىكى Italy بولغان خېرىدارلار تىزىمىسىغا ئېرىشىدۇ. ئەگەر

Console.WriteLine("\nItems after query definition: {0}", expr.Count());

نى ئىجرا قىلساق، بىزنىڭ بۇرۇن بېكىتتۇالغان مىسال ئۈچۈرلىرىمىزغا ئاساسەن،نەتىجە ئىككى چىقىدۇ، يەنى ئىپادىدىكى شەرتكە چۈشىدىغان ئىككى خېرىدار بار. ئەمدى يېڭى خېرىدارلارنى قوشۇش ئارقىلىق مەنبە تىزمىنى ئۆزگەرتەيلى (يۇقىرىقى كودنىڭ كەينىدىلا):

```
customersList.Add(
  new Customer {Name = "Roberto", City = "Firenze",
    Country = Countries.Italy, Orders = new Order[] {
    new Order {Quantity = 3, IdProduct = 1 , Shipped = false,
      Month = "March"}}});
```

ئەمدى نەتىجە تىزمىسى expr نىڭ چارلىساق ياكى ئۈنىڭدىكى ئەزا سانىنى تەكشۈرسەك، نەتىجە ئالدىنقى قېتىمدىكىدەك چىقمايدۇ (ئالدىنقى قېتىم ئىككى ئىدى). يەنى تۆۋەندىكى كودنى ئىجرا قىلىپ كۆرسەك:

كود 4.55

```
Console.WriteLine("\nItems after source sequence modification: {0}",
 expr.Count());

foreach (var item in expr) {
    Console.WriteLine(item);
}
```

نەتىجە تۆۋەندىكىدەك چىقىدۇ. دىققەت قىلىڭ، گەرچە Roberto ئىسىملىك خېرىدار سۈرۈشتۈرۈك ئىپادىسىنى ئىنىقلاپ (يېزىپ) بولغاندىن كېيىن قوشۇلغان بولسىمۇ، لېكىن نەتىجىدە يەنىلا بار.

```
Items after source sequence modification: 3
Paolo - Brescia  -  Italy
Marco - Torino  -  Italy
Roberto - Firenze - Italy
```

ئەسكەرتىش: Linq سۈرۈشتۈرۈكىنىڭ لوگىكىسىدىن قارىغاندا، ئۇنىڭ ئىپادىلەيدىغىنى «سۈرۈشتۈرۈك پىلانى» خالاس. چۈنكى ئۇ تاكى ئۇنى ئىشلەتكىچە ئىجرا بولمايدۇ. ھەر قېتىم قايتا ئىشلىتىش ئۇنىڭ قايتا ئىجرا بولۇشىنى كەلتۈرۈپ چىقىرىدۇ. بەزى Linq نىڭ ئەمەلگە ئاشۇرمىلىرى، مەسىلەن: Linq to Objects، ئۇشبۇ خۇلقىنى مۇۋەققەتلەر(delegates) ئارقىلىق ئەمەلگە ئاشۇرغان. لېكىن Linq to Sql بولسا ئۇنى ئىپادە دەرىخى ئارقىلىق ئەمەلگە ئاشۇرغان.

«كىچىكتۈرۈلمە سۈرۈشتۈرۈكنىڭ قىممەتلىنىشى» نى بىرلا قېتىم ئىنىقلاپ تەكرار ئىشلەتكىلى بولغانلىقى ئۈچۈن، ئۇ نۇرغۇن قولايلىقلارنى ئېلىپ كەلگەن. مەنبە تىزما مەزمۇنىغا قانداق ئۆزگىرىشلەرنىڭ بولۇشىدىن قەتئىي نەزەر، سۈرۈشتۈرۈك نەتىجىسى ھامان مەنبە تىزمىنىڭ ئەڭ يېڭى مەزمۇنىنى ئاساس قىلىدۇ.
بىراق، بۇنىسى دائىم قولايلىق بولۇشى ناتايىن. ئەگەر ئوخشىمىغان پەيتتىكى مەنبە تىزمىغا نىسبەتەن سۈرۈشتۈرۈك ئىپادە نەتىجىسىنى ساقلاپ قالماقچى بولسىڭىزچۇ؟... ئالماشتۇرۇش مەشغۇلاتچىلىرى ياردەم قىلىدۇ.

كېڭەيتىلمە مېتود چارىسى

كېڭەيتىلمە مېتود چارىسى Linq نى ئىگىلەش جەريانىڭىزدىكى ئەڭ مۇھىم ئۇقۇملارنىڭ بىرى. تۆۋەندىكى كودقا قاراڭ. ئۇنىڭدا، Customers نامىلىق ئۆزىمىزنىڭ خېرىدارلار تىزما تىپىنى ئېنىقلىدۇق. ئۇنىڭدىن باشقا Customers نى كېڭەيتىلگەن مېتودلار بىلەن تەمىنلەيدىغان CutomersEnumerable نامىلىق تۈر قۇردۇق. ئۇنىڭ ئىچىدە Customers ئالاھىدە مەشغۇلات ئېلىپ بارىدىغان Where نامىلىق كېڭەيمە مېتودنى ئېنىقلىدۇق:

```
public sealed class Customers: List<Customer> {
    public Customers(IEnumerable<Customer> items): base(items) {}
}

public static class CustomersEnumerable {
    public static IEnumerable<Customer> Where(
        this Customers source, Func<Customer, bool> predicate) { ... }

    public static IEnumerable<Customer> Where(
        this Customers source, Func<Customer, int, bool> predicate)
 { ... }
}
```

ئەگەر باشتىن بىرى ئىشلىتىپ كېلىۋاتقان customers تىزمىسىنى ئىشلىتىپ كود 4.56 دىكىدەك سۈرۈشتۈرۈك ئىپادىسى يازساق:

كود 4.56

```
Customers customersList = new Customers(customers);

var expr =
    from    c in customersList
    where   c.City == "Brescia"
    select c;

foreach (var item in expr) {
    Console.WriteLine(item);
}
```

كود ـ تەرجىمان سۇرۇشتۇرۇك ئىپادىسىنى تۆۋەندىكى كودقا ئالماشتۇرىدۇ.

```
var expr =
    customersList
    .Where(c => c.City == "Brescia")
    .Select(c => c);
```

بىز بايام قۇرىۋالغان Customers تۈرىمىز ئۆچۈن تۈزگەن CustomersEnumerable تۇراقلىق تۈرىدە Where ناملىق كېڭەيتمە مېتودنىڭ ئېنىقلىمىسى بىرلىگەنلىكى ئۈچۈن، يۇقىرىقى ئالماشتۇرۇش جەريانىدا ئىشلىتىلگەن Where(c => c.City == "Brescia")بولسا System.Linq.Enumerable دا تەمىنلەنگەن، تىزىملارغا ئورتاق ئىشلىتىلىدىغان Where بولماستىن بەلكى CustomersEnumerable دىكى كېڭەيمە مېتودتۇر.(CustomersEnumerable تۈرى چوقۇم مەزكۇر تۈر ئىچىدە بولۇشى، ياكى چاقىرىلغان بولۇشى كېرەك). ئەمدى Linq كۈچىنىڭ ھەققى ماھىيىتىنى چۈشەنگەندەك بولىۋاتقانسىز؟...

كېڭەيتىلمە مېتود ئارقىلىق نۆۋەتتە بار بولغان تۈرلەرگە ۋارىسلىق قىلماي تۈرۈپمۇ ئۇنىڭ ئىقتىدارىنى يۇقىرى كۆتۈرەلەيمىز. ئەمەلىيەتتە، بىز كېيىن سۆزلىمەكچى بولغان Linq to Sql, Linq to XML لەر بولسا، سۈرۈشتۈرۈك مەشغۇلاتچىلىرىنىڭ قايتا ئەمەلگە ئاشۇرۇلۇپ خاسلاشتۇرۇلغانلىرىدۇر، خالاس.

ئالماشتۇرۇش مەشغۇلاتچىلىرى

ئالماشتۇرۇش مەشغۇلاتچىلىرىغا تەۋە مەشغۇلاتچىلاردىن AsEnumerable، ToArray، ToList، ToDictionary، ToLookup، OfType ۋە Cast قاتارلىقلار. ئالماشتۇرۇش مەشغۇلاتچىلىرى بىز ئالدىنقى مەزمۇنلاردا ئۇچرىغان بىر قىسىم مەسىلىلەرنى ھەل قىلىش يۈزىسىدىن ئورۇنلاشتۇرۇلغان. بەزىدە بىز مۇقىم، ئىممونىتچانلىقى كۈچلۈك(ئاسان ئۆزگەرمەيدىغان) بولغان سۈرۈشتۈرۈك نەتىجىلىرىگە ئېھتىجاج بولۇپ قالساق، يەنە بەزىدە سۈرۈشتۈرۈك خاس سۆزلىرىنىڭ ئورنىغا كۆپماس كېڭەيتمە مېتودلارنى ئىشلىتىپ قېلىشىمىز مۇمكىن...

AsEnumerable

AsEnumerable نىڭ ئەندىزىلىرى تۆۋەندىكىچە:

```
public static IEnumerable<T> AsEnumerable<T>(
    this IEnumerable<T> source);
```

AsEnumerable مەشغۇلاتچىسى مەنبەنى ئاددى ھالدا IEnumerable<T> نىڭ ئوبىكتى ھالىتىدە قايتۇرۇپ بېرىدۇ. بۇنىڭغا ئوخشاش چاقماق تېزلىكىدىكى ئالماشتۇرۇش بىزنى مەنبە ئۈستىدە

ئالدىنقى مەزمۇنلاردا سۆزلەپ كېلىۋاتقان كېڭەيتىلمە مېتودلارنى قوللىنىش ئىمكانىيىتىگە ئىگە قىلىدۇ. مەسىلەن: تۆۋەندىكى مىسالغا قاراڭ:

كود 4.57

```
Customers customersList = new Customers(customers);

var expr =
    from   c in customersList.AsEnumerable()
    where  c.City == "Brescia"
    select c;

foreach (var item in expr) {
    Console.WriteLine(item);
}
```

كود 4.57 دا بىز System.Linq.Enumerable دا IEnumerable<T> ئۈچۈن ئېنىقلانغان ئۆلچەملىك Where مەشغۇلاتچىسىنى ئىشلەتتۇق.

ToArray بىلەن ToList

ToArray بىلەن ToList لار بولسا ناھايىتى قوللىنىشچان ئالماشتۇرۇش مەشغۇلاتچىلىرى بولۇپ ئايرىم-ئايرىم ھالدا، IEnumerable<T> تىپلىق مەنبە تىزمىنى T نىڭ رىتى []T گە ۋە T نىڭ كۆپماس تىزمىسى List<T> غا ئالماشتۇرىدۇ.

```
public static T[] ToArray<T>(
    this IEnumerable<T> source);
public static List<T> ToList<T>(
    this IEnumerable<T> source);
```

بۇ مەشغۇلاتچىلارنىڭ نەتىجىسى مەنبە تىزمىنىڭ ئالماشتۇرۇلغان ئەمەلى كۆپەيمىسى بولۇپ، ئەگەر ئۇلار سۈرۈشتۈرۈك ئىپادىسىنىڭ ئىچىدە ئىشلىتىلگەندە نەتىجە مەنبە تىزمىنىڭ ئۆزگىرىشىگە ئەگىشىپ ئۆزگەرمەيدۇ. تۆۋەندە ToList نىڭ ئىشلىتىشتىن بىر مىسال:

كود 4.58

```
List<Customer> customersList = new List<Customer>(customers);

var expr =
    from   c in customersList.ToList()
    where  c.Country == Countries.Italy
    select c;
```

```
foreach (var item in expr) {
    Console.WriteLine(item);
}
```

سۈرۈشتۈرۈك ئىپادىسىنىڭ نەتىجىسىنى قايتا- قايتا چارلاشقا توغرا كەلگەندە بۇ مەشغۇلاتچىلار ناھايىتى ئەس قاتىدۇ. چۈنكى نورمال ھالەتتە، سۈرۈشتۈرۈك ئىپادە نەتىجىسىنى ھەربىر قېتىم چارلاش ئىپادىنىڭ بىر قېتىم ئىجرا بولۇشىنى كەلتۈرۈپ چىقىرىدۇ. ئەگەر ئىپادە ساندانغا مەشغۇلات ئېلىپ بېرىۋاتقان بولسا، ئېنىقكى ئۈنۈمدىن ئۈتتۈرۈپ قويىمىز. شۇڭا بۇنداق ئەھۋاللاردا سۈرۈشتۈرۈك ئىپادە نەتىجىسىنى **ToList** ياكى **ToArray** ئارقىلىق ئەزالارنىڭ ئەمەلىي توپلىرىغا ئايلاندۇرىۋېلىپ ئاندىن مەشغۇلات ئېلىپ بارغان تۈزۈك.

تۆۋەندىكى مىسالدا **ToList** ئارقىلىق مەھسۇلاتلارغا ئېلىپ بېرىلغان سۈرۈشتۈرۈك نەتىجىسى كۆچۈرىۋالىدۇ:

كود 4.59

```
var productsQuery =
    (from p in products
     where p.Price >= 30
     select p)
    .ToList();

var ordersWithProducts =
    from c in customers
        from   o in c.Orders
        join   p in productsQuery
               on o.IdProduct equals p.IdProduct
        select new { p.IdProduct, o.Quantity, p.Price,
                         TotalAmount = o.Quantity * p.Price};

foreach (var order in ordersWithProducts) {
    Console.WriteLine(order);
}
```

ئەمدى **ordersWithProducts** ئىپادە نەتىجىسىنى چارلىغاندا، **foreach** بۆلىكىدىكى **productsQuery** قايتا- قايتا ئىجرا بولمايدىغان بولدى.(بۇلار بۇرۇنقى مەزمۇنلارنىڭ مىسالىدىكى ئىپادىلەرنىڭ ئىسملىرى)

ToDictionary

ToDictionary بولسا Dictionary<K, T> قۇرۇدىغان مەشغۇلاتچى بولۇپ، keySelector كۆرسەتكۈچىسى ھەربىر ئەزانىڭ ئاچقۇچلۇق سۆزىنى تەكشۈرىدۇ. elementSelector بولسا(ئەگەر تەمىنلەنسە) ھەربىر ئەزانى ئېلىشقا ئىشلىتىلىدۇ. بۇ كۆرسەتكۈچلەر تۆۋەندىكىدەك ئەندىزىلەردە ئىپادىلەنگەن:

```
public static Dictionary<K, T> ToDictionary<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
public static Dictionary<K, T> ToDictionary<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IEqualityComparer<K> comparer);
public static Dictionary<K, E> ToDictionary<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector);
public static Dictionary<K, E> ToDictionary<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector,
    IEqualityComparer<K> comparer);
```

ھەممىگە ئايان، Dictionary<K, V> تىپىدىكى ھەر بىر ئېلېمېنتنىڭ چوقۇم قىممىتى باشقىلارغا ئوخشىمايدىغان K تىپلىق ئاچقۇچلۇق سۆزى(مەلۇم تىپلىق ئوبيېكت بولسىمۇ بولىدۇ...) بولۇشى كېرەك. شۇڭا مەنبە تىزمىنى ToDictionary ئارقىلىق Dictionary<T, K> تىپلىق توپلامغا ئايلاندۇرغاندا، ھەر بىر ئەزا ئۈچۈن keySelector كۆرسەتمىسى ئارقىلىق بىردىن بىر ئاچقۇچلۇق سۆز تاللاپ بېرىش كېرەك. ئەگەر تاللاش جەريانىدا تەكرار ئاچقۇچلۇق سۆز بايقالسا ArgumentException تىپلىق بىنورماللىق چىقىرىدۇ. ئاچقۇچلۇق سۆزلەرنى سېلىشتۇرۇشتا comparer پارامېترى ئارقىلىق يوللانغان سېلىشتۇرغۇچ ئىشلىتىلىدۇ. ئەگەر تەمىنلەنمىسە سىستېما كۆڭۈلدىكى قىممىتىدىكى EqualityComparer<K>.Default نى ئىشلىتىدۇ.

كود 4.60 دا بىر ToDictionary مەشغۇلاتچىسىنى ئىشلىتىپ، خېرىدارلارنىڭ dictionary سىنى قۇردۇق. (بۇ يەردىكى dictionary لۇغەت دېگەن مەنىسى بار. بۇنداق ئاتاشتىكى سەۋەبمۇ ھەم لۇغەتتىكى ھەربىر ئاچقۇچلۇق سۆزنىڭ بىردىن- بىر بولغانلىقىدا)

كود 4.60

```
var customersDictionary =
    customers
```

```
    .ToDictionary(c => c.Name,
                   c => new {c.Name, c.City});
```

يۇقىرىقى مىسالدا، مەشغۇلاتچىنىڭ بىرىنچى پارامېترى **(c=?c.Name)** بولسا **keySelector** كۆرسەتمىسى بولۇپ، ئۇ «نەتىجە لۇغىتىدىكى ھەر بىر ئېلېمېنت ئۇچۇن خېرىدارنىڭ نام خاسلىقىنى ئاچقۇچ سۆز قىلىپ بېكىت» دېگەننى كۆرسىتىپ بېرىدۇ. ئىككىنچى پارامېترى **elementSelector** بولۇپ، ئۇ خېرىدارنىڭ نامى بىلەن شەھەر خاسلىقىنىلا ئۆز ئىچىگە ئالغان نامسىز تىپلىق ئوبيېكتنى لۇغەتتىكى ھەر بىر ئېلېمېنتنىڭ قىممەت بۆلىكى (K) قىلىپ بەلگىلەشنى كۆرسىتىدۇ. تۆۋەندىكىسى كود 4.60 دىكى سورۇشتۇرۇقنىڭ نەتىجىسىنىڭ مەزمۇنى:

K	E

```
[Paolo,  {Name=Paolo,  City=Brescia}]
[Marco,  {Name=Marco,  City=Torino}]
[James,  {Name=James,  City=Dallas}]
[Frank,  {Name=Frank,  City=Seattle}]
```

ئەسكەرتىش: خۇددى **ToList, ToArray** لارغا ئوخشا، **ToDictionary** ئارقىلىق ئېرىشكەن نەتىجمۇ سورۇشتۇرۇق ئىجرا نەتىجىسىنىڭ ئەمەلىي كۆپەيتىلمىسىدىن ئىبارەت.

ToLookup

ToLookup مەشغۇلاتچىسى مەنبە تىزمىدىن **Lookup<T, K>** تىپلىق توپلام ھاسىل قىلىش ۋەزىپىسىنى ئۆتەيدۇ. **Lookup<T, K>** نىڭ ئېنىقلىمىرى تۆۋەندىكىچە:

```
public class Lookup<K, T> : IEnumerable<IGrouping<K, T>> {
    public int Count { get; }
    public IEnumerable<T> this[K key] { get; }
    public bool Contains(K key);
    public IEnumerator<IGrouping<K, T>> GetEnumerator();
}
```

Lookup<T, K> تىپىنىڭ ھەر بىر ئوبيېكتى بىردانە «بىردىن كۆپكە» لۇغىتىگە ماس كېلىدۇ. يەنى **Linq** دىكى **GroupJoin** خاس سۆزىنىڭ نەتىجىسىگە ئوخشىشىپ كېتىدۇ. تۆۋەندىكىسى **ToLookup** مەشغۇلاتچىسىنىڭ ئەندىزىلىرى:

```
public static Lookup<K, T> ToLookup<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
public static Lookup<K, T> ToLookup<T, K>(
```

```
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IEqualityComparer<K> comparer);
public static Lookup<K, E> ToLookup<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector);
public static Lookup<K, E> ToLookup<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector,
    IEqualityComparer<K> comparer);
```

ToDictionary دىكىگە ئوخشاش، بۇنىڭدىمۇ **keySelector** كۆرسەتكۈچىسى، **elementSelector** كۆرسەتكۈچىسى ۋە **comparer** سىلىشتۇرغۇچىلىرى بار.

كود 4.61 دىكى مىسالدا مەزكۇر مەشغۇلاتچىنى ئىشلىتىپ تۇرۇپ ھەربىر مەھسۇلاتقا قارتا زاكازلارغا قانداق ئېرىشىش ئۇسۇلى كۆرسىتىلدۇ:

كود 4.61

```
var ordersByProduct =
    (from c in customers
        from   o in c.Orders
        select o)
    .ToLookup(o => o.IdProduct);

Console.WriteLine( "\n\nNumber of orders for Product 1: {0}\n",
                   ordersByProduct[1].Count());

foreach (var product in ordersByProduct) {
    Console.WriteLine("Product: {0}", product.Key);
    foreach(var order in product) {
        Console.WriteLine("  {0}", order);
```

تۆۋەندىكىسى يۇقىرىقى كودنىڭ ئىجرا نەتىجىسى:

```
Product: 1
  3 - False - January - 1
  10 - False - July - 1
Product: 2
  5 - True - May - 2
```

```
Product: 3
  20 - True - December  -  3
  20 - True - December  -  3
Product: 5
  20 - False - July - 5
```

OfType بىلەن Cast

OfType بىلەن Cast ئالماشتۇرۇش مەشغۇلاتچىلىرى ئىچىدىكى ئەڭ ئاخىرقى ئىككىسى.
OfType مەنبەدىكى ئەزالارنى چارلاپ پەقەت تىپى T بولغانلىرىلا يىغىۋالىدۇ(شۇنىڭدىن كۆرۈۋېلىشقا بولىدۇ، بىر تىزمىدىكى ئەزالار ئوخشىمىغان تىپلىق بولۇشىمۇ مۇمكىن). مەسىلەن: ئوبيېكتقا يۈزلەنگەن پروگراماما لايىھىلەش پرىنسىپى بويىچە، مەنبە تىزمىدا بەلكىم ئوخشاش ئاتىدىن بولغان ئوخشىمىغان تىپلىق ئەزالار بولۇشى مۇمكىن(ئوخشاش بىر تىپقا ۋارىسلىق قىلغان). گەرچە ئۇلارنىڭ ھەممىسى ئاتىسىنىڭ ئىقتىدارى بولسىمۇ لېكىن يەنىلا باشقا ـ باشقا تىپ.

```
public static IEnumerable<T> OfType<T>(
    this IEnumerable source);
```

ئەگەر تەمىنلەنگەن تىپ T نىڭ ئوبيېكتى مەنبە تىزمىدا بايقالمىسا، قۇرۇق تىزما قايتۇرىدۇ.
Cast مەشغۇلاتچىسى مەنبە تىزمىدىكى ھەربىر ئەزانى چارلاش جەريانىدا ئۇنىڭ تىپىنى بېرىلگەن T غا ئايلاندۇرۇپ يىغىۋالىدۇ. ئەگەر ئالماشتۇرۇش مۇۋاپىقيەتلىك بولمىسا InvalidCastException تىپلىق بىنورماللىق چىقىرىدۇ. ئۇنىڭ ئەندىزىسى تۆۋەندىكىچە:

```
public static IEnumerable<T> Cast<T>(
    this IEnumerable source);
```

ئۇنىڭ ئەندىزىسىدىن شۇنى كۆرۈۋالالايمىزكى، مەنبەگە IEnumerbale تىپلىق ھەرقانداق تىزما قوبۇل قىلىنىدىغان بولۇپ، ئەگەر ئۇنى IEnumerrable<T> غا ئايلاندۇرالىسا ، ئۇ چاغدا ئەسلىدىكى ئىنقىسز تىپلىق تىزمىغىمۇ Linq نى قوللىنالايمىز دېگەن گەپ.
ئەسكەرتىش: OfType بىلەن Cast دىن قايتقان ھەربىر ئەزا مەنبەدىكى ئەسلى ئوبيېكتنىڭ چاقىرىلمىسى بولۇپ، ئۇلار يېڭىدىن كۆچۈرۈلمەيدۇ. OfType نەتىجىسى مەنبە تىزمىنىڭ كۆچۈرۈلمىسى ئەمەس. ئۇ نەتىجىگە نىسبەتەن ئىشلىتىش ئېلىپ بېرىلگەندىلا ئاندىن قۇرۇلىدۇ. بۇ يېرى باشقا ئالماشتۇرۇش مەشغۇلاتچىلىرىدىن پەرقلىنىدۇ.