



Бесплатная электронная книга

учусь

C# Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#C#

	1
1: C # Language	2
	2
	2
Examples	2
(Visual Studio)	2
	3
	3
Visual Studio ()	5
Mono	9
.NET Core	10
	11
LinqPad	12
Xamarin Studio	16
2: BackgroundWorker	23
	23
	23
Examples	23
BackgroundWorker	23
BackgroundWorker	24
BackgroundWorker	24
BackgroundWorker	25
	26
3: BigInteger	28
	28
	28
	28
Examples	28
1000-	28
4: BindingList	30
Examples	30

N * 2.....	30
.....	30
5: CLSCompliantAttribute.....	31
.....	31
.....	31
.....	31
.....	31
Examples.....	31
, CLS.....	31
CLS: / sbyte.....	32
CLS:	33
CLS: _.....	33
CLS: CLSComplaint.....	34
6: Enum.....	35
.....	35
.....	35
.....	35
Examples.....	35
.....	35
Enum	36
enum	38
.....	39
== ZERO.....	39
Enum.....	40
.....	41
<<	42
.....	42
.....	43
.....	44
7: FileSystemWatcher.....	45
.....	45
.....	45
Examples.....	45

45	
IsFileReady.....	46
8: Generic Lambda Query Builder.....	47
.....	47
Examples.....	47
QueryFilter.....	47
GetExpression.....	48
GetExpression	49
.....	49
.....	50
ConstantExpression.....	50
.....	51
.....	51
9: Guid.....	52
.....	52
.....	52
Examples.....	52
.....	52
.....	53
GUID.....	53
10: ICloneable.....	54
.....	54
.....	54
Examples.....	54
ICloneable	54
ICloneable	55
11: IComparable.....	57
Examples.....	57
.....	57
12: IEnumerable.....	59
.....	59

59	
Examples.....	59
IEnumerable.....	59
IEnumerable Enumerator.....	59
13: ILGenerator.....	61
Examples.....	61
DynamicAssembly, UnixTimestamp.....	61
.....	63
14: Interoperability.....	64
.....	64
Examples.....	64
C ++.....	64
.....	64
com.....	65
C ++ name mangling.....	65
.....	66
DLL.....	67
Win32.....	68
.....	69
.....	70
15: Linq to Objects.....	72
.....	72
Examples.....	72
LINQ to Object	72
LINQ C #.....	73
16: LINQ to XML.....	77
Examples.....	77
XML LINQ to XML.....	77
17: Microsoft.Exchange.WebServices.....	79
Examples.....	79
.....	79

18: NullReferenceException	82
Examples.....	82
NullReferenceException.....	82
19: O (n)	84
.....	84
Examples.....	84
,	84
20: ObservableCollection	86
Examples.....	86
ObservableCollection.....	86
21: String.Format	87
.....	87
.....	87
.....	87
.....	87
.....	87
Examples.....	87
, String.Format «»	88
.....	88
.....	88
/ ,	89
.....	89
.....	90
.....	90
.....	90
.....	90
.....	91
C # 6.0.....	91
String.Format ().....	91
.....	91
().....	93
ToString ().....	94

.....	94
22: StringBuilder	95
Examples.....	95
StringBuilder	95
StringBuilder	96
23: System.DirectoryServices.Protocols.LdapConnection	98
Examples.....	98
SSL- LDAP, SSL- DNS.....	98
LDAP.....	99
24: System.Management.Automation	100
.....	100
Examples.....	100
.....	100
25: Verbatim Strings	102
.....	102
.....	102
Examples.....	102
.....	102
.....	103
Verbatim.....	103
Verbatim	103
26: Windows Communication Foundation	105
.....	105
Examples.....	105
.....	105
27: XDocument System.Xml.Linq	108
Examples.....	108
XML-.....	108
XML.....	108
XML-	110
28: XmlDocument System.Xml	111

Examples.....	111
XML-.....	111
XML.....	111
XmlDocument vs XDocument ().....	112
29:	116
.....	116
.....	116
.....	116
Examples.....	117
.....	117
.....	117
30:	118
Examples.....	118
DisplayNameAttribute ().....	118
EditableAttribute ().....	119
.....	121
: RequiredAttribute.....	121
: StringLengthAttribute.....	121
: RangeAttribute.....	121
: CustomValidationAttribute.....	122
.....	122
.....	123
.....	123
.....	124
.....	124
.....	124
.....	124
31:	125
Examples.....	125
.....	125
.....	125

.....	126
.....	126
.....	126
.....	127
32: /, ,	128
.....	128
Examples	128
ASP.NET Configure Await	128
.....	129
ConfigureAwait	129
/ Await	130
BackgroundWorker	131
.....	132
.....	133
« »	134
33:	136
.....	136
.....	136
Examples	137
(/)	137
34: -Await	145
.....	145
.....	145
Examples	145
.....	145
/ /	146
Web.config 4.5	146
.....	147
async	148
.....	149
.....	150
Async / await ,	151

35:	153
Examples.....	153
.....	153
.....	153
.....	153
DebuggerDisplay.....	154
.....	155
.....	156
.....	157
36:	158
.....	158
Examples.....	158
«Fontfamily»	158
.....	158
«Button».....	159
37:	160
.....	160
Examples.....	160
MEF.....	160
C # ASP.NET Unity.....	162
38:	166
Examples.....	166
RPG.....	166
39:	169
.....	169
Examples.....	169
.....	169
40: C # 3.0	171
.....	171
Examples.....	171
(var).....	171
(LINQ).....	171

Lambda expresions.....	172
.....	173
41: C # 4.0.....	175
Examples.....	175
.....	175
.....	176
ref COM.....	176
.....	177
42: C # 5.0.....	178
.....	178
.....	178
.....	178
Examples.....	178
Async & Await.....	179
.....	180
43: C # 6.0.....	182
.....	182
.....	182
Examples.....	182
.....	182
().....	184
.....	184
.....	184
.....	185
.....	185
.....	186
.....	186
.....	187
.....	187
,	188
.....	189

finally	190
: finally	191
.....	192
.....	192
.....	192
.....	193
.....	193
.....	193
.....	193
.....	194
.....	196
.....	198
.....	198
(pre C # 6.0)	198
.....	198
.....	199
.....	200
verbatim	198
.....	199
.....	200
FormattableString	201
.....	202
.....	202
.....	203
Linq	203
.....	203
.....	204
.....	205
.....	206
.....	207
.....	207
Null-Coalescing Operator (??)	208
.....	208
void	208
.....	209

.....	209
Gotchas	209
.....	211
.....	212
.....	213
.....	213
.....	214
44: C# 7.0	216
.....	216
Examples	216
var.....	216
.....	216
.....	217
.....	218
.....	218
.....	218
.....	219
.....	220
.....	220
.....	220
.....	222
h11	222
.....	222
.....	223
async	223
.....	223
ValueTuple Tuple	224
.....	225
.....	225
.....	225

225	226
.....	227
switch.....	227
is	227
.....	228
ref	228
.....	228
.....	229
ref.....	229
.....	230
.....	230
.....	231
ValueTask.....	232
1.	232
2.	232
:.....	233
.....	233
.....	233
45:	234
Examples.....	234
-	234
- char.....	234
- , int, long (16-, 32-, 64-).....	234
- ushort, uint, ulong (16 , 32 , 64).....	235
- bool.....	235
.....	236
.....	236
46: HTTP-.....	237
Examples.....	237
HTTP POST.....	237

HTTP GET.....	237
HTTP (404 Not Found).....	238
HTTP POST JSON.....	238
HTTP GET JSON.....	239
HTML - ().....	239
47: 4.....	240
.....	240
Examples.....	240
.....	240
48: C #.....	241
.....	241
.....	241
.....	241
.....	241
Examples.....	241
int.....	241
.....	242
int	242
.....	242
.....	242
.....	243
,	243
49:	245
.....	245
Examples.....	245
.....	245
.....	245
ISerializable.....	247
(ISerializationSurrogate).....	247
Serialization.....	250
.....	251
50:	255
.....	255

255	
: Action<...> , Predicate<T> Func<...,TResult>	255
.....	255
.....	255
.....	255
.....	256
Examples	256
.....	256
.....	257
Func ,	258
.....	259
.....	260
.....	260
.....	260
.....	260
()	261
.....	262
.....	264
.....	264
51: Func	266
.....	266
.....	266
Examples	266
.....	266
.....	267
.....	267
.....	268
52:	269
.....	269
.....	269
.....	269
.....	269

.....	269
.....	269
.....	270
.....	270
.....	271
LINQ	271
.....	271
Examples.....	272
API.....	272
.....	272
.....	272
.....	273
API.....	273
Basic.....	274
.....	275
53:	276
.....	276
.....	276
.....	276
Examples.....	276
().....	276
().....	277
().....	277
().....	278
().....	279
().....	280
().....	281
().....	282
.....	282
.....	282
.....	283
.....	284

286	287
.....	287
.....	287
.....	288
.....	289
.....	289
.....	290
.....	291
54:	293
Examples.....	293
Debug.WriteLine.....	293
TraceListeners.....	293
55:	294
.....	294
Examples.....	294
.....	294
.....	294
.....	295
,	295
56:	297
Examples.....	297
ADO.NET.....	297
.....	297
ADO.NET.....	297
Entity Framework.....	298
Entity.....	299
.....	300
.....	300
.....	300
57:	301
.....	301
Examples.....	301

.....	301
58:	304
.....	304
.....	304
Examples	305
.....	305
.....	305
.....	306
.....	306
Anti-Patterns gotchas	307
/	307
,	307
,	308
ValueType	309
,	310
59: LINQ	312
.....	312
.....	312
.....	314
Examples	314
.....	314
.....	315
.....	315
-	315
.....	316
.....	317
.....	317
.....	317
FirstOrDefault, Last, LastOrDefault, Single SingleOrDefault	318
0	318

FirstOrDefault ()	319
()	319
LastOrDefault ()	320
()	321
SingleOrDefault ()	321
.....	322
.....	323
SelectMany:	324
SelectMany	326
.....	327
1.	327
2. -	327
3.	328
/	328
.....	329
JOINS	329
(.....	329
.....	329
.....	330
.....	330
.....	330
.....	331
.....	332
GroupBy	332
Range Linq	333
- OrderBy () ThenBy () OrderByDescending () ThenByDescending ()	333
.....	334
.....	335
.....	335
.....	336
.....	337

1.	337
2. -	337
3.	337
ToDictionary	338
.....	338
Linq ()	339
SkipWhile	340
DefaultIfEmpty	340
:	341
SequenceEqual	342
Count LongCount	342
.....	342
-	344
GroupJoin	345
ElementAt ElementAtOrDefault	345
-	345
.....	346
.....	348
Func selector -	349
TakeWhile	350
.....	350
ToLookup	351
Linq IEnumerable	351
SelectMany	353
(OrDefault) -	353
GroupBy Sum Count	353
.....	355
.....	356
.....	358
OrderByDescending	359
Concat	360
.....	360
60: ASP.NET	362

.....	362
Examples.....	362
asp.net	362
61:	366
Examples.....	366
.....	366
.....	366
.....	367
.....	367
62:	369
.....	369
Examples.....	369
.....	369
.....	372
63: Google.....	374
.....	374
Examples.....	374
.....	374
.....	374
.....	377
64:	378
.....	378
.....	378
Examples.....	378
:	378
.....	378
PropertyChanged.....	379
PropertyChanged.....	380
.....	380
.....	381
.....	381
MVC.....	382

65:	383
.....	383
.....	383
Examples	383
.....	383
Indexer	384
SparseArray	384
66:	386
.....	386
Examples	386
.....	386
C # 6	387
.....	387
.....	388
.....	389
.....	389
67:	391
.....	391
.....	391
Examples	391
.....	391
.....	391
.....	392
68:	393
.....	393
Examples	393
C # 6.0:	393
.....	393
.....	393
.....	393
69:	395

395	395
Examples	395
.....	395
.....	395
.....	395
.....	396
.....	396
.....	396
.....	396
.....	396
.....	397
.....	397
.....	397
70: IDisposable	399
.....	399
Examples	399
.....	399
.....	399
.....	399
IDisposable, Dispose	400
.....	401
.....	401
71: INotifyPropertyChanged	403
.....	403
Examples	403
INotifyPropertyChanged C # 6	403
INotifyPropertyChanged	404
72: IQueryable	406
Examples	406
LINQ SQL	406
73:	407
Examples	407
.....	407
.....	407

.....	408
.....	409
.....	409
.....	409
.....	411
«»	413
IComparable	414
74: json.net	417
.....	417
Examples.....	417
JsonConverter	417
JSON (http://www.omdbapi.com/?i=tt1663662)	417
.....	418
RuntimeSerializer	418
.....	419
JSON.....	419
75: SQLite C #	422
Examples.....	422
CRUD SQLite C #.....	422
.....	426
76:	428
.....	428
Examples.....	428
.....	428
.....	428
.....	428
.....	429
.....	429
.....	430
77:	431
.....	431

431	431
Examples	431
.....	431
.....	432
.....	433
Gotcha: ,	434
.....	435
Gotcha: Dispose	435
.....	436
IDisposable	436
ADO.NET	436
DataContexts	437
Dispose Syntax	437
.....	438
78:	440
.....	440
Examples	440
.....	440
.....	440
79: C # Structs Union (C Unions)	443
.....	443
Examples	443
C-Style Unions C #	443
Union C # Struct	444
80:	446
.....	446
Examples	446
.....	446
.....	447
(`as`)	447
.....	447

.....	448
.....	448
.....	448
LINQ	449
81:	451
.....	451
.....	451
.....	451
.....	451
Examples.....	451
.....	451
.....	452
.....	453
.....	454
, Enumerable.....	455
.....	455
.....	456
yield IEnumator IEnumarable.....	457
.....	458
:	459
.....	460
82:	462
.....	462
.....	462
Examples.....	464
stackalloc.....	464
.....	465
.....	467
.....	467
.....	467
.....	468
.....	469
.....	469

471	472
.....	472
Const.....	472
.....	474
, , ,	474
.....	475
ref, out.....	476
checked, unchecked.....	478
.....	479
goto :	479
:	479
:	479
.....	480
.....	480
.....	481
.....	483
.....	484
.....	485
.....	487
float, double, decimal.....	488
.....	488
.....	489
.....	489
UINT.....	490
.....	490
.....	491
.....	492
.....	494
.....	494
.....	495
.....	495
.....	496

.....	496
.....	498
.....	498
.....	499
ULONG.....	499
.....	499
, ,	500
.....	500
.....	501
.....	503
.....	503
.....	504
,	504
.....	506
.....	506
.....	507
.....	508
.....	509
,	511
.....	512
BOOL.....	512
.....	513
.....	514
?.....	514
.....	514
if, if ... else, if ... else if.....	515
, ,	516
.....	516
.....	517
.....	519
.....	520
.....	521

.....	521
.....	524
true, false.....	524
.....	525
USHORT.....	525
SByte.....	525
.....	525
.....	527
.....	528
.....	528
83:	531
.....	531
.....	531
Examples.....	532
.....	532
.....	532
.....	532
.....	533
84:	536
Examples.....	536
.....	536
85:	538
Examples.....	538
.....	538
.....	538
.....	538
.....	539
.....	540
86: XML.....	542
.....	542
Examples.....	542
.....	542

.....	543
param return.....	543
XML	544
.....	545
87:	547
Examples.....	547
RoslynScript.....	547
CSharpCodeProvider.....	547
88:	548
.....	548
Examples.....	548
+	548
, System.Text.StringBuilder.....	548
Concat string String.Join.....	548
\$.....	549
89:	550
.....	550
.....	550
Examples.....	550
.....	550
.....	551
.....	552
.....	553
.....	554
Singleton.....	555
.....	555
.....	556
.....	556
.....	557
.....	558
90: (TPL).....	561
Examples.....	561

JoinBlock.....	561
BroadcastBlock.....	562
WriteOnceBlock.....	563
BatchedJoinBlock.....	564
TransformBlock.....	565
ActionBlock.....	565
TransformManyBlock.....	566
BatchBlock.....	567
BufferBlock.....	568
91: Async-Await.....	570
Examples.....	570
async / await.....	570
.....	570
SynchronizationContext ?.....	571
92:	573
Examples.....	573
.....	573
.....	573
.....	573
.....	574
93: (System.Security.Cryptography).....	575
Examples.....	575
.....	575
.....	587
.....	587
.....	587
.....	588
.....	589
.....	590
.....	591
94:	596
Examples.....	596

MemoryCache.....	596
95:	597
.....	597
Examples.....	597
int	597
uint	597
.....	597
.....	598
-.....	598
.....	598
.....	598
.....	599
.....	599
.....	599
.....	599
ulong literal.....	599
.....	599
ushort literal.....	599
bool.....	600
96: -.....	601
.....	601
Examples.....	601
.....	601
-.....	601
Lambdas `Func` ``Action`.....	601
-.....	602
.....	602
Lambdas `Func`, ``Expression`.....	602
Lambda	603
97: -.....	605
.....	605
.....	605
Examples.....	606

-	606
- LINQ	606
.....	606
Lambda	607
- System.Linq.Expressions	607
98:	608
Examples	608
.....	608
.....	608
()	609
.....	609
.....	610
.....	610
.....	610
99:	611
.....	611
.....	611
Examples	612
.....	612
.....	612
.....	612
.....	613
.....	614
.....	614
,	616
,	616
,	617
.....	618
:	618
.....	618
IEnumerable <>	619
100:	620

Examples.....	620
.....	620
.....	620
.....	621
.....	621
.....	622
.....	623
.....	624
.....	625
101: DateTime.....	626
Examples.....	626
DateTime.Add (TimeSpan).....	626
DateTime.AddDays ().....	626
DateTime.AddHours ().....	626
DateTime.AddMilliseconds ().....	626
DateTime.Compare (DateTime t1, DateTime t2).....	627
DateTime.DaysInMonth (Int32, Int32).....	627
DateTime.AddYears (Int32).....	627
DateTime.....	628
DateTime.Parse (String).....	628
DateTime.TryParse (String, DateTime).....	628
Parse TryParse	629
DateTime for-loop.....	629
DateTime ToString, ToShortDateString, ToLongDateString ToString	630
.....	630
.....	630
DateTime.ParseExact (String, String, IFormatProvider).....	631
DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime).....	632
102:	635
.....	635
.....	635
.....	635

Examples.....	636
.....	637
.....	640
.....	640
.....	640
.....	641
()	641
.....	642
.....	643
.....	645
.....	645
.....	646
.....	647
IList : 2	647
.....	648
DRY-.....	650
.....	651
.....	651
.....	653
.....	653
(, DictList).....	654
103:	657
.....	657
Examples.....	658
Threading Demo.....	658
.....	658
.....	659
.....	659
.....	659
.....	660
.....	660

.....	661
forEach.....	662
(,).....	663
().....	664
104:	668
.....	668
Examples.....	668
.....	668
.....	668
.....	669
.....	670
.....	670
.....	672
105:	674
.....	674
.....	674
Examples.....	674
.....	674
.....	675
.....	675
.....	676
.....	677
.....	677
.....	678
.....	681
.....	681
.....	681
.....	682
106: :Json C #	686
.....	686
Examples.....	686
Json.....	686

: Json.....	686
C #.....	686
.....	687
.....	688
- Common Utilities.....	688
107: .NET.....	689
.....	689
Examples.....	689
.....	689
.....	690
.....	690
108:	692
.....	692
.....	692
Examples.....	693
.....	693
, Nullable	693
NULL.....	694
NULL.....	694
, NULL.....	694
null.....	695
Nullable	695
109:	697
Examples.....	697
System.String.....	697
.....	697
110: -	699
.....	699
.....	699
Examples.....	699
Null-Conditional Operator.....	699
.....	700

Null-Coalescing.....	700
.....	700
NullReferenceExceptions.....	700
Null-.....	701
111:	703
Examples.....	703
Parallel.For.....	703
112: c#.....	704
Examples.....	704
HashSet.....	704
SortedSet.....	704
T [] (T).....	704
.....	705
.....	705
.....	706
.....	706
LinkedList.....	707
.....	707
113: FormatException	708
Examples.....	708
.....	708
114:	711
Examples.....	711
.....	711
.....	711
.....	711
,.....	713
IErrorHandler WCF.....	714
.....	717
.....	717
.....	718
.....	718

ParserException	719
.....	719
.....	720
.....	720
.....	720
.....	721
0	722
/	723
.....	724
.....	724
Cheatsheet	725
.....	725
.....	726
.....	726
,	727
.....	727
.....	728
115: C #	730
Examples	730
.....	730
116:	732
Examples	732
.....	732
.....	732
,	732
/	732
String.Trim()	732
String.TrimStart() String.TrimEnd()	733
.....	733
.....	733
.....	733

733	
ToString.....	734
x	735
String.IsNullOrEmpty () String.IsNullOrWhiteSpace ().....	736
.....	737
,	737
.....	738
.....	738
.....	740
.....	741
.....	741
.....	741
117: - C #.....	743
.....	743
Examples.....	743
.....	743
118: Null-Coalescing.....	744
.....	744
.....	744
.....	744
Examples.....	744
.....	744
Null fall-through and chaining.....	745
.....	746
.....	746
.....	747
.....	747
C # 6	747
MVVM.....	747
119:	749
Examples.....	749
c #	749

120:	750
.....	750
.....	750
.....	750
.....	750
.....	750
.....	750
.....	750
Examples.....	753
.....	753
.....	754
.....	756
.....	757
.....	758
:	759
: Null Conditional Member Access.....	759
:	759
:	760
:	760
«» «».....	760
.....	760
.....	761
.....	762
? :	762
.....	764
.....	764
(T: struct).....	764
(T:).....	765
.....	765
? . (Null).....	765
.....	766
=> -	766
'='.....	768
?? Null-Coalescing.....	768

121:	769
.....	769
.....	769
Examples	769
System.Type.....	769
.....	769
.....	770
.....	771
.....	771
.....	773
.....	773
.....	774
Generic	775
, (,).....	775
.....	776
Activator	776
Activator	777
.....	779
.....	780
122:	782
Examples	782
Parallel.ForEach.....	782
Parallel.For.....	782
Parallel.Invoke.....	783
,	783
CancellationTokenSource.....	784
PingUrl.....	785
123: LINQ (PLINQ)	786
.....	786
Examples	788
.....	788
WithDegreeOfParallelism.....	788

AsOrdered.....	789
AsUnordered.....	789
124:	790
Examples.....	790
.....	790
.....	791
.....	792
.....	793
.....	793
Do - While Loop.....	794
.....	795
.....	795
125:	797
Examples.....	797
.....	797
.....	797
.....	797
126: .NET (Roslyn).....	799
Examples.....	799
MSBuild.....	799
.....	799
.....	800
127:	802
Examples.....	802
.....	802
.....	803
.....	803
.....	804
128:	806
Examples.....	806
.....	806
129:	808

.....	808
Examples.....	808
MSDN.....	808
.....	809
130:	810
.....	810
.....	810
.....	810
Examples.....	811
.....	811
.....	812
.....	812
.....	813
.....	814
.....	814
Pragma.....	814
.....	814
.....	815
.....	815
131: AssemblyInfo.cs.....	817
.....	817
Examples.....	817
[AssemblyTitle].....	817
[AssemblyProduct].....	817
AssemblyInfo.....	817
[AssemblyVersion].....	818
.....	818
.....	819
.....	819
[AssemblyConfiguration].....	819
[InternalsVisibleTo].....	820
[AssemblyKeyFile].....	820

132:	822
.....	822
Examples.....	822
.....	822
.....	822
133:	823
Examples.....	823
.....	823
134: GetHashCode	825
.....	825
Examples.....	825
.....	825
GetHashCode	826
Equals GetHashCode	827
GetHashCode IEqualityComparer.....	828
135:	830
.....	830
Examples.....	830
.....	830
«params» ,	831
null	832
136: (Rx)	833
Examples.....	833
TextChanged TextBox.....	833
.....	833
137: Singleton	835
Examples.....	835
.....	835
, (Double Checked Locking).....	835
, (Lazy).....	836
Lazy, singleton (.NET 3.5 ,).....	837
Singleton,	837

138:	839
.....	839
Examples	839
.....	839
.....	840
.....	841
.....	844
.....	844
PowerOf	844
139: .Net	846
Examples	846
.....	846
.....	846
140:	849
.....	849
Examples	849
.....	849
Get	850
.....	850
.....	850
.....	852
.....	853
.....	853
.....	853
.....	853
.....	854
141:	855
.....	855
.....	855
Examples	855
.....	855
IsHighResolution	855
142:	857

.....	857
.....	857
Examples	857
TCP-.....	857
-.....	857
Async TCP Client.....	858
UDP.....	859
143:	861
.....	861
.....	861
.....	861
.....	861
Examples	862
.....	862
.....	862
.....	863
.....	863
.....	864
.....	865
EventArg,	866
.....	867
.....	868
144:	870
.....	870
.....	870
.....	870
.....	870
.....	870
.....	870
Examples	871
.....	871
Pascal	871
.....	871

.....	871
.....	871
.....	872
.....	872
.....	872
.....	873
.....	873
.....	873
.....	873
Enums.....	873
Enum,	873
« »	874
.....	874
.....	874
« »	874
145: Plain-Text	875
Examples.....	875
Plain-Text C #.....	875
.....	875
.....	875
146: MessageBox Windows Form.....	878
.....	878
.....	878
Examples.....	878
MessageBox.....	878
MessageBox W.....	880
147:	882
.....	882
Examples.....	882
Singleton.....	882
.....	884
Builder.....	887

.....	891
.....	893
148:	896
Examples.....	896
.....	896
.....	896
.....	897
149: escape-.....	899
.....	899
.....	899
Examples.....	899
Unicode.....	899
.....	900
.....	900
escape-	900
escape-	901
150:	902
.....	902
Examples.....	902
.....	902
151:	906
.....	906
Examples.....	906
.....	906
.....	907
.....	908
.....	909
152: C #.....	910
Examples.....	910
.....	910
153:	911
.....	911

.....	911
Examples.....	911
.....	911
:.....	912
.....	913
«Tick».....	913
:.....	914
154: vs	916
.....	916
.....	916
.....	916
.....	916
.....	916
.....	916
.....	916
,	917
,	917
,	917
Examples.....	917
.....	918
.....	919
ref.....	919
.....	920
ref out.....	920
ref vs out.....	922
155:	924
.....	924
unsafe	924
.....	924
,	924
Examples.....	924
.....	924
.....	925

.....	926
*	926
->	926
.....	927
156:	928
Examples.....	928
.....	928
.....	929
.....	929
.....	930
.....	931
157:	933
Examples.....	933
If-Else Statement.....	933
If-Else If-Else Statement.....	933
.....	934
.....	935
158: -	937
.....	937
.....	937
.....	937
.....	937
Examples.....	938
System.IO.File.....	938
System.IO.StreamWriter.....	938
System.IO.File.....	939
IEnumerable.....	939
.....	939
.....	940
.....	941
.....	941
.....	941

Async	StreamWriter	941
159:		943
Examples		943
		943
160:		945
Examples		945
Func	Action	945
		945
Null		947
		948
		949
		949
		949
IEnumerable		949
161:		950
		950
Examples		950
« » + « »		950
		950
		951
162: -		952
		952
Examples		952
MD5		952
SHA1		953
SHA256		953
SHA384		954
SHA512		954
PBKDF2		955
Pbkdf2		956
163:		960
		960

.....	960
.....	960
Examples.....	960
.....	960
.....	961
,	962
164: Stacktraces.....	963
.....	963
Examples.....	963
NullReferenceException Windows Forms.....	963
165: .zip-.....	965
.....	965
.....	965
Examples.....	965
zip-.....	965
ZIP-	965
Zip-.....	966
, zip- .txt-	966
.....	968

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [csharp-language](https://csharp-language.com)

It is an unofficial and free C# Language ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](https://csharp-language.com), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C# Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с C # Language

замечания

С # - это мультипарадигма, язык программирования С-потомок от Microsoft. С # - управляемый язык, который компилируется в [CIL](#) , промежуточный байт-код, который может быть выполнен в Windows, Mac OS X и Linux.

Версии 1.0, 2.0 и 5.0 были стандартизованы ECMA (как [ECMA-334](#)), и в настоящее время предпринимаются усилия по стандартизации для современного С #.

Версии

Версия	Дата выхода
1,0	2002-01-01
1.2	2003-04-01
2,0	2005-09-01
3.0	2007-08-01
4,0	2010-04-01
5.0	2013-06-01
6,0	2015-07-01
7,0	2017-03-07

Examples

Создание нового консольного приложения (Visual Studio)

1. Открыть Visual Studio
2. На панели инструментов перейдите в **Файл** → **Новый проект**
3. Выберите тип проекта **консоли**.
4. Откройте файл `Program.cs` в обозревателе решений
5. Добавьте следующий код в `Main()` :

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

```

    {
        // Prints a message to the console.
        System.Console.WriteLine("Hello, World!");

        /* Wait for the user to press a key. This is a common
           way to prevent the console window from terminating
           and disappearing before the programmer can see the contents
           of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}

```

6. На панели инструментов нажмите « **Отладка** » -> « **Начать отладки** » или нажмите « **F5** » или « **Ctrl + F5** » (работает без отладчика), чтобы запустить программу.

[Демо-версия на ideone](#)

объяснение

- `class Program` - это объявление класса. Класс `Program` содержит определение данных и методов , которые используются в вашей программе. Классы обычно содержат несколько методов. Методы определяют поведение класса. Однако класс `Program` имеет только один метод: `Main` .
 - `static void Main()` определяет метод `Main` , который является точкой входа для всех программ на C #. `Main` метод определяет, что делает класс при выполнении. Для каждого класса допускается только один `Main` метод.
 - `System.Console.WriteLine("Hello, world!");` метод выводит данные (в этом примере, `Hello, world!`) в качестве вывода в окне консоли.
 - `System.Console.ReadKey()` , гарантирует, что программа не будет закрываться сразу после отображения сообщения. Он делает это, ожидая, что пользователь нажмет клавишу на клавиатуре. Любое нажатие клавиши от пользователя прекратит выполнение программы. Программа завершается, когда она закончила последнюю строку кода в методе `main()` .
-

Использование командной строки

Для компиляции с помощью командной строки используйте либо `MSBuild` либо `csc.exe` (компилятор C #) , как часть пакета средств [Microsoft Build Tools](#) .

Чтобы скомпилировать этот пример, запустите следующую команду в том же каталоге, где находится `HelloWorld.cs` :

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs
```

Также возможно, что у вас есть два основных метода внутри одного приложения. В этом случае, вы должны сообщить компилятору, основной метод, чтобы выполнить, введя следующую команду в **консоли**. (Предположим, что класс `ClassA` также имеет основной метод в том же `HelloWorld.cs` файл `HelloWorld` имен)

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

где `HelloWorld` - пространство имен

Примечание. Это путь, в котором **.NET framework v4.0** находится в целом. Измените путь в соответствии с вашей версией .NET. Кроме того, каталог может быть **фреймворком** вместо **framework64**, если вы используете 32-разрядную .NET Framework. В командной строке Windows вы можете перечислить все пути `csc.exe` Framework, выполнив следующие команды (первая для 32-разрядных фреймворков):

```
dir %WINDIR%\Microsoft.NET\Framework\csc.exe /s/b  
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s/b
```

```
C:\Users\Main\Documents\helloworld>%WINDIR%\Microsoft.NET\Framework\v4.0.30319\csc.exe HelloWorld.cs  
Microsoft (R) Visual C# Compiler version 4.0.30319.17929  
for Microsoft (R) .NET Framework 4.5  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
C:\Users\Main\Documents\helloworld>dir  
Volume in drive C is System  
  
Directory of C:\Users\Main\Documents\helloworld  
07/26/2016  03:43 PM      <DIR>          .  
07/26/2016  03:43 PM      <DIR>          ..  
07/26/2016  03:41 PM            258 HelloWorld.cs  
07/26/2016  03:43 PM            3,584 HelloWorld.exe  
                2 File(s)       3,842 bytes  
                2 Dir(s)  99,699,073,024 bytes free  
  
C:\Users\Main\Documents\helloworld>
```

В этом же каталоге должен быть исполняемый файл `HelloWorld.exe`. Чтобы выполнить программу из командной строки, просто введите имя исполняемого файла и нажмите `Enter` следующим образом:

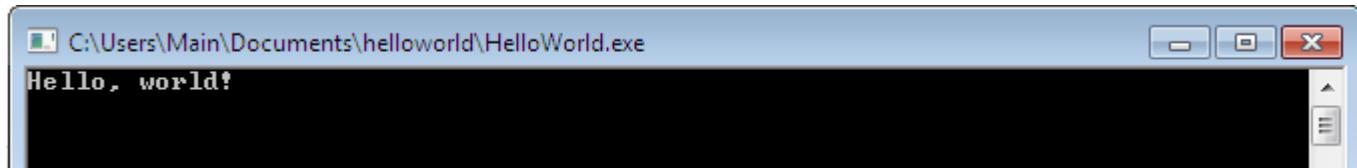
```
HelloWorld.exe
```

Это даст:

Привет, мир!

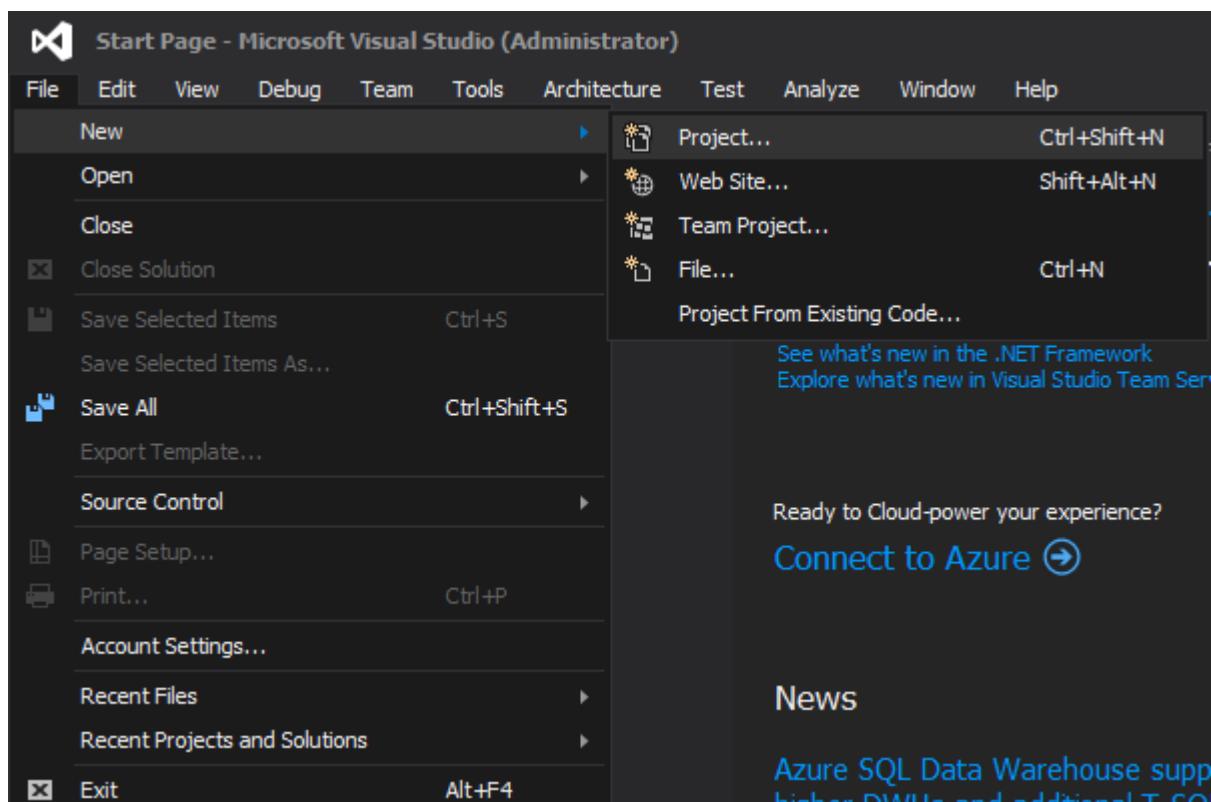
```
C:\Users\Main\Documents\helloworld>HelloWorld  
Hello, world!
```

Вы также можете дважды щелкнуть исполняемый файл и запустить новое окно консоли с сообщением « **Привет, мир!** ».

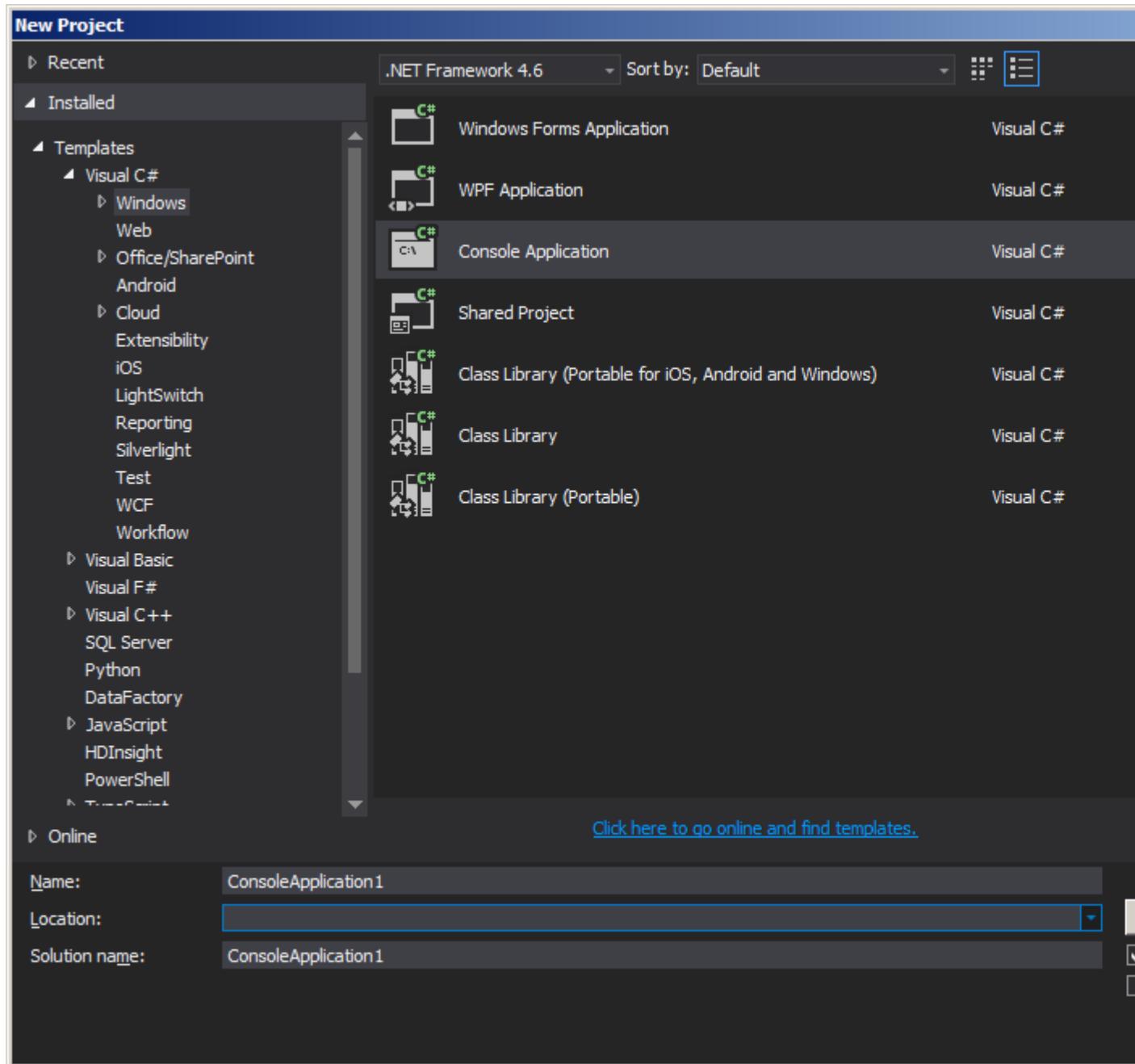


Создание нового проекта в Visual Studio (консольное приложение) и запуск его в режиме отладки

- 1. Загрузите и установите Visual Studio** . Visual Studio можно загрузить с сайта VisualStudio.com . Издание сообщества предлагается, во-первых, потому что оно бесплатное и второе, потому что оно включает в себя все общие функции и может быть расширено дальше.
- 2. Откройте Visual Studio.**
- 3. Добро пожаловать.** Откройте **Файл → Новый → Проект** .



- 4. Нажмите Шаблоны → Visual C # → Консольное приложение**



5. **Выбрав консольное приложение**, введите имя для своего проекта и место для сохранения и нажмите **OK**. Не беспокойтесь о имени решения.
6. **Создан проект**. Созданный проект будет выглядеть примерно так:

ConsoleApplication1 - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

146 %

Error List

Entire Solution 0 Errors 0 Warnings 0 Messages Build + IntelliSense

Code Description

Error List Output

(Всегда используйте описательные имена для проектов, чтобы их можно было легко отличить от других проектов. Рекомендуется не использовать пробелы в имени проекта или класса).

Всегда используйте описательные имена для проектов, чтобы их можно было легко отличить от других проектов. Рекомендуется не использовать пробелы в имени проекта или класса).

7. Напишите код. Теперь вы можете обновить программу `Program.cs` чтобы представить «Hello world!». для пользователя.

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Добавьте следующие две строки в объект `public static void Main(string[] args)` в `Program.cs` : (убедитесь, что он находится внутри фигурных скобок)

```
Console.WriteLine("Hello world!");
Console.Read();
```

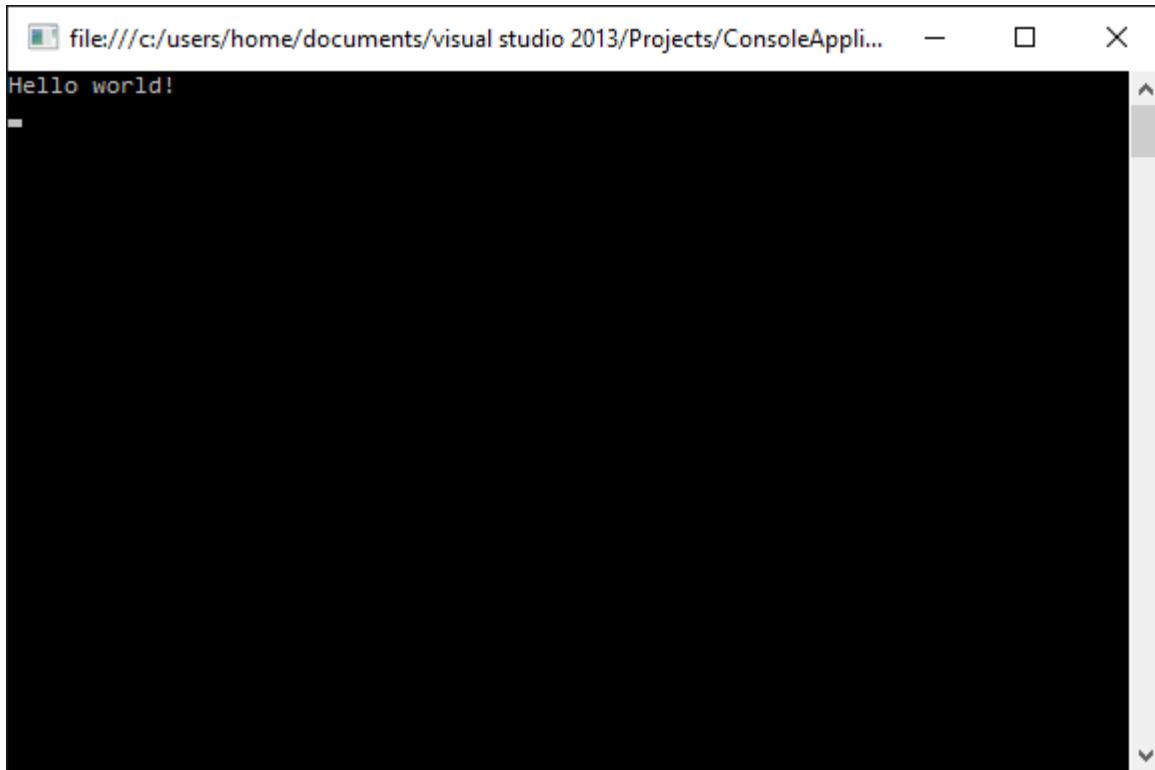
Почему `Console.Read()` ? Первая строка выводит текст «Привет мир!». на консоль, а вторая строка ожидает ввода одного символа; по сути, это заставляет программу приостанавливать выполнение, чтобы вы могли видеть вывод во время отладки. Без `Console.Read()` , когда вы начнете отлаживать приложение, оно просто напечатает «Hello world!». на консоль, а затем сразу же закрыть. Окно вашего кода теперь должно выглядеть следующим образом:

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. Отлаживайте свою программу. Нажмите кнопку «Пуск» на панели инструментов в

верхней части окна.  или нажмите **F5** на клавиатуре, чтобы запустить приложение. Если кнопки нет, вы можете запустить программу из главного меню: **Debug → Start Debugging**. Программа скомпилирует и откроет окно консоли. Он должен выглядеть примерно так, как показано на следующем снимке экрана:



9. Остановите программу. Чтобы закрыть программу, просто нажмите любую клавишу на клавиатуре. Добавленный нами `Console.Read()` был для этой же цели. Другой способ закрыть программу - это перейти в меню, где была кнопка «Пуск», и нажать кнопку «Стоп».

Создание новой программы с использованием Mono

Сначала установите **Mono**, выполнив инструкции по установке для выбранной вами платформы, как описано в [разделе их установки](#).

Моно доступно для Mac OS X, Windows и Linux.

После завершения установки создайте текстовый файл, назовите его `HelloWorld.cs` и скопируйте в него следующий контент:

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

Если вы используете Windows, запустите Mono Command Prompt, которая включена в установку Mono, и убедитесь, что установлены необходимые переменные среды. Если на Mac или Linux откройте новый терминал.

Чтобы скомпилировать вновь созданный файл, запустите следующую команду в каталоге, содержащем `HelloWorld.cs`:

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

Получаемый `HelloWorld.exe` можно выполнить с помощью:

```
mono HelloWorld.exe
```

который будет производить вывод:

```
Hello, world!  
Press any key to exit..
```

Создание новой программы с использованием .NET Core

Сначала установите [.NET Core SDK](#), выполнив инструкции по установке для выбранной вами платформы:

- [Windows](#)
- [OSX](#)
- [Linux](#)
- [докер](#)

По завершении установки откройте командную строку или окно терминала.

1. Создайте новый каталог с `mkdir hello_world` и перейдите во вновь созданный каталог `cd hello_world`.

2. Создайте новое консольное приложение с `dotnet new console`.

Это создаст два файла:

- **`hello_world.csproj`**

```
<Project Sdk="Microsoft.NET.Sdk">  
  
  <PropertyGroup>  
    <OutputType>Exe</OutputType>  
    <TargetFramework>netcoreapp1.1</TargetFramework>  
  </PropertyGroup>  
  
</Project>
```

- **`Program.cs`**

```
using System;

namespace hello_world
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

3. Восстановите необходимые пакеты с `dotnet restore`.

4. *Необязательно* Создайте приложение с `dotnet build` для `dotnet build Debug` или `dotnet build -c Release` for `Release`. `dotnet run` также запускает компилятор и `dotnet run` ошибки сборки, если они есть.

5. Запустите приложение с `dotnet run` для `dotnet run Debug` или `dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` для выпуска.

Вывод командной строки

```
Command Prompt

C:\dev>mkdir hello_world
C:\dev>cd hello_world
C:\dev\hello_world>dotnet new console
Content generation time: 75.7641 ms
The template "Console Application" created successfully.

C:\dev\hello_world>dotnet restore
Restoring packages for C:\dev\hello_world\hello_world.csproj...
Generating MSBuild file C:\dev\hello_world\obj\hello_world.csproj.nuget.g.props.
Generating MSBuild file C:\dev\hello_world\obj\hello_world.csproj.nuget.g.targets.
Writing lock file to disk. Path: C:\dev\hello_world\obj\project.assets.json
Restore completed in 3.35 sec for C:\dev\hello_world\hello_world.csproj.

NuGet Config files used:
  C:\Users\Ghost\AppData\Roaming\NuGet\NuGet.Config
  C:\Program Files (x86)\NuGet\Config\Microsoft.VisualStudio.Offline.config

Feeds used:
  https://api.nuget.org/v3/index.json
  C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

C:\dev\hello_world>dotnet build -c Release
Microsoft (R) Build Engine version 15.1.548.43366
Copyright (C) Microsoft Corporation. All rights reserved.

  hello_world -> C:\dev\hello_world\bin\Release\netcoreapp1.1\hello_world.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:03.58

C:\dev\hello_world>dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll
Hello World!

C:\dev\hello_world>
```

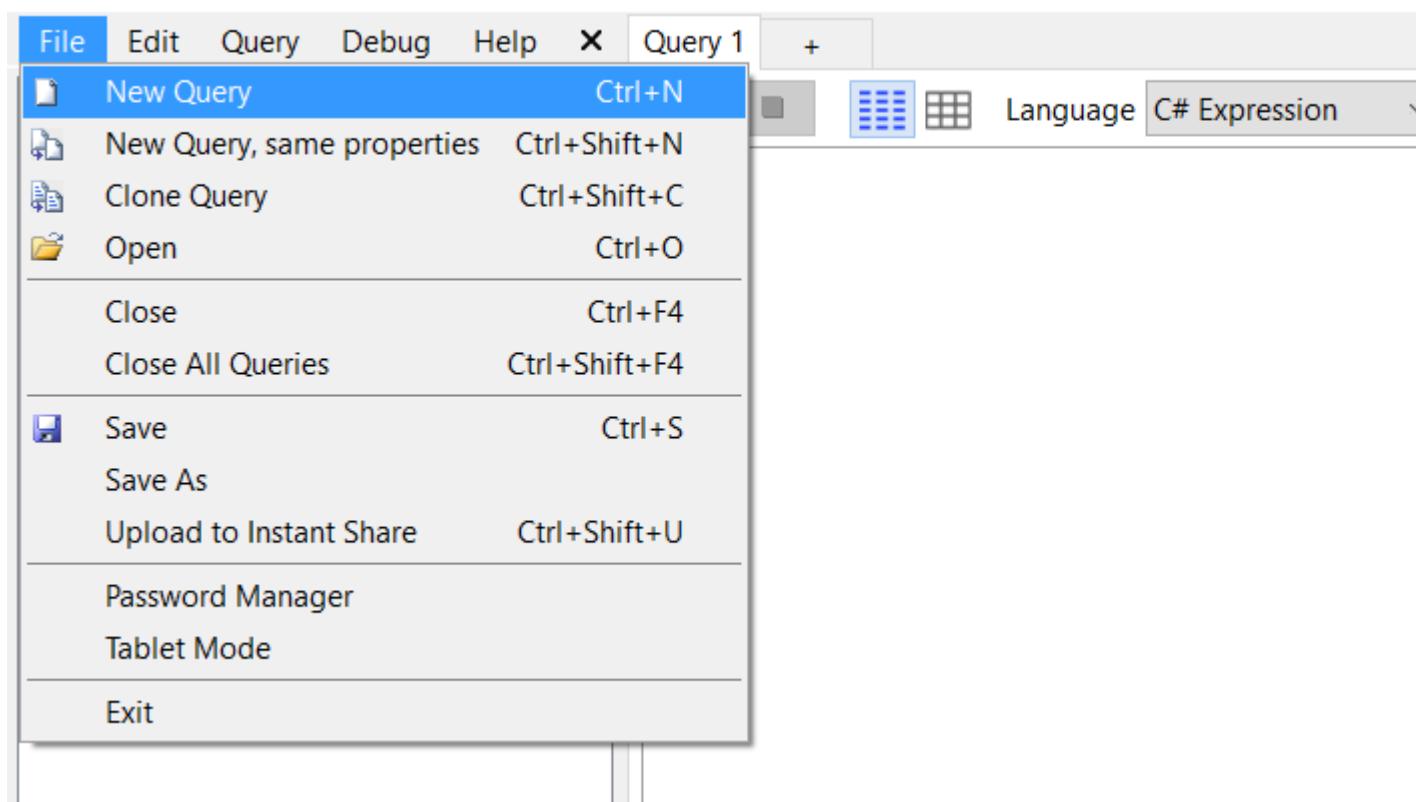
Создание нового запроса с использованием LinqPad

LinqPad - отличный инструмент, который позволяет вам изучать и тестировать функции .Net-языков (C #, F # и VB.Net.)

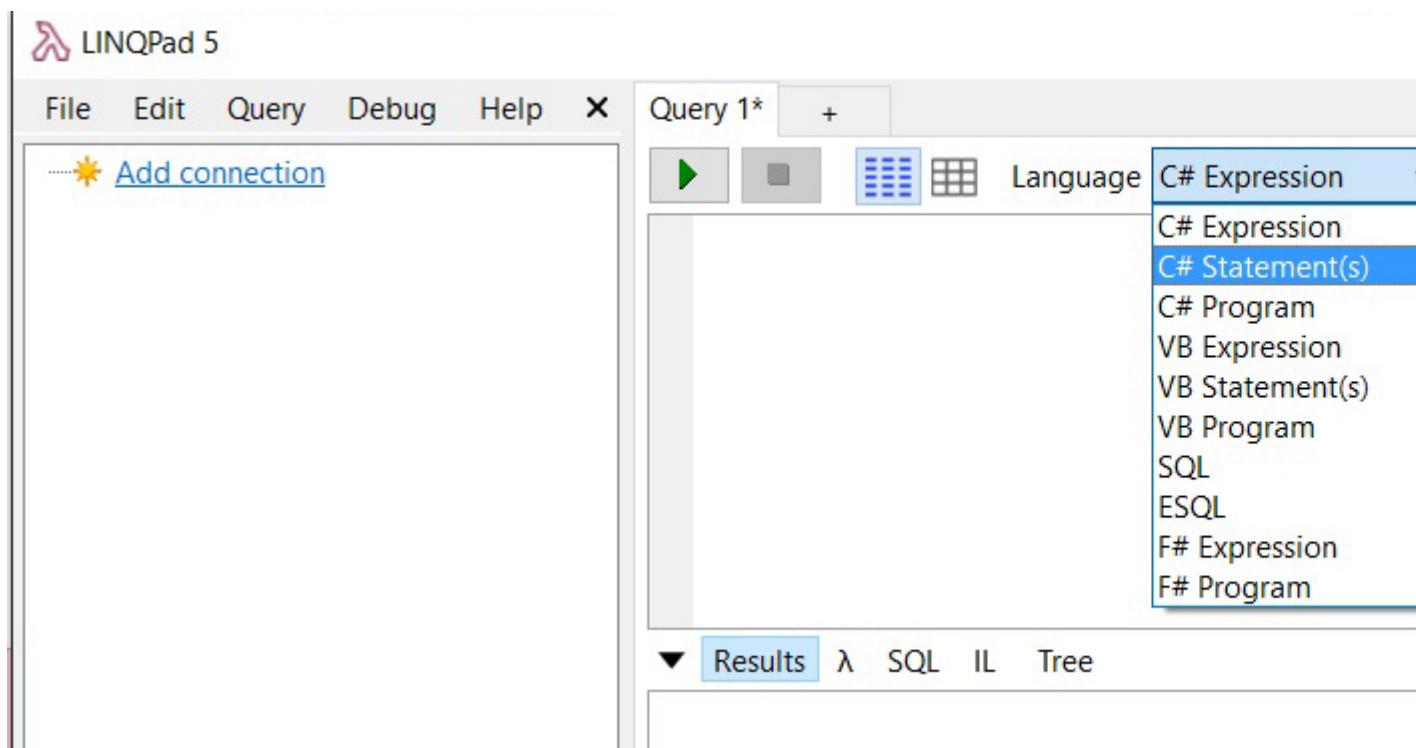
1. Установка [LinqPad](#)

2. Создайте новый запрос (**Ctrl + N**)

LINQPad 5

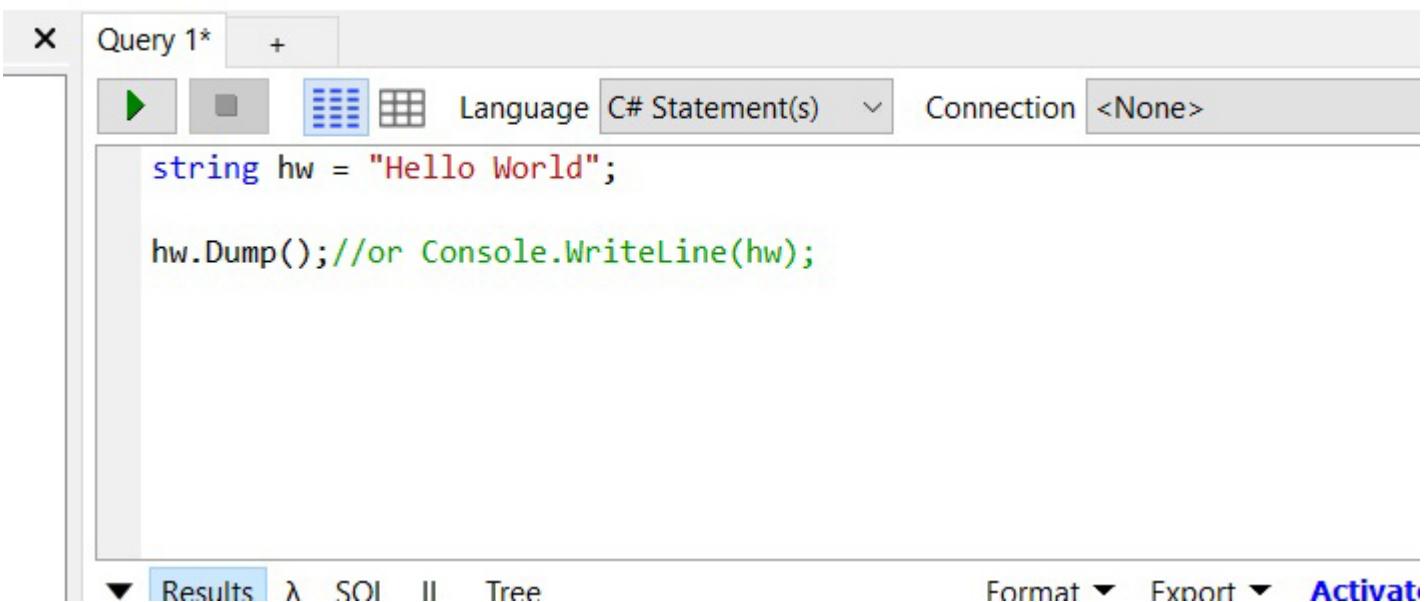


3. Под языком выберите «Операторы С #»,



4. Введите следующий код и нажмите пробег (F5)

```
string hw = "Hello World";  
hw.Dump(); //or Console.WriteLine(hw);
```



Query 1*

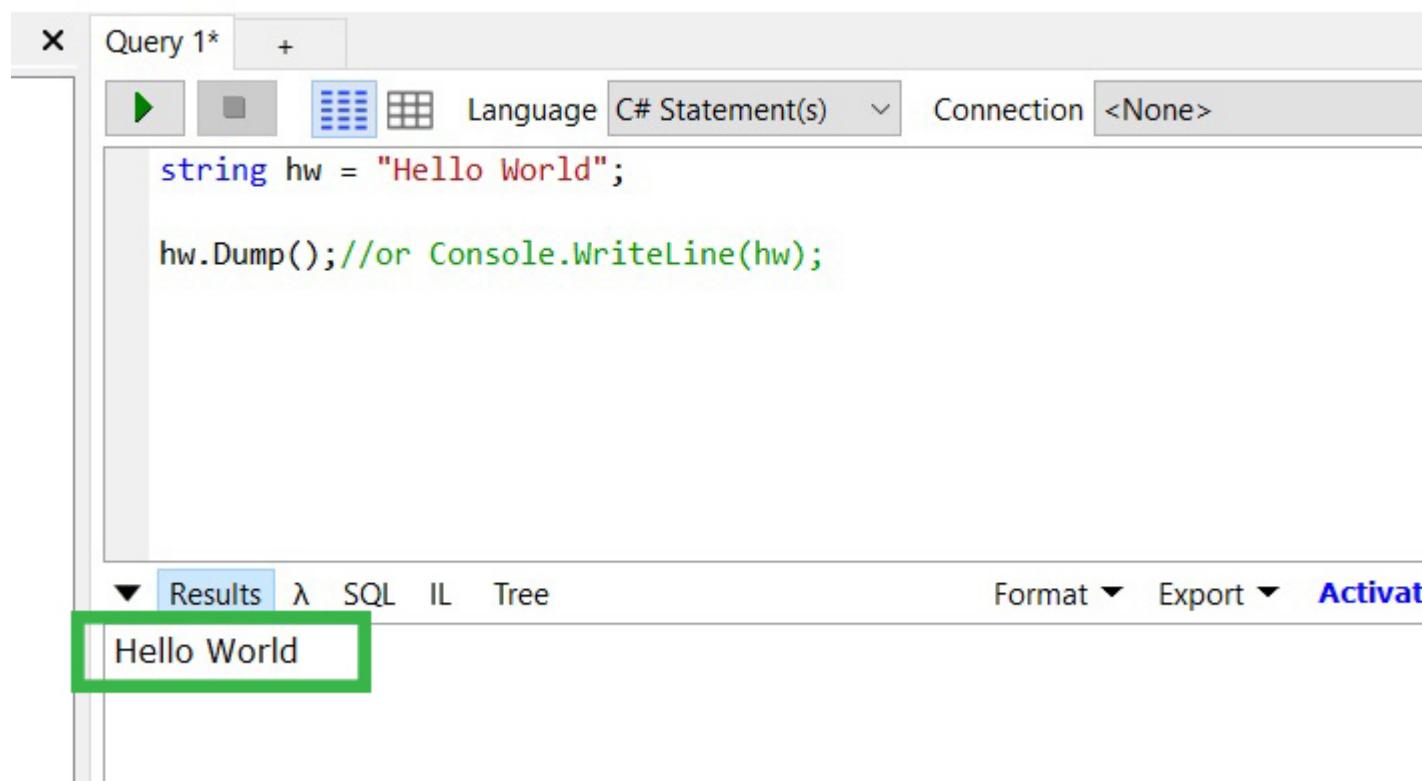
Language: C# Statement(s)

```
string hw = "Hello World";  
hw.Dump(); //or Console.WriteLine(hw);
```

Results

Format ▾ Export ▾ Activate

5. На экране результатов вы увидите «Hello World».



Query 1*

Language: C# Statement(s)

```
string hw = "Hello World";  
hw.Dump(); //or Console.WriteLine(hw);
```

Results

Format ▾ Export ▾ Activate

Hello World

6. Теперь, когда вы создали свою первую .Net-программу, пойдите и проверьте образцы, включенные в LinqPad, через браузер «Образцы». Есть много замечательных примеров, которые покажут вам много разных особенностей языков .Net.

LINQPad 5

File Edit Query Debug Help X

Query 1* +

Add connection

Language C# Statement(s)

```
string hw = "Hello World";  
hw.Dump(); //or Console.WriteLine(hw);
```

▼ Results λ SQL IL Tree

Hello World

My Queries Samples

- LINQPad 5 minute induction
- C# 6.0 in a Nutshell
- F# Tutorial

Download/import more samples

Query successful (00:00:000)

Заметки:

1. Если вы нажмете «IL», вы можете проверить код IL, который генерирует ваш .net-код. Это отличный инструмент обучения.

LINQPad 5

File Edit Query Debug Help ×

Query 1* +

Add connection

```
string hw = "Hello World";  
hw.Dump(); //or Console.WriteLine(hw);
```

▼ Results λ SQL IL Tree

```
IL_0000:  nop  
IL_0001:  ldstr      "Hello World"  
IL_0006:  stloc.0    // hw  
IL_0007:  ldloc.0    // hw  
IL_0008:  call       LINQPad.Extensions.D  
IL_000D:  pop  
IL_000E:  ret
```

My Queries Samples

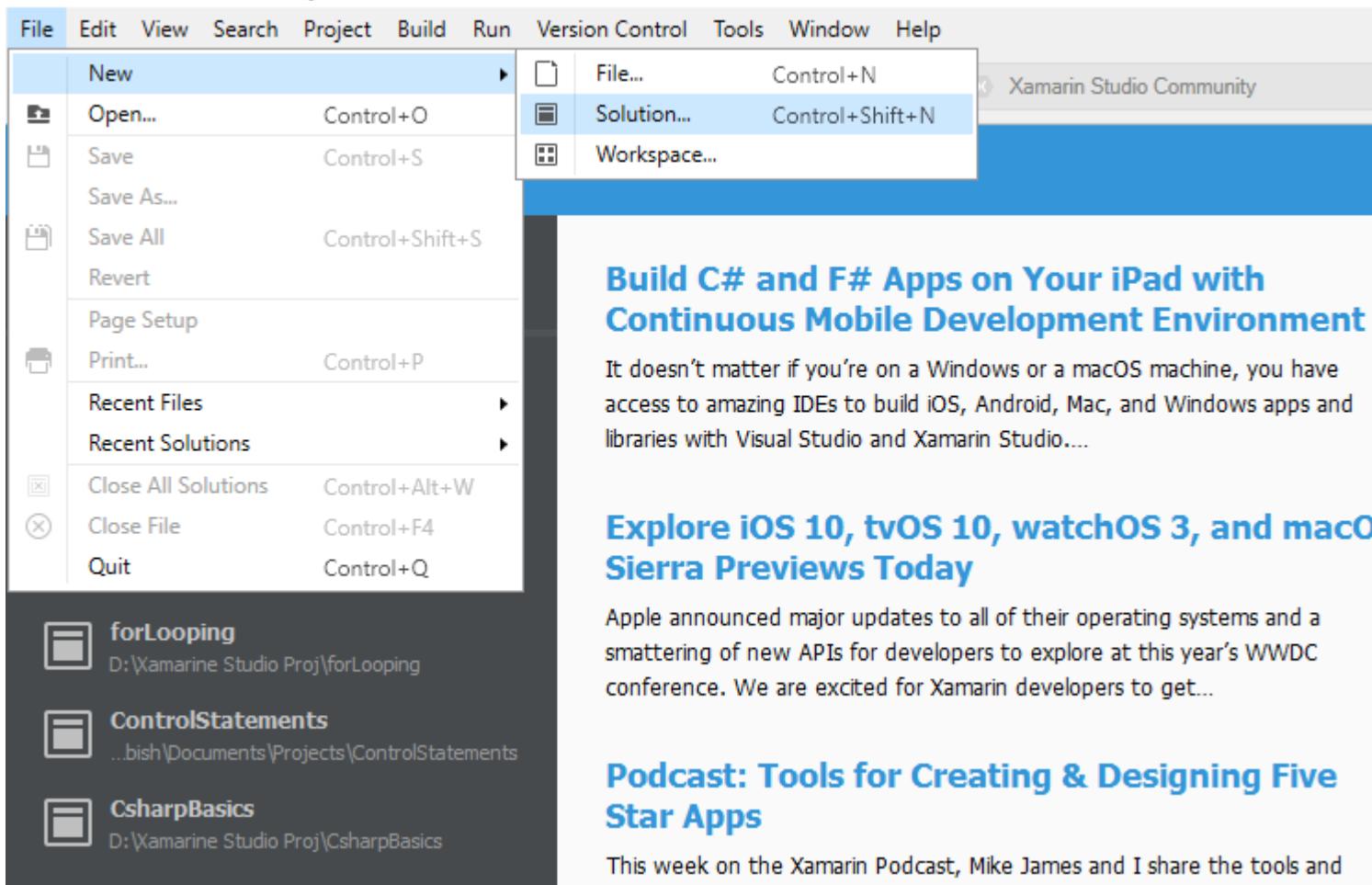
- LINQPad 5 minute induction
- C# 6.0 in a Nutshell
- F# Tutorial
- Download/import more samples...

Query successful (00:00:000)

2. При использовании LINQ to SQL ИЛИ Linq to Entities вы можете проверить создаваемый SQL, что является еще одним отличным способом узнать о LINQ.

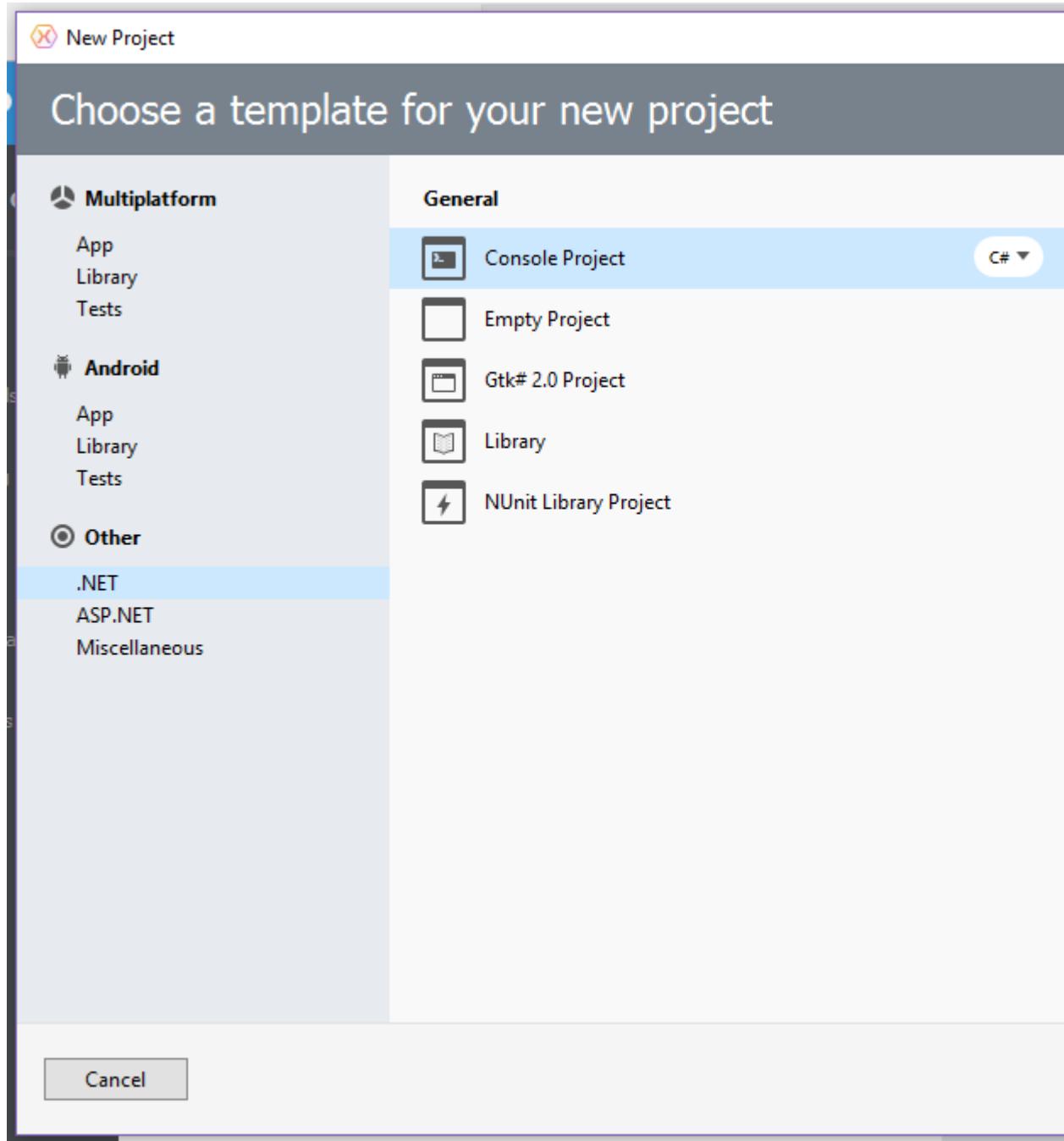
Создание нового проекта с использованием Xamarin Studio

1. Загрузите и установите сообщество Xamarin Studio .
2. Откройте Xamarin Studio.
3. Нажмите « Файл » → « Создать » → « Решение » .

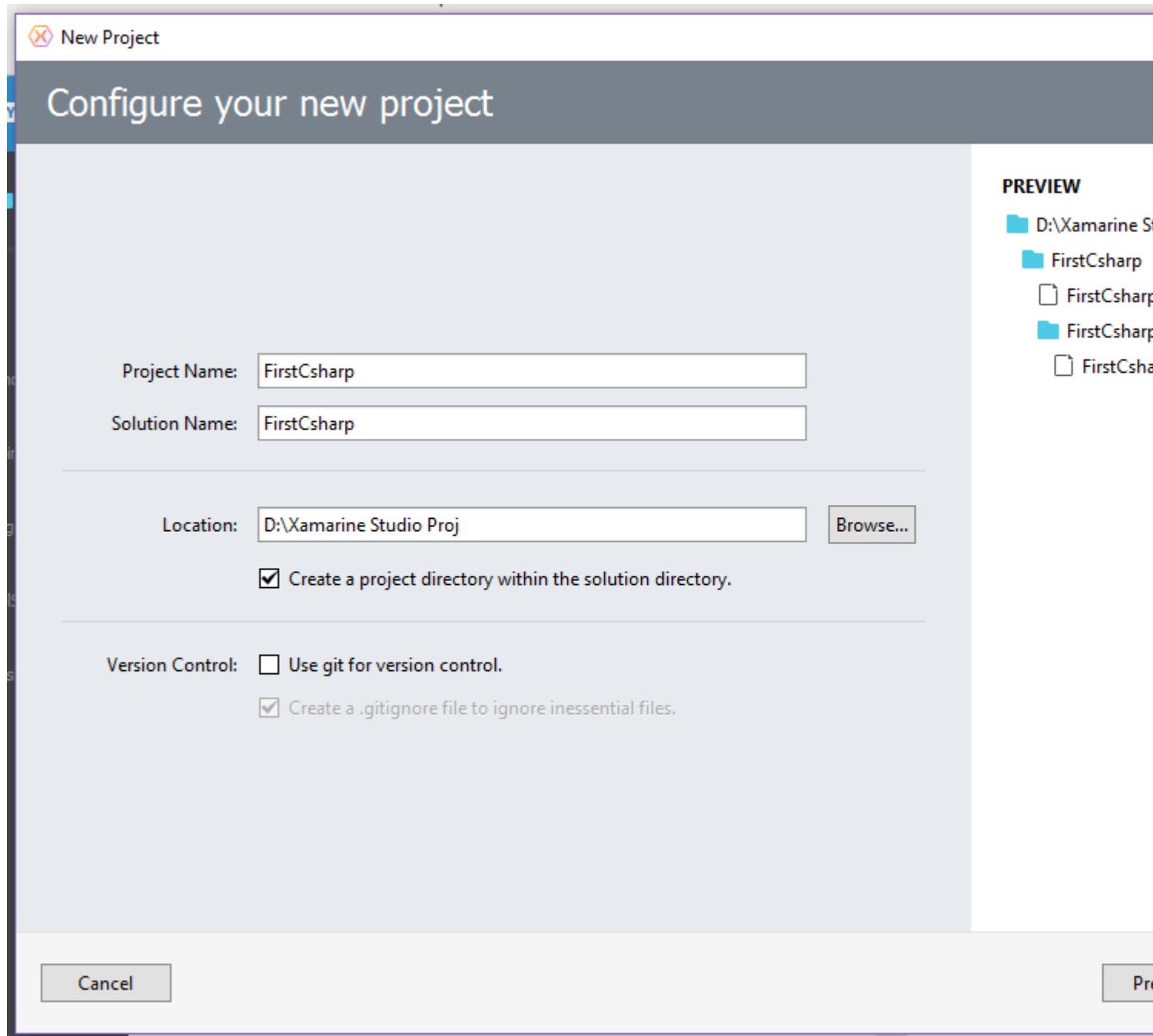


4. Нажмите « **NET** » → « **Проект консоли** » и выберите « **C #** » .

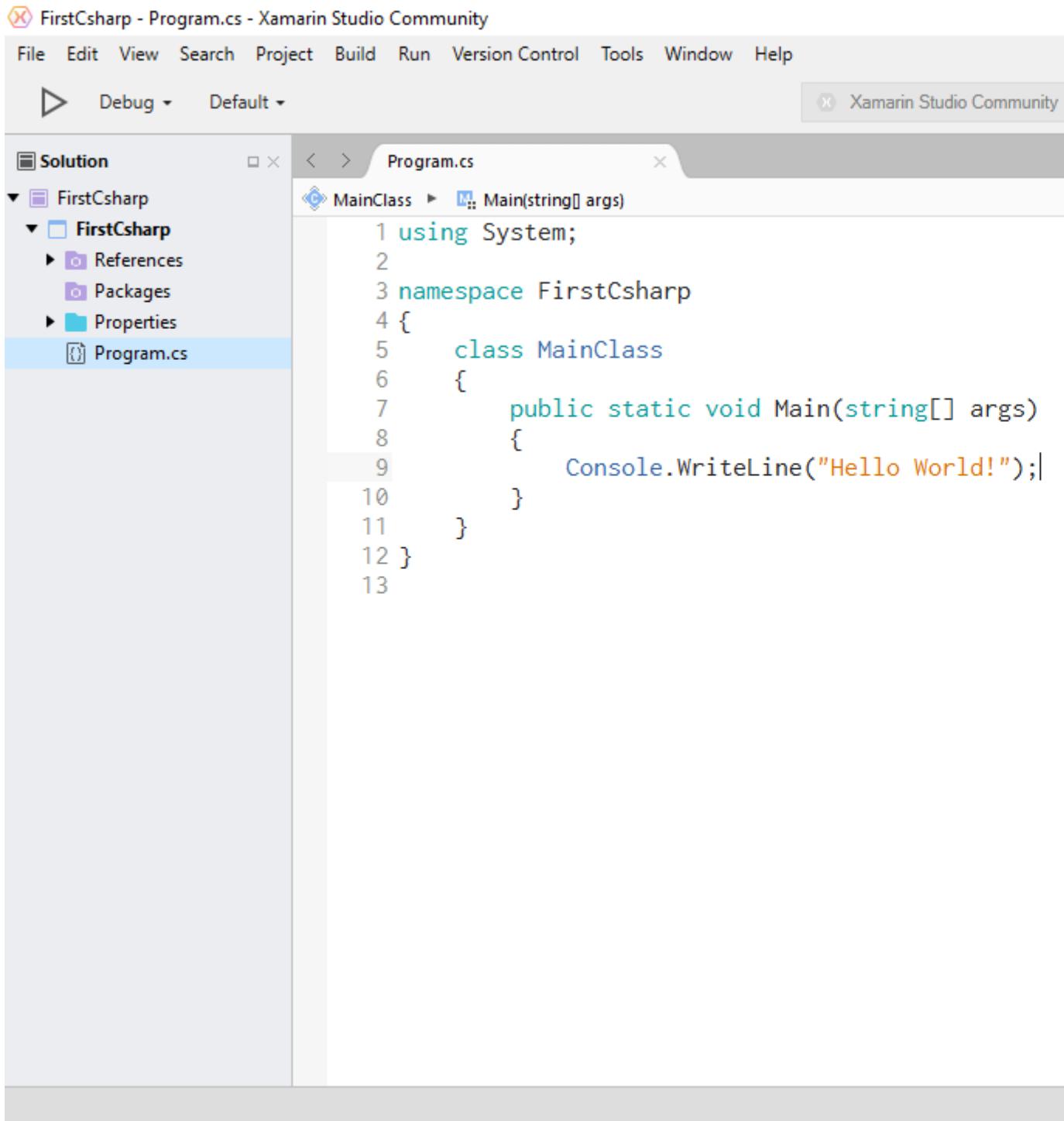
5. Нажмите « **далее** », чтобы продолжить.



6. Введите имя **проекта** и обзор ... для сохранения **местоположения** и нажмите кнопку «Создать» .



7. Созданный проект будет выглядеть примерно так:



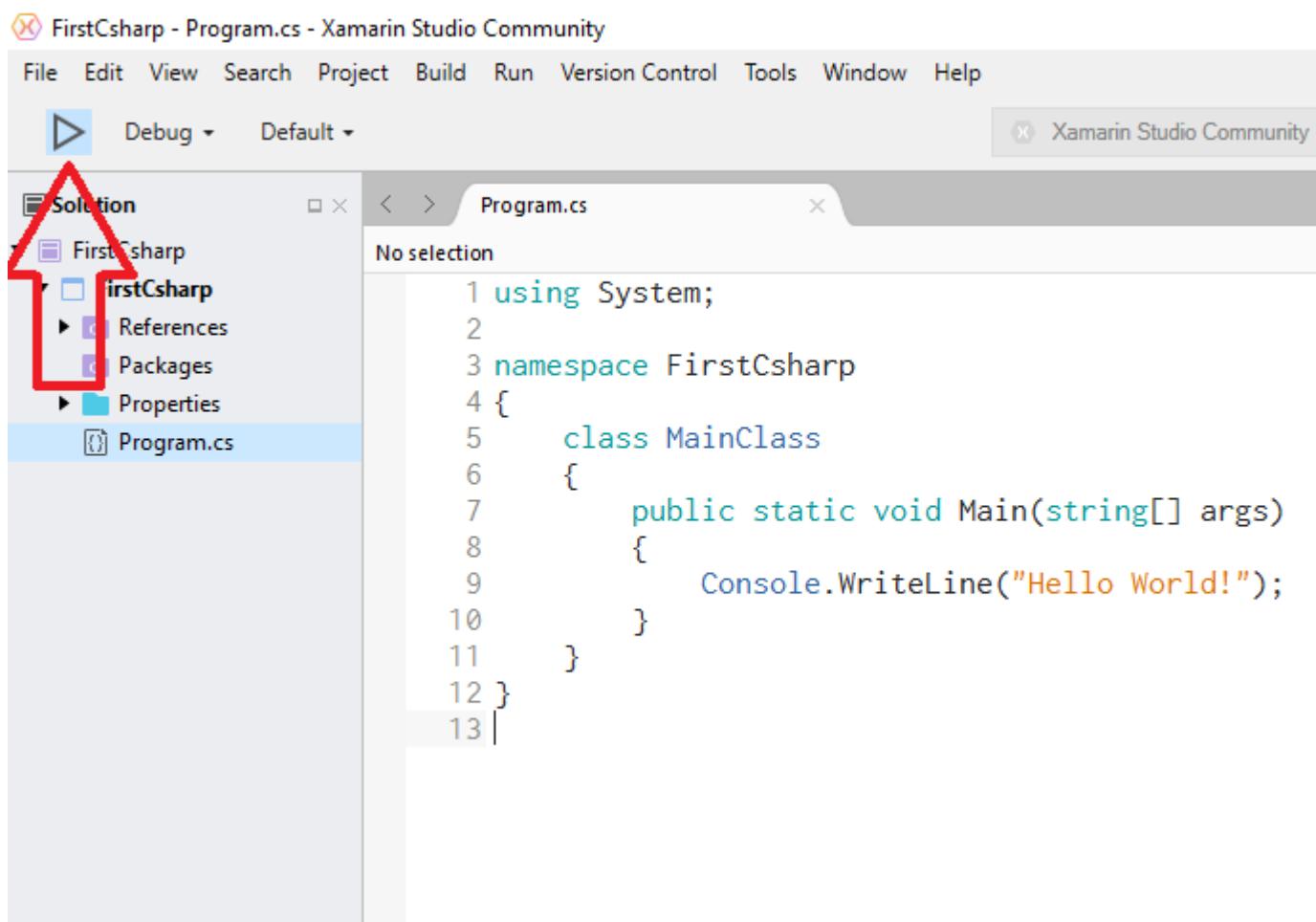
8. Это код в текстовом редакторе:

```
using System;

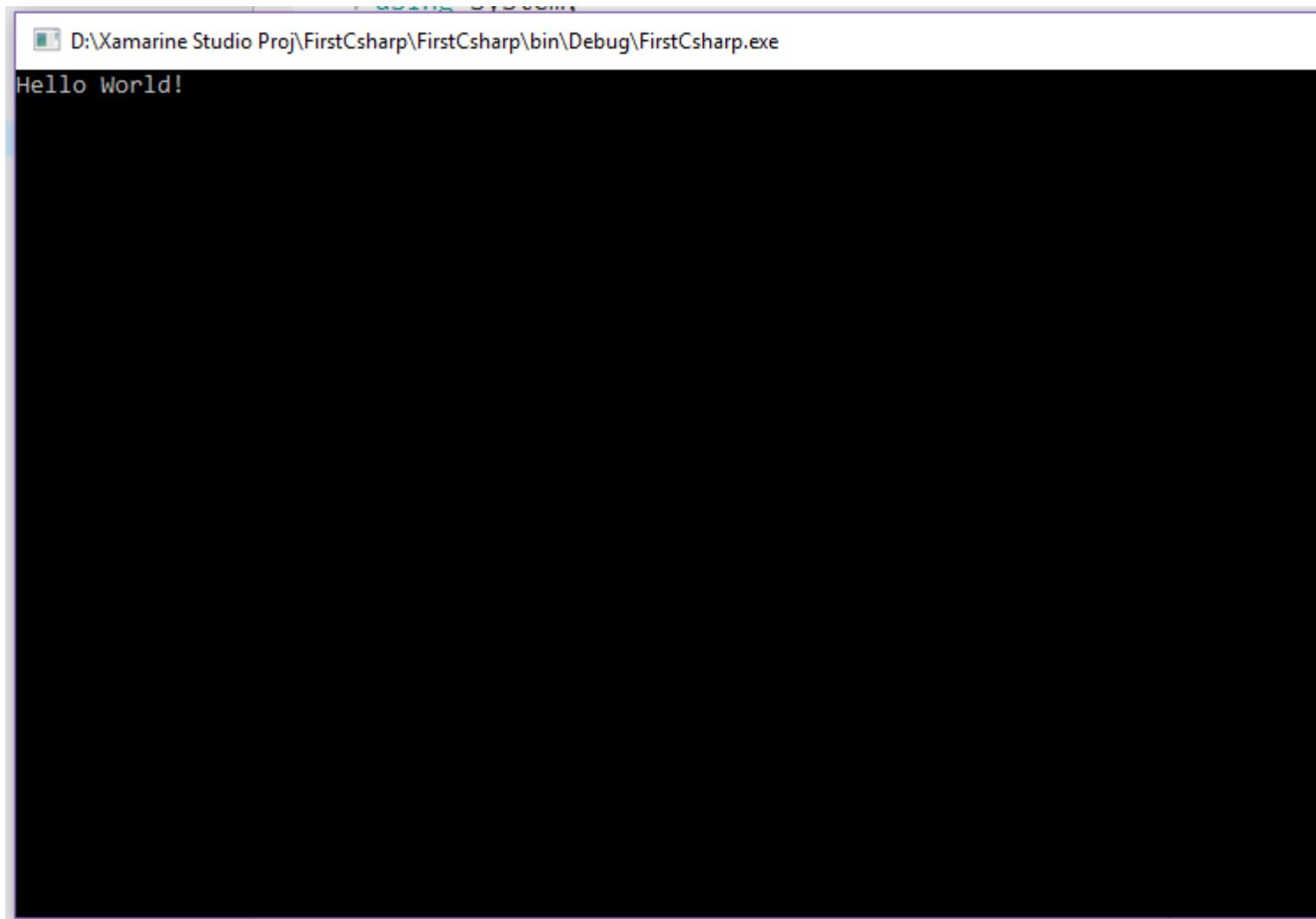
namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

```
}
```

9. Чтобы запустить код, нажмите **F5** или нажмите кнопку **воспроизведения**, как показано ниже:



10. Ниже приведен вывод:



Прочтайте Начало работы с C # Language онлайн: <https://riptutorial.com/ru/csharp/topic/15/начало-работы-с-c-sharp-language>

глава 2: BackgroundWorker

Синтаксис

- `bgWorker.CancellationPending` //returns whether the `bgWorker` was cancelled during its operation
- `bgWorker.IsBusy` //returns true if the `bgWorker` is in the middle of an operation
- `bgWorker.ReportProgress(int x)` //Reports a change in progress. Raises the "ProgressChanged" event
- `bgWorker.RunWorkerAsync()` //Starts the `BackgroundWorker` by raising the "DoWork" event
- `bgWorker.CancelAsync()` //instructs the `BackgroundWorker` to stop after the completion of a task.

замечания

Выполнение длительных операций в потоке пользовательского интерфейса может привести к тому, что ваше приложение перестанет реагировать на запросы пользователя и перестанет работать. Предпочтительно, чтобы эти задачи выполнялись в фоновом потоке. После завершения пользовательский интерфейс может быть обновлен.

Внесение изменений в пользовательский интерфейс во время работы `BackgroundWorker` требует вызова изменений в потоке пользовательского интерфейса, как правило, с помощью метода [Control.Invoke для элемента управления](#), который вы обновляете. Пренебрежение этим приведет к тому, что ваша программа выбросит исключение.

`BackgroundWorker` обычно используется только в приложениях Windows Forms. В приложениях WPF [Задачи](#) используются для разгрузки работы в фоновом потоке (возможно, в сочетании с `async / await`). Марширование обновлений в потоке пользовательского интерфейса обычно выполняется автоматически, когда обновляемое свойство реализует [INotifyPropertyChanged](#) или вручную, используя [Диспетчер потоков](#) пользовательского интерфейса.

Examples

Назначение обработчиков событий `BackgroundWorker`

Как только экземпляр `BackgroundWorker` был объявлен, ему должны быть предоставлены свойства и обработчики событий для выполняемых задач.

```
/* This is the backgroundworker's "DoWork" event handler. This
method is what will contain all the work you
wish to have your program perform without blocking the UI. */
```

```

bgWorker.DoWork += bgWorker_DoWork;

/*This is how the DoWork event method signature looks like:*/
private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
{
    // Work to be done here
    // ...
    // To get a reference to the current Backgroundworker:
    BackgroundWorker worker = sender as BackgroundWorker;
    // The reference to the BackgroundWorker is often used to report progress
    worker.ReportProgress(...);
}

/*This is the method that will be run once the BackgroundWorker has completed its tasks */

bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

/*This is how the RunWorkerCompletedEvent event method signature looks like:*/
private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
{
    // Things to be done after the backgroundworker has finished
}

/* When you wish to have something occur when a change in progress
occurs, (like the completion of a specific task) the "ProgressChanged"
event handler is used. Note that ProgressChanged events may be invoked
by calls to bgWorker.ReportProgress(...) only if bgWorker.WorkerReportsProgress
is set to true. */

bgWorker.ProgressChanged += bgWorker_ProgressChanged;

/*This is how the ProgressChanged event method signature looks like:*/
private void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    // Things to be done when a progress change has been reported

    /* The ProgressChangedEventArgs gives access to a percentage,
    allowing for easy reporting of how far along a process is*/
    int progress = e.ProgressPercentage;
}

```

Назначение свойств BackgroundWorker

Это позволяет отменить BackgroundWorker между задачами

```
bgWorker.WorkerSupportsCancellation = true;
```

Это позволяет работнику сообщать о прогрессе между выполнением задач ...

```
bgWorker.WorkerReportsProgress = true;

//this must also be used in conjunction with the ProgressChanged event
```

Создание нового экземпляра BackgroundWorker

BackgroundWorker обычно используется для выполнения задач, иногда отнимающих много времени, без блокировки потока пользовательского интерфейса.

```
// BackgroundWorker is part of the ComponentModel namespace.
using System.ComponentModel;

namespace BGWorkerExample
{
    public partial class ExampleForm : Form
    {

        // the following creates an instance of the BackgroundWorker named "bgWorker"
        BackgroundWorker bgWorker = new BackgroundWorker();

        public ExampleForm() { ... }
```

Использование BackgroundWorker для выполнения задачи.

Следующий пример демонстрирует использование BackgroundWorker для обновления WinForms ProgressBar. BackgroundWorker обновит значение индикатора выполнения без блокировки потока пользовательского интерфейса, тем самым показывая реактивный интерфейс, когда работа выполняется в фоновом режиме.

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
    {

        //a new instance of a backgroundWorker is created.
        BackgroundWorker bgWorker = new BackgroundWorker();

        public Form1()
        {
            InitializeComponent();

            prgProgressBar.Step = 1;

            //this assigns event handlers for the backgroundWorker
            bgWorker.DoWork += bgWorker_DoWork;
            bgWorker.RunWorkerCompleted += bgWorker_RunComplete;

            //tell the backgroundWorker to raise the "DoWork" event, thus starting it.
            //Check to make sure the background worker is not already running.
            if (!bgWorker.IsBusy)
                bgWorker.RunWorkerAsync();

        }

        private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
        {
            //this is the method that the backgroundworker will perform on in the background
            //thread.
            /* One thing to note! A try catch is not necessary as any exceptions will terminate
            the backgroundWorker and report
            the error to the "RunWorkerCompleted" event */
            CountToY();
        }

        private void bgWorker_RunComplete(object sender, RunWorkerCompletedEventArgs e)
        {
            if (e.Error != null)
                MessageBox.Show("An error occurred: " + e.Error.Message);
        }
    }
}
```

```

}

private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
{
    //e.Error will contain any exceptions caught by the backgroundWorker
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message);
    }
    else
    {
        MessageBox.Show("Task Complete!");
        prgProgressBar.Value = 0;
    }
}

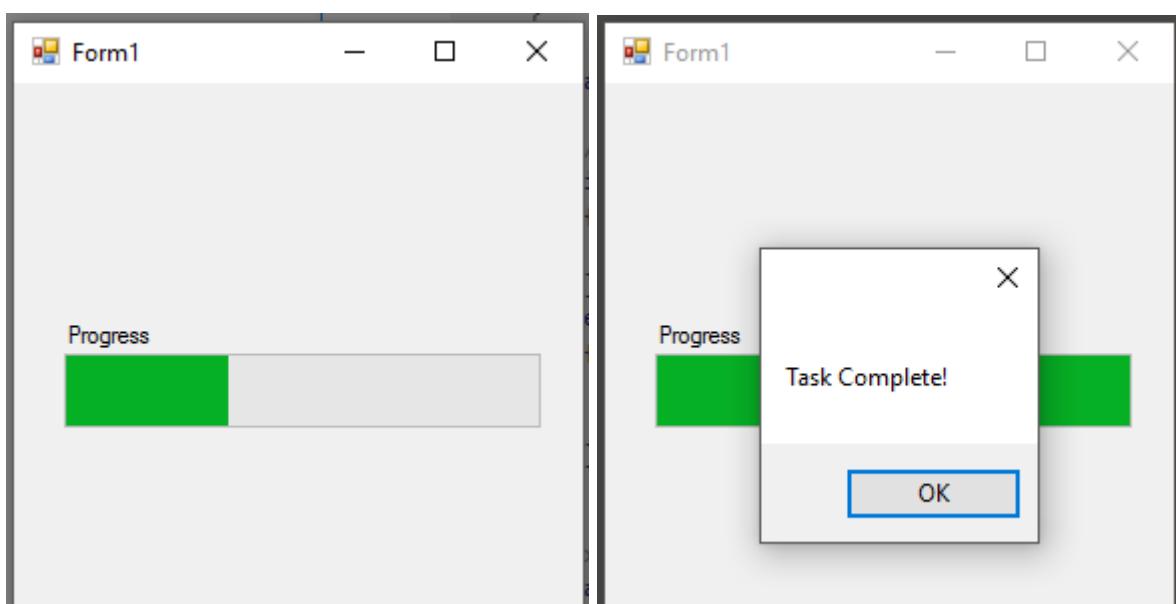
// example method to perform a "long" running task.
private void CountToY()
{
    int x = 0;

    int maxProgress = 100;
    prgProgressBar.Maximum = maxProgress;

    while (x < maxProgress)
    {
        System.Threading.Thread.Sleep(50);
        Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
        x += 1;
    }
}
}

```

Результатом является следующее ...



Прочтайте **BackgroundWorker** онлайн:

<https://riptutorial.com/ru/csharp/topic/1588/backgroundworker>

глава 3: BigInteger

замечания

Когда использовать

Объекты `BigInteger` по своей природе очень тяжелы в ОЗУ. Следовательно, они должны использоваться только тогда, когда они абсолютно необходимы, т. Е. Для чисел в истинно астрономическом масштабе.

Кроме того, все арифметические операции над этими объектами на порядок медленнее, чем их примитивные аналоги, эта проблема еще более усугубляется по мере того, как число растет, поскольку они не имеют фиксированного размера. Поэтому вполне возможно, что мошенник `BigInteger` может вызвать сбой, потребляя всю доступную оперативную память.

альтернативы

Если скорость необходима для вашего решения, может быть более эффективно реализовать эту функциональность самостоятельно, используя класс, оберывающий `byte[]` и перегружая необходимые операторы самостоятельно. Однако это требует значительных дополнительных усилий.

Examples

Вычислить первый 1000-значный номер Фибоначчи

Включите `using System.Numerics` и добавьте ссылку на `System.Numerics` в проект.

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1 = 1;
            BigInteger l2 = 1;
            BigInteger current = l1 + l2;
            while (current.ToString().Length < 1000)
            {
                l2 = l1;
                l1 = current;
                current = l1 + l2;
            }
        }
    }
}
```

```
        Console.WriteLine(current);
    }
}
```

Этот простой алгоритм выполняет итерацию по номерам Фибоначчи до тех пор, пока не достигнет одной длины не менее 1000 десятичных цифр, а затем распечатает ее. Это значение значительно больше, чем даже `ulong`.

Теоретически единственным пределом для класса `BigInteger` является объем оперативной памяти, которую может использовать ваше приложение.

Примечание. `BigInteger` доступен только в .NET 4.0 и выше.

Прочтайте `BigInteger` онлайн: <https://riptutorial.com/ru/csharp/topic/5654/biginteger>

глава 4: BindingList

Examples

Избегайте повторения N * 2

Это помещается в обработчик событий Windows Forms

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
```

Это займет много времени, чтобы исправить, выполните следующие действия:

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
nameList.RaiseListChangedEvents = false;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
nameList.RaiseListChangedEvents = true;
nameList.ResetBindings();
```

Добавить элемент в список

```
BindingList<string> listOfUIItems = new BindingList<string>();
listOfUIItems.Add("Alice");
listOfUIItems.Add("Bob");
```

Прочтайте **BindingList** онлайн: <https://riptutorial.com/ru/csharp/topic/182/bindinglist--t->

глава 5: CLSCompliantAttribute

Синтаксис

1. [Сборка: CLSCompliant (истина)]
2. [CLSCompliant (истина)]

параметры

Конструктор	параметр
CLSCompliantAttribute (Boolean)	Инициализирует экземпляр класса CLSCompliantAttribute с логическим значением, указывающим, является ли указанный программный элемент CLS-совместимым.

замечания

Общая спецификация языка (CLS) - это набор базовых правил, которым должен соответствовать любой язык, ориентированный на CLI (язык, который подтверждает спецификации общей языковой инфраструктуры), чтобы взаимодействовать с другими языками, совместимыми с CLS.

Список языков CLI

Вы должны отметить свою сборку как CLSCompliant в большинстве случаев, когда вы распространяете библиотеки. Этот атрибут гарантирует, что ваш код будет использоваться всеми CLS-совместимыми языками. Это означает, что ваш код может быть использован любым языком, который может быть скомпилирован и запущен на CLR ([Common Language Runtime](#))

Когда ваша сборка отмечена `CLSCompliantAttribute`, компилятор проверяет, не нарушает ли ваш код какой-либо из правил CLS и возвращает **предупреждение**, если это необходимо.

Examples

Модификатор доступа, к которому применяются правила CLS

```
using System;  
  
[assembly:CLSCompliant(true)]  
namespace CLSDoc  
{
```

```

public class Cat
{
    internal UInt16 _age = 0;
    private UInt16 _daysTillVaccination = 0;

    //Warning CS3003 Type of 'Cat.DaysTillVaccination' is not CLS-compliant
    protected UInt16 DaysTillVaccination
    {
        get { return _daysTillVaccination; }
    }

    //Warning CS3003 Type of 'Cat.Age' is not CLS-compliant
    public UInt16 Age
    { get { return _age; } }

    //valid behaviour by CLS-compliant rules
    public int IncreaseAge()
    {
        int increasedAge = (int)_age + 1;

        return increasedAge;
    }

}
}

```

Правила соблюдения CLS применяются только к общедоступным / защищенным компонентам.

Нарушение правила CLS: Неподписанные типы / sbyte

```

using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

        //Warning CS3008 Identifier '_numberOfDoors' is not CLS-compliant
        //Warning CS3003 Type of 'Car._numberOfDoors' is not CLS-compliant
        public UInt32 _numberOfDoors = 0;

        //Warning CS3003 Type of 'Car.YearOfCreation' is not CLS-compliant
        public UInt16 YearOfCreation
        {
            get { return _yearOfCreation; }
        }

        //Warning CS3002 Return type of 'Car.CalculateDistance()' is not CLS-compliant
        public UInt64 CalculateDistance()
        {
            return 0;
        }
    }
}

```

```

//Warning CS3002  Return type of 'Car.TestDummyUnsignedPointerMethod()' is not CLS-
compliant
public UIntPtr TestDummyUnsignedPointerMethod()
{
    int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    UIntPtr ptr = (UIntPtr)arr[0];

    return ptr;
}

//Warning CS3003  Type of 'Car.age' is not CLS-compliant
public sbyte age = 120;

}
}

```

Нарушение правила CLS: одно и то же имя

```

using System;

[assembly: CLSCompliant(true)]
namespace CLSDoc
{
    public class Car
    {
        //Warning      CS3005      Identifier 'Car.CALCULATEAge()' differing only in case is not
CLS-compliant
        public int CalculateAge()
        {
            return 0;
        }

        public int CALCULATEAge()
        {
            return 0;
        }
    }
}

```

Visual Basic не чувствителен к регистру

Нарушение правила CLS: Идентификатор _

```

using System;

[assembly: CLSCompliant(true)]
namespace CLSDoc
{
    public class Car
    {

```

```
//Warning CS3008 Identifier '_age' is not CLS-compliant
public int _age = 0;
}

}
```

Вы не можете запустить переменную с _

Нарушение правила CLS: наследовать от класса не CLSCompliant

```
using System;

[assembly:CLSSCompliant(true)]
namespace CLSDoc
{
    [CLSSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //Warning CS3009 'Dog': base type 'Animal' is not CLS-compliant
    public class Dog : Animal
    {
    }
}
```

Прочтайте CLSCompliantAttribute онлайн:

<https://riptutorial.com/ru/csharp/topic/7214/clscompliantattribute>

глава 6: Enum

Вступление

Перечисление может быть получено из любого из следующих типов: байт, sbyte, short, ushort, int, uint, long, ulong. Значение по умолчанию - int, и его можно изменить, указав тип в определении перечисления:

```
public enum Weekday: byte {Понедельник = 1, вторник = 2, среда = 3, четверг = 4, пятница = 5}
```

Это полезно, когда Р / Invoking для собственного кода, сопоставление с источниками данных и аналогичные обстоятельства. В общем случае следует использовать значение по умолчанию int, поскольку большинство разработчиков ожидают, что enum будет int.

Синтаксис

- enum Colors {Red, Green, Blue} // Объявление enum
- enum Цвета: байт {Красный, Зеленый, Синий} // Декларация определенного типа
- enum Colors {Red = 23, Green = 45, Blue = 12} // Объявление с заданными значениями
- Colors.Red // Доступ к элементу Enum
- int value = (int) Colors.Red // Получить значение int элемента перечисления
- Цвета color = (Цвета) intValue // Получить элемент перечисления из int

замечания

Enum (сокращение от «перечисляемого типа») представляет собой тип, состоящий из набора именованных констант, представленного идентификатором типа.

Перечисления наиболее полезны для представления понятий, которые имеют (как правило, небольшое) число возможных дискретных значений. Например, они могут использоваться для представления дня недели или месяца года. Они также могут использоваться в качестве флагов, которые можно комбинировать или проверять, используя побитовые операции.

Examples

Получить все значения элементов перечисления

```
enum MyEnum
{
    One,
```

```

    Two,
    Three
}

foreach (MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);

```

Это напечатает:

```

One
Two
Three

```

Enum как флаги

FlagsAttribute может быть применен к перечислению, изменяющему поведение `ToString()` чтобы соответствовать характеру перечисления:

```

[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);

```

Поскольку `FlagsAttribute` полагается на константы перечисления как полномочия двух (или их комбинаций), а значения перечисления в конечном счете являются числовыми значениями, вы ограничены размером базового числового типа. Самый большой доступный числовой тип, который вы можете использовать, - `UInt64`, который позволяет указать 64 различных (не комбинированных) константы перечисления флага. Ключевое слово `enum` умолчанию относится к базовому типу `int`, который является `Int32`. Компилятор позволит объявить значения шириной более 32 бит. Они обернутся без предупреждения и приведут к тому, что два или более члена перечисления будут иметь одинаковое значение. Поэтому, если перечисление предназначено для размещения битового набора из более чем 32 флагов, вам необходимо указать более крупный тип:

```

public enum BigEnum : ulong
{
    BigValue = 1 << 63
}

```

Хотя флаги часто имеют только один бит, их можно объединить в названные «наборы» для более легкого использования.

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Чтобы не указывать десятичные значения степеней двух, [оператор сдвига слева \(<<\)](#) также может быть использован для объявления того же перечисления

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Начиная с С # 7.0, [бинарные литералы также могут использоваться](#).

Чтобы проверить, имеет ли значение переменной enum определенный флаг, может использоваться метод [HasFlag](#). Допустим, у нас есть

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

И value

```
var value = MyEnum.One | MyEnum.Two;
```

С [HasFlag](#) мы можем проверить, установлен ли какой-либо из флагов

```
if(value.HasFlag(MyEnum.One))
    Console.WriteLine("Enum has One");

if(value.HasFlag(MyEnum.Two))
```

```
Console.WriteLine("Enum has Two");

if (value.HasFlag(MyEnum.Three))
    Console.WriteLine("Enum has Three");
```

Также мы можем перебирать все значения enum, чтобы получить все установленные флаги

```
var type = typeof(MyEnum);
var names = Enum.GetNames(type);

foreach (var name in names)
{
    var item = (MyEnum)Enum.Parse(type, name);

    if (value.HasFlag(item))
        Console.WriteLine("Enum has " + name);
}
```

Или же

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if (value.HasFlag(flagToCheck))
    {
        Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

Все три примера будут напечатаны:

```
Enum has One
Enum has Two
```

Тестирование флагов в стиле enum с побитовой логикой

Значение переименования в стиле флагов должно быть проверено с помощью побитовой логики, поскольку оно может не соответствовать ни одному значению.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

Значение по Default представляет собой комбинацию двух других, объединенных с побитовым ИЛИ. Поэтому для проверки наличия флага нам нужно использовать побитовое

И.

```
var value = FlagsEnum.Default;

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

Перечисление в строку и обратно

```
public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"

// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET
// versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive
// match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)

// String to enum (all .NET versions)
DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturday"); // Throws
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));
```

Значение по умолчанию для перечисления == ZERO

Значение по умолчанию для перечисления равно нулю . Если перечисление не

определяет элемент со значением нуля, его значение по умолчанию будет равно нулю.

```
public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
        else
            Console.WriteLine("Unknown");
    }
}
```

Пример: <https://dotnetfiddle.net/l5Rwie>

Основы Enum

Из [MSDN](#) :

Тип перечисления (также называемый перечислением или enum) обеспечивает эффективный способ определения набора именованных **интегральных констант**, которые могут быть **назначены переменной** .

По сути, перечисление - это тип, который допускает только набор конечных опций, и каждый параметр соответствует числу. По умолчанию эти числа увеличиваются в том порядке, в котором объявлены значения, начиная с нуля. Например, можно объявить перечисление для дней недели:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Это перечисление можно использовать следующим образом:

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;
```

```
// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

По умолчанию базовым типом каждого элемента в `enum` является `int`, но также могут использоваться `byte`, `sbyte`, `short`, `ushort`, `uint`, `long` и `ulong`. Если вы используете тип, отличный от `int`, вы должны указать тип, используя двоеточие после имени перечисления:

```
public enum Day : byte
{
    // same as before
}
```

Числа после имени теперь являются байтами вместо целых чисел. Вы можете получить базовый тип перечисления следующим образом:

```
Enum.GetUnderlyingType(typeof(Days));
```

Выход:

```
System.Byte
```

Демо: [.NET скрипка](#)

Побитовая манипуляция с использованием перечислений

`FlagsAttribute` следует использовать всякий раз, когда перечисляемый представляет собой набор флагов, а не одно значение. Числовое значение, присвоенное каждому значению перечисления, помогает при манипулировании перечислениями с использованием побитовых операторов.

Пример 1: С помощью [Флаги]

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

печатает красный, синий

Пример 2: Без [Флаги]

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

отпечатки 3

Использование << обозначение для флагов

Оператор левого сдвига (`<<`) может использоваться в объявлениях перечисления флагов, чтобы гарантировать, что каждый флаг имеет ровно один 1 в двоичном представлении, как и должны быть флаги.

Это также помогает улучшить читаемость больших перечислений с большим количеством флагов в них.

```
[Flags]
public enum MyEnum
{
    None = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

Теперь очевидно, что `MyEnum` содержит только соответствующие флаги, а не какие-то грязные вещи, такие как `Flag30 = 1073741822` (или `11111111111111111111111111111110` в двоичном формате), что неуместно.

Добавление дополнительной информации описания к значению перечисления

В некоторых случаях вам может потребоваться добавить дополнительное описание к значению перечисления, например, когда значение enum становится менее читаемым, чем то, что вы можете отобразить пользователю. В таких случаях вы можете использовать класс `System.ComponentModel.DescriptionAttribute`.

Например:

```

public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}

```

Теперь, если вы хотите вернуть описание определенного значения перечисления, вы можете сделать следующее:

```

public static string GetDescriptionAttribute(PossibleResults result)
{
    return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())), typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}

```

Это также может быть легко преобразовано в метод расширения для всех перечислений:

```

static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())), typeof(DescriptionAttribute))).Description;
    }
}

```

И тогда легко использовать так: `Console.WriteLine(result.GetDescription());`

Добавление и удаление значений из помеченного значком перечисления

Этот код предназначен для добавления и удаления значения из помеченного enum-экземпляра:

```

[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

```

```
// set additional value
value |= MyEnum.Flag2; //value is now Flag1, Flag2
value |= MyEnum.Flag3; //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3
```

Перечисления могут иметь неожиданные значения

Поскольку перечисление может быть отнесено к своему базовому интегральному типу и от него, значение может выходить за пределы значений, указанных в определении типа перечисления.

Хотя приведенный ниже тип перечисления `DaysOfWeek` имеет только 7 определенных значений, он все равно может содержать любое значение `int`.

```
public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

DaysOfWeek s = DaysOfWeek.Sunday;
s++; // No error
```

В настоящее время нет способа определить перечисление, которое не имеет такого поведения.

Однако неопределенные значения перечисления могут быть обнаружены с помощью метода `Enum.IsDefined`. Например,

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek), d)); // prints False
```

Прочтайте `Enum` онлайн: <https://riptutorial.com/ru/csharp/topic/931/enum>

глава 7: FileSystemWatcher

Синтаксис

- public FileSystemWatcher ()
- public FileSystemWatcher (строка)
- public FileSystemWatcher (строковый путь, строковый фильтр)

параметры

дорожка	фильтр
Каталог для мониторинга в стандартной или универсальной нотации (UNC).	Тип файлов для просмотра. Например, «*.txt» отслеживает изменения всех текстовых файлов.

Examples

Основной файл

В следующем примере создается `FileSystemWatcher` для просмотра каталога, указанного во время выполнения. Компонент настроен на просмотр изменений в `LastWrite` и `LastAccess`, создание, удаление или переименование текстовых файлов в каталоге. Если файл изменен, создан или удален, путь к файлу будет распечатан на консоль. Когда файл переименовывается, старые и новые пути печатаются на консоли.

Для этого примера используйте пространства имен `System.Diagnostics` и `System.IO`.

```
FileSystemWatcher watcher;

private void watch()
{
    // Create a new FileSystemWatcher and set its properties.
    watcher = new FileSystemWatcher();
    watcher.Path = path;

    /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
    watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
                           | NotifyFilters.FileName | NotifyFilters.DirectoryName;

    // Only watch text files.
    watcher.Filter = "*.txt*";

    // Add event handler.
    watcher.Changed += new FileSystemEventHandler(OnChanged);
}
```

```

// Begin watching.
watcher.EnableRaisingEvents = true;
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
    //Copies file to another directory or another action.
    Console.WriteLine("File: " + e.FullPath + " " + e.ChangeType);
}

```

IsFileReady

Общей ошибкой многих людей, начинающих работу с FileSystemWatcher, не учитывается, что событие FileWatcher возникает, как только создается файл. Однако для завершения файла может потребоваться некоторое время.

Пример :

Например, возьмите файл размером 1 ГБ. Запрос файла арг создается другой программой (Explorer.exe копирует ее откуда-то), но для завершения этого процесса потребуется несколько минут. Событие связано с тем, что время создания и вам нужно дождаться, когда файл будет готов к копированию.

Это метод проверки готовности файла.

```

public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
    catch (Exception)
    {
        return false;
    }
}

```

[Прочтайте FileSystemWatcher онлайн:](#)

<https://riptutorial.com/ru/csharp/topic/5061/filesystemwatcher>

глава 8: Generic Lambda Query Builder

замечания

Класс называется `ExpressionBuilder`. Он имеет три свойства:

```
private static readonly MethodInfo ContainsMethod = typeof(string).GetMethod("Contains",
new[] { typeof(string) });
private static readonly MethodInfo StartsWithMethod = typeof(string).GetMethod("StartsWith",
new[] { typeof(string) });
private static readonly MethodInfo EndsWithMethod = typeof(string).GetMethod("EndsWith",
new[] { typeof(string) });
```

Один общедоступный метод `GetExpression` который возвращает выражение лямбда и три частных метода:

- `Expression GetExpression<T>`
- `BinaryExpression GetExpression<T>`
- `ConstantExpression GetConstant`

Все методы подробно описаны в примерах.

Examples

Класс `QueryFilter`

Этот класс содержит значения предикатных фильтров.

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro") }
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

Перечислите значения операций:

```
public enum Operator
{
```

```

Contains,
GreaterThan,
GreaterThanOrEqual,
LessThan,
LessThanOrEqual,
StartsWith,
EndsWith,
Equals,
NotEqual
}

```

Метод GetExpression

```

public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param
    part
    // To create a ParameterExpression need the type of the entity that the query is against
    an a name
    // The type is possible to find with the generic T and the name is fixed parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good practice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
        if (filters.Count == 2)
            // It is result from direct call.
            // For simplicity sake the private overloads will be explained in another example.
            exp = GetExpression<T>(param, filters[0], filters[1]);
        else
        {
            // As there is no method for more than two filters,
            // I iterate through all the filters and put 2 in the query two at a time
            while (filters.Count > 0)
            {
                // Retreive the first two filters
                var f1 = filters[0];
                var f2 = filters[1];

                // To build a expression with a conditional AND operation that evaluates
                // the second operand only if the first operand evaluates to true.
                // It needed to use the BinaryExpression a Expression derived class
                // That has the AndAlso method that join two expression together
                exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
                Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

                // Remove the two just used filters, for the method in the next iteration
                finds the next filters
                filters.Remove(f1);
                filters.Remove(f2);

                // If it is that last filter, add the last one and remove it
                if (filters.Count == 1)

```

```

        {
            exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
            filters.RemoveAt(0);
        }
    }
}
else
    // It is result from direct call.
    exp = GetExpression<T>(param, filters[0]);

    // converts the Expression into Lambda and retuns the query
return Expression.Lambda<Func<T, bool>>(exp, param);
}

```

GetExpression Частная перегрузка

Для одного фильтра:

Здесь создается запрос, он получает параметр выражения и фильтр.

```

private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
    MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for
    //example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
    ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
    // every operator has it one way to call, so the switch will do.
    switch (queryFilter.Operator)
    {
        case Operator.Equals:
            return Expression.Equal(member, constant);

        case Operator.Contains:
            return Expression.Call(member, ContainsMethod, constant);

        case Operator.GreaterThan:
            return Expression.GreaterThan(member, constant);

        case Operator.GreaterThanOrEqual:
            return Expression.GreaterThanOrEqual(member, constant);

        case Operator.LessThan:
            return Expression.LessThan(member, constant);

        case Operator.LessThanOrEqualTo:
            return Expression.LessThanOrEqual(member, constant);

        case Operator.StartsWith:
    }

```

```

        return Expression.Call(member, StartsWithMethod, constant);

    case Operator.EndsWith:
        return Expression.Call(member, EndsWithMethod, constant);
    }

    return null;
}

```

Для двух фильтров:

Он возвращает экземпляр `BinaryExpression` вместо простого выражения.

```

private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter
filter1, QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}

```

Метод `ConstantExpression`

`ConstantExpression` **должен быть одним и тем же типом** `MemberExpression`. **Значение в этом примере представляет собой строку, которая преобразуется перед созданием экземпляра** `ConstantExpression`.

```

private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if (type == typeof(string))
    {
        constant = Expression.Constant(value);
    }
    else if (type == typeof(DateTime))
    {
        DateTime date;
        DateTime.TryParse(value, out date);
        constant = Expression.Constant(date);
    }
    else if (type == typeof(bool))
    {
        bool flag;
        if (bool.TryParse(value, out flag))
        {
            flag = true;
            constant = Expression.Constant(flag);
        }
    }
}

```

```
        }
        constant = Expression.Constant(flag);
    }
    else if (type == typeof(decimal))
    {
        decimal number;
        decimal.TryParse(value, out number);
        constant = Expression.Constant(number);
    }
    return constant;
}
```

Использование

Фильтры коллекции = новый список (); Фильтр QueryFilter = новый QueryFilter («Имя», «Burger», Operator.StartsWith); filters.Add (фильтр);

```
Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

В этом случае это запрос к субъекту Food, который хочет найти все продукты, которые начинаются с «Burger» в названии.

Выход:

```
query = {parm => a.parm.StartsWith("Burger")}

Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

Прочтайте Generic Lambda Query Builder онлайн:

<https://riptutorial.com/ru/csharp/topic/6721/generic-lambda-query-builder>

глава 9: Guid

Вступление

GUID (или UUID) - это аббревиатура «Глобально уникальный идентификатор» (или «Универсальный уникальный идентификатор»). Это 128-разрядное целое число, используемое для идентификации ресурсов.

замечания

Guid являются глобально уникальными идентификаторами, также известными как универсальные уникальные идентификаторы UUID.

Это 128-битные псевдослучайные значения. Существует так много действительных Guid (около 10^{18} Guid для каждой ячейки каждого человека на Земле), что, если они генерируются хорошим псевдослучайным алгоритмом, их можно считать уникальными во всей вселенной всеми практическими способами.

Guid чаще всего используются в качестве первичных ключей в базах данных. Их преимуществом является то, что вам не нужно вызывать базу данных, чтобы получить новый идентификатор, который (почти) гарантированно будет уникальным.

Examples

Получение строкового представления руководства

Строковое представление Guid может быть получено с помощью встроенного метода `ToString`

```
string myGuidString = myGuid.ToString();
```

В зависимости от ваших потребностей вы также можете отформатировать Guid, добавив аргумент типа формата к вызову `ToString`.

```
var guid = new Guid("7feb16f-651b-43b0-a5e3-0da8da49e90d");

// None          "7feb16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens       "7feb16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces        "{7feb16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));
```

```
// Parentheses  "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex          "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

Создание руководства

Это наиболее распространенные способы создания экземпляра Guid:

- Создание пустого guid (00000000-0000-0000-000000000000):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Создание нового (псевдослучайного) руководства:

```
Guid g = Guid.NewGuid();
```

- Создание гидов с определенным значением:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

Объявление допустимого идентификатора GUID

Как и другие типы значений, GUID также имеет тип с нулевым значением, который может принимать нулевое значение.

Декларация:

```
Guid? myGuidVar = null;
```

Это особенно полезно при извлечении данных из базы данных, когда есть вероятность, что значение из таблицы равно NULL.

Прочтайте Guid онлайн: <https://riptutorial.com/ru/csharp/topic/1153/guid>

глава 10: ICloneable

Синтаксис

- объект ICloneable.Clone () {return Clone ();} // Частная реализация метода интерфейса, который использует нашу пользовательскую функцию public Clone ().
- public Foo Clone () {return new Foo (this);} // Метод public clone должен использовать логику конструктора копирования.

замечания

Для CLR требуется object Clone() определения метода object Clone() который не является безопасным по типу. Общепринятой практикой является переопределение этого поведения и определение безопасного метода типа, который возвращает копию содержащего класса.

Автор должен решить, означает ли клонирование только мелкую копию или глубокую копию. Для неизменяемых структур, содержащих ссылки, рекомендуется сделать глубокую копию. Для классов, являющихся самими ссылками, вероятно, хорошо реализовать мелкую копию.

ПРИМЕЧАНИЕ. В C# метод интерфейса может быть реализован в частном порядке с приведенным выше синтаксисом.

Examples

Реализация ICloneable в классе

Внедрить ICloneable в класс с твист. Выполните открытый clone() публичный тип и внедрите object Clone() частном порядке.

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }
}
```

```

#region ICloneable Members
// Type safe Clone
public Person Clone() { return new Person(this); }
// ICloneable implementation
object ICloneable.Clone()
{
    return Clone();
}
#endregion
}

```

Позже будет использоваться следующим образом:

```

{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);

    bob.Age=56;
    Debug.Assert(bob.Age!=bob.Age);
}

```

Обратите внимание, что изменение возраста `bob` не изменяет возраст `bob_clone`. Это связано с тем, что в проекте используется клонирование вместо назначения (ссылочных) переменных.

Внедрение ICloneable в структуре

Реализация `ICloneable` для структуры, как правило, не требуется, потому что `structs` делает копию по порядку с оператором присваивания `=`. Но для дизайна может потребоваться реализация другого интерфейса, который наследуется от `ICloneable`.

Другая причина заключалась бы в том, что структура содержит ссылочный тип (или массив), который также нуждается в копировании.

```

// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }
}

```

```
}

#region ICloneable Members
// Type safe Clone
public Person Clone() { return new Person(this); }
// ICloneable implementation
object ICloneable.Clone()
{
    return Clone();
}
#endregion
}
```

Позже будет использоваться следующим образом:

```
static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

Прочтайте **ICloneable** онлайн: <https://riptutorial.com/ru/csharp/topic/7917/icloneable>

глава 11: IComparable

Examples

Сортировать версии

Учебный класс:

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[\d]+(\.[\d]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

Тестовое задание:

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1
```

```

a = new Version("2.8.4");
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");

```

Выход:

НОЛЬ
1
1.0.1
1.1.5
2,0
3.0.10

Демо - версия:

[Демо-версия Liveone на Ideone](#)

Прочтайте **IComparable** онлайн: <https://riptutorial.com/ru/csharp/topic/4222/icomparable>

глава 12: **IEnumerable**

Вступление

`IEnumerable` - это базовый интерфейс для всех не общих коллекций, таких как `ArrayList`, которые могут быть перечислены. `IEnumerator<T>` - это базовый интерфейс для всех общих счетчиков, таких как `List <T>`.

`IEnumerable` - это интерфейс, реализующий метод `GetEnumerator`. Метод `GetEnumerator` возвращает `IEnumerator` который предоставляет опции для итерации через коллекцию, например `foreach`.

замечания

`IEnumerable` - это базовый интерфейс для всех не общих коллекций, которые могут быть перечислены

Examples

`IEnumerable`

В своей основной форме объект, реализующий `IEnumerable`, представляет собой серию объектов. Объекты, о которых идет речь, могут быть повторены с использованием ключевого слова с `# foreach`.

В приведенном ниже примере `sequenceOfNumbers` объектов `OfNumbers` реализует `IEnumerable`. Он представляет собой серию целых чисел. Цикл `foreach` выполняет итерацию по очереди.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

`IEnumerable` с пользовательским `Enumerator`

Реализация интерфейса `IEnumerable` позволяет перечислять классы так же, как коллекции BCL. Это требует расширения класса `Enumerator`, который отслеживает состояние перечисления.

Примеры, кроме повторения стандартной коллекции, включают:

- Использование диапазонов чисел на основе функции, а не набора объектов
- Реализация различных алгоритмов итераций по коллекциям, например DFS или BFS, в коллекции графов

```
public static void Main(string[] args) {  
  
    foreach (var coffee in new CoffeeCollection()) {  
        Console.WriteLine(coffee);  
    }  
}  
  
public class CoffeeCollection : IEnumerable {  
    private CoffeeEnumerator enumerator;  
  
    public CoffeeCollection() {  
        enumerator = new CoffeeEnumerator();  
    }  
  
    public IEnumerator GetEnumerator() {  
        return enumerator;  
    }  
  
    public class CoffeeEnumerator : IEnumerator {  
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };  
        int currentIndex = -1;  
  
        public object Current {  
            get {  
                return beverages[currentIndex];  
            }  
        }  
  
        public bool MoveNext() {  
            currentIndex++;  
  
            if (currentIndex < beverages.Length) {  
                return true;  
            }  
  
            return false;  
        }  
  
        public void Reset() {  
            currentIndex = 0;  
        }  
    }  
}
```

Прочтайте `IEnumerable` онлайн: <https://riptutorial.com/ru/csharp/topic/2220/ienumerable>

глава 13: ILGenerator

Examples

Создает DynamicAssembly, который содержит вспомогательный метод UnixTimestamp

В этом примере показано использование ILGenerator путем генерации кода, который использует уже существующие и новые созданные члены, а также базовую обработку исключений. Следующий код испускает DynamicAssembly, который содержит эквивалент этого кода с #:

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds =
checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's too late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}
```

```
//Get the required methods
var dateTimeCtor = typeof (DateTime)
    .GetConstructor(new[] {typeof (int), typeof (int), typeof (int)});
var dateTimeSubtract = typeof (DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof (DateTime)});
var timeSpanSecondsGetter = typeof (TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationExceptionCtor = typeof (InvalidOperationException)
    .GetConstructor(new[] {typeof (string), typeof (Exception)});

if (dateTimeCtor == null || dateTimeSubtract == null ||
    timeSpanSecondsGetter == null || invalidOperationExceptionCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an,
```

```

AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");

var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof (DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
        MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
        MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
        Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof (int), new[] {typeof (DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof (TimeSpan));
methodGen.DeclareLocal(typeof (int));
methodGen.DeclareLocal(typeof (OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte) 0); //To call a method on a struct we need to load the
address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubtract); //Call the subtract method on the input
DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte) 0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on
overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
//The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof (OverflowException)); //Begin the catch block
//When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's too late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
//When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value

```

```
methodGen.Emit(OpCodes.Ret); //Return it

dynType.CreateType();

dynAsm.Save(an.Name + ".dll");
```

Создать метод переопределения

В этом примере показано, как переопределить метод `ToString` в сгенерированном классе

```
// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name,
AssemblyBuilderAccess.RunAndSave);
var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public |
TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());
```

Прочтайте `ILGenerator` онлайн: <https://riptutorial.com/ru/csharp/topic/667/ilgenerator>

глава 14: Interoperability

замечания

Работа с API Win32 с использованием C

Windows предоставляет множество функций в виде Win32 API. Используя эти API, вы можете выполнять прямую работу в окнах, что повышает производительность вашего приложения. [Источник](#)

Windows предоставляет широкий спектр API. Чтобы получить информацию о различных API, вы можете проверять сайты, подобные [pinvoke](#).

Examples

Функция импорта из неуправляемой библиотеки C ++

Вот пример того, как импортировать функцию, определенную в неуправляемой DLL C ++. В исходном коде C ++ для «myDLL.dll» определена функция `add`:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Затем он может быть включен в программу C # следующим образом:

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

См. «[Соглашения о __stdcall extern "C" и __stdcall C ++](#)» для объяснения причин, по которым необходимы `extern "C"` и `__stdcall`.

Поиск динамической библиотеки

При первом вызове метода `extern` программа C # будет искать и загружать соответствующую DLL. Для получения дополнительной информации о том, где искать, найти DLL, и как вы можете влиять на места поиска, см. [Этот вопрос в stackoverflow](#) .

Простой код для публикации класса для COM

```
using System;
using System.Runtime.InteropServices;

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }

        public int StopTime()
        {
            return (int)stopWatch.ElapsedMilliseconds;
        }
    }
}
```

C ++ name mangling

Компиляторы C ++ кодируют дополнительную информацию в именах экспортимемых функций, таких как типы аргументов, чтобы сделать возможными перегрузки с различными аргументами. Этот процесс называется изменением [имени](#) . Это приводит к проблемам с импортированием функций в C # (и взаимодействием с другими языками вообще), поскольку имя функции `int add(int a, int b)` больше не `add` , это может быть `?add@0УАНН@Z` , `_add@8` или что-либо еще, в зависимости от компилятора и соглашения о вызове.

Существует несколько способов решения проблемы смены имени:

- Экспортирование функций с использованием `extern "C"` для переключения на внешнюю связь C, которая использует сглаживание имени C:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)  
[DllImport("myDLL.dll")]
```

Имя функции все равно будет `_add@8` (`_add@8`), но C `StdCall` компилятора C # распознается `StdCall + extern "C"`.

- Указание имен экспортаемых функций в `myDLL.def` определения модуля `myDLL.def`:

```
EXPORTS  
add  
  
int __stdcall add(int a, int b)  
[DllImport("myDLL.dll")]
```

В этом случае имя функции будет чисто `add`.

- Импортирование измененного имени. Для просмотра искаженного имени вам понадобится некоторое средство просмотра DLL, и вы можете указать его явно:

```
__declspec(dllexport) int __stdcall add(int a, int b)  
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

Вызовы

Существует несколько соглашений о вызовах, указывающих, кто (вызывающий или вызываемый) выдает аргументы из стека, как передаются аргументы и в каком порядке. C ++ использует `Cdecl` вызове `Cdecl` по умолчанию, но C # ожидает `StdCall`, который обычно используется Windows API. Вам нужно изменить одно или другое:

- Изменение соглашения о `StdCall` в `StdCall` в C ++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)  
[DllImport("myDLL.dll")]
```

- Или, измените соглашение о `Cdecl` на `Cdecl` в C #:

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

Если вы хотите использовать функцию с `Cdecl` вызове `Cdecl` и `Cdecl` именем, ваш код будет выглядеть так:

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall** (`__thiscall`) в основном используется в функциях, входящих в класс.
- Когда функция использует **thiscall** (`__thiscall`), указатель на класс передается в качестве первого параметра.

Динамическая загрузка и выгрузка неуправляемых библиотек DLL

При использовании атрибута `DllImport` вы должны знать правильное имя DLL и имя метода во время компиляции. Если вы хотите быть более гибкими и решать во время выполнения, какие DLL и методы загружать, вы можете использовать методы Windows API `LoadLibrary()`, `GetProcAddress()` и `FreeLibrary()`. Это может быть полезно, если используемая библиотека зависит от условий выполнения.

Указатель, возвращаемый `GetProcAddress()` может быть переведен в делегат, используя `Marshal.GetDelegateForFunctionPointer()`.

Следующий пример кода демонстрирует это с помощью `myDLL.dll` из предыдущих примеров:

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }
    }
}
```

```

    }

    // get a "pointer" to the method
    IntPtr method = GetProcAddress(module, "add");
    if (method == IntPtr.Zero) // error handling
    {
        Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
        FreeLibrary(module); // unload library
        return;
    }

    // convert "pointer" to delegate
    AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
    typeof(AddDelegate));

    // use function
    int result = add(750, 300);

    // unload library
    FreeLibrary(module);
}
}

```

Работа с ошибками Win32

При использовании методов interop вы можете использовать **GetLastError** API для получения дополнительной информации о ваших вызовах API.

Атрибут **DllImport** Атрибут **SetLastError**

SetLastError = верно

Указывает, что вызываемый вызовет вызов SetLastError (функция Win32 API).

SetLastError = ложь

Указывает, что вызываемый вызов **не будет** вызывать SetLastError (функция Win32 API), поэтому вы не получите информацию об ошибке.

- Если SetLastError не установлен, он устанавливается в значение `false` (значение по умолчанию).
- Вы можете получить код ошибки с помощью метода `Marshal.GetLastWin32Error`:

Пример:

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

Если вы пытаетесь открыть мьютекс, которого не существует, `GetLastError` вернет **ERROR_FILE_NOT_FOUND**.

```
var lastErrorCode = Marshal.GetLastWin32Error();

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

Коды ошибок системы можно найти здесь:

[https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx)

API GetLastError

Существует собственный API **GetLastError**, который вы также можете использовать:

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- При вызове Win32 API из управляемого кода вы всегда должны использовать **Marshal.GetLastWin32Error** .

Вот почему:

Между вашим вызовом Win32, который устанавливает ошибку (вызывает **SetLastError**), среда CLR может вызывать другие вызовы Win32, которые также могут вызвать **SetLastError** , это поведение может переопределить ваше значение ошибки. В этом случае, если вы вызываете **GetLastError**, вы можете получить недопустимую ошибку.

Установка **SetLastError = true** , убедитесь, что CLR извлекает код ошибки до того, как он выполнит другие вызовы Win32.

Связанный объект

GC (Garbage Collector) несет ответственность за очистку нашего мусора.

В то время как **GC** очищает наш мусор, он удаляет неиспользуемые объекты из управляемой кучи, которые вызывают фрагментацию кучи. Когда **GC** выполняется с удалением, он выполняет сжатие кучи (defragmentation), которое включает в себя перемещение объектов в куче.

Поскольку **GC** не является детерминированным, когда передается ссылка на управляемый объект / указатель на собственный код, **GC** может ударить в любое время, если это происходит сразу после вызова `Inerop`, есть очень хорошая вероятность того, что объект (эта ссылка передана на native) будет перемещаться по управляемой куче - в результате мы получаем неверную ссылку на управляемую сторону.

В этом случае вы должны **привязать** объект до передачи его в собственный код.

Связанный объект

Прикрепленный объект - объект, который не разрешен для перемещения по GC.

Gc Pinned Handle

Вы можете создать контактный объект, используя метод **Gc.Alloc**

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- Получение привязанного **GCHandle** к управляемому объекту маркирует определенный объект как тот, который не может быть перемещен **GC**, пока не освободит дескриптор

Пример:

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finally
    {
        handle.Free();
    }
}
```

Меры предосторожности

- При фиксации (особенно больших) объектов старайтесь как можно быстрее освободить закрепленный **GCHandle**, так как он прерывает дефрагментацию кучи.
- Если вы забудете освободить **GCHandle**, ничего не будет. Сделайте это в разделе безопасного кода (например, finaly)

Чтение структур с маршалом

Класс маршала содержит функцию **PtrToStructure**, эта функция дает нам возможность считывания структур неуправляемым указателем.

Функция **PtrToStructure** получила много перегрузок, но все они имеют одно и то же намерение.

Общая структура **PtrToStructure** :

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - тип структуры.

ptr - указатель на неуправляемый блок памяти.

Пример:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- Если вы работаете с управляемыми объектами при чтении собственных структур, не забудьте прикрепить свой объект :)

```
T Read<T>(byte[] buffer)
{
    T result = default(T);

    var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

    try
    {
        result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
    }
    finally
    {
        gch.Free();
    }

    return result;
}
```

Прочтайте Interoperability онлайн: <https://riptutorial.com/ru/csharp/topic/3278/interoperability>

глава 15: Linq to Objects

Вступление

LINQ to Objects ссылается на использование запросов LINQ с любой коллекцией `IEnumerable`.

Examples

Как LINQ to Object выполняет запросы

Запросы LINQ не выполняются немедленно. Когда вы строите запрос, вы просто сохраняете запрос для будущего выполнения. Только когда вы действительно запрашиваете итерацию запроса, выполняется запрос (например, в цикле `for`, при вызове `ToList`, `Count`, `Max`, `Average`, `First` и т. Д.).

Это считается *отсроченным исполнением*. Это позволяет вам создавать запрос в несколько этапов, потенциально модифицируя его на основе условных операторов, а затем выполнять его позже только после того, как потребуется результат.

Учитывая код:

```
var query = from n in numbers
            where n % 2 != 0
            select n;
```

В приведенном выше примере сохраняется только запрос в переменной `query`. Он не выполняет сам запрос.

Оператор `foreach` заставляет выполнить запрос:

```
foreach(var n in query) {
    Console.WriteLine($"Number selected {n}");
}
```

Некоторые методы LINQ также инициируют выполнение запроса, `Count`, `First`, `Max`, `Average`. Они возвращают одиночные значения. `ToList` и `ToArray` собирает результат и превращает их в список или массив соответственно.

Имейте в виду, что вы можете многократно перебирать запрос по нескольким запросам, если вы вызываете несколько функций LINQ в одном запросе. Это может дать вам разные результаты при каждом вызове. Если вы хотите работать только с одним набором данных, обязательно сохраните его в списке или массиве.

Использование LINQ для объектов в C

Простой запрос SELECT в Linq

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

В приведенном выше примере массив строк (автомобилей) используется как набор объектов для запроса с использованием LINQ. В запросе LINQ предложение `from` приходит первым, чтобы ввести источник данных (автомобили) и переменную диапазона (автомобиль). Когда запрос выполняется, переменная диапазона будет служить ссылкой на каждый последующий элемент в автомобилях. Поскольку компилятор может вывести тип автомобиля, вам не нужно указывать его явно

Когда приведенный выше код компилируется и выполняется, он производит следующий

```
VW Golf
Opel Astra
Audi A4
Ford Focus
Seat Leon
VW Passat
VW Polo
Mercedes C-Class
```

результат:

SELECT с предложением WHERE

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

Предложение WHERE используется для запроса массива строк (автомобилей) для поиска и возврата подмножества массива, которое удовлетворяет условию WHERE.

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

```
VW Golf
VW Passat
VW Polo
```

Создание упорядоченного списка

```
var list = from car in cars
            orderby car ascending
            select car;
```

Иногда полезно сортировать возвращаемые данные. Предложение orderby приведет к сортировке элементов в соответствии со стандартным компаратором для сортируемого типа.

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

```
Audi A4
Ford Focus
Mercedes C-Class
Opel Astra
Seat Leon
VW Golf
VW Passat
VW Polo
```

Работа с настраиваемым типом

В этом примере создается типичный список, заполняется, а затем запрашивается

```
public class Car
{
    public String Name { get; private set; }
    public int UnitsSold { get; private set; }

    public Car(string name, int unitsSold)
    {
        Name = name;
        UnitsSold = unitsSold;
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {

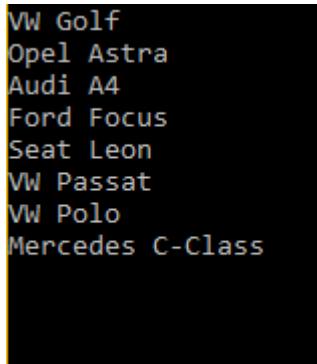
        var car1 = new Car("VW Golf", 270952);
        var car2 = new Car("Opel Astra", 56079);
        var car3 = new Car("Audi A4", 52493);
        var car4 = new Car("Ford Focus", 51677);
        var car5 = new Car("Seat Leon", 42125);
        var car6 = new Car("VW Passat", 97586);
        var car7 = new Car("VW Polo", 69867);
        var car8 = new Car("Mercedes C-Class", 67549);

        var cars = new List<Car> {
            car1, car2, car3, car4, car5, car6, car7, car8 };
        var list = from car in cars
                   select car.Name;

        foreach (var entry in list)
        {
            Console.WriteLine(entry);
        }
        Console.ReadLine();
    }
}

```

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:



```

VW Golf
Opel Astra
Audi A4
Ford Focus
Seat Leon
VW Passat
VW Polo
Mercedes C-Class

```

До сих пор примеры не кажутся удивительными, так как можно просто перебирать массив, чтобы сделать в основном то же самое. Однако в приведенных ниже примерах вы можете увидеть, как создавать более сложные запросы с LINQ to Objects и добиваться большего с гораздо меньшим количеством кода.

В приведенном ниже примере мы можем выбрать автомобили, которые были проданы более чем 60000 единиц, и отсортировать их по количеству проданных единиц:

```

var list = from car in cars
           where car.UnitsSold > 60000
           orderby car.UnitsSold descending
           select car;

StringBuilder sb = new StringBuilder();

```

```
foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());
```

Когда приведенный выше код компилируется и выполняется, он производит следующий

```
VW Golf - 270952
VW Passat - 97586
VW Polo - 69867
Mercedes C-Class - 67549
```

результат:

В приведенном ниже примере мы можем выбрать автомобили, которые продали нечетное количество единиц и упорядочить их в алфавитном порядке по имени:

```
var list = from car in cars
           where car.UnitsSold % 2 != 0
           orderby car.Name ascending
           select car;
```

Когда приведенный выше код компилируется и выполняется, он производит следующий

```
Audi A4 - 52493
Ford Focus - 51677
Mercedes C-Class - 67549
Opel Astra - 56079
Seat Leon - 42125
VW Polo - 69867
```

результат:

Прочтайте Linq to Objects онлайн: <https://riptutorial.com/ru/csharp/topic/9405/linq-to-objects>

глава 16: LINQ to XML

Examples

Чтение XML с использованием LINQ to XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
<Employee>
<EmpId>1</EmpId>
<Name>Sam</Name>
<Sex>Male</Sex>
<Phone Type="Home">423-555-0124</Phone>
<Phone Type="Work">424-555-0545</Phone>
<Address>
<Street>7A Cox Street</Street>
<City>Acampo</City>
<State>CA</State>
<Zip>95220</Zip>
<Country>USA</Country>
</Address>
</Employee>
<Employee>
<EmpId>2</EmpId>
<Name>Lucy</Name>
<Sex>Female</Sex>
<Phone Type="Home">143-555-0763</Phone>
<Phone Type="Work">434-555-0567</Phone>
<Address>
<Street>Jess Bay</Street>
<City>Alta</City>
<State>CA</State>
<Zip>95701</Zip>
<Country>USA</Country>
</Address>
</Employee>
</Employees>
```

Чтобы прочитать этот XML-файл, используя LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable< XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

Для доступа к одному элементу

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable< XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
```

```
{  
    Console.WriteLine(employee.Element("Name").Value);  
}
```

Доступ к нескольким элементам

```
XElement xelement = XElement.Load("Employees.xml");  
IEnumerable< XElement> employees = xelement.Root.Elements();  
Console.WriteLine("List of all Employee Names along with their ID:");  
foreach (var employee in employees)  
{  
    Console.WriteLine("{0} has Employee ID {1}",  
        employee.Element("Name").Value,  
        employee.Element("EmpId").Value);  
}
```

Чтобы получить доступ ко всем элементам, имеющим определенный атрибут

```
XElement xelement = XElement.Load("Employees.xml");  
var name = from nm in xelement.Root.Elements("Employee")  
    where (string)nm.Element("Sex") == "Female"  
    select nm;  
Console.WriteLine("Details of Female Employees:");  
foreach (XElement xEle in name)  
    Console.WriteLine(xEle);
```

Для доступа к определенному элементу, имеющему определенный атрибут

```
XElement xelement = XElement.Load("../\\..\\Employees.xml");  
var homePhone = from phoneno in xelement.Root.Elements("Employee")  
    where (string)phoneno.Element("Phone").Attribute("Type") == "Home"  
    select phoneno;  
Console.WriteLine("List HomePhone Nos.");  
foreach (XElement xEle in homePhone)  
{  
    Console.WriteLine(xEle.Element("Phone").Value);  
}
```

Прочтайте LINQ to XML онлайн: <https://riptutorial.com/ru/csharp/topic/2773/linq-to-xml>

глава 17: Microsoft.Exchange.WebServices

Examples

Получить заданные настройки пользователя вне офиса

Сначала давайте создадим объект `ExchangeManager` , где конструктор будет подключаться к сервисам для нас. Он также имеет метод `GetOutOfSettings` , который вернет объект `oofSettings` для указанного адреса электронной почты:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password =
WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }
        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

            Service = service;
        }
        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return
redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }
        public OofSettings GetOutOfSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

Теперь мы можем назвать это в другом месте следующим образом:

```
var em = new ExchangeManager();
var oofSettings = em.GetOutOfSettings("testemail@domain.com");
```

Обновление специальных настроек пользователя вне офиса

Используя нижеприведенный класс, мы можем подключиться к Exchange, а затем установить настройки `UpdateUserOof` с помощью `UpdateUserOof`:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }

    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com",
        RedirectionUrlValidationCallback);

        Service = service;
    }

    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }

    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
    externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

Обновите настройки пользователя следующим образом:

```
var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>"
```

```
var theUser = "theuser@domain.com";  
  
var em = new ExchangeManager();  
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage,  
externalMessage, theUser);
```

Обратите внимание, что вы можете отформатировать сообщения с помощью стандартных тегов `html`.

Прочтайте `Microsoft.Exchange.WebServices` онлайн:

<https://riptutorial.com/ru/csharp/topic/4863/microsoft-exchange-webservices>

глава 18: NullReferenceException

Examples

Исправлено исключение NullReferenceException

NullReferenceException при попытке получить доступ к нестатическому члену (свойство, метод, поле или событие) ссылочного объекта, но оно равно null.

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

Чтобы отладить такое исключение, это довольно просто: в строке, где выбрано исключение, вам просто нужно посмотреть перед каждым » . 'или' [', или в редких случаях " ('.

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

Откуда мое исключение? Или:

- myGarage = null
- myGarage.CarCollection = null
- currentIndex **равен** null
- myGarage.CarCollection[currentIndex.Value] **имеет значение** null
- theCarInTheStreet **имеет значение** null

В режиме отладки вам нужно только поместить курсор мыши на каждый из этих элементов, и вы найдете свою нулевую ссылку. Тогда вам нужно понять, почему это не имеет значения. Коррекция полностью зависит от цели вашего метода.

Вы забыли создать / инициализировать его?

```
myGarage.CarCollection = new Car[10];
```

Вы должны делать что-то другое, если объект имеет значение null?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

Или, может быть, кто-то дал вам нулевой аргумент и не должен был:

```
if (theCarInTheStreet == null)
{
    throw new ArgumentNullException("theCarInTheStreet");
}
```

В любом случае помните, что метод никогда не должен генерировать исключение NullReferenceException. Если да, значит, вы забыли что-то проверить.

Прочтайте NullReferenceException онлайн:

<https://riptutorial.com/ru/csharp/topic/2702/nullreferenceexception>

глава 19: O (n) Алгоритм кругового вращения массива

Вступление

На моем пути к изучению программирования были простые, но интересные проблемы для решения как упражнения. Одна из этих проблем заключалась в том, чтобы повернуть массив (или другую коллекцию) на определенное значение. Здесь я поделюсь с вами простой формулой для этого.

Examples

Пример общего метода, который вращает массив с заданным сдвигом

Я хотел бы указать, что мы поворачиваем влево, когда значение сдвига отрицательное, и мы поворачиваем вправо, когда значение положительное.

```
public static void Main()
{
    int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int shiftCount = 1;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    shiftCount = 15;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

    array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    shiftCount = -1;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

    array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    shiftCount = -35;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
}

private static void Rotate<T>(ref T[] array, int shiftCount)
{
    T[] backupArray= new T[array.Length];

    for (int index = 0; index < array.Length; index++)
    {
```

```

        backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
    }

    array = backupArray;
}

```

То, что важно в этом коде, - это формула, с которой мы получаем новое значение индекса после вращения.

(index + array.Length + shiftCount% array.Length)% array.Length

Вот немного больше информации об этом:

(shiftCount% array.Length) -> мы нормализуем значение сдвига, которое должно быть в длине массива (поскольку в массиве длиной 10, сдвиг 1 или 11 - это то же самое, то же самое происходит для -1 и -11) ,

array.Length + (shiftCount% array.Length) -> это делается из-за левых поворотов, чтобы убедиться, что мы не переходим в отрицательный индекс, а поворачиваем его до конца массива. Без него для массива с длиной 10 для индекса 0 и вращения -1 мы бы пошли в отрицательное число (-1) и не получили значение индекса вращения, равное 9. $(10 + (-1 \% 10)) \% 10 = 9$

index + array.Length + (shiftCount% array.Length) -> не так много сказать здесь, когда мы применяем поворот к индексу для получения нового индекса. $(0 + 10 + (-1 \% 10)) \% 10 = 9$

index + array.Length + (shiftCount% array.Length)% array.Length -> вторая нормализация гарантирует, что новое значение индекса не выходит за пределы массива, но вращает значение в начале массива. Это для правильных вращений, так как в массиве с длиной 10 без него для индекса 9 и вращением 1 мы бы вошли в индекс 10, который находится вне массива, а не для того, чтобы значение реального вращения было равно 0. $((9 + 10 + (1 \% 10)) \% 10 = 0)$

Прочтите О (n) Алгоритм кругового вращения массива онлайн:

<https://riptutorial.com/ru/csharp/topic/9770/o--n--алгоритм-кругового-вращения-массива>

глава 20: ObservableCollection

Examples

Инициализировать ObservableCollection

ObservableCollection - это коллекция типа `T` like `List<T>` что означает, что она содержит объекты типа `T`

Из документации мы читаем, что:

ObservableCollection представляет собой динамический сбор данных, который предоставляет уведомления, когда элементы добавляются, удаляются или когда весь список обновляется.

Основное отличие от других коллекций является то, что ObservableCollection реализует интерфейсы `INotifyCollectionChanged` и `INotifyPropertyChanged` и немедленно поднять событие уведомления при добавлении или удалении новый объект, и когда коллекция очищается.

Это особенно полезно для связывания пользовательского интерфейса и бэкэнд приложения без необходимости писать дополнительный код, потому что, когда объект добавляется или удаляется из наблюдаемой коллекции, пользовательский интерфейс автоматически обновляется.

Первым шагом для его использования является включение

```
using System.Collections.ObjectModel
```

Вы можете либо создать пустой экземпляр коллекции, например `string` типа

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

или экземпляр, заполненный данными

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
    "First_String", "Second_String"
};
```

Помните, что во всей коллекции `IList` индекс начинается с 0 ([свойство `IList.Item`](#)).

Прочтайте ObservableCollection онлайн:

<https://riptutorial.com/ru/csharp/topic/7351/observablecollection--t->

глава 21: String.Format

Вступление

Методы `Format` - это набор [перегрузок](#) в классе `System.String` используется для создания строк, которые объединяют объекты в конкретные представления строк. Эта информация может быть применена к `String.Format`, различным методам `WriteLine` а также другим методам в .NET framework.

Синтаксис

- `string.Format` (строковый формат, `params object [] args`)
- `string.Format` (поставщик `IFormatProvider`, строковый формат, `params object [] args`)
- `$"string {text} blablabla" // Поскольку C # 6`

параметры

параметр	подробности
формат	Строка составного формата , определяющая способ объединения аргументов в строку.
арг	Последовательность объектов, которые должны быть объединены в строку. Поскольку это использует аргумент <code>params</code> , вы можете использовать список аргументов, разделенных запятыми, или фактический массив объектов.
поставщик	Набор способов форматирования объектов для строк. Типичные значения включают <code>CultureInfo.InvariantCulture</code> и <code>CultureInfo.CurrentCulture</code> .

замечания

Заметки:

- `String.Format()` обрабатывает `null` аргументы, не `String.Format()` исключения.
- Существуют перегрузки, которые заменяют параметр `args` **одним**, **двумя** или **тремя** параметрами объекта.

Examples

Места, где String.Format «встроен» в структуру

Есть несколько мест, где вы можете использовать `String.Format` **косвенно**: секрет заключается в `String.Format` **перегрузки** с `string format, params object[] args` **ПОДПИСИ** `string format, params object[] args`, **например**:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Может быть заменен более короткой версией:

```
Console.WriteLine("{0} - {1}", name, value);
```

Существуют и другие методы, которые также используют `String.Format` **например**:

```
Debug.WriteLine(); // and Print()  
StringBuilder.AppendFormat();
```

Использование формата пользовательского номера

`NumberFormatInfo` может использоваться для форматирования чисел `integer` и `float`.

```
// invariantResult is "1,234,567.89"  
var invariantResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);  
  
// NumberFormatInfo is one of classes that implement IFormatProvider  
var customProvider = new NumberFormatInfo  
{  
    NumberDecimalSeparator = "_NS_", // will be used instead of ','  
    NumberGroupSeparator = "_GS_", // will be used instead of '.'  
};  
  
// customResult is "1_GS_234_GS_567_NS_89"  
var customResult = string.Format(customProvider, "{0:#,###.##}", 1234567.89);
```

Создание поставщика настраиваемого формата

```
public class CustomFormat : IFormatProvider, ICustomFormatter  
{  
    public string Format(string format, object arg, IFormatProvider formatProvider)  
    {  
        if (!this.Equals(formatProvider))  
        {  
            return null;  
        }  
  
        if (format == "Reverse")  
        {  
            return String.Join("", arg.ToString().Reverse());  
        }  
  
        return arg.ToString();  
    }  
}
```

```

    }

    public object GetFormat(Type formatType)
    {
        return formatType==typeof(ICustomFormatter) ? this:null;
    }
}

```

Использование:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Выход:

```
-> dlroW olleH <-
```

Выровнять влево / вправо, прокладка с пробелами

Второе значение в фигурных скобках определяет длину строки замены. При корректировке второго значения, которое должно быть положительным или отрицательным, выравнивание строки может быть изменено.

```
string.Format("LEFT: string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Выход:

```
LEFT: string: ->abc <- int: ->123 <-
RIGHT: string: -> abc<- int: -> 123<-
```

Числовые форматы

```

// Integral types as hex
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');

// Integers with thousand separators
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);

// Integer with leading zeroes
string.Format("Integer, leading zeroes: {0:00}; ", 1);

// Decimals
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);

```

Выход:

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<
Integer, leading zeroes: 01;
Decimal, fixed precision: 0.120; as percents: 12.00%
```

Форматирование валюты

Спецификатор формата «с» (или валюты) преобразует число в строку, представляющую сумму валюты.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

ТОЧНОСТЬ

Значение по умолчанию - 2. Используйте с1, с2, с3 и т. д. Для контроля точности.

```
string.Format("{0:C1}", 112.236677) // $112.2
string.Format("{0:C3}", 112.236677) // $112.237
string.Format("{0:C4}", 112.236677) // $112.2367
string.Format("{0:C9}", 112.236677) // $112.236677000
```

Символ валюты

1. CultureInfo экземпляр CultureInfo для использования символа пользовательской культуры.

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); // $112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); // 112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); // ₹ 112.24
```

2. Используйте любую строку в качестве символа валюты. Используйте NumberFormatInfo для настройки символа валюты.

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
string.Format(nfi, "{0:C}", 112.236677); // ?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); // ?%^&112.24
```

Позиция валютного символа

Используйте [CurrencyPositivePattern](#) для положительных значений и [CurrencyNegativePattern](#) для отрицательных значений.

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;
nfi.CurrencyPositivePattern = 0;
string.Format(nfi, "{0:C}", 112.236677); // $112.24 - default
nfi.CurrencyPositivePattern = 1;
string.Format(nfi, "{0:C}", 112.236677); // 112.24$
nfi.CurrencyPositivePattern = 2;
string.Format(nfi, "{0:C}", 112.236677); // $ 112.24
```

```
nfi.CurrencyPositivePattern = 3;
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

Отрицательное использование шаблона совпадает с положительным. Гораздо больше случаев использования см. В исходной ссылке.

Пользовательский десятичный разделитель

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //112..24
```

Поскольку С # 6.0

6,0

Начиная с С # 6.0, вместо `String.Format` можно использовать интерполяцию `String.Format`.

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Привет, Джон Доу!

Дополнительные примеры для этого в разделе С # 6.0: [String-интерполяция](#).

Выделение фигурных скобок внутри выражения `String.Format()`

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);

//Outputs "{I am in brackets!} I am outside of bracket"
```

Форматирование даты

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds

Console.WriteLine(String.Format("{0:dd}", date));

//Format by Culture info
String.Format(new System.Globalization.CultureInfo("mn-MN"), "{0:dddd}", date);
```

6,0

```
Console.WriteLine($"{date:ddd}");
```

ВЫХОД :

06
Лхагва
06

Тендерный	Имея в виду	Образец	Результат
d	Дата	{0:d}	7/6/2016
дд	День, с нулевой	{0:dd}	06
ddd	Короткое название дня	{0:ddd}	Мы бы
dddd	Полное имя дня	{0:dddd}	среда
D	Долгосрочная дата	{0:D}	Среда, 6 июля 2016 г.
e	Полная дата и время, короткие	{0:f}	Среда, 6 июля 2016 года 18:30
Ф.Ф.	Вторая фракция, 2 цифры	{0:ff}	20
FFF	Вторая фракция, 3 цифры	{0:fff}	201
FFFF	Вторая фракция, 4 цифры	{0:ffff}	2016
F	Полная дата и время, длинные	{0:F}	Среда, 6 июля 2016 года 6:30:14
г	Дата и время по умолчанию	{0:g}	6/6/2016 18:30
гг	эпоха	{0:gg}	ОБЪЯВЛЕНИЕ
чч	Час (2 цифры, 12H)	{0:hh}	06
HH	Час (2 цифры, 24 часа)	{0:HH}	18
М	Месяц и день	{0:M}	6 июля
мм	Минуты с нулевым запасом	{0:mm}	30
М.М.	Месяц, с нулевым запасом	{0:MM}	07
МММ	3-месячное название месяца	{0:MMM}	июль
ММММ	Полное название месяца	{0:MMMM}	июль
сс	секунд	{0:ss}	14
р	Дата RFC1123	{0:r}	Ср, 06 июля 2016 18:30:14

Тендерный	Имя в виду	Образец	Результат
s	Строка сортируемой даты	{0:s}	2016-07-06T18: 30: 14
T	Короткое время	{0:t}	6:30 ВЕЧЕРА
T	Много времени	{0:T}	6:30:14 PM
TT	ДО ПОЛУДНЯ ПОСЛЕ ПОЛУДНЯ	{0:tt}	ВЕЧЕРА
U	Универсальное сортируемое местное время	{0:u}	2016-07-06 18: 30: 14Z
U	Универсальный GMT	{0:U}	Среда, 6 июля 2016 года 9:30:14
Y	Месяц и год	{0:Y}	Июль 2016 г.
yy	2-значный год	{0:yy}	16
ГГГГ	4-значный год	{0:yyyy}	2016
ZZ	Двухзначное смещение часового пояса	{0:zz}	+09
ZZZ	полное смещение часового пояса	{0:zzz}	+09: 00

Нанизывать()

Метод `ToString()` присутствует во всех типах ссылочных объектов. Это происходит из-за того, что все ссылочные типы выводятся из `Object`, на котором есть метод `ToString()`.

Метод `ToString()` в базовом классе объекта возвращает имя типа. Оставшийся фрагмент выведет «Пользователь» на консоль.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
}

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Тем не менее, пользователь класса может также переопределить `ToString()`, чтобы

изменить возвращаемую строку. Ниже фрагмент кода выводит на консоль «Id: 5, Name: User1».

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
    public override ToString()
    {
        return string.Format("Id: {0}, Name: {1}", Id, Name);
    }
}

...
var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Связь с `ToString()`

Хотя метод `String.Format()` безусловно, полезен при форматировании данных в виде строк, он часто может быть немного перегружен, особенно при работе с одним объектом, как показано ниже:

```
String.Format("{0:C}", money); // yields "$42.00"
```

Более простым подходом может быть просто использование метода `ToString()` доступного для всех объектов в C#. Он поддерживает все одинаковые [стандартные и настраиваемые строки форматирования](#), но не требует необходимого сопоставления параметров, поскольку будет только один аргумент:

```
money.ToString("C"); // yields "$42.00"
```

Ограничения и форматирование ограничений

Хотя в некоторых сценариях этот подход может быть проще, подход `ToString()` ограничен в отношении добавления левого или правого заполнения, как вы могли бы сделать в `String.Format()`:

```
String.Format("{0,10:C}", money); // yields "      $42.00"
```

Чтобы достичь такого же поведения с помощью метода `ToString()`, вам нужно будет использовать другой метод, например `PadLeft()` или `PadRight()`:

```
money.ToString("C").PadLeft(10); // yields "      $42.00"
```

Прочтайте `String.Format` онлайн: <https://riptutorial.com/ru/csharp/topic/79/string-format>

глава 22: StringBuilder

Examples

Что такое `StringBuilder` и когда использовать его

`StringBuilder` представляет серию символов, которые, в отличие от обычной строки, изменяемы. Часто бывает необходимо изменить строки, которые мы уже сделали, но стандартный строковый объект не изменен. Это означает, что каждый раз, когда строка изменяется, необходимо создать новый, строковый объект, скопировать в него и затем переназначить.

```
string myString = "Apples";
myString += " are my favorite fruit";
```

В приведенном выше примере `myString` изначально имеет значение "Apples". Однако, когда мы объединяем «мой любимый плод», то, что должен делать внутренний класс строки, включает в себя:

- Создание нового массива символов, равного длине `myString` и новой строки, которую мы добавляем.
- Копирование всех символов `myString` в начало нашего нового массива и копирование новой строки в конец массива.
- Создайте новый строковый объект в памяти и переназначьте его в `myString`.

Для одного конкатенации это относительно тривиально. Однако, что делать, если нужно выполнить много операций добавления, например, в цикле?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

Благодаря многократному копированию и созданию объекта это значительно ухудшит производительность нашей программы. Мы можем избежать этого, используя `StringBuilder`.

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

Теперь, когда выполняется один и тот же цикл, производительность и скорость выполнения программы будут значительно быстрее, чем при использовании обычной строки. Чтобы вернуть `StringBuilder` в обычную строку, мы можем просто вызвать метод `ToString()` для `StringBuilder`.

Однако это не единственная оптимизация, которую имеет `StringBuilder`. Для дальнейшей оптимизации функций мы можем использовать другие свойства, которые помогают повысить производительность.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

Если мы заранее знаем, как долго должен быть наш `StringBuilder`, мы можем указать его размер раньше времени, что не позволит ему изменить размер массива символов, который он внутренне.

```
sb.Append('k', 2000);
```

Хотя использование `StringBuilder` для добавления намного быстрее, чем строка, оно может работать даже быстрее, если вам нужно только добавить один символ много раз.

После того, как вы завершили построение своей строки, вы можете использовать метод `ToString()` в `StringBuilder` чтобы преобразовать его в базовую `string`. Это часто необходимо, потому что класс `StringBuilder` не наследует `string`.

Например, вот как вы можете использовать `StringBuilder` для создания `string`:

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

В заключение, `StringBuilder` следует использовать вместо строки, когда необходимо внести множество изменений в строку с учетом производительности.

Используйте `StringBuilder` для создания строки из большого количества записей

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of
    records, say, 10000+

    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
        customerNamesCsv
            .Append(record.LastName)
```

```
        .Append(',')
        .Append(record.FirstName)
        .Append(Environment.NewLine);
    }

    return customerNamesCsv.ToString();
}
```

Прочтайте **StringBuilder** онлайн: <https://riptutorial.com/ru/csharp/topic/4675/stringbuilder>

глава 23:

System.DirectoryServices.Protocols.LdapConnection

Examples

Аутентифицированное SSL-соединение LDAP, SSL-сертификат не соответствует обратному DNS

Настройте некоторые константы для сервера и информацию аутентификации.

Предполагая LDAPv3, но достаточно легко изменить это.

```
// Authentication, and the name of the server.
private const string LDAPUser =
"cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };
private const string TargetServer = "ldap.example.com";

// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

На самом деле создайте соединение с тремя частями: `LdapDirectoryIdentifier` (сервер) и `NetworkCredentials`.

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.
// If you don't have SSL, don't give it the SSL port.
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);

// Configure network credentials (userid and password)
var secureString = new SecureString();
foreach (var character in password)
    secureString.AppendChar(character);
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);

// Actually create the connection
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType = AuthType.Basic,
    SessionOptions =
    {
        ProtocolVersion = 3,
        SecureSocketLayer = true
    }
};

// Override SChannel reverse DNS lookup.
// This gets us past the "The LDAP server is unavailable." exception
// Could be
//     connection.SessionOptions.VerifyServerCertificate += { return true; };
// but some certificate validation is probably good.
connection.SessionOptions.VerifyServerCertificate +=
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0},",
```

```
TargetServer));
```

Используйте сервер LDAP, например, ищите кого-то по идентификатору пользователя для всех значений objectClass. Объект Classass присутствует, чтобы продемонстрировать сложный поиск: Амперсанд является логическим оператором «и» для двух предложений запроса.

```
SearchRequest searchRequest = new SearchRequest(
    CompanyDN,
    string.Format((&(objectClass=*)(uid={0})), uid),
    SearchScope.Subtree,
    null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}
```

Супер простой анонимный LDAP

Предполагая LDAPv3, но достаточно легко изменить это. Это анонимное, незашифрованное создание LDAPv3 LdapConnection.

```
private const string TargetServer = "ldap.example.com";
```

На самом деле создайте соединение с тремя частями: LdapDirectoryIdentifier (сервер) и NetworkCredentials.

```
// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};
```

Чтобы использовать соединение, что-то вроде этого получило бы людей с фамилией Смит

```
SearchRequest searchRequest = new SearchRequest("dn=example, dn=com", "(sn=Smith)",
    SearchScope.Subtree, null);
```

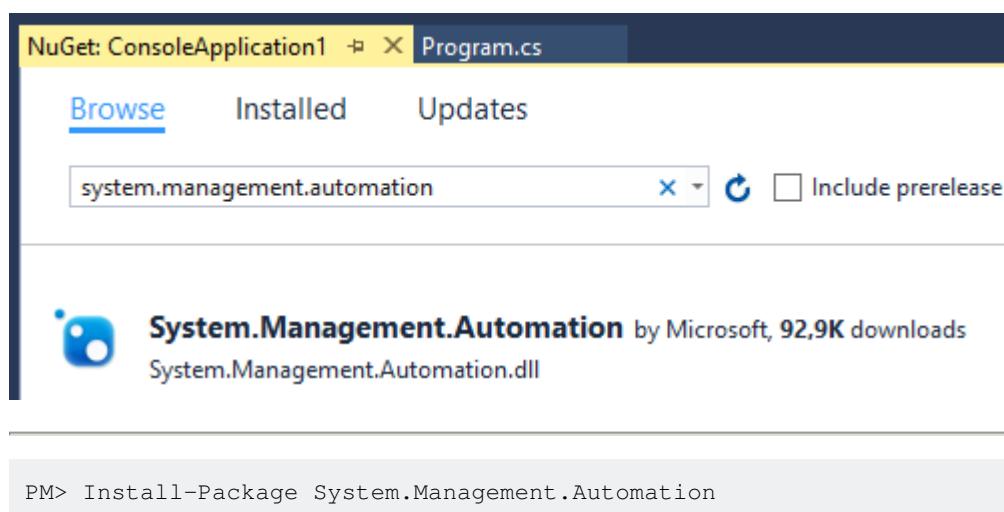
Прочтайте [System.DirectoryServices.Protocols.LdapConnection](#) онлайн:
<https://riptutorial.com/ru/csharp/topic/5177/system-directoryservices-protocols-ldapconnection>

глава 24: System.Management.Automation

замечания

Пространство имен *System.Management.Automation* является корневым пространством имен для Windows PowerShell.

System.Management.Automation является библиотекой расширений от Microsoft и может быть добавлена в проекты Visual Studio через диспетчер пакетов NuGet или консоль диспетчера пакетов.



Examples

Вызывать простой синхронный конвейер

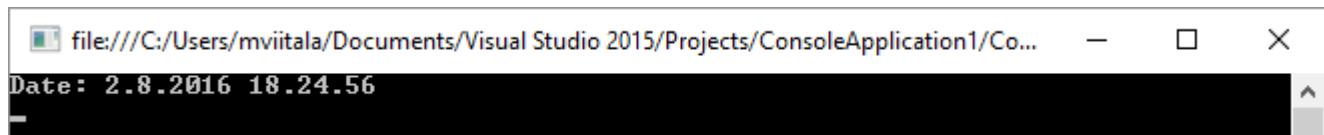
Получить текущую дату и время.

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
        PowerShell ps = PowerShell.Create();

        // add command
        ps.AddCommand("Get-Date");

        // run command(s)
        Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
}
```



Прочтайте `System.Management.Automation` онлайн:

<https://riptutorial.com/ru/csharp/topic/4988/system-management-automation>

глава 25: Verbatim Strings

Синтаксис

- @ "verbatim strings - это строки, содержимое которых не экранировано, поэтому в этом случае \n не представляет символ новой строки, а два отдельных символа: \ и n. Строки Verbatim создаются с префиксом содержимого строки символом @
- @ «Чтобы избежать кавычек», используются «двойные кавычки».

замечания

Чтобы объединить строковые литералы, используйте символ @ в начале каждой строки.

```
var combinedString = @"\\t means a tab" + @" and \\n means a newline";
```

Examples

Многострочные струны

```
var multiLine = @"This is a  
multiline paragraph";
```

Выход:

Это

многострочный абзац

[Живая демонстрация на .NET скрипке](#)

Многострочные строки, содержащие двойные кавычки, также могут быть экранированы так же, как и в одной строке, потому что они являются стендографическими строками.

```
var multilineWithDoubleQuotes = @"I went to a city named  
    ""San Diego""  
    during summer vacation.;"
```

[Живая демонстрация на .NET скрипке](#)

Следует отметить, что пробелы / табуляции в начале строк 2 и 3 здесь действительно

присутствуют в значении переменной; проверьте [этот вопрос на](#) предмет возможных решений.

Выход из двойных котировок

Двойные кавычки внутри строк verbatim можно экранировать, используя две последовательные двойные кавычки "" для представления одной двойной кавычки " в полученной строке.

```
var str = @"""I don't think so,"" he said.";  
Console.WriteLine(str);
```

Выход:

«Я так не думаю», - сказал он.

Живая демонстрация на .NET скрипке

Интерполированные стенограммы Verbatim

Строки Verbatim могут быть объединены с новыми функциями [интерполяции String](#), найденными на C # 6.

```
Console.WriteLine(${@"Testing \n 1 2 {5 - 2}  
New line"});
```

Выход:

Тестирование \ n 1 2 3

Новая линия

Живая демонстрация на .NET скрипке

Как и ожидалось из стенографической строки, обратные косые черты игнорируются как escape-символы. И, как и ожидалось из интерполированной строки, любое выражение внутри фигурных фигурных скобок вычисляется перед вставкой в строку в этой позиции.

Строки Verbatim инструктируют компилятор не использовать экранирование символов

В обычной строке символ обратной косой черты - это символ escape, который инструктирует компилятор для поиска следующего символа (ов) для определения фактического символа в строке. ([Полный список экранов символов](#))

В стенографических строках не существует символьных экранов (кроме "" который превращен в "). Чтобы использовать строку verbatim, просто добавьте @ перед стартовыми

кавычками.

Эта стенографическая строка

```
var filename = @"c:\temp\newfile.txt"
```

Выход:

C:\Temp\newfile.txt

В отличие от использования обычной (невербальной) строки:

```
var filename = "c:\temp\newfile.txt"
```

который будет выводить:

```
c:    emp  
ewfile.txt
```

используя экранирование символов. (\t заменяется символом табуляции, а \n заменяется на новую строку.)

[Живая демонстрация на .NET скрипке](#)

Прочтайте Verbatim Strings онлайн: <https://riptutorial.com/ru/csharp/topic/16/verbatim-strings>

глава 26: Windows Communication Foundation

Вступление

Windows Communication Foundation (WCF) - это основа для создания сервис-ориентированных приложений. Используя WCF, вы можете отправлять данные как асинхронные сообщения с одной конечной точки службы на другую. Конечная точка службы может быть частью постоянно доступной службы, размещенной в IIS, или может быть службой, размещенной в приложении. Сообщения могут быть такими же простыми, как один символ или слово, отправленное как XML, или сложное, как поток двоичных данных.

Examples

Начало работы с образцом

Служба описывает операции, которые он выполняет в контракте на обслуживание, который он публично раскрывает как метаданные.

```
// Define a service contract.  
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]  
public interface ICalculator  
{  
    [OperationContract]  
    double Add(double n1, double n2);  
}
```

Реализация службы вычисляет и возвращает соответствующий результат, как показано в следующем примере кода.

```
// Service class that implements the service contract.  
public class CalculatorService : ICalculator  
{  
    public double Add(double n1, double n2)  
    {  
        return n1 + n2;  
    }  
}
```

Служба предоставляет конечную точку для связи с сервисом, определенную с использованием файла конфигурации (Web.config), как показано в следующей примерной конфигурации.

```
<services>
```

```

<service
  name="StackOverflow.ServiceModel.Samples.CalculatorService"
  behaviorConfiguration="CalculatorServiceBehavior">
  <!-- ICalculator is exposed at the base address provided by
      host: http://localhost/servicemodelsamples/service.svc. -->
  <endpoint address=""
    binding="wsHttpBinding"
    contract="StackOverflow.ServiceModel.Samples.ICalculator" />
  ...
</service>
</services>

```

Структура не предоставляет метаданные по умолчанию. Таким образом, служба включает ServiceMetadataBehavior и предоставляет конечную точку обмена метаданными (MEX) по адресу <http://localhost/servicemodelsamples/service.svc/mex>. Следующая конфигурация демонстрирует это.

```

<system.serviceModel>
  <services>
    <service
      name="StackOverflow.ServiceModel.Samples.CalculatorService"
      behaviorConfiguration="CalculatorServiceBehavior">
      ...
      <!-- the mex endpoint is exposed at
          http://localhost/servicemodelsamples/service.svc/mex -->
      <endpoint address="mex"
        binding="mexHttpBinding"
        contract="IMetadataExchange" />
    </service>
  </services>

  <!--For debugging purposes set the includeExceptionDetailInFaults
      attribute to true-->
  <behaviors>
    <serviceBehaviors>
      <behavior name="CalculatorServiceBehavior">
        <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>

```

Клиент связывается с использованием определенного типа контракта, используя класс клиента, который создается средством служебной программы метаданных ServiceModel (Svchost.exe).

Выполните следующую команду из командной строки SDK в каталоге клиента, чтобы создать типизированный прокси:

```

svchost.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs

```

Как и сервис, клиент использует файл конфигурации (App.config), чтобы указать конечную

точку, с которой он хочет общаться. Конфигурация конечной точки клиента состоит из абсолютного адреса для конечной точки службы, привязки и контракта, как показано в следующем примере.

```
<client>
  <endpoint
    address="http://localhost/servicemodelsamples/service.svc"
    binding="wsHttpBinding"
    contract="StackOverflow.ServiceModel.Samples.IColorator" />
</client>
```

Клиентская реализация создает экземпляр клиента и использует типизированный интерфейс для начала связи с сервисом, как показано в следующем примере кода.

```
// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();
```

Прочтите Windows Communication Foundation онлайн:
<https://riptutorial.com/ru/csharp/topic/10760/windows-communication-foundation>

глава 27: XDocument и пространство имен System.Xml.Linq

Examples

Создание XML-документа

Цель состоит в том, чтобы сгенерировать следующий XML-документ:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Код:

```
XNamespace xmlns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
 XElement xRoot = new XElement(xns + "FruitBasket");
 xDoc.Add(xRoot);

 XElement xelFruit1 = new XElement(xns + "Fruit");
 XAttribute idAttribute1 = new XAttribute("ID", "F0001");
 xelFruit1.Add(idAttribute1);
 XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
 XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
 xelFruit1.Add(xelFruitName1);
 xelFruit1.Add(xelFruitColor1);
 xRoot.Add(xelFruit1);

 XElement xelFruit2 = new XElement(xns + "Fruit");
 XAttribute idAttribute2 = new XAttribute("ID", "F0002");
 xelFruit2.Add(idAttribute2);
 XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
 XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
 xelFruit2.Add(xelFruitName2);
 xelFruit2.Add(xelFruitColor2);
 xRoot.Add(xelFruit2);
```

Изменение файла XML

Чтобы изменить файл XML с помощью `XDocument`, вы загружаете файл в переменную типа `XDocument`, модифицируете его в памяти, а затем сохраняете, перезаписывая исходный

файл. Общей ошибкой является изменение XML в памяти и ожидание изменения файла на диске.

Учитывая XML-файл:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Вы хотите изменить цвет банана на коричневый:

1. Нам нужно знать путь к файлу на диске.
2. Одна перегрузка `XDocument.Load` получает URI (путь к файлу).
3. Поскольку XML-файл использует пространство имен, мы должны запросить имя пространства имен AND.
4. Запрос Linq, использующий синтаксис C # 6 для размещения нулевых значений. Каждый . используемый в этом запросе, может возвращать нулевой набор, если условие не находит элементов. До C # 6 вы сделали бы это несколькими шагами, проверяя нуль на этом пути. Результатом является элемент `<Fruit>` который содержит Банану. На самом деле `IEnumerable< XElement >`, поэтому на следующем шаге используется `FirstOrDefault()`.
5. Теперь мы извлекаем элемент `FruitColor` из элемента `Fruit`, который мы только что нашли. Здесь мы предполагаем, что есть только один, или мы заботимся только о первом.
6. Если это не `null`, мы устанавливаем `FruitColor` в «Brown».
7. Наконец, мы сохраняем `XDocument`, перезаписывая исходный файл на диске.

```
// 1.
string xmlFilePath = "c:\\users\\public\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?.
  Elements(ns + "FruitName")?.
  Where(x => x.Value == "Banana")?.
  Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();
```

```
// 6.  
if (elColor != null)  
{  
    elColor.Value = "Brown";  
}  
  
// 7.  
xdoc.Save(xmlFilePath);
```

Теперь файл выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>  
<FruitBasket xmlns="http://www.fruitauthority.fake">  
    <Fruit>  
        <FruitName>Banana</FruitName>  
        <FruitColor>Brown</FruitColor>  
    </Fruit>  
    <Fruit>  
        <FruitName>Apple</FruitName>  
        <FruitColor>Red</FruitColor>  
    </Fruit>  
</FruitBasket>
```

Создание XML-документа с использованием свободного синтаксиса

Цель:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">  
    <Fruit>  
        <FruitName>Banana</FruitName>  
        <FruitColor>Yellow</FruitColor>  
    </Fruit>  
    <Fruit>  
        <FruitName>Apple</FruitName>  
        <FruitColor>Red</FruitColor>  
    </Fruit>  
</FruitBasket>
```

Код:

```
XNamespace xns = "http://www.fruitauthority.fake";  
XDocument xDoc =  
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),  
        new XElement(xns + "FruitBasket",  
            new XElement(xns + "Fruit",  
                new XElement(xns + "FruitName", "Banana"),  
                new XElement(xns + "FruitColor", "Yellow")),  
            new XElement(xns + "Fruit",  
                new XElement(xns + "FruitName", "Apple"),  
                new XElement(xns + "FruitColor", "Red"))  
        ));
```

Прочтайте XDocument и пространство имен System.Xml.Linq онлайн:

<https://riptutorial.com/ru/csharp/topic/1866/xdocument-и-пространство-имен-system-xml-linq>

глава 28: XmlDocument и пространство имен System.Xml

Examples

Основное взаимодействие с XML-документами

```
public static void Main()
{
    var xml = new XmlDocument();
    var root = xml.CreateElement("element");
    // Creates an attribute, so the element will now be "<element attribute='value' />"
    root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
        day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}
```

Чтение из документа XML

Пример файла XML

```
<Sample>
<Account>
    <One number="12"/>
    <Two number="14"/>
</Account>
<Account>
    <One number="14"/>
    <Two number="16"/>
```

```
</Account>  
</Sample>
```

Чтение из этого XML-файла:

```
using System.Xml;  
using System.Collections.Generic;  
  
public static void Main(string fullpath)  
{  
    var xmldoc = new XmlDocument();  
    xmldoc.Load(fullpath);  
  
    var oneValues = new List<string>();  
  
    // Getting all XML nodes with the tag name  
    var accountNodes = xmldoc.GetElementsByTagName("Account");  
    for (var i = 0; i < accountNodes.Count; i++)  
    {  
        // Use Xpath to find a node  
        var account = accountNodes[i].SelectSingleNode("./One");  
        if (account != null && account.Attributes != null)  
        {  
            // Read node attribute  
            oneValues.Add(account.Attributes["number"].Value);  
        }  
    }  
}
```

XmIDocument vs XDocument (пример и сравнение)

Существует несколько способов взаимодействия с Xml-файлом.

1. Документ Xml
2. XDocument
3. XmlReader / XmlWriter

До LINQ to XML мы использовали XmlDocument для манипуляций в XML, таких как добавление атрибутов, элементов и т. д. Теперь LINQ to XML использует XDocument для такого же типа вещей. Синтаксисы намного проще, чем XmlDocument, и для этого требуется минимальное количество кода.

Кроме того, XDocument работает быстрее, чем XmlDocument. XmlDocument - это старое и грязное решение для запроса XML-документа.

Я собираюсь показать некоторые примеры класса XmlDocument и класса класса XDocument :

Загрузить файл XML

```
string filename = @"C:\temp\test.xml";
```

XmlDocument

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

Создать XmlDocument

XmlDocument

```
 XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

XDocument

```
XDocument doc = new XDocument(
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>TextNode</child>
</root>
```

Изменить InnerText узла в XML

XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

XDocument

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNote.Value = "New Value";
```

Сохранить файл после редактирования

Обязательно сохраните xml после любых изменений.

```
// Save XmlDocument and XDocument
_doc.Save(filename);
```

Retreive Values из XML

XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

XDocument

```
XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
string text = node.Value;
```

Извлеките значение из всех дочерних элементов, где атрибут = что-то.

XmlDocument

```
List<string> valueList = new List<string>();
foreach (XmlNode n in nodelist)
{
    if(n.Attributes["type"].InnerText == "City")
    {
        valueList.Add(n.Attributes["type"].InnerText);
    }
}
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c =>
c.Attribute("type").Value == "setting").Select(c => c.Value);
```

Добавить узел

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/*After a change make sure to save the document*/
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new
 XElement("SecondLevelNode"));

/*After a change make sure to save the document*/
_doc.Save(fileName);
```

Прочтайте XmlDocument и пространство имен System.Xml онлайн:

<https://riptutorial.com/ru/csharp/topic/1528/xmldocument-и-пространство-имен-system-xml>

глава 29: Анализ регулярных выражений

Синтаксис

- `new Regex(pattern);` // Создает новый экземпляр с определенным шаблоном.
- `Regex.Match(input);` // Запускает поиск и возвращает совпадение.
- `Regex.Matches(input);` // Запускает поиск и возвращает MatchCollection

параметры

название	подробности
Шаблон	<code>string</code> шаблон, который должен использоваться для поиска. Для получения дополнительной информации: msdn
RegexOptions [Необязательно]	Общие варианты здесь: <code>Singleline</code> И <code>Multiline</code> . Они меняют поведение элементов шаблона, таких как точка <code>(.)</code> , <code>SingleLine</code> -Mode не будет покрывать <code>NewLine</code> <code>(\n)</code> в <code>Multiline</code> -Mode но в <code>SingleLine</code> -Mode . По умолчанию: msdn
Таймаут [Дополнительно]	Там, где модели становятся более сложными, поиск может потреблять больше времени. Это пропущенный тайм-аут для поиска, как известно из сетевого программирования.

замечания

Необходимо использовать

```
using System.Text.RegularExpressions;
```

Хорошо бы иметь

- Вы можете проверить свои шаблоны онлайн без необходимости компиляции своего решения, чтобы получить результаты здесь: [Нажмите меня](#)
- Пример Regex101: [нажмите мне](#)

Особенно новички имеют тенденцию перегружать свои задачи с помощью регулярного выражения, потому что он чувствует себя мощным и в нужном месте для сложных текстовых поисков. Это тот момент, когда люди пытаются разбирать xml-документы с регулярным выражением, даже не спрашивая себя, может ли быть уже готовый класс для этой задачи, такой как `XmlDocument` .

Regex должен быть последним оружием, чтобы выбрать сложность повтора. По крайней мере, не забудьте приложить некоторые усилия, чтобы найти *right way* прежде чем записывать 20 строк шаблонов.

Examples

Единый матч

```
using System.Text.RegularExpressions;

string pattern = ":(.*?):";
string lookup = "--:text in here:--";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
// Get the match from your regex-object
Match mLookup = rgxLookup.Match(lookup);

// The group-index 0 always covers the full pattern.
// Matches inside parentheses will be accessed through the index 1 and above.
string found = mLookup.Groups[1].Value;
```

Результат:

```
found = "text in here"
```

Несколько совпадений

```
using System.Text.RegularExpressions;

List<string> found = new List<string>();
string pattern = ":(.*?):";
string lookup = "--:text in here:--:another one:-:third one:---!123:fourth:--;

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
MatchCollection mLookup = rgxLookup.Matches(lookup);

foreach (Match match in mLookup)
{
    found.Add(match.Groups[1].Value);
}
```

Результат:

```
found = new List<string>() { "text in here", "another one", "third one", "fourth" }
```

Прочтайте Анализ регулярных выражений онлайн:

<https://riptutorial.com/ru/csharp/topic/3774/анализ-регулярных-выражений>

глава 30: Аннотации данных

Examples

DisplayNameAttribute (атрибут отображения)

DisplayName устанавливает отображаемое имя для свойства, события или общедоступного метода void с нулевыми (0) аргументами.

```
public class Employee
{
    [DisplayName(@"Employee first name")]
    public string FirstName { get; set; }
}
```

Простой пример использования в приложении XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="100" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:DisplayNameConverter x:Key="DisplayNameConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- Label (DisplayName attribute) -->
        <Label Content="{Binding Employee, Converter={StaticResource DisplayNameConverter},
ConverterParameter=FirstName}" />
        <!-- TextBox (FirstName property value) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>
</Window>
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee();

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }
}
```

```

        public Employee Employee
    {
        get { return _employee; }
        set { _employee = value; }
    }
}

```

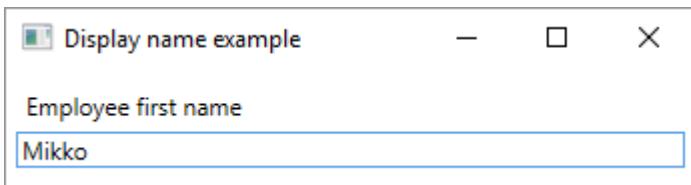
```

namespace WpfApplication
{
    public class DisplayNameConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // Get display name for given instance type and property name
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<DisplayNameAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.DisplayName : string.Empty;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}

```



EditableAttribute (атрибут моделирования данных)

EditableAttribute устанавливает, должны ли пользователи изменять значение свойства класса.

```

public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}

```

Простой пример использования в приложении XAML

```
<Window x:Class="WpfApplication.MainWindow"
```

```

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wpfApplication="clr-namespace:WpfApplication"
Height="70" Width="360" Title="Display name example">

<Window.Resources>
    <wpfApplication:EditableConverter x:Key="EditableConverter"/>
</Window.Resources>

<StackPanel Margin="5">
    <!-- TextBox Text (FirstName property value) -->
    <!-- TextBox IsEnabled (Editable attribute) -->
    <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
        IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
</StackPanel>

</Window>

```

```

namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "This is not editable" };

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}

```

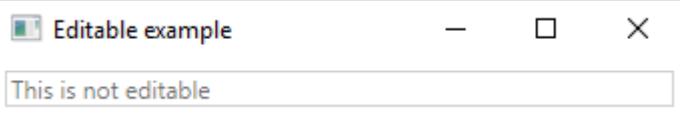
```

namespace WpfApplication
{
    public class EditableConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // return editable attribute's value for given instance property,
            // defaults to true if not found
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<EditableAttribute>()
                .FirstOrDefault();

```

```
        return attribute != null ? attribute.AllowEdit : true;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        throw new NotImplementedException();
    }
}
```



Атрибуты проверки

Атрибуты проверки используются для принудительного применения различных правил проверки в декларативном виде для классов или членов класса. Все атрибуты проверки основаны на базовом классе [ValidationAttribute](#).

Пример: RequiredAttribute

При проверке с помощью метода `ValidationAttribute.Validate` этот атрибут возвращает ошибку, если свойство `Name` равно `null` или содержит только пробелы.

```
public class ContactModel
{
    [Required(ErrorMessage = "Please provide a name.")]
    public string Name { get; set; }
}
```

Пример: StringLengthAttribute

`StringLengthAttribute` проверяет, `StringLengthAttribute` ли строка, чем максимальная длина строки. Он может дополнительно указать минимальную длину. Оба значения включены.

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "A name must be between five and
twenty characters.")]
    public string Name { get; set; }
}
```

Пример: RangeAttribute

RangeAttribute **дает максимальное и минимальное значение для числового поля.**

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "Price must be between 0.01 and 100.00")]
    public decimal Price { get; set; }
}
```

Пример: CustomValidationAttribute

Класс `CustomValidationAttribute` **позволяет использовать специальный static метод для проверки. Пользовательский метод должен быть** `static ValidationResult [MethodName] (object input)`.

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

Объявление метода:

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("Apples are not allowed.");
        }

        return result;
    }
}
```

Создание настраиваемого атрибута проверки

Пользовательские атрибуты проверки могут быть созданы путем получения базового класса `ValidationAttribute` и **последующего переопределения** `virtual` методов по мере необходимости.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;
```

```

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }

        return isValid;
    }
}

```

Затем этот атрибут можно использовать следующим образом:

```

public class Model
{
    [NotABanana(ErrorMessage = "Bananas are not allowed.")]
    public string FavoriteFruit { get; set; }
}

```

Основы аннотаций данных

Аннотации данных - это способ добавления дополнительной контекстной информации к классам или членам класса. Существуют три основные категории аннотаций:

- Атрибуты проверки: добавить критерии проверки данных
- Атрибуты отображения: укажите, как данные должны отображаться пользователю
- Атрибуты моделирования: добавьте информацию об использовании и отношениях с другими классами

ИСПОЛЬЗОВАНИЕ

Вот пример, в котором используются два параметра `ValidationAttribute` и `один DisplayAttribute`:

```

class Kid
{
    [Range(0, 18)] // The age cannot be over 18 and cannot be negative
    public int Age { get; set; }

    [StringLength(MaximumLength = 50, MinimumLength = 3)] // The name cannot be under 3 chars
    or more than 50 chars
    public string Name { get; set; }

    [DataType(DataType.Date)] // The birthday will be displayed as a date only (without the
    time)
    public DateTime Birthday { get; set; }
}

```

Аннотации данных в основном используются в таких рамках, как `ASP.NET`. Например, в `ASP.NET MVC`, когда модель получена методом контроллера, `ModelState.IsValid()` может использоваться, чтобы определить, соответствует ли полученная модель всему ее `ValidationAttribute`. `DisplayAttribute` также используется в `ASP.NET MVC` для определения способа отображения значений на веб-странице.

Вручную выполнять атрибуты проверки

В большинстве случаев атрибуты проверки используются внутри фреймворков (например, ASP.NET). Эти рамки заботятся о выполнении атрибутов проверки. Но что, если вы хотите вручную выполнить атрибуты проверки? Просто используйте класс `Validator` (не требуется никакого отражения).

Контекст проверки

Любая проверка требует контекста, чтобы дать некоторую информацию о том, что проверяется. Это может включать в себя различную информацию, такую как проверяемый объект, некоторые свойства, имя для отображения в сообщении об ошибке и т. д.

```
ValidationContext vc = new ValidationContext(objectToValidate); // The simplest form of validation context. It contains only a reference to the object being validated.
```

После создания контекста существует несколько способов проверки.

Проверка объекта и всех его свойств

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the results of the validation
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // Validates the object and its properties using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

Проверка свойства объекта

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the results of the validation
bool isValid = Validator.TryValidateProperty(objectToValidate.PropertyToValidate, vc, results, true); // Validates the property using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

И больше

Подробнее о ручной проверке см. в:

- [Документация класса ValidationContext](#)
- [Документация класса валидатора](#)

Прочтите Аннотации данных онлайн: <https://riptutorial.com/ru/csharp/topic/4942/аннотации-данных>

глава 31: Анонимные типы

Examples

Создание анонимного типа

Поскольку анонимные типы не называются, переменные этих типов должны быть неявно напечатаны (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

Если имена членов не указаны, они устанавливаются на имя свойства / переменной, используемого для инициализации объекта.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Обратите внимание, что имена могут быть опущены только тогда, когда выражение в объявлении анонимного типа является простым доступом к свойствам; для вызовов методов или более сложных выражений должно быть указано имя свойства.

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

Анонимный и динамический

Анонимные типы позволяют создавать объекты без явного определения их типов заранее, сохраняя при этом статическую проверку типов.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

И наоборот, `dynamic` динамическая проверка типов, выбор ошибок времени выполнения, а не ошибок времени компиляции.

```
dynamic val = "foo";
```

```
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Общие методы с анонимными типами

Общие методы позволяют использовать анонимные типы посредством вывода типа.

```
void Log<T>(T obj) {  
    // ...  
}  
Log(new { Value = 10 });
```

Это означает, что выражения LINQ могут использоваться с анонимными типами:

```
var products = new[] {  
    new { Amount = 10, Id = 0 },  
    new { Amount = 20, Id = 1 },  
    new { Amount = 15, Id = 2 }  
};  
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);  
// idsByAmount: 0, 2, 1
```

Создание типичных типов с анонимными типами

Использование универсальных конструкторов потребует назвать анонимные типы, что невозможно. Альтернативно, общие методы могут использоваться для разрешения вывода типа.

```
var anon = new { Foo = 1, Bar = 2 };  
var anon2 = new { Foo = 5, Bar = 10 };  
List<T> CreateList<T>(params T[] items) {  
    return new List<T>(items);  
}  
  
var list1 = CreateList(anon, anon2);
```

В случае `List<T>` неявно типизированные массивы могут быть преобразованы в `List<T>` через метод `ToList` LINQ:

```
var list2 = new[] {anon, anon2}.ToList();
```

Неопределенное равенство типа

Равенство анонимного типа задается методом экземпляра `Equals`. Два объекта равны, если они имеют одинаковый тип и равные значения (через `a.Prop.Equals(b.Prop)`) для каждого свойства.

```
var anon = new { Foo = 1, Bar = 2 };  
var anon2 = new { Foo = 1, Bar = 2 };  
var anon3 = new { Foo = 5, Bar = 10 };
```

```
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

Два анонимных типа считаются одинаковыми, если и только если их свойства имеют одно и то же имя и тип и отображаются в одном порядке.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have different types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

Неявно типизированные массивы

Массивы анонимных типов могут быть созданы с неявной типизацией.

```
var arr = new[] {
    new { Id = 0 },
    new { Id = 1 }
};
```

Прочтайте Анонимные типы онлайн: <https://riptutorial.com/ru/csharp/topic/765/анонимные-типы>

глава 32: Асинхронные / ожидающие, фоновые работы, задачи и примеры потоков

замечания

Чтобы запустить любой из этих примеров, просто назовите их так:

```
static void Main()
{
    new Program().ProcessDataAsync();
    Console.ReadLine();
}
```

Examples

ASP.NET Configure Await

Когда ASP.NET обрабатывает запрос, поток создается из пула потоков и создается **контекст запроса**. Контекст запроса содержит информацию о текущем запросе, к которому можно получить доступ через статическое свойство `HttpContext.Current`. Затем контекст запроса для запроса присваивается потоку, обрабатывающему запрос.

Конкретный контекст запроса **может быть активен только по одному потоку за раз**.

Когда выполнение достигает `await`, поток, обрабатывающий запрос, возвращается в пул потоков, пока выполняется асинхронный метод, и контекст запроса свободен для использования другого потока.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    var products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    // Execution continues using the original request context.
    return View(products);
}
```

Когда задача завершается, пул потоков назначает другой поток для продолжения выполнения запроса. Затем контекст запроса присваивается этому потоку. Это может быть или не быть исходной нитью.

блокировка

Когда результат вызова метода `async` **синхронно**, могут возникнуть взаимоблокировки. Например, следующий код приведет к взаимоблокировке при `IndexSync()`:

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

Это связано с тем, что по умолчанию ожидаемая задача, в этом случае `dbContext.Products.ToListAsync()` захватит контекст (в случае ASP.NET контекст запроса) и попытается использовать его после его завершения.

Когда весь стек вызовов асинхронен, проблем нет, потому что, как только `await` достигнуто, исходный поток освобождается, освобождая контекст запроса.

Когда мы блокируем синхронное использование `Task.Result` или `Task.Wait()` (или других методов блокировки), исходный поток все еще активен и сохраняет контекст запроса. Ожидаемый метод все еще работает асинхронно, и как только обратный вызов пытается выполнить, т. Е. Как только ожидаемая задача вернулась, он пытается получить контекст запроса.

Поэтому тупик возникает из-за того, что, пока блокирующий поток с контекстом запроса ожидает завершения асинхронной операции, асинхронная операция пытается получить контекст запроса для завершения.

ConfigureAwait

По умолчанию вызовы ожидаемой задачи будут захватывать текущий контекст и пытаться возобновить выполнение в контексте после завершения.

Используя `ConfigureAwait(false)` это может быть предотвращено, а взаимоблокировки можно избежать.

```
public async Task<ActionResult> Index()
```

```

{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // Execution resumes on a "random" thread from the pool without the original request
    context
        return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}

```

Это позволяет избежать блокировок, когда необходимо блокировать асинхронный код, однако это происходит за счет потери контекста в продолжении (код после ожидания вызова).

В ASP.NET это означает, что если ваш код после вызова `await someTask.ConfigureAwait(false);` пытается получить доступ к информации из контекста, например `HttpContext.Current.User` тогда информация была потеряна. В этом случае `HttpContext.Current` имеет значение `NULL`. Например:

```

public async Task<ActionResult> Index()
{
    // Contains information about the user sending the request
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // Null Reference Exception, Current is null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
}

```

Если `ConfigureAwait(true)` используется (эквивалентно отсутствию `ConfigureAwait`), то как `user` и `user2` заполняются теми же данными.

По этой причине часто рекомендуется использовать `ConfigureAwait(false)` в библиотечном коде, где контекст больше не используется.

Асинхронный / Await

Ниже приведен простой пример того, как использовать `async / await` для выполнения

некоторых интенсивных операций времени в фоновом режиме, сохраняя при этом возможность делать некоторые другие вещи, которые не нужно ждать от времени, необходимого для завершения.

Однако, если вам нужно позже работать с результатом интенсивного времени, вы можете сделать это, ожидая выполнения.

```
public async Task ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> task = TimeintensiveMethod(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    // Wait for TimeintensiveMethod to complete and get its result
    int x = await task;
    Console.WriteLine("Count: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = await reader.ReadToEndAsync();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

BackgroundWorker

Ниже приведен простой пример использования объекта `BackgroundWorker` для выполнения интенсивных операций в фоновом потоке.

Вам нужно:

1. Определите рабочий метод, который выполняет трудоемкую работу и вызывается из обработчика события для `DoWork` события `BackgroundWorker`.
2. Запустите выполнение с помощью `RunWorkerAsync`. Любой аргумент требуется по методе работника, прикрепленной к `DoWork` может быть передан в через `DoWorkEventArgs` параметр `RunWorkerAsync`.

В дополнение к событию `DoWork` класс `BackgroundWorker` также определяет два события, которые должны использоваться для взаимодействия с пользовательским интерфейсом.

Это необязательно.

- Событие `RunWorkerCompleted` запускается, когда обработчики `DoWork` завершены.
- Событие `ProgressChanged` запускается при `ReportProgress` метода `ReportProgress`.

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

// Method that will be called after BwDoWork exits
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // we can access possible return values of our Method via the Parameter e
    Console.WriteLine("Count: " + e.Result);
}

// execution of our time intensive Method
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

задача

Ниже приведен простой пример того, как использовать `Task` для выполнения некоторого времени в фоновом процессе.

Все, что вам нужно сделать, это обернуть ваш метод интенсивного времени в `Task.Run()`.

```
public void ProcessDataAsync()
```

```

{
    // Start the time intensive method
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    Console.WriteLine("Count: " + t.Result);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}

```

Нить

Ниже приведен простой пример того, как использовать `Thread` для выполнения некоторого времени в фоновом процессе.

```

public async void ProcessDataAsync()
{
    // Start the time intensive method
    Thread t = new Thread(TimeintensiveMethod);

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}

```

Как вы можете видеть, мы не можем вернуть значение из нашего `TimeIntensiveMethod` ПОТОМУ

ЧТО Thread ожидает, что в качестве параметра будет использоваться метод void.

Чтобы получить возвращаемое значение из Thread используйте либо событие, либо следующее:

```
int ret;
Thread t = new Thread(() =>
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something to demonstrate the coolness of await-async
    ret = new Random().Next(100);
});
t.Start();
t.Join(1000);
Console.WriteLine("Count: " + ret);
```

Задание «запустить и забыть»

В некоторых случаях (например, ведение журнала) может оказаться полезным запустить задачу и не ждать результата. Следующее расширение позволяет запустить задачу и продолжить выполнение кода останова:

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

Результат ожидается только внутри метода расширения. Поскольку используется `async / await`, можно поймать исключение и вызвать необязательный метод его обработки.

Пример использования расширения:

```
var task = Task.FromResult(0); // Or any other task from e.g. external lib.
task.RunAndForget(
    e =>
{
    // Something went wrong, handle it.
});
```

Прочтайте Асинхронные / ожидающие, фоновые работы, задачи и примеры потоков онлайн: <https://riptutorial.com/ru/csharp/topic/3824/асинхронные---ожидающие--фоновые-работы--задачи-и-примеры-потоков>

глава 33: Асинхронный разъем

Вступление

Используя асинхронные сокеты, сервер может прослушивать входящие подключения и выполнять некоторую другую логику в среднем времени в отличие от синхронного сокета, когда они прослушивают, они блокируют основной поток, и приложение становится невосприимчивым, и он замерзнет, пока клиент не подключится.

замечания

Розетка и сеть

Как получить доступ к серверу за пределами моей собственной сети? Это общий вопрос, и когда его спрашивают, в основном помечены как темы.

Серверная сторона

В сети вашего сервера вам необходимо перенаправить маршрутизатор на ваш сервер.

Для примера ПК, на котором работает сервер:

локальный IP = 192.168.1.115

Сервер прослушивает порт 1234.

Перенаправить входящие соединения на маршрутизаторе `Port 1234 на 192.168.1.115`

Страна клиента

Единственное, что вам нужно изменить, это IP. Вы не хотите подключаться к вашему loopback-адресу, но к общедоступному IP-адресу из сети, на которой работает ваш сервер. Этот IP-адрес вы можете получить [здесь](#).

```
_connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("10.10.10.10"), 1234));
```

Итак, теперь вы создаете запрос на эту конечную точку: `10.10.10.10:1234` если вы сделали порт свойств, перенаправляете ваш маршрутизатор, ваш сервер и клиент будут подключаться без каких-либо проблем.

Если вы хотите подключиться к локальному IP-адресу, вам не придется переносить portforward, просто изменив адрес loopback на `192.168.1.178` или что-то в этом роде.

Отправка данных:

Данные отправляются в байтовый массив. Вам необходимо упаковать данные в массив байтов и распаковать их с другой стороны.

Если вы знакомы с сокетом, вы также можете попытаться зашифровать массив байтов перед отправкой. Это предотвратит кражу вашего пакета.

Examples

Пример асинхронного сокета (клиент / сервер).

Пример сервера

Создать прослушиватель для сервера

Начните с создания сервера, который будет обрабатывать клиентов, которые подключаются, и запросы, которые будут отправляться. Поэтому создайте класс Listener, который будет обрабатывать это.

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming
    connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
    }
}
```

Сначала нам нужно инициализировать сокет Listener, где мы можем прослушивать любые подключения. Мы собираемся использовать Tcp Socket, поэтому мы используем SocketType.Stream. Также мы указываем, что порт which должен прослушивать

Затем мы начинаем слушать любые входящие соединения.

Используемые здесь древовидные методы:

1. [ListenerSocket.Bind \(\)](#);

Этот метод связывает сокет с [IPEndPoint](#). Этот класс содержит информацию о хосте и локальном или удаленном порту, необходимую приложению для подключения к службе на хосте.

2. [ListenerSocket.Listen \(10\)](#);

Параметр backlog указывает количество входящих соединений, которые могут быть поставлены в очередь для принятия.

3. [ListenerSocket.BeginAccept \(\)](#)

Сервер начнет прослушивать входящие соединения и продолжит работу с другой логикой. Когда есть соединение, сервер переключается на этот метод и запускает метод `AcceptCallBack`

```
public void StartListening()
{
    try
    {
        MessageBox.Show($"Listening started port:{Port} protocol type:
{ProtocolType.Tcp}");
        ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
        ListenerSocket.Listen(10);
        ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
    }
    catch (Exception ex)
    {
        throw new Exception("listening error" + ex);
    }
}
```

Поэтому, когда клиент подключается, мы можем принять их с помощью этого метода:

Здесь используются три метода:

1. [ListenerSocket.EndAccept \(\)](#)

Мы начали обратный вызов с помощью `Listener.BeginAccept()` теперь мы должны завершить этот вызов. `The EndAccept()` принимает параметр `IAsyncResult`, это сохранит состояние асинхронного метода. Из этого состояния мы можем извлечь сокет, из которого поступало входящее соединение.

2. [ClientController.AddClient \(\)](#)

С помощью сокета, который мы получили от `EndAccept()` мы создаем Клиент с собственным методом (код `ClientController` ниже примера сервера) .

3. [ListenerSocket.BeginAccept \(\)](#)

Нам нужно снова начать прослушивание, когда сокет будет обработан новым соединением. Передайте метод, который поймает этот обратный вызов. А также передайте `int socket Listener`, чтобы мы могли повторно использовать этот сокет для предстоящих подключений.

```
public void AcceptCallback(IAsyncResult ar)
{
    try
    {
        Console.WriteLine($"Accept CallBack port:{Port} protocol type:
{ProtocolType.Tcp}");
        Socket acceptedSocket = ListenerSocket.EndAccept(ar);
```

```

        ClientController.AddClient(acceptedSocket);

        ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
    }
    catch (Exception ex)
    {
        throw new Exception("Base Accept error" + ex);
    }
}

```

Теперь у нас есть Listening Socket, но как мы получаем передачу данных клиентом, что показывает следующий код.

Создавать серверный приемник для каждого клиента

Сначала создайте класс получения с конструктором, который принимает параметр Socket as:

```

public class ReceivePacket
{
    private byte[] _buffer;
    private Socket _receiveSocket;

    public ReceivePacket(Socket receiveSocket)
    {
        _receiveSocket = receiveSocket;
    }
}

```

В следующем методе мы сначала начинаем с предоставления буфера размером 4 байта (Int32) или пакета для частей {длина, фактические данные}. Таким образом, первые 4 байта резервируют для длины данных остальное для фактических данных.

Затем мы используем [метод BeginReceive \(\)](#). Этот метод используется для начала приема от подключенных клиентов, и когда он получит данные, он будет запускать функцию ReceiveCallback .

```

public void StartReceiving()
{
    try
    {
        _buffer = new byte[4];
        _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
        ReceiveCallback, null);
    }
    catch {}
}

private void ReceiveCallback(IAsyncResult AR)
{
    try
    {
        // if bytes are less than 1 takes place when a client disconnect from the server.
        // So we run the Disconnect function on the current client
        if (_receiveSocket.EndReceive(AR) > 1)
    }
}

```

```

    {
        // Convert the first 4 bytes (int 32) that we received and convert it to an
        Int32 (this is the size for the coming data).
        _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
        // Next receive this data into the buffer with size that we did receive before
        _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
        // When we received everything its onto you to convert it into the data that
        you've send.
        // For example string, int etc... in this example I only use the
        implementation for sending and receiving a string.

        // Convert the bytes to string and output it in a message box
        string data = Encoding.Default.GetString(_buffer);
        MessageBox.Show(data);
        // Now we have to start all over again with waiting for a data to come from
        the socket.
        StartReceiving();
    }
    else
    {
        Disconnect();
    }
}
catch
{
    // if exception is throw check if socket is connected because than you can
    startreive again else Dissconeect
    if (!_receiveSocket.Connected)
    {
        Disconnect();
    }
    else
    {
        StartReceiving();
    }
}
}

private void Disconnect()
{
    // Close connection
    _receiveSocket.Disconnect(true);
    // Next line only apply for the server side receive
    ClientController.RemoveClient(_clientId);
    // Next line only apply on the Client Side receive
    Here you want to run the method TryToConnect()
}
}

```

Поэтому мы настроили сервер, который может принимать и слушать входящие соединения. Когда клиенты подключаются, он будет добавлен в список клиентов, и каждый клиент имеет свой собственный класс приема. Чтобы прослушать сервер:

```

Listener listener = new Listener();
listener.StartListening();

```

Некоторые классы, которые я использую в этом примере

```

class Client

```

```

{
    public Socket _socket { get; set; }
    public ReceivePacket Receive { get; set; }
    public int Id { get; set; }

    public Client(Socket socket, int id)
    {
        Receive = new ReceivePacket(socket, id);
        Receive.StartReceiving();
        _socket = socket;
        Id = id;
    }
}

static class ClientController
{
    public static List<Client> Clients = new List<Client>();

    public static void AddClient(Socket socket)
    {
        Clients.Add(new Client(socket, Clients.Count));
    }

    public static void RemoveClient(int id)
    {
        Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));
    }
}

```

Пример стороны клиента

Подключение к серверу

Прежде всего, мы хотим создать класс, который соединяется с именем сервера, которое мы даем: Connector:

```

class Connector
{
    private Socket _connectingSocket;
}

```

Следующий метод для этого класса - TryToConnect ()

Этот метод немного интересует:

1. Создайте сокет;
2. Затем я контактирую до сокета
3. В каждом цикле он просто удерживает Thread в течение 1 секунды, мы не хотим DOS сервера XD
4. С [Connect \(\)](#) он попытается подключиться к серверу. Если он терпит неудачу, он выдает исключение, но while будет поддерживать соединение с сервером. Вы можете

использовать метод [Connect CallBack](#) для этого, но я просто перейду к вызову метода при подключении Socket.

5. Обратите внимание, что Клиент теперь пытается подключиться к вашему локальному компьютеру на порту 1234.

```
public void TryToConnect()
{
    _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

    while (!_connectingSocket.Connected)
    {
        Thread.Sleep(1000);

        try
        {
            _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
1234));
        }
        catch { }
    }
    SetupForReceiveing();
}
}

private void SetupForReceiveing()
{
    // View Client Class bottom of Client Example
    Client.SetClient(_connectingSocket);
    Client.StartReceiving();
}
```

Отправка сообщения на сервер

Итак, теперь у нас есть почти законченное приложение или приложение Socket. Единственное, что у нас нет, это класс для отправки сообщения на сервер.

```
public class SendPacket
{
    private Socket _sendSocket;

    public SendPacket(Socket sendSocket)
    {
        _sendSocket = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what happens here:
            1. Create a list of bytes
            2. Add the length of the string to the list.
               So if this message arrives at the server we can easily read the length of
            the coming message.
            3. Add the message(string) bytes
        }
    }
}
```

```

        */

        var fullPacket = new List<byte>();
        fullPacket.AddRange(BitConverter.GetBytes(data.Length));
        fullPacket.AddRange(Encoding.Default.GetBytes(data));

        /* Send the message to the server we are currently connected to.
        Or package structure is {length of data 4 bytes (int32), actual data}*/
        _sendSocked.Send(fullPacket.ToArray());
    }
    catch (Exception ex)
    {
        throw new Exception();
    }
}

```

Завершить ящик двумя кнопками один для подключения, а другой для отправки сообщения:

```

private void ConnectClick(object sender, EventArgs e)
{
    Connector tpp = new Connector();
    tpp.TryToConnect();
}

private void SendClick(object sender, EventArgs e)
{
    Client.SendString("Test data from client");
}

```

Класс клиента, который я использовал в этом примере

```

public static void SetClient(Socket socket)
{
    Id = 1;
    Socket = socket;
    Receive = new ReceivePacket(socket, Id);
    SendPacket = new SendPacket(socket);
}

```

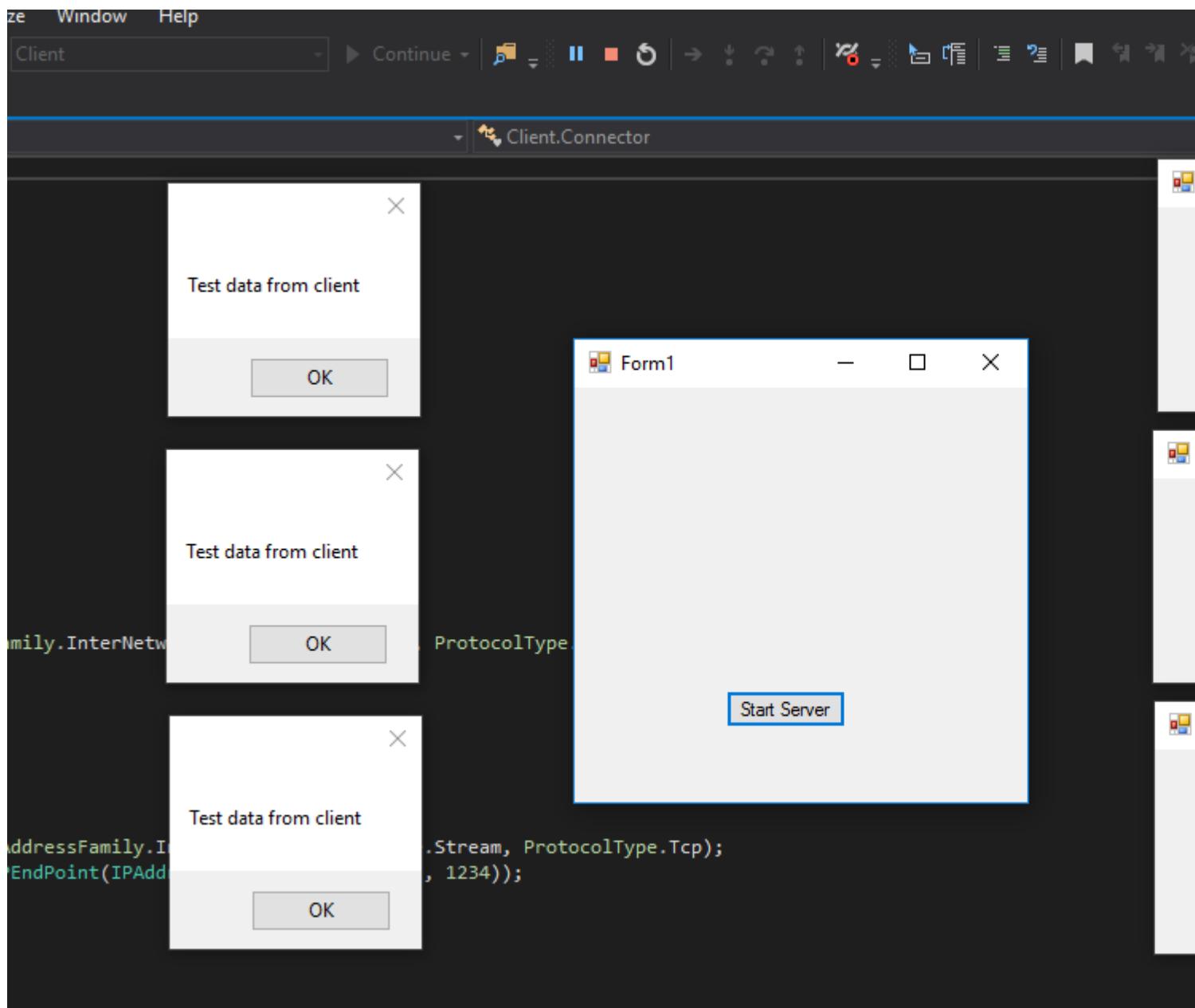
уведомление

Класс приема с сервера совпадает с классом приема от клиента.

Заключение

Теперь у вас есть сервер и клиент. Вы можете использовать этот основной пример. Например, убедитесь, что сервер также может получать файлы или другие мелодии. Или отправьте сообщение клиенту. На сервере у вас есть список клиентов, поэтому, когда вы получаете что-то, о чем узнаете, с клиентом он пришел.

Конечный результат:



Прочтите Асинхронный разъем онлайн: <https://riptutorial.com/ru/csharp/topic/9638/асинхронный-разъем>

глава 34: Асинхронный-Await

Вступление

В C # метод, объявленный `async`, не будет блокироваться в синхронном процессе, если вы используете операции на основе ввода-вывода (например, веб-доступ, работа с файлами, ...). Результат таких асинхронных методов может ожидаться с помощью ключевого слова `await`.

замечания

Метод `async` может возвращать `void`, `Task` ИЛИ `Task<T>`.

Тип возвращаемого `Task` будет ждать завершения метода, и результат будет `void`. `Task<T>` вернет значение из типа `T` после завершения метода.

методы `async` должны возвращать `Task` ИЛИ `Task<T>`, в отличие от `void`, практически во всех случаях. `async void` методы не могут быть `await`, что приводит к множеству проблем.

Единственный сценарий, в котором `async` должен возвращать `void` относится к обработчику событий.

`async / await` работает, преобразуя ваш `async` метод в `async` автомат. Он делает это, создавая структуру за кулисами, которая хранит текущее состояние и любой контекст (например, локальные переменные) и предоставляет метод `MoveNext()` для продвижения состояний (и запуска любого связанного кода) всякий раз, когда ожидается ожидаемое завершение.

Examples

Простые последовательные звонки

```
public async Task<JobResult> GetDataFromWebAsync()
{
    var nextJob = await _database.GetNextJobAsync();
    var response = await _httpClient.GetAsync(nextJob.Uri);
    var pageContents = await response.Content.ReadAsStringAsync();
    return await _database.SaveJobResultAsync(pageContents);
}
```

Главное, чтобы отметить здесь, что в то время как каждый `await`-ED метод вызывается асинхронно - и за все время, что называют управления привели обратно в систему - поток внутри метода является линейным и не требует какого-либо специального лечения из-за асинхронность. Если какой-либо из методов, называемых `fail`, исключение будет обработано «как ожидалось», что в этом случае означает, что выполнение метода будет

прервано, и исключение будет расти вверх по стеку.

Попробуйте / Поймать / Наконец

6,0

Начиная с C # 6.0 ключевое слово `await` теперь можно использовать в блоке `catch` и `finally`.

```
try {
    var client = new AsyncClient();
    await client.DoSomething();
} catch (MyException ex) {
    await client.LogExceptionAsync();
    throw;
} finally {
    await client.CloseAsync();
}
```

5.0 6.0

До C # 6.0 вам нужно сделать что-то в следующих строках: Обратите внимание, что 6.0 также очистил нулевые проверки с помощью [оператора Null Propagating](#).

```
AsynClient client;
MyException caughtException;
try {
    client = new AsyncClient();
    await client.DoSomething();
} catch (MyException ex) {
    caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
        await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

Обратите внимание, что если вы ожидаете задачи, не созданные `async` (например, задача, созданная `Task.Run`), некоторые отладчики могут ломаться над исключениями, `Task.Run` этой задачей, даже если она, по-видимому, обрабатывается окружением `try / catch`. Это происходит потому, что отладчик считает, что он не обрабатывается в отношении кода пользователя. В Visual Studio есть опция «[Только мой код](#)», которую можно отключить, чтобы предотвратить отладчик в таких ситуациях.

Настройка Web.config для установки 4.5 для правильного асинхронного поведения.

Web.config `system.web.httpRuntime` должен быть нацелен на 4.5, чтобы гарантировать, что

поток будет обрабатывать контекст запроса, прежде чем возобновлять ваш метод `async`.

```
<httpRuntime targetFramework="4.5" />
```

Async и ждут неопределенного поведения на ASP.NET до 4.5. Async / await возобновится на произвольном потоке, который может не иметь контекста запроса. Приложения под нагрузкой случайным образом терпят неудачу с нулевыми ссылочными исключениями, доступ к `HttpContext` после ожидания. [Использование `HttpContext.Current` в WebApi опасно из-за `async`](#)

Параллельные вызовы

Одновременно можно ожидать нескольких вызовов, сначала вызвав ожидаемые задачи, а затем ожидая их.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

Кроме того, `Task.WhenAll` можно использовать для группировки нескольких задач в одну `Task`, которая завершается, когда все переданные задачи завершены.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

Вы также можете сделать это внутри цикла, например:

```
List<Task> tasks = new List<Task>();
while (something) {
    // do stuff
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

Чтобы получить результаты из задачи после ожидания нескольких задач с помощью `Task.WhenAll`, просто подождите снова. Поскольку задача уже завершена, она вернет результат обратно

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();

await Task.WhenAll(task1, task2);

var result = await task2;
```

Кроме того, `Task.WhenAny` может использоваться для одновременного выполнения нескольких задач, таких как `Task.WhenAll` выше, с той разницей, что этот метод будет завершен, когда будет завершена любая из поставленных задач.

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

Task возвращаемая `RunConcurrentTasksWhenAny`, завершится, когда завершатся все из `firstTask`, `secondTask` ИЛИ `thirdTask`.

Оператор ожидания и ключевое слово `async`

`await` оператора и ключевое слово `async` объединяются:

Асинхронный метод, в котором используется `await`, должен быть изменен с помощью ключевого слова `async`.

Противоположность не всегда верна: вы можете пометить метод как `async` не используя `await` в своем теле.

`await` самом деле заключается в том, чтобы приостановить выполнение кода до тех пор, пока ожидаемая задача не завершится; любая задача может быть ожидаемой.

Примечание: вы не можете ждать метода `async`, который ничего не возвращает (`void`).

На самом деле слово «`suspends`» немного вводит в заблуждение, потому что не только выполнение останавливается, но поток может стать бесплатным для выполнения других операций. Под капотом `await` реализуется с помощью магии компилятора: он разбивает метод на две части - до и после `await`. Последняя часть выполняется, когда ожидаемая задача завершается.

Если мы проигнорируем некоторые важные детали, компилятор примерно сделает это за вас:

```
public async Task<TResult> DoIt()
{
    // do something and acquire someTask of type Task<TSomeResult>
```

```
var awaitedResult = await someTask;
// ... do something more and produce result of type TResult
return result;
}
```

будет выглядеть так:

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
{
    // ...
}
```

Любой обычный метод можно превратить в `async` следующим образом:

```
await Task.Run(() => YourSyncMethod());
```

Это может быть полезно, когда вам нужно выполнить длинный запуск в потоке пользовательского интерфейса без замораживания пользовательского интерфейса.

Но здесь есть очень важное замечание: **асинхронный не всегда означает одновременный (параллельный или даже многопоточный)**. Даже в одном потоке `async await` до сих пор позволяет использовать асинхронный код. Например, см. Этот настраиваемый [планировщик задач](#). Такой «сумасшедший» планировщик задач может просто превращать задачи в функции, которые вызывают в обработке цикла сообщений.

Нам нужно спросить себя: какой поток выполнит продолжение нашего метода `DoIt_Continuation`?

По умолчанию оператор `await` выполняет выполнение продолжения с текущим [контекстом синхронизации](#). Это означает, что по умолчанию для продолжений WinForms и WPF в потоке пользовательского интерфейса. Если по какой-то причине вам нужно изменить это поведение, используйте [метод](#) `Task.ConfigureAwait()`:

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

Возвращение задачи без ожидания

Методы, которые выполняют асинхронные операции не нужно использовать `await`, если:

- Внутри метода есть только один асинхронный вызов

- Асинхронный вызов находится в конце метода
- Исключение исключения / обработки, которое может произойти в Задаче, не обязательно

Рассмотрим этот метод, который возвращает `Task` :

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return await dataStore.GetByKeyAsync(lookupKey);
}
```

Если `GetByKeyAsync` имеет ту же подпись, что и `GetUserAsync` (возвращает `Task<User>`), метод может быть упрощен:

```
public Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return dataStore.GetByKeyAsync(lookupKey);
}
```

В этом случае метод не должен быть помечен как `async`, даже если он выполняет предварительную асинхронную операцию. Задача, возвращаемая `GetByKeyAsync`, передается непосредственно вызывающему методу, где он будет `await`.

Важно : возвращая `Task` вместо ожидания, изменяет поведение исключения для метода, поскольку оно не будет генерироваться внутри метода, который запускает задачу, но в ожидающем ее методе.

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch (Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

// If exception happens, it will be thrown here, not inside SaveAsync()
await SaveAsync();
```

Это улучшит производительность, так как это спасет компилятор от генерации дополнительного **асинхронного конечного автомата**.

Блокирование асинхронного кода может привести к блокировкам

Плохая практика блокировать асинхронные вызовы, так как это может вызвать взаимоблокировки в средах, которые имеют контекст синхронизации. Лучшей практикой является использование `async / await` «полностью вниз». Например, следующий код Windows Forms вызывает тупик:

```
private async Task<bool> TryThis()
{
    Trace.TraceInformation("Starting TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("In TryThis task");
        for (int i = 0; i < 100; i++)
        {
            // This runs successfully - the loop runs to completion
            Trace.TraceInformation("For loop " + i);
            System.Threading.Thread.Sleep(10);
        }
    });
    // This never happens due to the deadlock
    Trace.TraceInformation("About to return");
    return true;
}

// Button click event handler
private void button1_Click(object sender, EventArgs e)
{
    // .Result causes this to block on the asynchronous call
    bool result = TryThis().Result;
    // Never actually gets here
    Trace.TraceInformation("Done with result");
}
```

По сути, после завершения асинхронного вызова он ожидает, что контекст синхронизации станет доступным. Однако обработчик событий «держится» в контексте синхронизации, пока он ждет завершения `TryThis()`, что вызывает циклическое ожидание.

Чтобы исправить это, код должен быть изменен на

```
private async void button1_Click(object sender, EventArgs e)
{
    bool result = await TryThis();
    Trace.TraceInformation("Done with result");
}
```

Примечание. Обработчики событий - это единственное место, где следует использовать `async void` (потому что вы не можете ждать метода `async void`).

Async / await будет только улучшать производительность, если он позволяет машине выполнять дополнительную работу

Рассмотрим следующий код:

```

public async Task MethodA()
{
    await MethodB();
    // Do other work
}

public async Task MethodB()
{
    await MethodC();
    // Do other work
}

public async Task MethodC()
{
    // Or await some other async work
    await Task.Delay(100);
}

```

Это не будет лучше, чем

```

public void MethodA()
{
    MethodB();
    // Do other work
}

public void MethodB()
{
    MethodC();
    // Do other work
}

public void MethodC()
{
    Thread.Sleep(100);
}

```

Основная цель `async / await` - позволить машине выполнять дополнительную работу - например, разрешить вызывающему потоку выполнять другую работу, пока он ждет результата от какой-либо операции ввода-вывода. В этом случае вызывающему потоку никогда не разрешается выполнять больше работы, чем это было бы в противном случае, так что нет никакого увеличения производительности при простом вызове `MethodA()` , `MethodB()` И `MethodC()` СИНХРОННО.

Прочтайте Асинхронный-Await онлайн: <https://riptutorial.com/ru/csharp/topic/48/асинхронный-await>

глава 35: Атрибуты

Examples

Создание настраиваемого атрибута

```
//1) All attributes should be inherited from System.Attribute
//2) You can customize your attribute usage (e.g. place restrictions) by using
System.AttributeUsage Attribute
//3) You can use this attribute only via reflection in the way it is supposed to be used
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.
[MethodMetadata("This text could be retrieved via reflection")].
//5) You can overload an attribute constructors
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]
public class MethodMetadataAttribute : System.Attribute
{
    //this is custom field given just for an example
    //you can create attribute without any fields
    //even an empty attribute can be used - as marker
    public string Text { get; set; }

    //this constructor could be used as [MethodMetadata]
    public MethodMetadataAttribute ()
    {
    }

    //This constructor could be used as [MethodMetadata("String")]
    public MethodMetadataAttribute (string text)
    {
        Text = text;
    }
}
```

Использование атрибута

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

Чтение атрибута

Метод `GetCustomAttributes` возвращает массив пользовательских атрибутов, применяемых к элементу. После извлечения этого массива вы можете искать один или несколько конкретных атрибутов.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

Или итерации через них

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {  
    Console.WriteLine(attribute.GetType());  
}
```

GetCustomAttribute **расширения** GetCustomAttribute **из** System.Reflection.CustomAttributeExtensions извлекает пользовательский атрибут указанного типа, его можно применить к любому MemberInfo .

```
var attribute = (MyCustomAttribute)  
    typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

GetCustomAttribute **также** имеет общую подпись для указания типа атрибута для поиска.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

inherit логических аргументов может быть передано обоим этим методам. Если для этого значения установлено значение true предки элемента также будут проверяться.

Атрибут DebuggerDisplay

Добавление атрибута DebuggerDisplay изменит способ отображения класса отладчика при его зависании.

Выражения, которые завернуты в {} будут оцениваться отладчиком. Это может быть простое свойство, как в следующем примере или более сложной логике.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]  
public class AnObject  
{  
    public int ObjectId { get; set; }  
    public string StringProperty { get; set; }  
    public int IntProperty { get; set; }  
}
```



Добавляя , нq перед закрывающей скобкой удаляет кавычки при выводе строки.

```
[DebuggerDisplay("{StringProperty, nq} - {IntProperty}")]
```

Даже если общие выражения разрешены в {} они не рекомендуются. Атрибут `DebuggerDisplay` будет записан в метаданные сборки как строку. Выражения в {} не проверяются на достоверность. Таким образом, атрибут `DebuggerDisplay` содержащий более сложную логику, чем простая простая арифметика, может отлично работать на C#, но одно и то же выражение, оцениваемое в VB.NET, вероятно, не будет синтаксически корректным и приведет к ошибке при отладке.

Способ сделать `DebuggerDisplay` более агностиком языка - написать выражение в методе или свойстве и вызвать его вместо этого.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }

    private string DebuggerDisplay()
    {
        return $"{StringProperty} - {IntProperty}";
    }
}
```

Возможно, `DebuggerDisplay` может выводить все или только некоторые из свойств, а также при отладке и проверке типа объекта.

Пример ниже также окружает вспомогательный метод с `#if DEBUG` поскольку `DebuggerDisplay` используется в средах отладки.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }

    #if DEBUG
    private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
            Type:{this.GetType()}";
    }
    #endif
}
```

Атрибуты информации об абоненте

Атрибуты информации о вызывающем абоненте могут использоваться для передачи информации о вызове вызываемому методу. Декларация выглядит так:

```
using System.Runtime.CompilerServices;
```

```
public void LogException(Exception ex,
    [CallerMemberName] string callerMemberName = "",
    [CallerLineNumber] int callerLineNumber = 0,
    [CallerFilePath] string callerFilePath = "")  
{  
    //perform logging  
}
```

И вызов выглядит так:

```
public void Save(DBContext context)  
{  
    try  
    {  
        context.SaveChanges();  
    }  
    catch (Exception ex)  
    {  
        LogException(ex);  
    }  
}
```

Обратите внимание, что только первый параметр передается явно методу `LogException` тогда как остальные из них будут предоставлены во время компиляции соответствующими значениями.

Параметр `callerMemberName` получит значение "Save" - имя вызывающего метода.

Параметр `callerLineNumber` будет получать количество строк, на которые `LogException` вызов метода `LogException`.

И параметр 'callerFilePath' получит полный путь к файлу. Объявлен метод `Save`.

Чтение атрибута из интерфейса

Нет простого способа получить атрибуты из интерфейса, поскольку классы не наследуют атрибуты от интерфейса. Всякий раз, когда вы выполняете интерфейс или переопределяете члены производного класса, вам нужно повторно объявить атрибуты. Таким образом, в приведенном ниже примере вывод будет `True` во всех трех случаях.

```
using System;  
using System.Linq;  
using System.Reflection;  
  
namespace InterfaceAttributesDemo {  
  
    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]  
    class MyCustomAttribute : Attribute {  
        public string Text { get; set; }  
    }  
  
    [MyCustomAttribute(Text = "Hello from interface attribute")]  
    interface IMyClass {
```

```

        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True

            var attribute2 =
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
            Console.WriteLine(attribute2 == null); // True

            var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
            Console.WriteLine(attribute3 == null); // True
        }
    }
}

```

Одним из способов получения атрибутов интерфейса является поиск их через все интерфейсы, реализованные классом.

```

var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute

```

Устаревший атрибут

`System.Obsolete` - это атрибут, который используется для обозначения типа или члена, который имеет лучшую версию и, следовательно, не должен использоваться.

```

[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}

```

В случае использования класса выше компилятор выдаст предупреждение «Этот класс устарел. Вместо этого используйте `SomeOtherClass`».

Прочтайте Атрибуты онлайн: <https://riptutorial.com/ru/csharp/topic/1062/атрибуты>

глава 36: Включая ресурсы шрифтов

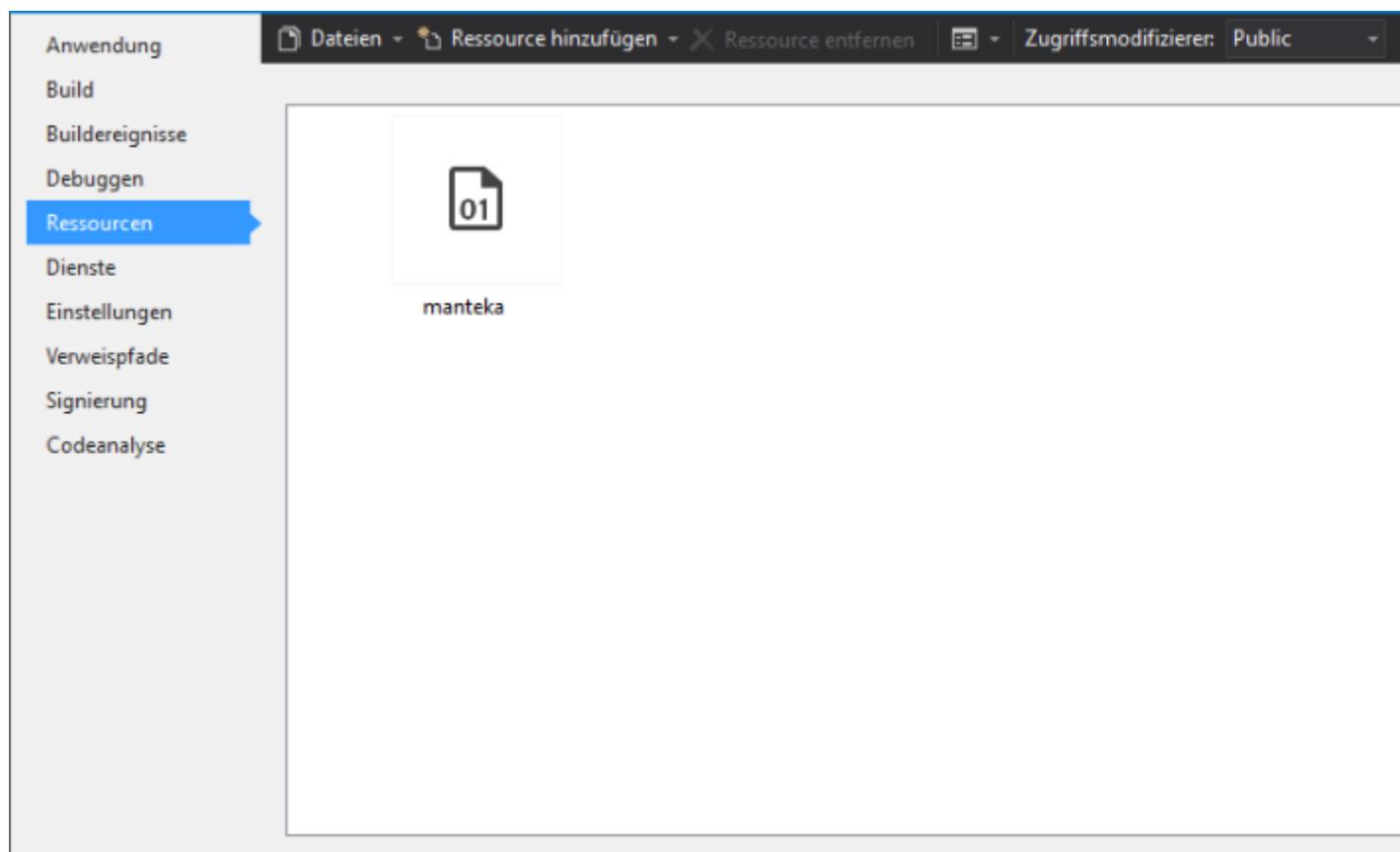
параметры

параметр	подробности
fontbytes	байтовый массив из двоичного .ttf

Examples

Выполнить «Fontfamily» из ресурсов

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



Интеграционный метод

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
}
```

```
    return pfc.Families[0];
}
```

Использование с кнопкой «Button»

```
public static class Res
{
    /// <summary>
    /// URL: https://www.behance.net/gallery/2846011/Manteka
    /// </summary>
    public static FontFamily Maneteke =
        GetResourceFontFamily(Properties.Resources.manteka);

    public static FontFamily GetResourceFontFamily(byte[] fontbytes)
    {
        PrivateFontCollection pfc = new PrivateFontCollection();
        IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
        Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
        pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
        Marshal.FreeCoTaskMem(fontMemPointer);
        return pfc.Families[0];
    }
}

public class FlatButton : Button
{
    public FlatButton() : base()
    {
        Font = new Font(Res.Maneteke, Font.Size);
    }

    protected override void OnFontChanged(EventArgs e)
    {
        base.OnFontChanged(e);
        this.Font = new Font(Res.Maneteke, this.Font.Size);
    }
}
```

Прочтайте Включая ресурсы шрифтов онлайн: <https://riptutorial.com/ru/csharp/topic/9789/включая-ресурсы-шрифтов>

глава 37: Внедрение зависимости

замечания

Википедия: определение инъекции зависимостей:

В программной инженерии инъекция зависимостей представляет собой шаблон разработки программного обеспечения, который реализует инверсию управления для разрешения зависимостей. Зависимость - это объект, который может использоваться (услуга). Инъекция - это передача зависимости зависимому объекту (клиенту), который будет использовать его.

** На этом сайте есть ответ на вопрос. **Как объяснить инъекцию зависимостей 5-летнему ребенку.** Самый высоко оцененный ответ, предоставленный Джоном Муншем, дает удивительно точную аналогию, нацеленную на (воображаемого) пятилетнего инквизитора: когда вы идете и извлекаете вещи из холодильника для себя, вы можете создавать проблемы. Вы можете оставить дверь открытой, вы можете получить то, что мама или папа не хотят, чтобы вы имели. Возможно, вы даже ищете что-то, чего у нас нет или которое истекло. То, что вы должны делать, это заявить о необходимости: «Мне нужно что-нибудь выпить с обедом», а затем мы обязательно удостовериться, что у вас есть что-то, когда вы садитесь, чтобы поесть. То, что это означает с точки зрения объектно-ориентированного развития программного обеспечения, таково: сотрудничающие классы (пятилетние) должны полагаться на инфраструктуру (родителей), чтобы обеспечить

** Этот код использует MEF для динамической загрузки DLL и разрешения зависимостей. Зависимость ILogger разрешается с помощью MEF и вводится в класс пользователя. Пользовательский класс никогда не получает конкретную реализацию ILogger и не знает, какой или какой тип регистратора он использует. **

Examples

Инъекция с использованием MEF

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
```

```

        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message) ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _loggers.First((n) =>
((n.Metadata.Name.ToUpper() == "Console")));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory = Path.Combine(Utilities.GetApplicationDirectory(),
"Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
    }
}

```

```

catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory));

//Create the CompositionContainer with the parts in the catalog
_container = new CompositionContainer(catalog);

//Fill the imports of this object
try
{
    this._container.ComposeParts(this);
}
catch (CompositionException compositionException)
{
    throw new CompositionException(compositionException.Message);
}
}
}
}

```

Инъекция зависимостей C # и ASP.NET с Unity

Во-первых, почему мы должны использовать инъекцию dependency в нашем коде? Мы хотим отделить другие компоненты от других классов нашей программы. Например, у нас есть класс AnimalController, который имеет такой код:

```

public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
        Console.WriteLine("");
    }
}

```

Мы смотрим на этот код, и мы думаем, что все в порядке, но теперь наш AnimalController зависит от объекта _SantaAndHisReindeer. Автоматически мой контроллер плохо тестирует и повторное использование моего кода будет очень тяжелым.

Очень хорошее объяснение, почему мы должны использовать Dependency Injection и интерфейсы [здесь](#).

Если мы хотим, чтобы Unity обрабатывала DI, дорога для достижения этого очень проста :) С NuGet (менеджер пакетов) мы можем легко импортировать единство в наш код.

в Visual Studio Tools -> Диспетчер пакетов NuGet -> Управление пакетами для решения -> в поле ввода ввода для ввода единства -> выберите наш проект-> нажмите кнопку установки

Теперь будут созданы два файла с хорошими комментариями.

в папке App-Data UnityConfig.cs и UnityMvcActivator.cs

UnityConfig - в методе RegisterTypes мы можем видеть тип, который будет

ВСТРАИВАТЬСЯ В НАШИ КОНСТРУКТОРЫ.

```
namespace Vegan.WebUi.App_Start
{
    public class UnityConfig
    {
        #region Unity Container
        private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
        {
            var container = new UnityContainer();
            RegisterTypes(container);
            return container;
        });

        /// <summary>
        /// Gets the configured Unity container.
        /// </summary>
        public static IUnityContainer GetConfiguredContainer()
        {
            return container.Value;
        }
        #endregion

        /// <summary>Registers the type mappings with the Unity container.</summary>
        /// <param name="container">The unity container to configure.</param>
        /// <remarks>There is no need to register concrete types such as controllers or API
        controllers (unless you want to
        /// change the defaults), as Unity allows resolving a concrete type even if it was not
        previously registered.</remarks>
        public static void RegisterTypes(IUnityContainer container)
        {
            // NOTE: To load from web.config uncomment the line below. Make sure to add a
            Microsoft.Practices.Unity.Configuration to the using statements.
            // container.LoadConfiguration();

            // TODO: Register your types here
            // container.RegisterType<IProductRepository, ProductRepository>();

            container.RegisterType<ISanta, SantaAndHisReindeer>();

        }
    }
}
```

UnityMvcActivator -> также с хорошими комментариями, которые говорят, что этот класс объединяет Unity с ASP.NET MVC

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]
```

```

namespace Vegan.WebUi.App_Start
{
    /// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
    public static class UnityWebActivator
    {
        /// <summary>Integrates Unity when the application starts.</summary>
        public static void Start()
        {
            var container = UnityConfig.GetConfiguredContainer();

            FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>());
            FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

            DependencyResolver.SetResolver(new UnityDependencyResolver(container));

            // TODO: Uncomment if you want to use PerRequestLifetimeManager
            // Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRe
        }

        /// <summary>Disposes the Unity container when the application is shut down.</summary>
        public static void Shutdown()
        {
            var container = UnityConfig.GetConfiguredContainer();
            container.Dispose();
        }
    }
}

```

Теперь мы можем отделить наш контроллер от класса `SantAndHisReindeer` :

```

public class AnimalController()
{
    private readonly SantaAndHisReindeer _SantaAndHisReindeer;

    public AnimalController(SantaAndHisReindeer SantaAndHisReindeer) {
        _SantAndHisReindeer = SantaAndHisReindeer;
    }
}

```

Перед тем, как запустить наше приложение, мы должны сделать все возможное.

В `Global.asax.cs` мы должны добавить новую строку: `UnityWebActivator.Start()`, которая запустится, настроит `Unity` и зарегистрирует наши типы.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

```

```
namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}
```

Прочтайте Внедрение зависимости онлайн: <https://riptutorial.com/ru/csharp/topic/5766/внедрение-зависимости>

глава 38: Внедрение шаблона дизайна **мухи**

Examples

Реализация карты в игре RPG

Flyweight - один из структурных шаблонов проектирования. Он используется для уменьшения объема используемой памяти путем совместного использования как можно большего количества данных с похожими объектами. В этом документе вы научитесь правильно использовать Flyweight DP.

Позвольте мне объяснить вам это на простом примере. Представьте, что вы работаете в RPG-игре, и вам нужно загрузить огромный файл, содержащий несколько символов. Например:

- # трава. Вы можете ходить по нему.
 - \$ - начальная точка
 - @ - рок. Вы не можете ходить по нему.
 - % - сундук с сокровищами

Пример карты:

Поскольку эти объекты имеют сходную характеристику, вам не нужно создавать отдельный объект для каждого поля карты. Я покажу вам, как использовать мухи.

Давайте определим интерфейс, который будут реализовывать наши поля:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Теперь мы можем создавать классы, представляющие наши поля. Мы также должны идентифицировать их как-то (я использовал перечисление):

```
public enum FieldType
{
    GRASS,
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}
```

Как я уже сказал, нам не нужно создавать отдельный экземпляр для каждого поля. Мы должны создать **репозиторий** полей. Суть Flyweight DP заключается в том, что мы динамически создаем объект только в том случае, если он нам нужен, и он еще не существует в нашем репо или возвращает его, если он уже существует. Давайте напишем простой класс, который будет обрабатывать это для нас:

```
public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
        }
        lstFields.Add(f);
        return f;
    }
}
```

```

        case FieldType.CHEST:
        default: f = new TreasureChest(); break;
    }
    lstFields.Add(f); //add it to repository
    Console.WriteLine("Created new instance of {0}", f.Name);
    return f;
}
public IField GetField(FieldType type)
{
    IField f = lstFields.Find(x => x.Type == type);
    if (f != null) return f;
    else return AddField(type);
}
}

```

Большой! Теперь мы можем проверить наш код:

```

public class Program
{
    public static void Main(string[] args)
    {
        FieldRepository f = new FieldRepository();
        IField grass = f.GetField(FieldType.GRASS);
        grass = f.GetField(FieldType.ROCK);
        grass = f.GetField(FieldType.GRASS);
    }
}

```

Результат в консоли должен быть:

Создал новый экземпляр Grass

Создал новый экземпляр Rock

Но почему трава появляется только один раз, если мы хотим получить ее дважды? Это потому, что первый раз, когда мы вызываем экземпляр `GetField`, в нашем **репозитории** не существует, поэтому он создан, но в следующий раз нам нужна трава, которая уже существует, поэтому мы возвращаем ее только.

Прочтайте Внедрение шаблона дизайна мухи онлайн:

<https://riptutorial.com/ru/csharp/topic/4619/внедрение-шаблона-дизайна-мухи>

глава 39: Внедрение шаблона проектирования декоратора

замечания

Плюсы использования Decorator:

- вы можете добавлять новые функции во время выполнения в разных конфигурациях
- хорошая альтернатива для наследования
- клиент может выбрать конфигурацию, которую он хочет использовать

Examples

Моделирование столовой

Декоратор - один из структурных шаблонов дизайна. Он используется для добавления, удаления или изменения поведения объекта. В этом документе вы узнаете, как правильно использовать Decorator DP.

Позвольте мне объяснить вам это на простом примере. Представьте, что вы сейчас в знаменитой кофейной компании Starbobs. Вы можете заказать заказ на любой кофе, который вы хотите - со сливками и сахаром, со сливками и топпингом и намного больше комбинаций! Но в основе всех напитков кофе - темный, горький напиток, вы можете изменить. Давайте напишем простую программу, имитирующую кофеварку.

Во-первых, нам нужно создать и абстрактный класс, который описывает наш базовый напиток:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Теперь давайте создадим некоторые дополнительные услуги, такие как сахар, молоко и топпинг. Созданные классы должны реализовать `AbstractCoffee` - они украсят его:

```
public class Milk : AbstractCoffee
{
```

```

public Milk(AbstractCoffee c) : base(c) { }
public override string ShowCoffee()
{
    if (k != null)
        return k.ShowCoffee() + " with Milk";
    else return "Milk";
}
public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}
public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
        else return "Topping";
    }
}

```

Теперь мы можем создать наш любимый кофе:

```

public class Program
{
    public static void Main(string[] args)
    {
        AbstractCoffee coffee = null; //we cant create instance of abstract class
        coffee = new Topping(coffee); //passing null
        coffee = new Sugar(coffee); //passing topping instance
        coffee = new Milk(coffee); //passing sugar
        Console.WriteLine("Coffee with " + coffee.ShowCoffee());
    }
}

```

Запуск кода приведет к следующему выводу:

Кофе с топкой с сахаром с молоком

Прочтайте Внедрение шаблона проектирования декоратора онлайн:

<https://riptutorial.com/ru/csharp/topic/4798/внедрение-шаблона-проектирования-декоратора>

глава 40: Возможности С # 3.0

замечания

Версия С # 3.0 была выпущена как часть .Net версии 3.5. Многие из функций, добавленных в эту версию, поддерживались LINQ (языковые INtegrated Queries).

Список дополнительных функций:

- LINQ
- Лямбда-выражения
- Методы расширения
- Анонимные типы
- Неявно типизированные переменные
- Инициализаторы объектов и коллекций
- Автоматически реализованные свойства
- Деревья выражений

Examples

Неявно введенные переменные (var)

Ключевое слово `var` позволяет программисту неявно вводить переменную во время компиляции. объявления `var` имеют тот же тип, что и явно объявленные переменные.

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

Типы вышеперечисленных переменных - это `int`, `double`, `StringBuilder` и анонимный тип соответственно.

Важно отметить, что переменная `var` не является динамически типизированной.

`SquaredNumber = Builder` недействителен, так как вы пытаетесь установить `int` в экземпляр `StringBuilder`

Языковые интегрированные запросы (LINQ)

```
//Example 1
```

```
int[] array = { 1, 5, 2, 10, 7 };

// Select squares of all odd numbers in the array sorted in descending order
IQueryable<int> query = from x in array
                         where x % 2 == 1
                         orderby x descending
                         select x * x;

// Result: 49, 25, 1
```

Пример из статьи в Википедии на C # 3.0, подраздел LINQ

В примере 1 используется синтаксис запроса, который был спроектирован так, чтобы выглядеть аналогично SQL-запросам.

```
//Example 2
IQueryable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

Пример из статьи в Википедии на C # 3.0, подраздел LINQ

В примере 2 используется синтаксис метода для достижения того же результата, что и в примере 1.

Важно отметить, что в C # синтаксис запроса LINQ представляет собой [синтаксический сахар](#) для синтаксиса метода LINQ. Компилятор переводит запросы в вызовы методов во время компиляции. Некоторые запросы должны быть выражены в синтаксисе метода. [Из MSDN](#) - «Например, вы должны использовать вызов метода для выражения запроса, который извлекает количество элементов, которые соответствуют указанному условию».

Lambda expresions

Lambda Expressions - это расширение [анонимных методов](#), которые допускают неявно типизированные параметры и возвращаемые значения. Их синтаксис менее подробный, чем анонимный, и следует за функциональным стилем программирования.

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

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

Вышеприведенный код выводит сумму квадратов от 1 до 10 на консоль.

Первое лямбда-выражение квадратизирует числа в списке. Поскольку существует только одна скобка параметра, ее можно опустить. Вы можете включить скобки, если хотите:

```
.Select( number => number * number);
```

или явно введите параметр, но затем требуются скобки:

```
.Select( int number => number * number);
```

Тело лямбда является выражением и имеет неявное возвращение. Вы можете использовать тело оператора, если хотите. Это полезно для более сложных лямбдов.

```
.Select( number => { return number * number; } );
```

Метод `select` возвращает новый `IEnumerable` с вычисленными значениями.

Второе лямбда-выражение суммирует числа в списке, возвращаемом методом `select`. Скобки необходимы, поскольку существует несколько параметров. Типы параметров явно напечатаны, но это необязательно. Следующий метод эквивалентен.

```
.Aggregate( first, second => { return first + second; } );
```

Как и этот:

```
.Aggregate( int first, int second => first + second );
```

Анонимные типы

Анонимные типы предоставляют удобный способ инкапсулировать набор свойств только для чтения в один объект без необходимости явно определять тип в первую очередь. Имя типа генерируется компилятором и не доступно на уровне исходного кода. Тип каждого свойства определяется компилятором.

Вы можете создавать анонимные типы, используя `new` ключевое слово, за которым следует фигурная скобка `()`. Внутри фигурных скобок вы можете определить свойства, подобные приведенному ниже.

```
var v = new { Amount = 108, Message = "Hello" };
```

Также возможно создать массив анонимных типов. См. Следующий код:

```
var a = new[] {
    new {
        Fruit = "Apple",
```

```
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

Или используйте его с запросами LINQ:

```
var productQuery = from prod in products
                    select new { prod.Color, prod.Price };
```

Прочтайте **Возможности C # 3.0** онлайн: <https://riptutorial.com/ru/csharp/topic/3820/возможности-c-sharp-3-0>

глава 41: Возможности С # 4.0

Examples

Необязательные параметры и именованные аргументы

Мы можем опустить аргумент в вызове, если этот аргумент является необязательным аргументом. Каждый необязательный аргумент имеет собственное значение по умолчанию. Оно будет принимать значение по умолчанию, если мы не укажем значение. Значение по умолчанию для необязательного аргумента должно быть

1. Постоянное выражение.
2. Должен быть тип значения, например enum или struct.
3. Должно быть выражение формы default (valueType)

Он должен быть установлен в конце списка параметров

Параметры метода со значениями по умолчанию:

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)
{
    //...
}
```

Как сказано в MSDN, именованный аргумент,

Позволяет передавать аргумент функции, связывая имя параметра. Нет необходимости запоминать позицию параметров, о которой мы не знаем всегда. Не нужно искать порядок параметров в списке параметров вызываемой функции. Мы можем указать параметр для каждого аргумента по его имени.

Именованные аргументы:

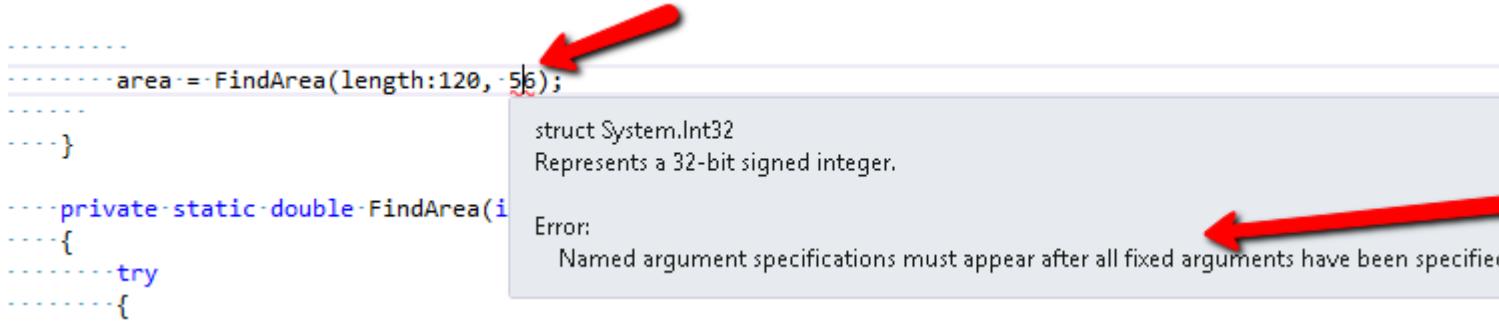
```
// required = 3, optValue = "test", optNum = 4
ExampleMethod(3, optNum: 4);
// required = 2, optValue = "foo", optNum = 42
ExampleMethod(2, optValue: "foo");
// required = 6, optValue = "bar", optNum = 1
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

Ограничение использования именованного аргумента

Именованная спецификация аргумента должна появиться после указания всех фиксированных аргументов.

Если вы используете именованный аргумент перед фиксированным аргументом, вы

получите ошибку времени компиляции следующим образом.



```
area = FindArea(length:120, 56);
```

struct System.Int32
Represents a 32-bit signed integer.

Error:
Named argument specifications must appear after all fixed arguments have been specified

Именованная спецификация аргумента должна появляться после указания всех фиксированных аргументов

ОТКЛОНЕНИЕ

Общие интерфейсы и делегаты могут иметь параметры своего типа, помеченные как *ковариантные* или *контравариантные*, используя ключевые слова `out` и `in`. Затем эти декларации учитываются для преобразований типов, как неявных, так и явных, и как время компиляции, так и время выполнения.

Например, существующий интерфейс `IEnumerable<T>` был переопределен как ковариантный:

```
interface IEnumerable<out T>
{
    Ienumerator<T> GetEnumerator();
}
```

Существующий интерфейс `IComparer` был переопределен как контравариантный:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Необязательное ключевое слово `ref` при использовании СОМ

Ключевое слово `ref` для вызывающих методов теперь является необязательным при вызове методов, предоставляемых интерфейсами СОМ. Учитывая метод СОМ с подписью

```
void Increment(ref int x);
```

теперь вызов можно записать как либо

```
Increment(0); // no need for "ref" or a place holder variable any more
```

Динамический поиск элементов

В систему типов C# вводится новая псевдо-`dynamic`. Он рассматривается как `System.Object`, но кроме того, любой членский доступ (вызов метода, поле, свойство или доступ индексатора или вызов делегата) или приложение оператора по значению такого типа разрешено без проверки типа, и его разрешение откладывается до времени выполнения. Это называется утиным типом или поздним связыванием. Например:

```
// Returns the value of Length property or field of any object
int GetLength(dynamic obj)
{
    return obj.Length;
}

GetLength("Hello, world");           // a string has a Length property,
GetLength(new int[] { 1, 2, 3 });    // and so does an array,
GetLength(42);                     // but not an integer - an exception will be thrown
                                    // in GetLength method at run-time
```

В этом случае динамический тип используется, чтобы избежать более подробного отражения. Он по-прежнему использует Reflection под капотом, но обычно это происходит быстрее благодаря кешированию.

Эта функция в первую очередь ориентирована на взаимодействие с динамическими языками.

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

Динамический тип имеет приложения даже в основном статически типизированном коде, например, делает [двойную отправку](#) возможной без реализации шаблона посетителя.

Прочтайте [Возможности C# 4.0](#) онлайн: <https://riptutorial.com/ru/csharp/topic/3093/возможности-c-sharp-4-0>

глава 42: Возможности С # 5.0

Синтаксис

- **Async & Await**

- `public Task MyTask Async () {doSomething (); }`

ждать `MyTaskAsync ()`;

- `public Task <string> MyStringTask Async () {return getSomeString (); }`

`string MyString = ожидание MyStringTaskAsync ()`;

- **Атрибуты информации о вызывающем абоненте**

- `public void MyCallerAttributes (строка MyMessage,`

`[CallerMemberName] string MemberName = "",`

`[CallerFilePath] string SourceFilePath = "",`

`[CallerLineNumber] int LineNumber = 0)`

- `Trace.WriteLine ("Мое сообщение:" + MyMessage);`

`Trace.WriteLine ("Member:" + MemberName);`

`Trace.WriteLine ("«Путь исходного файла:» + SourceFilePath);`

`Trace.WriteLine ("Номер строки:" + LineNumber);`

параметры

Метод / модификатор с параметром	подробности
<code>Type<T></code>	<code>T</code> - тип возврата

замечания

С # 5.0 сочетается с Visual Studio .NET 2012

Examples

Async & Await

`async` И `await` два оператора, которые предназначены для повышения производительности, освобождая потоки и ожидая завершения операций перед тем, как двигаться вперед.

Вот пример получения строки перед возвратом ее длины:

```
//This method is async because:  
//1. It has async and Task or Task<T> as modifiers  
//2. It ends in "Async"  
async Task<int> GetStringLengthAsync(string URL){  
    HttpClient client = new HttpClient();  
    //Sends a GET request and returns the response body as a string  
    Task<string> getString = client.GetStringAsync(URL);  
    //Waits for getString to complete before returning its length  
    string contents = await getString;  
    return contents.Length;  
}  
  
private async void doProcess(){  
    int length = await GetStringLengthAsync("http://example.com/");  
    //Waits for all the above to finish before printing the number  
    Console.WriteLine(length);  
}
```

Вот еще один пример загрузки файла и обработки того, что происходит, когда его прогресс изменился, и когда загрузка завершена (есть два способа сделать это):

Способ 1:

```
//This one using async event handlers, but not async coupled with await  
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){  
    WebClient web = new WebClient();  
    //Assign the event handler  
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);  
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);  
    //Download the file asynchronously  
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);  
}  
  
//event called for when download progress has changed  
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){  
    //example code  
    int i = 0;  
    i++;  
    doSomething();  
}  
  
//event called for when download has finished  
private void FileCompleted(object sender, AsyncCompletedEventArgs e){  
    Console.WriteLine("Completed!")  
}
```

Способ 2:

```

//however, this one does
//Refer to first example on why this method is async
private void DownloadAndUpdateAsync(string uri, string DownloadLocation) {
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    //Download the file async
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
    //Notice how there is no complete event, instead we're using techniques from the first
example
}
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e) {
    int i = 0;
    i++;
    doSomething();
}
private void doProcess() {
    //Wait for the download to finish
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))
    doSomething();
}

```

Атрибуты информации о вызывающем абоненте

ЦРУ предназначены как простой способ получения атрибутов из того, что вызывается целевым методом. Существует только один способ использовать их, и есть только 3 атрибута.

Пример:

```

//This is the "calling method": the method that is calling the target method
public void doProcess()
{
    GetMessageCallerAttributes("Show my attributes.");
}
//This is the target method
//There are only 3 caller attributes
public void GetMessageCallerAttributes(string message,
    //gets the name of what is calling this method
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    //gets the path of the file in which the "calling method" is in
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    //gets the line number of the "calling method"
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    //Writes lines of all the attributes
    System.Diagnostics.Trace.WriteLine("Message: " + message);
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);
}

```

Пример:

```

//Message: Show my attributes.
//Member: doProcess

```

```
//Source File Path: c:\Path\To\The\File  
//Line Number: 13
```

Прочтайте Возможности C # 5.0 онлайн: <https://riptutorial.com/ru/csharp/topic/4584/возможности-c-sharp-5-0>

глава 43: Возможности C # 6.0

Вступление

Эта шестнадцатая итерация языка C # предоставляет компилятором Roslyn. Этот компилятор вышел с версией 4.6 .NET Framework, однако он может генерировать код с обратной совместимостью, чтобы разрешить таргетинг на ранние версии рамок. Код C # версии 6 может быть скомпилирован полностью совместимым образом с .NET 4.0. Его также можно использовать для ранних фреймворков, однако некоторые функции, требующие дополнительной поддержки фреймворка, могут работать некорректно.

замечания

Шестая версия C # была выпущена в июле 2015 года вместе с Visual Studio 2015 и .NET 4.6.

Помимо добавления некоторых новых функций языка, он включает полную переработку компилятора. Ранее `csc.exe` было родным приложением Win32, написанным на C ++, а C # 6 теперь является управляемым .NET-программным обеспечением, написанным на C #. Этот переписчик был известен как проект «Roslyn», и код теперь открыт с открытым исходным кодом и доступен на [GitHub](https://github.com/dotnet/roslyn) .

Examples

Имя оператора

Оператор `nameof` возвращает имя элемента кода в виде `string` . Это полезно при бросании исключений, связанных с аргументами метода, а также при внедрении `INotifyPropertyChanged` .

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

Оператор `nameof` вычисляется во время компиляции и изменяет выражение в строковый литерал. Это также полезно для строк, названных в честь их члена, который их раскрывает. Рассмотрим следующее:

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
```

```
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

Поскольку выражения `nameof` являются константами времени компиляции, их можно использовать в атрибутах, метках `case`, операторах `switch` и т. д.

Удобно использовать `nameof` с `Enum` с. Вместо:

```
Console.WriteLine(Enum.One.ToString());
```

можно использовать:

```
Console.WriteLine(nameof(Enum.One))
```

В обоих случаях выход будет `One`.

Оператор `nameof` может обращаться к `nameof` элементам, используя синтаксис типа `static`. Вместо того, чтобы делать:

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

Может быть заменено на:

```
string lengthName = nameof(string.Length);
```

В обоих примерах выход будет `Length`. Однако последний препятствует созданию ненужных экземпляров.

Хотя `nameof` оператора работает с большинством языковых конструкций, существуют некоторые ограничения. Например, вы не можете использовать оператор `nameof` для открытых общих типов или возвращаемых значений метода:

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

Кроме того, если вы примените его к родовому типу, параметр родового типа будет проигнорирован:

```
Console.WriteLine(nameof(List<int>)); // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

Дополнительные примеры см. В [этом разделе](#), посвященном `nameof`.

Обходной путь для предыдущих версий (подробнее)

Хотя оператор `nameof` не существует в C# для версий до 6.0, аналогичную функциональность можно использовать с помощью `MemberExpression` как `MemberExpression` ниже:

6,0

Выражение:

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

Использование:

```
string variableName = NameOf(() => variable);
string propertyName = NameOf((Foo o) => o.Bar);
```

Обратите внимание, что этот подход заставляет `nameof` дерево выражений при каждом вызове, поэтому производительность намного хуже по сравнению с `nameof` оператора, который оценивается во время компиляции и имеет нулевые служебные данные во время выполнения.

Элементы функции выражения

Элементы функции, выражающие выражение, позволяют использовать лямбда-выражения в качестве элементов-членов. Для простых членов это может привести к более чистым и понятным кодам.

Функциональные выражения могут использоваться для свойств, индексаторов, методов и операторов.

Свойства

```
public decimal TotalPrice => BasePrice + Taxes;
```

Эквивалентно:

```
public decimal TotalPrice
{
    get
    {
        return BasePrice + Taxes;
    }
}
```

Когда функция expression-bodied используется с свойством, свойство реализуется как свойство только для getter.

[Посмотреть демо](#)

Индексаторы

```
public object this[string key] => dictionary[key];
```

Эквивалентно:

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

Методы

```
static int Multiply(int a, int b) => a * b;
```

Эквивалентно:

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

Который также может использоваться с `void` методами:

```
public void Dispose() => resource?.Dispose();
```

В класс `Pair<T>` можно добавить переопределение `ToString`:

```
public override string ToString() => $"{First}, {Second}";
```

Кроме того, этот упрощенный подход работает с ключевым словом `override`:

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

операторы

Это также может использоваться операторами:

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

Ограничения

Элементы функции с выражением имеют некоторые ограничения. Они не могут содержать операторы блоков и любые другие операторы, которые содержат блоки: `if`, `switch`, `for`, `foreach`, `while`, `do`, `try` и т. д.

Некоторые операторы `if` могут быть заменены на тернарные операторы. Некоторые операторы `for` и `foreach` могут быть преобразованы в запросы LINQ, например:

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

Во всех остальных случаях можно использовать старый синтаксис для членов функции.

Элементы функции Expression-body могут содержать `async / await`, но это часто избыточно:

```
async Task<int> Foo() => await Bar();
```

Может быть заменено на:

```
Task<int> Foo() => Bar();
```

Исключительные фильтры

Фильтры исключений дают разработчикам возможность добавить условие (в виде `boolean` выражения) в блок `catch`, позволяя `catch` выполнять, только если условие оценивается как `true`.

Фильтры исключений позволяют распространять информацию об отладке в исходном исключении, где при использовании оператора `if` внутри блока `catch` и повторного выброса исключения останавливает распространение отладочной информации в исходном исключении. При использовании фильтров исключений исключение продолжает распространяться вверх в стеке вызовов, если условие не выполняется. В результате фильтры исключений значительно облегчают процесс отладки. Вместо того, чтобы останавливаться на инструкции `throw`, отладчик остановится на выражении, исключающем исключение, с сохранением текущего состояния и всех локальных переменных.

Аналогичным образом затрагиваются аварийные свалки.

Фильтры исключений поддерживались **CLR** с самого начала, и они были доступны из VB.NET и F# в течение более десяти лет, подвергая часть модели обработки исключений CLR. Только после выпуска C# 6.0 функциональность также была доступна для разработчиков C#.

Использование фильтров исключений

Фильтры исключений используются путем добавления предложения `when` в выражение `catch`. Можно использовать любое выражение, возвращающее `bool` в предложении `when` (кроме **ожидания**). Объявленная переменная `exception ex` доступна из предложения `when`:

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
```

```
{  
    throw new Exception("An error occurred accessing the database", ex);  
}
```

Множество блоков `catch when` предложения могут быть объединены. Первое, `when` предложение, возвращающее `true` приведет к тому, что исключение будет обнаружено. Его блок `catch` будет введен, в то время как другие предложения `catch` будут проигнорированы (их, `when` предложения не будут оцениваться). Например:

```
try  
{ ... }  
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,  
                           //the rest of the catches are ignored.  
{ ... }  
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if  
                           //someCondition evaluates to false  
{ ... }  
catch (Exception ex) // If both when clauses evaluate to false  
{ ... }
```

Опасно, когда статья

предосторожность

Может быть опасно использовать фильтры исключений: когда `Exception` выбрано из предложения `when`, `Exception` из предложения `when` игнорируется и считается `false`. Такой подход позволяет разработчикам писать, `when` клаузулы, не заботясь о недопустимых случаях.

Следующий пример иллюстрирует такой сценарий:

```
public static void Main()  
{  
    int a = 7;  
    int b = 0;  
    try  
    {  
        DoSomethingThatMightFail();  
    }  
    catch (Exception ex) when (a / b == 0)  
    {  
        // This block is never reached because a / b throws an ignored  
        // DivideByZeroException which is treated as false.  
    }  
    catch (Exception ex)  
    {  
        // This block is reached since the DivideByZeroException in the  
        // previous when clause is ignored.  
    }  
}  
  
public static void DoSomethingThatMightFail()
```

```
{  
    // This will always throw an ArgumentNullException.  
    Type.GetType(null);  
}
```

Посмотреть демо

Обратите внимание, что фильтры исключений исключают проблемы с запутанными номерами строк, связанные с использованием `throw` когда код ошибки находится в одной и той же функции. Например, в этом случае номер строки отображается как 6 вместо 3:

```
1. int a = 0, b = 0;  
2. try {  
3.     int c = a / b;  
4. }  
5. catch (DivideByZeroException) {  
6.     throw;  
7. }
```

Номер строки исключений сообщается как 6, потому что ошибка была поймана и повторно выбрана с помощью оператора `throw` в строке 6.

То же самое происходит с фильтрами исключений:

```
1. int a = 0, b = 0;  
2. try {  
3.     int c = a / b;  
4. }  
5. catch (DivideByZeroException) when (a != 0) {  
6.     throw;  
7. }
```

В этом примере `a` равно 0, тогда предложение `catch` игнорируется, но 3 указывается как номер строки. Это происходит потому, что они **не разматывают стек**. Более конкретно, исключение **не попадает** в строку 5, потому `a` на самом деле оно равно 0 и, следовательно, нет возможности исключить исключение в строке 6, потому что строка 6 не выполняется.

Регистрация в качестве побочного эффекта

Вызовы метода в состоянии могут вызывать побочные эффекты, поэтому фильтры исключения могут использоваться для запуска кода на исключениях, не вылавливая их. Общим примером, который использует это, является метод `Log` который всегда возвращает `false`. Это позволяет отслеживать информацию журнала во время отладки без необходимости повторного выброса исключения.

Имейте в виду, что, хотя это, кажется, удобный способ ведения журнала, это может быть рискованным, особенно если используются сторонние протоколирующие сборки. Они могут генерировать исключения при входе в не

очевидные ситуации, которые могут быть легко обнаружены (см. **Рискованное, when(...)** выше).

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex) when (Log(ex, "An error occurred"))
{
    // This catch block will never be reached
}

// ...

static bool Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
    return false;
}
```

[Посмотреть демо](#)

Общий подход в предыдущих версиях C# заключался в регистрации и повторном выбросе исключения.

6,0

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
    Log(ex, "An error occurred");
    throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
}
```

[Посмотреть демо](#)

finally блок

Блок `finally` выполняется каждый раз, независимо от того, выбрано ли исключение или нет. Одна тонкость с выражениями, `when` есть фильтры исключений, выполняется дальше в стеке, прежде чем вводить внутренние блоки `finally`. Это может привести к неожиданным результатам и поведению, когда код пытается изменить глобальное состояние (например,

пользователь или культура текущего потока) и установить его обратно в блок `finally`.

Пример: `finally` блок

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Производительность:

Начните
EvaluatesTo: True
Внутренний Наконец
Ловить
Внешнее окончание

[Посмотреть демо](#)

В приведенном выше примере, если метод `SomeOperation` не желает «утечки», глобальное состояние изменяется на предложения вызывающего, `when` оно должно быть, оно также

должно содержать блок `catch` для изменения состояния. Например:

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
        Flag = false;
        throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Также часто бывает, что `IDisposable` вспомогательные классы используют семантику **использования** блоков для достижения той же цели, что и `IDisposable.Dispose` всегда `IDisposable.Dispose` до того, как исключение, вызванное внутри `using` блока, начнет разворачивать стек.

Инициализаторы автоистории

Вступление

Свойства можно инициализировать с помощью оператора `=` после закрытия `} . В приведенном ниже разделе « Coordinate » показаны доступные параметры инициализации свойства:`

6,0

```
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer

    public int Y { get; } = 89; // read-only auto-property with initializer
}
```

Аксессоры с различной видимостью

Вы можете инициализировать автоматические свойства, которые имеют различную видимость для их аксессуаров. Вот пример с защищенным сеттером:

```
public string Name { get; protected set; } = "Cheeze";
```

Аксессор также может быть `internal`, `internal protected` ИЛИ `private`.

Свойства только для чтения

Помимо гибкости с видимостью, вы также можете инициализировать автоматические свойства только для чтения. Вот пример:

```
public List<string> Ingredients { get; } =  
    new List<string> { "dough", "sauce", "cheese" };
```

В этом примере также показано, как инициализировать свойство сложным типом. Кроме того, авто-свойства не могут быть только для записи, поэтому также исключает инициализацию только для записи.

Старый стиль (pre C # 6.0)

До C # 6 это требовало гораздо более подробного кода. Мы использовали одну дополнительную переменную, называемую свойство `backing` для свойства, чтобы присвоить значение по умолчанию или инициализировать публичное свойство, как показано ниже,

6,0

```
public class Coordinate  
{  
    private int _x = 34;  
    public int X { get { return _x; } set { _x = value; } }  
  
    private readonly int _y = 89;  
    public int Y { get { return _y; } }  
  
    private readonly int _z;  
    public int Z { get { return _z; } }  
  
    public Coordinate()  
    {  
        _z = 42;  
    }  
}
```

Примечание. До C # 6.0 вы все равно можете инициализировать чтение и запись **автоматически реализованных свойств** (свойств с помощью `getter` и `setter`) внутри конструктора, но вы не можете инициализировать свойство `inline` с его объявлением

[Посмотреть демо](#)

ИСПОЛЬЗОВАНИЕ

Инициализаторы должны оценивать статические выражения, как и инициализаторы полей. Если вам нужно ссылаться на нестатические элементы, вы можете либо инициализировать свойства в конструкторах, как раньше, либо использовать свойства с выражением. Нестатические выражения, подобные приведенным ниже (закомментированные), генерируют ошибку компилятора:

```
// public decimal X { get; set; } = InitMe(); // generates compiler error

decimal InitMe() { return 4m; }
```

Но статические методы **могут** использоваться для инициализации автоматических свойств:

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

Этот метод также можно применять к свойствам с различным уровнем доступа:

```
public short Type { get; private set; } = 15;
```

Инициализатор автоматического свойства позволяет присваивать свойства непосредственно в объявлении. Для свойств только для чтения он выполняет все требования, необходимые для обеспечения неизменности свойства. Рассмотрим, например, класс `FingerPrint` в следующем примере:

```
public class FingerPrint
{
    public DateTime TimeStamp { get; } = DateTime.UtcNow;

    public string User { get; } =
        System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

    public string Process { get; } =
        System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

[Посмотреть демо](#)

Предупреждающие примечания

Постарайтесь не путать инициализаторы авто-свойств или поля с похожими [способами выражения-тела](#), которые используют `=>` в противоположность `=`, и поля, которые не включают `{ get; }`.

Например, каждая из следующих деклараций различна.

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

Отсутствует `{ get; }` в объявлении свойства приводит к публичному полю. И автообновление `Users1` чтения, и поле чтения-записи `Users2` инициализируются только один раз, но общедоступное поле позволяет изменять экземпляр коллекции вне класса, что обычно нежелательно. Изменение автоматического свойства только для чтения с телом выражения в свойство только для чтения с инициализатором требует не только удаления `> from =>`, но и добавления `{ get; }`.

`Users3` символ `(=> вместо =)` в `Users3` приводит к каждому доступу к свойству, возвращающему новый экземпляр `HashSet<UserDto>` который, хотя действительный C # (с точки зрения компилятора) вряд ли будет желательным, когда используется для члена коллекции.

Вышеприведенный код эквивалентен:

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // This is a property which returns a new HashSet<UserDto> as
    // an ICollection<UserDto> on each call to it.
    public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
}
```

Инициализаторы индекса

Инициализаторы индексов позволяют одновременно создавать и инициализировать объекты с индексами.

Это упрощает инициализацию словарей:

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

Любой объект, который имеет индексированный `getter` или `setter`, может использоваться с этим синтаксисом:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42
        };

        Console.ReadKey();
    }
}
```

Выход:

```
Индекс: foo, значение: 34
Индекс: bar, значение: 42
```

[Посмотреть демо](#)

Если класс имеет несколько индексаторов, их можно назначить в одной группе операторов:

```
class Program
{
```

```

public class MyClassWithIndexer
{
    public int this[string index]
    {
        set
        {
            Console.WriteLine($"Index: {index}, value: {value}");
        }
    }
    public string this[int index]
    {
        set
        {
            Console.WriteLine($"Index: {index}, value: {value}");
        }
    }
}

public static void Main()
{
    var x = new MyClassWithIndexer()
    {
        ["foo"] = 34,
        ["bar"] = 42,
        [10] = "Ten",
        [42] = "Meaning of life"
    };
}
}

```

Выход:

```

Индекс: foo, значение: 34
Индекс: bar, значение: 42
Индекс: 10, значение: Десять
Индекс: 42, значение: Значение жизни

```

Следует отметить, что аксессор `set` индексов может вести себя по-разному по сравнению с методом `Add` (используется в инициализаторах коллекции).

Например:

```

var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // does not throw, second value overwrites the first one

```

против:

```

var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.

```

Строчная интерполяция

Строковая интерполяция позволяет разработчику комбинировать `variables` и текст для формирования строки.

Основной пример

int **две переменные** int : foo И bar .

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";

Console.WriteLine(resultString);
```

Выход :

Foo - 34, а бара - 42.

[Посмотреть демо](#)

Скобки внутри строк все еще можно использовать, например:

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is {{foo}}, and the bar is {{bar}}.");
```

Это дает следующий результат:

Foo - {foo}, а bar - {bar}.

Использование интерполяции со стенографическими строками `verbatim`

Использование `@` перед строкой приведет к тому, что строка будет интерпретирована дословно. Таким образом, например, символы Unicode или разрывы строк будут оставаться такими же, как они были напечатаны. Однако это не повлияет на выражения в интерполированной строке, как показано в следующем примере:

```
Console.WriteLine(${@"In case it wasn't clear: "});
```

```
\u00B9
The foo
is {foo},
and the bar
is {bar}.";
```

Выход:

В случае непонятности:

```
\u00B9
Foo
составляет 34,
и бар
составляет 42.
```

[Посмотреть демо](#)

Выражения

При строковой интерполяции **выражения** в фигурных скобках {} также могут быть оценены. Результат будет вставлен в соответствующее место в строке. Например, чтобы вычислить максимум `foo` и `bar` и вставить его, используйте `Math.Max` в фигурных скобках:

```
Console.WriteLine($"And the greater one is: { Math.Max(foo, bar) }");
```

Выход:

И тем больше: 42

Примечание. Любые ведущие или конечные пробелы (включая пробел, вкладку и CRLF / новую строку) между фигурной скобкой и выражением полностью игнорируются и не включаются в вывод

[Посмотреть демо](#)

В качестве другого примера переменные могут быть отформатированы как валюта:

```
Console.WriteLine($"Foo formatted as a currency to 4 decimal places: {foo:c4}");
```

Выход:

Foo отформатирован как валюта до 4 знаков после запятой: \$ 34,0000

[Посмотреть демо](#)

Или они могут быть отформатированы как даты:

```
Console.WriteLine($"Today is: {DateTime.Today:ddd, MMMM dd - yyyy}");
```

Выход:

Сегодня: понедельник, 20 июля - 2015 г.

[Посмотреть демо](#)

Операции с [условным \(тройным\) оператором](#) также могут быть оценены внутри интерполяции. Однако они должны быть заключены в круглые скобки, поскольку двоеточие в противном случае используется для указания форматирования, как показано выше:

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")});
```

Выход:

Бар больше, чем foo!

[Посмотреть демо](#)

Условные выражения и спецификаторы формата могут быть смешаны:

```
Console.WriteLine($"Environment: {(Environment.Is64BitProcess ? 64 : 32):00}'-bit'{process}");
```

Выход:

Окружающая среда: 32-битный процесс

Эквивалентные последовательности

Символы обратной косой черты (\) и quote (") работают точно так же в интерполированных строках, как и в неинтерполированных строках, как для дословных, так и для невербальных строковых литералов:

```
Console.WriteLine($"Foo is: {foo}. In a non-verbatim string, we need to escape \" and \\" with backslashes.");
Console.WriteLine(${@"Foo is: {foo}. In a verbatim string, we need to escape "" with an extra quote, but we don't need to escape \"");
```

Выход:

Foo - 34. В строке, отличной от слов, нам нужно избежать «и \ с обратными косыми чертами.

Foo - 34. В стенографической строке нам нужно сбежать с дополнительной

цитатой, но нам не нужно бежать \

Чтобы включить фигурные скобки { или } в интерполированной строке, используйте две фигурные скобки {{ или }} :

```
$"{{foo}} is: {{foo}}"
```

Выход:

{foo}: 34

[Посмотреть демо](#)

Тип **FormattableString**

Тип выражения интерполяции строки \$"..." [не всегда](#) является простой строкой.

Компилятор решает, какой тип назначить в зависимости от контекста:

```
string s = $"hello, {name}";  
System.FormattableString s = $"Hello, {name}";  
System.IFormattable s = $"Hello, {name}";
```

Это также порядок предпочтений типа, когда компилятору необходимо выбрать, какой перегруженный метод будет вызываться.

[Новый тип](#), `System.FormattableString`, представляет собой строку составного формата вместе с аргументами, которые нужно отформатировать. Используйте это, чтобы писать приложения, которые обрабатывают аргументы интерполяции:

```
public void AddLogItem(FormattableString formattableString)  
{  
    foreach (var arg in formattableString.GetArguments())  
    {  
        // do something to interpolation argument 'arg'  
    }  
  
    // use the standard interpolation and the current culture info  
    // to get an ordinary String:  
    var formatted = formattableString.ToString();  
  
    // ...  
}
```

Вызовите вышеуказанный метод с помощью:

```
AddLogItem($"The foo is {foo}, and the bar is {bar}.");
```

Например, можно было бы отказаться от затрат на производительность форматирования

строки, если уровень ведения журнала уже собирался отфильтровать элемент журнала.

Неявные преобразования

Имеются неявные преобразования типов из интерполированной строки:

```
var s = $"Foo: {foo}";  
System.IFormattable s = $"Foo: {foo}";
```

Вы также можете создать переменную `IFormattable` которая позволяет вам преобразовать строку с инвариантным контекстом:

```
var s = $"Bar: {bar}";  
System.FormattableString s = $"Bar: {bar}";
```

Методы текущей и инвариантной культуры

Если анализ кода включен, интерполированные строки будут выдавать предупреждение [CA1305 \(Укажите `IFormatProvider`\)](#). Статический метод может использоваться для применения текущей культуры.

```
public static class Culture  
{  
    public static string Current(FormattableString formattableString)  
    {  
        return formattableString?.ToString(CultureInfo.CurrentCulture);  
    }  
    public static string Invariant(FormattableString formattableString)  
    {  
        return formattableString?.ToString(CultureInfo.InvariantCulture);  
    }  
}
```

Затем, чтобы создать правильную строку для текущей культуры, просто используйте выражение:

```
Culture.Current($"interpolated {typeof(string).Name} string.")  
Culture.Invariant($"interpolated {typeof(string).Name} string.")
```

Примечание : `Current` и `Invariant` не могут быть созданы как методы расширения, потому что по умолчанию компилятор присваивает тип `String` *интерполированному строковому выражению*, из-за которого невозможно скомпилировать следующий код:

```
$"interpolated {typeof(string).Name} string.".Current();
```

Класс `FormattableString` уже содержит метод `Invariant()`, поэтому самым простым способом

перехода на инвариантную культуру является `using static`:

```
using static System.FormattableString;

string invariant = Invariant($"Now = {DateTime.Now}");
string current = $"Now = {DateTime.Now}";
```

За кулисами

Интерполированные строки - это просто синтаксический сахар для `String.Format()`.

Компилятор ([Roslyn](#)) превратит его в `String.Format` за кулисами:

```
var text = $"Hello {name + lastName}";
```

Вышеуказанное будет преобразовано в следующее:

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

Интерполяция строк и Linq

Можно использовать интерполированные строки в операторах Linq для дальнейшего повышения удобочитаемости.

```
var fooBar = (from DataRow x in fooBarTable.Rows
    select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

Может быть переписано как:

```
var fooBar = (from DataRow x in fooBarTable.Rows
    select $"{x["foo"]}{x["bar"]}") .ToList();
```

Многоразовые интерполированные строки

С помощью `string.Format` вы можете создавать строки многократного использования:

```
public const string ErrorFormat = "Exception caught:\r\n{0}";
// ...
```

```
Logger.Log(string.Format(ErrorFormat, ex));
```

Интерполированные строки, однако, не будут компилироваться с заполнителями, ссылаясь на несуществующие переменные. Следующие не будут компилироваться:

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";  
// CS0103: The name 'error' does not exist in the current context
```

Вместо этого создайте `Func<>` который использует переменные и возвращает `String`:

```
public static Func<Exception, string> FormatError =  
    error => $"Exception caught:\r\n{error}";  
  
// ...  
  
Logger.Log(FormatError(ex));
```

Строчная интерполяция и локализация

Если вы локализуете свое приложение, вы можете задаться вопросом, можно ли использовать интерполяцию строк вместе с локализацией. Действительно, было бы неплохо иметь возможность хранить в файлах ресурсов `String` вроде:

```
"My name is {name} {middlename} {surname}"
```

вместо гораздо менее читаемого:

```
"My name is {0} {1} {2}"
```

Процесс интерполяции `String` происходит *во время компиляции*, в отличие от строки форматирования со строкой `string.Format` которая встречается *во время выполнения*. Выражения в интерполированной строке должны ссылаться на имена в текущем контексте и должны храниться в файлах ресурсов. Это означает, что если вы хотите использовать локализацию, вам нужно сделать это так:

```
var FirstName = "John";  
  
// method using different resource file "strings"  
// for French ("strings.fr.resx"), German ("strings.de.resx"),  
// and English ("strings.en.resx")  
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")  
{  
    // get localized string  
    var localizedMyNameIs = Properties.strings.Hello;  
    // insert spaces where necessary  
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");  
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
```

```

surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
// display it
Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// switch to French and greet John
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// switch to German and greet John
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);

```

Если строки ресурсов для языков, используемых выше, правильно сохраняются в отдельных файлах ресурсов, вы должны получить следующий результат:

Bonjour, mon nom est John

Hallo, mein Имя ist John

Привет меня зовут Джон

Обратите внимание, что это означает, что имя следует за локализованной строкой на каждом языке. Если это не так, вам нужно добавить заполнители в строки ресурсов и изменить вышеприведенную функцию или вам нужно запросить информацию о культуре в этой функции и предоставить оператор case switch, содержащий разные случаи. Дополнительные сведения о файлах ресурсов см. В разделе [Как использовать локализацию в C #](#).

Хорошей практикой является использование стандартного резервного языка, который большинство людей поймет, если перевод недоступен. Я предлагаю использовать английский язык по умолчанию.

Рекурсивная интерполяция

Хотя это и не очень полезно, разрешено использовать интерполированную `string` рекурсивно внутри фигурных скобок другого:

```

Console.WriteLine($"String has ${"My class is called {nameof(MyClass)}.".Length} chars:");
Console.WriteLine($"My class is called {nameof(MyClass)}.");

```

Выход:

Строка имеет 27 символов:

Мой класс называется MyClass.

Ожидание в условие и, наконец,

Можно использовать выражение `await` для применения оператора ожидания к задачам или задаче (из `TResult`) в `catch` и, `finally` блоках на С # 6.

Невозможно использовать выражение `await` в `catch` и, `finally` блоки в более ранних версиях из-за ограничений компилятора. С # 6 делает ожидание асинхронных задач намного проще, разрешая выражение `await`.

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

В С # 5 требовалось использовать `bool` или объявить `Exception` вне `try catch` для выполнения асинхронных операций. Этот метод показан в следующем примере:

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(e);
}

// Close the service, since this isn't possible in the finally
```

```
await service.CloseAsync();
```

Нулевое распространение

?. оператор и оператор ?[...] называются **нуль-условным оператором**. Его иногда называют другие имена, такие как **безопасный оператор навигации**.

Это полезно, потому что если . (член accessor) применяется к выражению, которое вычисляет значение `null`, программа будет генерировать `NullReferenceException`. Если разработчик вместо этого использует ?. (нуль-условный) оператор, выражение будет вычисляться как `null` вместо того, чтобы бросать исключение.

Заметим, что если ?. используется оператор, а выражение не равно `null` ?. и . эквивалентны.

ОСНОВЫ

```
var teacherName = classroom.GetTeacher().Name;  
// throws NullReferenceException if GetTeacher() returns null
```

[Посмотреть демо](#)

Если `classroom` не имеет учителя, `GetTeacher()` может возвращать значение `null`. Когда он равен `null` и к нему будет применено свойство `Name`, будет `NullReferenceException`.

Если мы изменим это утверждение, чтобы использовать ?. синтаксис, результат всего выражения будет `null`:

```
var teacherName = classroom.GetTeacher()?.Name;  
// teacherName is null if GetTeacher() returns null
```

[Посмотреть демо](#)

Впоследствии, если `classroom` также может быть `null`, мы могли бы также написать это утверждение как:

```
var teacherName = classroom?.GetTeacher()?.Name;  
// teacherName is null if GetTeacher() returns null OR classroom is null
```

[Посмотреть демо](#)

Это пример короткого замыкания: когда любая операция условного доступа, использующая оператор с нулевым условием, имеет значение `null`, все выражение немедленно вычисляется до нуля, без обработки остальной части цепочки.

Когда терминальный член выражения, содержащего оператор с нулевым условием, имеет тип значения, выражение оценивается как `Nullable<T>` этого типа и поэтому не может использоваться как прямая замена выражения без `..`,

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// compile time error: implicit conversion from bool? to bool not allowed

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null

bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

Использовать с Null-Coalescing Operator (??)

Вы можете комбинировать оператор с нулевым условием с [Null-coalescing Operator](#) (`??`), чтобы вернуть значение по умолчанию, если выражение разрешает `null`. Используя наш пример выше:

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

Использование с индексаторами

Оператор с нулевым условием может использоваться с [индексаторами](#):

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

В приведенном выше примере:

- Первый `..` гарантирует, что `classroom` не является `null`.
- Второй `?` гарантирует, что вся коллекция `Students` не равна `null`.
- Третий `..` после того, как индексатор гарантирует, что индексирующее устройство `[0]` не вернет `null` объект. Следует отметить, что эта операция все **равно** может вызвать `IndexOutOfRangeException`.

Использование с функциями void

Нуль-условный оператор также может использоваться с функциями `void`. Однако в этом случае оператор не будет оценивать значение `null`. Это просто предотвратит исключение `NullReferenceException`.

```
List<string> list = null;
list?.Add("hi");           // Does not evaluate to null
```

Использование с помощью вызова события

Предполагая следующее определение события:

```
private event EventArgs OnCompleted;
```

При вызове события традиционно лучше всего проверять, является ли событие `null` если нет подписчиков:

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

Поскольку введен оператор с нулевым условием, вызов можно свести к одной строке:

```
OnCompleted?.Invoke(EventArgs.Empty);
```

Ограничения

Null-условный оператор производит `rvalue`, а не `lvalue`, то есть он не может использоваться для присвоения свойств, подписки на события и т. д. Например, следующий код не будет работать:

```
// Error: The left-hand side of an assignment must be a variable, property or indexer
Process.GetProcessById(1337)?.EnableRaisingEvents = true;
// Error: The event can only appear on the left hand side of += or ==
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

Gotchas

Обратите внимание, что:

```
int? nameLength = person?.Name.Length;      // safe if 'person' is null
```

Это не то же самое, что:

```
int? nameLength = (person?.Name).Length; // avoid this
```

потому что первое соответствует:

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

и последнее соответствует:

```
int? nameLength = (person != null ? person.Name : null).Length;
```

Несмотря на тернарный оператор ?: Используется здесь для объяснения разницы между двумя случаями, эти операторы не эквивалентны. Это можно легко продемонстрировать в следующем примере:

```
void Main()
{
    var foo = new Foo();
    Console.WriteLine("Null propagation");
    Console.WriteLine(foo.Bar?.Length);

    Console.WriteLine("Ternary");
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);
}

class Foo
{
    public string Bar
    {
        get
        {
            Console.WriteLine("I was read");
            return string.Empty;
        }
    }
}
```

Какие результаты:

Нулевое распространение

Я был прочитан

0

```
троичный
Я был прочитан
Я был прочитан
0
```

[Посмотреть демо](#)

Чтобы избежать эквивалента нескольких вызовов, можно:

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

И эта разница несколько объясняет, почему оператор распространения пустоты еще [не поддерживается](#) в деревьях выражений.

Использование статического типа

`using static [Namespace.Type]` директивы позволяет импортировать статические элементы типов и значений перечисления. Методы расширения импортируются как методы расширения (всего один тип), а не в область верхнего уровня.

6,0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;

class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

Демо-скрипты

6,0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

Улучшенное разрешение перегрузки

Следующий фрагмент показывает пример передачи группы методов (в отличие от лямбда), когда ожидается делегат. Разрешение перегрузки теперь разрешит это вместо повышения неоднозначной ошибки перегрузки из-за возможности **С # 6** проверить тип возвращаемого метода.

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
        Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
        Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
    }
}
```

Результаты:

6,0

Выход

перегрузка с помощью Func <int>

[Посмотреть демо](#)

5.0

ошибка

ошибка CS0121: вызов неоднозначен между следующими методами или свойствами: «Program.Overloaded (System.ction)» и «Program.Overloaded (System.Func)»

С # 6 также может хорошо обработать следующий случай точного соответствия для лямбда-выражений, что привело бы к ошибке в **С # 5**.

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

Незначительные изменения и исправления

Круглые скобки теперь запрещены по именованным параметрам. Следующие компиляции в C # 5, но не C # 6

5.0

```
Console.WriteLine((value: 23));
```

Операнды `is` и `as` больше не могут быть группами методов. Следующие компиляции в C # 5, но не C # 6

5.0

```
var result = "".Any is byte;
```

Собственный компилятор допустил это (хотя он и сделал предупреждение), и на самом деле даже не проверял совместимость метода расширения, позволяя сумасшедшие вещи, такие как `1.Any is string` ИЛИ `IDisposable.Dispose is object`.

См. [Эту ссылку](#) для получения обновлений об изменениях.

Использование метода расширения для инициализации коллекции

Синтаксис инициализации синтаксиса можно использовать при создании экземпляра любого класса, который реализует `IEnumerable` и имеет метод `Add` который принимает один параметр.

В предыдущих версиях этот метод `Add` должен был быть методом **экземпляра** для инициализированного класса. В C # 6 он также может быть методом расширения.

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }
}
```

```

public IEnumarator GetEnumerator()
{
    // Some implementation here
}

}

public class CollectionWithoutAdd : IEnumarable
{
    public IEnumarator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1,2,3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4,5,6}; // Valid only since C# 6
    }
}

```

Это приведет к выводу:

```

Элемент добавлен с помощью метода добавления экземпляра: 1
Элемент добавлен с помощью метода добавления экземпляра: 2
Элемент добавлен с помощью метода добавления экземпляра: 3
Добавлен элемент с добавлением метода добавления: 4
Добавлен элемент с добавлением метода добавления: 5
Добавлен элемент с добавлением метода добавления: 6

```

Отключить улучшения предупреждений

В C # 5.0 и ранее разработчик мог только подавлять предупреждения по номеру. С введением Roslyn Analyzers C # нужен способ отключения предупреждений, выпущенных из конкретных библиотек. С C # 6.0 директива pragma может подавлять предупреждения по имени.

До:

```
#pragma warning disable 0501
```

C # 6.0:

```
#pragma warning disable CS0501
```

Прочтайте Возможности C # 6.0 онлайн: <https://riptutorial.com/ru/csharp/topic/24/возможности-c-sharp-6-0>

глава 44: Возможности С # 7.0

Вступление

С # 7.0 - седьмая версия С #. Эта версия содержит некоторые новые функции: поддержка языков для Tuples, локальных функций, `out var`, разделителей цифр, двоичных литералов, сопоставления шаблонов, выражений `throw`, `ref return` и `ref local` и `extended expression bodied members list`.

Официальная ссылка: [Что нового в С # 7](#)

Examples

декларация var

Обычным шаблоном в С # является использование `bool TryParse(object input, out object value)` для безопасного анализа объектов.

Объявление `out var` является простой функцией улучшения удобочитаемости. Он позволяет объявлять переменную в то же самое время, которое передается как параметр `out`.

Переменная, объявленная таким образом, привязана к остальной части тела в точке, в которой она объявлена.

пример

Используя `TryParse` до С # 7.0, вы должны объявить переменную для получения значения перед вызовом функции:

7,0

```
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok
```

В С # 7.0 вы можете вставить объявление переменной, переданной в параметр `out`,

исключая необходимость в отдельном объявлении переменной:

7,0

```
if (int.TryParse(input, out var value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // still ok, the value is in scope within the remainder of the body
```

Если некоторые из параметров, возвращаемых функцией `out` не нужны, вы можете использовать оператор `discard_`.

```
p.GetCoordinates(out var x, out _); // I only care about x
```

`out var` объявление может быть использовано с любой существующей функцией, которая уже имеет `out` параметров. Синтаксис объявления функции остается тем же, и никаких дополнительных требований не требуется для обеспечения совместимости функции с объявлением `out var`. Эта особенность - просто синтаксический сахар.

Другая особенность объявления `out var` заключается в том, что ее можно использовать с анонимными типами.

7,0

```
var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
{
    Source = x,
    Mod2 = x % 2
})
.GroupBy(x => x.Mod2)
.ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}
```

В этом коде мы создаем `Dictionary` с ключом `int` и массив анонимного значения типа. В предыдущей версии C# было невозможно использовать метод `TryGetValue` потому что вам нужно было объявить переменную `out` (которая имеет анонимный тип!). Однако, `out var` нам не нужно явно указывать тип переменной `out`.

Ограничения

Обратите внимание, что объявления `var` ограничены в запросах LINQ, поскольку выражения интерпретируются как выражения лямбда-тел, поэтому объем вводимых переменных ограничен этими лямбдами. Например, следующий код не будет работать:

```
var nums =
    from item in seq
    let success = int.TryParse(item, out var tmp)
    select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context
```

Рекомендации

- Предложение публикации оригинального предложения `var` на GitHub

Бинарные литералы

Предфикс **0b** может использоваться для представления бинарных литералов.

Бинарные литералы позволяют строить числа от нулей и единиц, что делает видение того, какие биты устанавливаются в двоичном представлении числа намного проще. Это может быть полезно для работы с бинарными флагами.

Ниже приведены эквивалентные способы указания `int` со значением $34 (= 2^5 + 2^1)$:

```
// Using a binary literal:
// bits: 76543210
int a1 = 0b00100010;           // binary: explicitly specify bits

// Existing methods:
int a2 = 0x22;                // hexadecimal: every digit corresponds to 4 bits
int a3 = 34;                   // decimal: hard to visualise which bits are set
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

Перечисления флагов

Раньше указание значений флага для `enum` могло быть выполнено только с использованием одного из трех методов в этом примере:

```
[Flags]
public enum DaysOfWeek
{
    // Previously available methods:
    //      decimal      hex      bit shifting
    Monday     = 1,    //      = 0x01      = 1 << 0
    Tuesday    = 2,    //      = 0x02      = 1 << 1
    Wednesday  = 4,    //      = 0x04      = 1 << 2
    Thursday   = 8,    //      = 0x08      = 1 << 3
    Friday     = 16,   //      = 0x10      = 1 << 4
    Saturday   = 32,   //      = 0x20      = 1 << 5
```

```

Sunday      = 64,      //      = 0x40      = 1 << 6

Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
Weekends = Saturday | Sunday
}

```

С бинарными литералами более очевидно, какие биты установлены, и использование их не требует понимания шестнадцатеричных чисел и побитовой арифметики:

```

[Flags]
public enum DaysOfWeek
{
    Monday      = 0b00000001,
    Tuesday     = 0b00000010,
    Wednesday   = 0b00000100,
    Thursday    = 0b00001000,
    Friday      = 0b00010000,
    Saturday    = 0b00100000,
    Sunday      = 0b01000000,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}

```

Цифровые разделители

Подчеркивание _ может использоваться как разделитель цифр. Возможность группировать цифры в больших числовых литералах оказывает значительное влияние на читаемость.

Подчеркивание может происходить в любом месте в числовом литерале, за исключением случаев, указанных ниже. Различные группировки могут иметь смысл в разных сценариях или с разными числовыми базами.

Любая последовательность цифр может быть разделена одним или несколькими символами подчеркивания _. Допускается как десятичными знаками, так и экспонентами. Сепараторы не имеют семантического воздействия - их просто игнорируют.

```

int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2_3_4_5_6_7_8_9;
double real = 1_000.111_1e-1_000;

```

Где _ Разделитель цифр не может быть использован:

- в начале значения (_121)
- в конце значения (121_ или 121.05_)
- рядом с десятичной (10_.0)
- рядом с символом экспоненты (1.1e_1)

- рядом с спецификатором типа (`10_f`)
- сразу после `0x` или `0b` в двоичных и шестнадцатеричных литералах (`может быть изменено, чтобы разрешить, например, 0b_1001_1000`)

Языковая поддержка для кортежей

ОСНОВЫ

Кортеж представляет собой упорядоченный конечный список элементов. Кортежи обычно используются в программировании как средство совместной работы с одним отдельным объектом вместо индивидуальной работы с каждым из элементов кортежа и для представления отдельных строк (т. Е. «Записей») в реляционной базе данных.

В С # 7.0 методы могут иметь несколько возвращаемых значений. За кулисами компилятор будет использовать новую структуру `ValueTuple`.

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

Замечание : для этого в Visual Studio 2017 вам необходимо получить пакет `System.ValueTuple`.

Если результат метода возврата кортежа присваивается одной переменной, вы можете получить доступ к членам по их определенным именам в сигнатуре метода:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

Деконструкция кортежа

Деконструкция кортежа разделяет кортеж на его части.

Например, `GetTallies` и присвоение возвращаемого значения двум отдельным переменным деконструирует кортеж для этих двух переменных:

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` также работает:

```
(var s, var c) = GetTallies();
```

Вы также можете использовать более короткий синтаксис, с `var` вне `()`:

```
var (s, c) = GetTallies();
```

Вы также можете деконструировать существующие переменные:

```
int s, c;
(s, c) = GetTallies();
```

Обмен теперь намного проще (без переменной `temp`):

```
(b, a) = (a, b);
```

Интересно, что любой объект может быть деконструирован путем определения метода `Deconstruct` в классе:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

В этом случае синтаксис `(localFirstName, localLastName) = person` вызывает `(localFirstName, localLastName) = person Deconstruct` на `person`.

Деконструкцию можно даже определить в методе расширения. Это эквивалентно приведенному выше:

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

Альтернативным подходом для класса `Person` является определение самого `Name` как `Tuple`.

Рассмотрим следующее:

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

Тогда вы можете создать экземпляр человека таким образом (где мы можем взять кортеж в качестве аргумента):

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last; // "Smith"
```

Инициализация кортежа

Вы также можете произвольно создавать кортежи в коде:

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

При создании кортежа вы можете назначать имена объявлений ad-hoc членам кортежа:

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

Вывод типа

Множественные кортежи, определенные с одной и той же сигнатурой (совпадающие типы и количество), будут выведены как соответствующие типы. Например:

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
```

```
        return stats;
    }
```

`stats` может быть возвращена, так как объявление переменной `stats` и ответная подпись метода совпадают.

Имена полей отражения и кортежа

Имена членов не существуют во время выполнения. Отражение будет рассматривать кортежи с одинаковым числом и типами членов одинаково, даже если имена участников не совпадают. Преобразование кортежа в `object` а затем в кортеж с теми же типами членов, но с разными именами, также не вызывает исключений.

Хотя сам класс `ValueTuple` не сохраняет информацию для имен членов, информация доступна через отражение в `TupleElementNamesAttribute`. Этот атрибут не применяется к самому кортежу, а к параметрам метода, возвращаемым значениям, свойствам и полям. Это позволяет сохранять имена элементов кортежа в сборках, т. Е. Если метод возвращает (имя строки, `int count`), имя и количество имен будут доступны вызывающим лицам метода в другой сборке, потому что возвращаемое значение будет отмечено с помощью `TupleElementNameAttribute`, содержащим значения «имя» и «счет».

Использование с дженерикиами и `async`

Новые функции кортежа (с использованием базового типа `ValueTuple`) полностью поддерживают дженерики и могут использоваться как общий тип параметра. Это позволяет использовать их с шаблоном `async / await`:

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackoverflow.GetStringAsync();
    int num = await _stackoverflow.GetIntAsync();

    return (fooBar, num);
}
```

Использование с коллекциями

Может оказаться полезным иметь набор кортежей в (например) сценарий, в котором вы пытаетесь найти соответствующий кортеж на основе условий, чтобы избежать разветвления кода.

Пример:

```

private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string, string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}

```

С новыми кортежами могут стать:

```

private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)> labels = new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}

```

Хотя именование на примере кортежа выше является довольно общим, идея соответствующих ярлыков позволяет глубже понять, что делается в коде, ссылаясь на «item1», «item2» и «item3».

Различия между ValueTuple и Tuple

Основной причиной внедрения ValueTuple является производительность.

Название типа	ValueTuple	Tuple
Класс или структура	struct	class

Название типа	ValueTuple	Tuple
Мутабильность (изменение значений после создания)	изменчивый	неизменный
Имена членов и поддержка других языков	да	нет (TBD)

Рекомендации

- [Предложение оригинального набора языков для GitHub](#)
- [Реализованное решение VS 15 для функций C # 7.0](#)
- [Пакет NuGet Tuple](#)

Локальные функции

Локальные функции определяются внутри метода и недоступны вне его. Они имеют доступ ко всем локальным переменным и поддерживают итераторы, синтаксис `async / await` и лямбда. Таким образом, повторения, специфичные для функции, могут функционировать без перенаселения класса. Как побочный эффект, это улучшает производительность предложения intellisense.

пример

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

Локальные функции значительно упрощают код для операторов LINQ, где вам обычно приходится отделять проверки аргументов от фактической логики, чтобы мгновенно проверять аргументы, а не откладываться до начала итерации.

пример

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
```

```

{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}

```

Локальные функции также поддерживают ключевые слова `async` и `await`.

пример

```

async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?! [a-z0-9_.+-]+@[a-z0-9-]+\.[a-z0-9-\.]+)");
    IEnumerable<string> emails1 = await getEmailsFromFileAsync("input1.txt");
    IEnumerable<string> emails2 = await getEmailsFromFileAsync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");

    async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
    {
        string text;

        using (StreamReader reader = File.OpenText(fileName))
        {
            text = await reader.ReadToEndAsync();
        }

        return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
    }

    async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
    {
        using (StreamWriter writer = File.CreateText(fileName))
        {
            foreach (string line in lines)
            {
                await writer.WriteLineAsync(line);
            }
        }
    }
}

```

Одна важная вещь, которую вы, возможно, заметили, заключается в том, что локальные функции могут быть определены в операторе `return`, их **не** нужно определять над ним. Кроме того, локальные функции обычно следуют принципу именования «`lowerCamelCase`», чтобы более легко отличить себя от функций класса.

Соответствие шаблону

Расширения соответствия шаблонов для C # позволяют использовать многие из преимуществ сопоставления шаблонов с функциональных языков, но таким образом, что плавно интегрируется с ощущением основного языка

выражение `switch`

Поиск по шаблону расширяет `switch` заявление на включение типов:

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}

class Square : Geometry
{
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

is выражением

Соответствие шаблону расширяет оператор `is` для проверки типа и объявления новой переменной одновременно.

пример

7,0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

можно переписать как:

7,0

```
if(o is string s)
{
    //Do something with s
};
```

Также обратите внимание, что область действия переменной шаблона `s` распространяется за пределы блока `if` достигающего конца охватывающей области, например:

```
if(someCondition)
{
    if(o is string s)
    {
        //Do something with s
    }
    else
    {
        // s is unassigned here, but accessible
    }

    // s is unassigned here, but accessible
}
// s is not accessible here
```

ref возвратить и указать местный

`Ref` возвращает и `ref locals` полезны для манипулирования и возврата ссылок на блоки памяти вместо копирования памяти, не прибегая к небезопасным указателям.

Вернуться

```
public static ref TValue Choose<TValue>(
```

```
Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

При этом вы можете передать два значения по ссылке, при этом один из них будет возвращен на основе некоторого условия:

```
Matrix3D left = ..., right = ...;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

Ссылка

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}
...
int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

Небезопасные операции ref

В `System.Runtime.CompilerServices.Unsafe` определен набор небезопасных операций, которые позволяют вам манипулировать значениями `ref` как если бы они были указателями, в основном.

Например, переинтерпретировать адрес памяти (`ref`) как другой тип:

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

Остерегайтесь **энтианности** при этом, например, проверьте `BitConverter.IsLittleEndian` если необходимо, и обработайте соответствующим образом.

Или перебирайте массив небезопасным образом:

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };
```

```
ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

Или аналогичный Subtract :

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

Кроме того, можно проверить, совпадают ли два значения ref, т.е. тот же адрес:

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

СВЯЗИ

[Рослин Гитуб](#)

[System.Runtime.CompilerServices.Unsafe on github](#)

ВЫЗЫВАТЬ ВЫРАЖЕНИЯ

С # 7.0 позволяет метать как выражение в определенных местах:

```
class Person
{
    public string Name { get; }

    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No
name!");
    }
}
```

```
    public string GetLastName() => throw new NotImplementedException();
}
```

До C # 7.0, если вы хотите выбросить исключение из тела выражения, вам необходимо:

```
var spoons = "dinner,desert,soup".Split(',');
var spoonsArray = spoons.Length > 0 ? spoons : null;
if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}
```

Или же

```
var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
    {
        throw new Exception("There are no spoons");
    })();
```

В C # 7.0 приведенное выше упрощено:

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

Список расширенных выражений

C # 7.0 добавляет аксессоры, конструкторы и финализаторы в список вещей, которые могут иметь тела выражений:

```
class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int, string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _); // finalizers

    public string Name
    {
        get => names[id]; // getters
        set => names[id] = value; // setters
    }
}
```

Также см. Раздел [объявления var var](#) для оператора discard.

ValueTask

`Task<T>` - это **класс** и вызывает ненужные накладные расходы на его распределение, когда результат сразу доступен.

`ValueTask<T>` является **структурой** и был введен для предотвращения выделения объекта `Task` в случае, если результат операции `async` уже доступен во время ожидания.

Итак, `ValueTask<T>` предоставляет два преимущества:

1. Увеличение производительности

Вот пример `Task<T>` :

- Требуется распределение кучи
- Принимает 120 нс с JIT

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

Вот `ValueTask<T>` аналогового значения `ValueTask<T>` :

- Нет распределение кучи, если результат известен синхронно (что не в этом случае из-за `Task.Delay`, но часто не во многих реальных `async / await` сценариев)
- Принимает 65нс с JIT

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

2. Повышенная гибкость внедрения

Реализация асинхронного интерфейса, желающего быть синхронным, в противном случае была бы вынуждена использовать либо `Task.Run` либо `Task.FromResult` (что привело к `Task.FromResult` производительности, описанному выше). Таким образом, существует некоторое давление на синхронные реализации.

Но с `ValueTask<T>` реализации более свободны выбирать между синхронными или асинхронными, не затрагивая вызывающих.

Например, вот интерфейс с асинхронным методом:

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

... и вот как можно вызвать этот метод:

```
IFoo<T> thing = getThing();
var x = await thing.BarAsync();
```

С `ValueTask` приведенный выше код будет работать **либо с синхронными, либо с асинхронными реализациями**:

Синхронная реализация:

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

Асинхронная реализация

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

Заметки

Несмотря на то, что в **C # 7.0** планировалось добавить конструкцию `ValueTask`, она пока сохраняется как еще одна библиотека. **Пакет `ValueTask <T>` System.Threading.Tasks.Extensions** можно загрузить из галереи **Nuget**

Прочтайте Возможности C # 7.0 онлайн: <https://riptutorial.com/ru/csharp/topic/1936/возможности-c-sharp-7-0>

глава 45: Встроенные типы

Examples

Неизменяемый ссылочный тип - строка

```
// assign string from a string literal
string s = "hello";

// assign string from an array of characters
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };
string s = new string(chars, 0, chars.Length);

// assign string from a char pointer, derived from a string
string s;
unsafe
{
    fixed (char* charPointer = "hello")
    {
        s = new string(charPointer);
    }
}
```

Тип значения - char

```
// single character s
char c = 's';

// character s: casted from integer value
char c = (char)115;

// unicode character: single character s
char c = '\u0073';

// unicode character: smiley face
char c = '\u263a';
```

Тип значения - короткий, int, long (16-битные, 32-битные, 64-битные целые числа)

```
// assigning a signed short to its minimum value
short s = -32768;

// assigning a signed short to its maximum value
short s = 32767;

// assigning a signed int to its minimum value
int i = -2147483648;

// assigning a signed int to its maximum value
int i = 2147483647;
```

```
// assigning a signed long to its minimum value (note the long postfix)
long l = -9223372036854775808L;

// assigning a signed long to its maximum value (note the long postfix)
long l = 9223372036854775807L;
```

Также возможно сделать эти типы обнуляемыми, что означает, что дополнительно к обычным значениям также может быть присвоен нуль. Если переменная с нулевым типом не инициализируется, она будет равна нулю, а не 0. Нулевые типы помечены добавлением вопросительного знака (?) После типа.

```
int a; //This is now 0.
int? b; //This is now null.
```

Тип значения - ushort, uint, ulong (неподписанные 16 бит, 32 бит, 64 битные целые числа)

```
// assigning an unsigned short to its minimum value
ushort s = 0;

// assigning an unsigned short to its maximum value
ushort s = 65535;

// assigning an unsigned int to its minimum value
uint i = 0;

// assigning an unsigned int to its maximum value
uint i = 4294967295;

// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

Также возможно сделать эти типы обнуляемыми, что означает, что дополнительно к обычным значениям также может быть присвоен нуль. Если переменная с нулевым типом не инициализируется, она будет равна нулю, а не 0. Нулевые типы помечены добавлением вопросительного знака (?) После типа.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

Тип значения - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
```

```
        Console.WriteLine("Boolean has true value");
    }
```

Ключевое слово `bool` является псевдонимом `System.Boolean`. Он используется для объявления переменных для хранения логических значений, `true` и `false`.

Сравнение со значениями в штучной упаковке

Если типы значений присваиваются переменным типа `object` они в [штучной упаковке](#) - значение сохраняется в экземпляре `System.Object`. Это может привести к непреднамеренным последствиям при сравнении значений с `==`, например:

```
object left = (int)1; // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right; // false
```

Этого можно избежать, используя перегруженный метод `Equals`, который даст ожидаемый результат.

```
var comparison2 = left.Equals(right); // true
```

В качестве альтернативы, то же самое можно сделать, распакуя `left` и `right` переменные, чтобы сравнить значения `int`:

```
var comparison3 = (int)left == (int)right; // true
```

Преобразование типов значений в штучной упаковке

Типы в [штучной упаковке](#) могут быть только распакованы в исходный `Type`, даже если выполняется преобразование двух `Type`’s, например:

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

Этого можно избежать при первом распаковке исходного `Type`, например:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Прочтайте Встроенные типы онлайн: <https://riptutorial.com/ru/csharp/topic/42/встроенные-типы>

глава 46: Выполнение HTTP-запросов

Examples

Создание и отправка запроса HTTP POST

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

Создание и отправка запроса HTTP GET

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
```

```

string responseBodyFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}

```

Обработка ошибок определенных кодов ответа HTTP (таких как 404 Not Found)

```

using System.Net;
...
string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as
        WebExceptionStatus.ConnectFailure.
        // ...
    }
}

```

Отправка асинхронного запроса HTTP POST с корпусом JSON

```

public static async Task PostAsync(this Uri uri, object value)
{
    var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());
    using (var client = new HttpClient())
    {

```

```

        return await client.PostAsync(uri, content);
    }
}

. . .

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });

```

Отправка асинхронного запроса HTTP GET и чтение запроса JSON

```

public static async Task<TResult> GetAnsync<TResult>(this Uri uri)
{
    using (var client = new HttpClient())
    {
        var message = await client.GetAsync(uri);

        if (!message.IsSuccessStatusCode)
            throw new Exception();

        return message.ReadAsAsync<TResult>();
    }
}

. . .

public class Result
{
    public double foo { get; set; }

    public string bar { get; set; }
}

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
var result = await uri.GetAnsync<Result>();

```

Получить HTML для веб-страницы (простой)

```

string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);

```

Прочтайте Выполнение HTTP-запросов онлайн: <https://riptutorial.com/ru/csharp/topic/1971/ выполнение-http-запросов>

глава 47: Генерация кода Т4

Синтаксис

- **Синтаксис Т4**

- `<#@ ... #>` // Объявление свойств, включая шаблоны, сборки и пространства имен, а также язык, используемый шаблоном
- `Plain Text` // Объявление текста, который может быть закодирован для файлов, сгенерированных
- `<#= ... #>` // Объявление скриптов
- `<#+ ... #>` // Объявление сценариев
- `<#... #>` // Объявление текстовых блоков

Examples

Генерация кода времени выполнения

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

Прочтайте Генерация кода Т4 онлайн: <https://riptutorial.com/ru/csharp/topic/4824/генерация-кода-т4>

глава 48: Генерация случайных чисел в C

Синтаксис

- Random ()
- Случайный (int Seed)
- int Next ()
- int Next (int maxValue)
- int Next (int minValue, int maxValue)

параметры

параметры	подробности
семя	Значение для генерации случайных чисел. Если не установлено, значение по умолчанию определяется текущим системным временем.
MinValue	Сгенерированные числа не будут меньше этого значения. Если значение не установлено, значение по умолчанию равно 0.
.MaxValue	Сгенерированные числа будут меньше этого значения. Если не установлено, значением по умолчанию является <code>Int32.MaxValue</code> .
возвращаемое значение	Возвращает число со случайным значением.

замечания

Случайное семя, генерируемое системой, не одинаково в каждом отдельном прогоне.

Семена, сгенерированные в одно и то же время, могут быть одинаковыми.

Examples

Создать случайный int

В этом примере генерируются случайные значения от 0 до 2147483647.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

Создать случайный двойной

Создайте случайное число от 0 до 1.0. (не включая 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

Генерировать случайный int в заданном диапазоне

Создайте случайное число между minValue и maxValue - 1.

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

Генерирование одной и той же последовательности случайных чисел снова и снова

При создании Random экземпляров с одним и тем же семенем будут генерироваться одинаковые числа.

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
    Console.WriteLine("Random instance " + i);
    Random rnd = new Random(seed);
    for (int j = 0; j < 5; j++)
    {
        Console.Write(rnd.Next());
        Console.Write(" ");
    }

    Console.WriteLine();
}
```

Выход:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

Создавать несколько случайных классов с разными семенами одновременно

Два случайных класса, созданные одновременно, будут иметь одинаковое начальное

значение.

Использование `System.Guid.NewGuid().GetHashCode()` может получить другое семя даже в одно и то же время.

```
Random rnd1 = new Random();
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());

Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());
```

Другим способом достижения разных семян является использование другого экземпляра `Random` для извлечения значений семян.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

Это также позволяет контролировать результат всех `Random` экземпляров, устанавливая только начальное значение для `rndSeeds`. Все остальные экземпляры будут детерминистически получены из этого единственного значения семени.

Создать случайный символ

Создайте случайную букву между `a` и `z`, используя перегрузку `Next()` для заданного диапазона чисел, а затем преобразование полученного `int` в `char`

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a', 'z');
// 'a' and 'z' are interpreted as ints for parameters for Next()
```

Создайте число, которое представляет собой процент от максимального значения

Общей потребностью в случайных числах является генерация числа, которое составляет $x\%$ от некоторого максимального значения. Это можно сделать, обработав результат `NextDouble()` в процентах:

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

Прочтайте Генерация случайных чисел в C # онлайн:

<https://riptutorial.com/ru/csharp/topic/1975/генерация-случайных-чисел-в-c-sharp>

глава 49: Двоичная сериализация

замечания

Двоичный механизм сериализации является частью платформы .NET, но приведенные здесь примеры относятся к C#. По сравнению с другими механизмами сериализации, встроенными в платформу .NET, двоичный сериализатор работает быстро и эффективно и обычно требует очень мало дополнительного кода, чтобы заставить его работать. Однако он также менее терпим к изменениям кода; то есть, если вы сериализуете объект и затем слегка изменяете определение объекта, оно, скорее всего, не будет десериализоваться правильно.

Examples

Создание объекта сериализации

Добавьте атрибут `[Serializable]` чтобы пометить весь объект для двоичной сериализации:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;

    [OptionalField]
    public string Name;
}
```

Все члены будут сериализованы, если мы явно не `[NonSerialized]` атрибута `[NonSerialized]`. В нашем примере `X`, `Y`, `Z` и `Name` все сериализованы.

Все члены должны присутствовать при десериализации, если не отмечены `[NonSerialized]` или `[OptionalField]`. В нашем примере `X`, `Y` и `Z` являются обязательными, и десериализация не выполняется, если они не присутствуют в потоке. `DontSerializeThis` всегда будет установлен по `default(decimal)` (который равен 0). Если `Name` присутствует в потоке, то он будет установлен в это значение, в противном случае он будет установлен по `default(string)` (который является нулевым). Цель `[OptionalField]` - предоставить немного допуск к версии.

Управление сериализацией с атрибутами

Если вы используете `[NonSerialized]`, этот член всегда будет иметь свое значение по умолчанию после десериализации (например, для `int`, `null` для `string`, `false` для `bool` и т. д.). Независимо от инициализации, сделанной в самом объекте (конструкторы, декларации и т. д.). Для компенсации атрибутов `[OnDeserializing]` (называемых только ПЕРЕД десериализацией) и `[OnDeserialized]` (называемых просто после десериализации) вместе со своими аналогами предусмотрены `[OnSerializing]` и `[OnSerialized]`.

Предположим, мы хотим добавить «Rating» к нашему `Vector`, и мы хотим убедиться, что значение всегда начинается с 1. Так, как это написано ниже, после десериализации будет 0:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

Чтобы устранить эту проблему, мы можем просто добавить следующий метод внутри класса, чтобы установить его в 1:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

Или, если мы хотим установить его на вычисленное значение, мы можем дождаться завершения десериализации и затем установить его:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

Точно так же мы можем контролировать, как вещи выписываются с помощью `[OnSerializing]` и `[OnSerialized]`.

Добавление большего контроля за счет внедрения ISerializable

Это обеспечит больший контроль над сериализацией, как сохранить и загрузить типы

Внедрить интерфейс ISerializable и создать пустой конструктор для компиляции

```
[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("_name", _name, typeof(string));
    }
}
```

Для сериализации данных вы можете указать желаемое имя и нужный тип

```
info.AddValue("_name", _name, typeof(string));
```

Когда данные десериализованы, вы сможете прочитать нужный тип

```
_name = (string)info.GetValue("_name", typeof(string));
```

Сериализация суррогатов (внедрение ISerializationSurrogate)

Реализует сериализационный селектор, который позволяет одному объекту выполнять сериализацию и десериализацию другого объекта

Также позволяет правильно сериализовать или десериализовать класс, который не является сериализуемым

Внедрить интерфейс ISerializationSurrogate

```
public class ItemSurrogate : ISerializationSurrogate
```

```

{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}

```

Затем вы должны сообщить своему знатоку IFormatter о суррогатах, указав и инициализируя SurrogateSelector и назначив его вашему IFormatter

```

var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All),
new ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};

```

Даже если класс не помечен сериализуемым.

```

//this class is not serializable
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

```

Полное решение

```

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
        }
    }
}

```

```

        set { _name = value; }
    }

}

class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext
context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext
context, ISurrogateSelector selector)
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new Item
        {
            Name = "Orange"
        };

        var bytes = SerializeData(item);
        var deserializedData = (Item)DeserializeData(bytes);
    }

    private static byte[] SerializeData(object obj)
    {
        var surrogateSelector = new SurrogateSelector();
        surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

        var binaryFormatter = new BinaryFormatter
        {
            SurrogateSelector = surrogateSelector
        };

        using (var memoryStream = new MemoryStream())
        {
            binaryFormatter.Serialize(memoryStream, obj);
            return memoryStream.ToArray();
        }
    }

    private static object DeserializeData(byte[] bytes)
    {
        var surrogateSelector = new SurrogateSelector();
        surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

        var binaryFormatter = new BinaryFormatter
        {
    
```

```
        SurrogateSelector = surrogateSelector
    };
}

using (var memoryStream = new MemoryStream(bytes))
    return binaryFormatter.Deserialize(memoryStream);
}
```

Связывание Serialization

Связующее дает вам возможность проверить, какие типы загружаются в ваш домен приложения.

Создайте класс, унаследованный от `SerializationBinder`

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Теперь мы можем проверить, какие типы загружаются, и на этой основе решить, что мы действительно хотим получить

Для использования связующего необходимо добавить его в `BinaryFormatter`.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

Полное решение

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            else
                return null;
        }
    }
}
```

```

        return typeof(Item);
    return null;
}

}

[Serializable]
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new Item
        {
            Name = "Orange"
        };

        var bytes = SerializeData(item);
        var deserializedData = (Item)DeserializeData(bytes);
    }
}

private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter
    {
        Binder = new MyBinder()
    };

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
}

```

Некоторые исправления в обратной совместимости

Этот небольшой пример показывает, как вы можете потерять обратную совместимость в своих программах, если заранее не позаботитесь об этом. И способы получения большего контроля над процессом сериализации

Сначала мы напишем пример первой версии программы:

Версия 1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

А теперь предположим, что во второй версии программы добавлен новый класс. И нам нужно сохранить его в массиве.

Теперь код будет выглядеть так:

Версия 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

```
}
```

И код для сериализации и десериализации

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

Итак, что произойдет, когда вы сериализуете данные в программе v2 и попытаетесь десериализовать их в программе v1?

Вы получаете исключение:

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem
in the Formatter.Source=mscorlib
StackTrace:
   at System.Runtime.Serialization.ObjectManager.DoFixups()
   at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler
handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
   at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
   at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
   at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

Зачем?

ObjectManager имеет другую логику для разрешения зависимостей для массивов, для ссылочных и значений типов. Мы добавили массив нового ссылочного типа, отсутствующего в нашей сборке.

Когда ObjectManager пытается разрешить зависимости, он строит график. Когда он видит массив, он не может исправить его немедленно, так что он создает фиктивную ссылку, а

затем исправляет массив позже.

И поскольку этот тип не находится в сборке, и зависимости не могут быть исправлены. По какой-то причине он не удаляет массив из списка элементов для исправлений и в конце, он выдает исключение «`IncorrectNumberOfFixups`».

Это некоторые «`gotchas`» в процессе сериализации. По какой-то причине он работает некорректно только для массивов новых ссылочных типов.

A Note:

Similar code will work correctly if you do not use arrays with new classes

И первый способ исправить это и поддерживать совместимость?

- Используйте коллекцию новых структур, а не классы или используйте словарь (возможные классы), потому что словарь представляет собой набор ключей`valuepair` (его структура)
- Используйте `ISerializable`, если вы не можете изменить старый код

Прочтайте Двоичная сериализация онлайн: <https://riptutorial.com/ru/csharp/topic/4120/двоичная-сериализация>

глава 50: Делегаты

замечания

Резюме

Тип **делегата** - это тип, представляющий конкретную подпись метода. Экземпляр этого типа относится к определенному методу с соответствующей сигнатурой. Параметры метода могут иметь типы делегирования, и поэтому этому одному методу передается ссылка на другой метод, который затем может быть вызван

Встроенные типы делегатов: `Action<...>`, `Predicate<T>` И `Func<..., TResult>`

System имена содержат `Action<...>`, `Predicate<T>` И `Func<..., TResult>` делегатов, где "..." составляет от 0 до 16 параметров универсального типа (для 0 параметров, `Action` являются не- родовое).

`Func` представляет методы с типом возвращаемого типа `TResult`, а `Action` представляет методы без возвращаемого значения (void). В обоих случаях дополнительные параметры типового типа соответствуют параметрам метода.

`Predicate` представляет метод с булевым типом возврата, `T` - входной параметр.

Пользовательские типы делегатов

Именованные типы делегатов могут быть объявлены с использованием ключевого слова `delegate`.

Вызов делегатов

Делегаты могут быть вызваны с использованием того же синтаксиса, что и методы: имя экземпляра делегата, за которым следуют скобки, содержащие любые параметры.

Присвоение делегатам

Делегатам можно назначить следующие способы:

- Назначение именованного метода
- Назначение анонимного метода с использованием лямбда
- Назначение именованного метода с использованием ключевого слова `delegate`.

Объединение делегатов

Несколько экземпляров делегата могут быть назначены одному экземпляру делегата с помощью оператора `+`. Оператор `-` может использоваться для удаления делегата компонента из другого делегата.

Examples

Базовые ссылки делегатов именованного метода

При назначении именованных методов делегатам они будут ссылаться на один и тот же базовый объект, если:

- Они являются одним и тем же методом экземпляра, в том же экземпляре класса
- Они представляют собой один и тот же статический метод для класса

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

Объявление типа делегата

Следующий синтаксис создает тип `delegate` с именем `NumberInOutDelegate`, представляющий метод, который принимает `int` и возвращает `int`.

```
public delegate int NumberInOutDelegate(int input);
```

Это можно использовать следующим образом:

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

example Экземпляра делегата выполняется так же, как метод `Square`. Экземпляр делегата буквально выступает в качестве делегата для вызывающего: вызывающий вызывает делегата, а затем делегат вызывает целевой метод. Это направление отделяет вызывающего абонента от целевого метода.

Вы можете объявить **общий** тип делегата, и в этом случае вы можете указать, что тип является ковариантным (`out`) или контравариантным (`in`) в некоторых аргументах типа. Например:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Как и другие общие типы, общие типы делегатов могут иметь ограничения, например, `where TFrom : struct, IConvertible where TTo : new()`.

Избегайте совместной и контравариантности для типов делегатов, которые

предназначены для использования для делегатов многоадресной передачи, таких как типы обработчиков событий. Это связано с тем, что конкатенация (+) может завершиться неудачей, если тип выполнения отличается от типа времени компиляции из-за дисперсии. Например, избегайте:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

Вместо этого используйте инвариантный общий тип:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Также поддерживаются делегаты, где некоторые параметры изменяются с помощью `ref` или `out`, как в:

```
public delegate bool TryParser<T>(string input, out T result);
```

(пример использования `TryParser<decimal> example = decimal.TryParse;`) или делегатов, где последний параметр имеет модификатор `params`. Типы делегатов могут иметь необязательные параметры (значения по умолчанию для поставки). Типы делегатов могут использовать типы указателей, такие как `int*` или `char*` в своих подписях или типах возврата (используйте ключевое слово `unsafe`). Тип делегата и его параметры могут нести пользовательские атрибуты.

Func , Действие и предикат типы делегатов

Пространство имен `System` содержит `Func<..., TResult>` делегировать типы с 0 до 15 общих параметров, возвращая тип `TResult`.

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but
    the first as parameters of that type
    Console.WriteLine(output);
}
```

Пространство имен `System` также содержит типы делегатов `Action<...>` с различным количеством общих параметров (от 0 до 16). Он похож на `Func<T1, ..., Tn>`, но он всегда возвращает `void`.

```
private void UseAction(Action action)
{
```

```

        action(); // The non-generic Action has no parameters
    }

    private void UseAction(Action<int, string> action)
    {
        action(4, "two"); // The generic action is invoked with parameters matching its type
        arguments
    }

```

`Predicate<T>` также является формой `Func` но он всегда возвращает `bool`. Предикат - это способ указания пользовательских критериев. В зависимости от значения входа и логики, определенных в предикате, он вернет либо `true` либо `false`. Таким образом, `Predicate<T>` ведет себя так же, как `Func<T, bool>` и оба могут быть инициализированы и использованы одинаково.

```

Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");

// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");

```

Выбор того, следует ли использовать `Predicate<T>` или `Func<T, bool>`, действительно является вопросом мнения. `Predicate<T>`, возможно, более выразителен в намерениях автора, тогда как `Func<T, bool>`, скорее всего, будет знаком с большей частью разработчиков С#.

В дополнение к этому, есть некоторые случаи, когда доступен только один из вариантов, особенно при взаимодействии с другим API. Например, `List<T>` и `Array<T>` обычно используют `Predicate<T>` для своих методов, в то время как большинство расширений LINQ принимают только `Func<T, bool>`.

Назначение именованного метода делегату

Именованные методы могут быть назначены делегатам с соответствующими сигнатурами:

```

public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}

```

```
Func<int, int> addOne = Example.AddOne
```

```
Example.AddOne
```

принимает `int` и возвращает `int`, его подпись соответствует делегату `Func<int,int>`. `Example.AddOne` может быть напрямую назначено `addOne` поскольку они имеют соответствующие подписи.

Равенство делегатов

Вызов `.Equals()` для делегата сравнивается по ссылочному равенству:

```
Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True
```

Эти правила также применяются при выполнении команды `+=` или `-=` в многоадресном делегете, например, при подписке и отмене подписки на события.

Назначение делегата лямбдой

Lambdas можно использовать для создания анонимных методов для назначения делегату:

```
Func<int,int> addOne = x => x+1;
```

Обратите внимание, что явное объявление типа требуется при создании переменной таким образом:

```
var addOne = x => x+1; // Does not work
```

Передача делегатов в качестве параметров

Делегаты могут использоваться в качестве типизированных указателей функций:

```
class FuncAsParameters
{
    public void Run()
    {
        DoSomething(ErrorHandler1);
        DoSomething(ErrorHandler2);
    }

    public bool ErrorHandler1(string message)
    {
        Console.WriteLine(message);
        var shouldWeContinue = ...
        return shouldWeContinue;
    }

    public bool ErrorHandler2(string message)
    {
```

```

// ...Write message to file...
var shouldWeContinue = ...
return shouldWeContinue;
}

public void DoSomething(Func<string, bool> errorHandler)
{
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
        if (!errorHandler("Some error occurred!"))
        {
            // The handler decided we can't continue
            return;
        }
    }
}
}
}

```

Объединение делегатов (многоадресные делегаты)

Сложение + и вычитание - операции могут быть использованы для объединения экземпляров делегата. Делегат содержит список назначенных делегатов.

```

using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
        }
    }
}

```

```
    d3(1);  
  
    MyDelegate d4 = d3 - d2;  
    // Output:  
    // 1  
    d4(1);  
  
    // Output:  
    // True  
    Console.WriteLine(d1 == d4);  
}  
}  
}
```

В этом примере `d3` представляет собой комбинацию делегатов `d1` и `d2`, поэтому при вызове программы выводят как строки `1` и `System.Int32`.

Объединение делегатов с **непустыми** возвращаемыми типами:

Если групповой делегат имеет `nonvoid` типа возвращаемого, вызывающий абонент получает возвращаемое значение из последнего метода, который будет вызвано. Предыдущие методы все еще вызываются, но их возвращаемые значения отбрасываются.

```
class Program
{
    public delegate int Transformer(int x);

    static void Main(string[] args)
    {
        Transformer t = Square;
        t += Cube;
        Console.WriteLine(t(2)); // O/P 8
    }

    static int Square(int x) { return x * x; }

    static int Cube(int x) { return x*x*x; }
}
```

т (2) вызовет первый `square` а затем `cube`. Возвращаемое значение `square` отбрасывается и возвращается значение последнего метода, т.е. `cube` сохраняется.

Деактивировать многоадресную рассылку

Когда-либо хотелось вызвать делегата многоадресной рассылки, но вы хотите, чтобы весь список вызовов был вызван, даже если исключение происходит в любом из цепочки. Тогда вам повезло, я создал метод расширения, который делает именно это, бросая `AggregateException` только после завершения полного списка:

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del, params object[] args)
```

```

{
    var exceptions = new List<Exception>();

    foreach (var handler in del.GetInvocationList())
    {
        try
        {
            handler.Method.Invoke(handler.Target, args);
        }
        catch (Exception ex)
        {
            exceptions.Add(ex);
        }
    }

    if(exceptions.Any())
    {
        throw new AggregateException(exceptions);
    }
}

public class Test
{
    public delegate void SampleDelegate();

    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}

```

Эти результаты:

```

Target 2 executed
Target 1 executed

```

Вызов напрямую, без `SafeInvoke`, выполнит только Target 2.

Закрытие внутри делегата

Закрытие - это встроенные анонимные методы, которые имеют возможность использовать переменные метода `Parent` и другие анонимные методы, которые определены в области родителя.

По сути, закрытие представляет собой блок кода, который может быть выполнен в более позднее время, но который поддерживает среду, в которой он был сначала создан, т. Е. Он все еще может использовать локальные переменные и т. Д. Метода, который его создал, даже после этого метод завершил выполнение. - **Джон Скит**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();
}
```

Пример, взятый из [Closures в .NET](#).

Инкапсулирующие преобразования в функциях

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now) {
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

В духе чистого кодирования инкапсуляция проверок и преобразований, подобных описанной выше как `Func`, может упростить чтение и понимание кода. В то время как приведенный выше пример очень прост, что, если было несколько свойств `DateTime`, каждый из которых имеет свои собственные правила проверки достоверности, и мы хотели проверить разные комбинации? Простые односрочные `Funcs`, у которых установлена

логика возврата, могут быть как читаемыми, так и уменьшать кажущуюся сложность вашего кода. Рассмотрим приведенные ниже вызовы Func и представьте, сколько еще кода будет загромождать метод:

```
public void CheckForIntegrity() {
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)) {
        SendPassingTestNotification();
    }
}
```

Прочтайте Делегаты онлайн: <https://riptutorial.com/ru/csharp/topic/1194/делегаты>

глава 51: Делегаты Func

Синтаксис

- `public delegate TResult Func<in T, out TResult>(T arg)`
- `public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)`
- `public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)`
- `public delegate TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)`

параметры

параметр	подробности
<code>arg</code> или <code>arg1</code>	(первый) параметр метода
<code>arg2</code>	второй параметр метода
<code>arg3</code>	третий параметр метода
<code>arg4</code>	четвертый параметр метода
<code>T</code> или <code>T1</code>	тип (первого) параметра метода
<code>T2</code>	тип второго параметра метода
<code>T3</code>	тип третьего параметра метода
<code>T4</code>	тип четвертого параметра метода
<code>TResult</code>	возвращаемый тип метода

Examples

Без параметров

В этом примере показано, как создать делегат, который инкапсулирует метод, который возвращает текущее время

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
```

```

        return DateTime.Now;
    }

static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time

    DateTime localNow = method();
}

```

С несколькими переменными

```

static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
    Func<int, int, int> method = Sum;
    // method points to the Sum method
    // that retuns 1 int variable and takes 2 int variables
    int sum = method(1, 1);

    method = Multiplication;
    // now method points to the Multiplication method

    int multiplication = method(1, 1);
}

```

Лямбда и анонимные методы

Анонимный метод может быть назначен везде, где ожидается делегат:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Лямбда-выражения могут использоваться для выражения того же:

```
Func<int, int> square = x => x * x;
```

В любом случае мы теперь можем вызвать метод, хранящийся внутри `square` следующим образом:

```
var sq = square.Invoke(2);
```

Или как сокращение:

```
var sq = square(2);
```

Обратите внимание, что для присвоения типа безопасным типа типы параметров и тип возврата анонимного метода должны соответствовать типу типа делегата:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

Параметры ковариантного и контравариантного типов

Func также поддерживает [ковариантные и контравариантные](#)

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Прочтайте Делегаты Func онлайн: <https://riptutorial.com/ru/csharp/topic/2769/делегаты-func>

глава 52: Деревья выражений

Вступление

Деревья выражений - это выражения, расположенные в древовидной структуре данных. Каждый узел в дереве является представлением выражения, причем выражение является кодом. Представление Lambda с внутренней памятью было бы деревом выражений, которое содержит фактические элементы (то есть код) запроса, но не его результат. Деревья выражений делают структуру лямбда-выражения прозрачной и явной.

Синтаксис

- Выражение `<TDelegate> name = lambdaExpression;`

параметры

параметр	подробности
<code>TDelegate</code>	Тип делегата, который будет использоваться для выражения
<code>lambdaExpression</code>	Выражение лямбда (например, <code>num => num < 5</code>)

замечания

Введение в деревья выражений

Откуда мы пришли

Деревья выражений - все о потреблении «исходного кода» во время выполнения. Рассмотрим метод, который рассчитывает налог с продаж, причитающийся по заказу клиента `decimal CalculateTotalTaxDue(SalesOrder order)`. Использование этого метода в .NET-программе легко - просто назовите его `decimal taxDue = CalculateTotalTaxDue(order);`, Что делать, если вы хотите применить его ко всем результатам удаленного запроса (SQL, XML, удаленный сервер и т. Д.)? Эти источники удаленного запроса не могут вызвать метод! Традиционно вам придется инвертировать поток во всех этих случаях. Сделайте весь запрос, сохраните его в памяти, затем просмотрите результаты и рассчитайте налог за каждый результат.

Как избежать ошибок инверсии потока и проблем с задержкой

Деревья выражений - это структуры данных в формате дерева, где каждый узел содержит выражение. Они используются для перевода скомпилированных инструкций (например, методов, используемых для фильтрации данных) в выражениях, которые могут использоваться вне программной среды, например внутри запроса к базе данных.

Проблема здесь в том, что удаленный запрос *не может получить доступ к нашему методу*. Мы могли бы избежать этой проблемы, если вместо этого мы отправили *инструкции* для метода в удаленный запрос. В нашем примере `CalculateTotalTaxDue` это означает, что мы отправляем эту информацию:

1. Создать переменную для хранения общего налога
2. Прокрутите все строки по порядку
3. Для каждой строки проверьте, облагается ли продукт
4. Если это так, умножьте общую сумму по применимой ставке налога и добавьте эту сумму к общей сумме
5. В противном случае ничего не делать

С помощью этих инструкций удаленный запрос может выполнять работу по мере создания данных.

Для этого есть две проблемы. Как преобразовать скомпилированный метод .NET в список инструкций и как вы отформатируете инструкции таким образом, чтобы их можно было использовать удаленной системой?

Без деревьев выражений вы могли бы решить первую проблему с MSIL. (MSIL - это ассемблерный код, созданный компилятором .NET.) Разбор MSIL возможен, но это непросто. Даже если вы правильно разобрали его, может быть трудно определить, какова была цель оригинального программиста с конкретной процедурой.

Деревья выражений сохраняют день

Деревья выражений адресуют эти точные проблемы. Они представляют собой программные инструкции для структуры данных дерева, где каждый узел представляет *одну инструкцию* и имеет ссылки на всю информацию, необходимую для выполнения этой инструкции. Например, `MethodCallExpression` имеет отношение к 1) `MethodInfo` он собирается позвонить, 2) список `Expression` `S` будет переходить к этому методу, 3) для методов экземпляра, то `Expression` вы будете вызывать метод. Вы можете «ходить по дереву» и применять инструкции к вашему удаленному запросу.

Создание деревьев выражений

Самый простой способ создать дерево выражений - с помощью выражения лямбда. Эти выражения выглядят почти так же, как обычные методы C#. Важно понимать, что это **магия компилятора**. Когда вы сначала создаете лямбда-выражение, компилятор проверяет, к чему вы его назначили. Если это тип `Delegate` (включая `Action` или `Func`), компилятор преобразует лямбда-выражение в делегат. Если это `LambdaExpression` (или `Expression<Action<T>>` или `Expression<Func<T>>`) которое строго типизировано `LambdaExpression`, компилятор преобразует его в `LambdaExpression`. Это - то, где волшебство начинает. За кулисами, компилятор *использует API дерева выражений*, чтобы преобразовать Ваше `lambda выражение` в `LambdaExpression`.

Лямбда-выражения не могут создавать каждый тип дерева выражений. В этих случаях вы можете использовать API выражений вручную, чтобы создать дерево, в котором вы нуждаетесь. В примере «[Понимание примеров выражений](#)» мы создаем выражение `CalculateTotalSalesTax` с использованием API.

ПРИМЕЧАНИЕ. Названия здесь немного запутывают. *Лямбда-выражение* (два слова, нижний регистр) относится к блоку кода с индикатором `a =>`. Он представляет анонимный метод в C# и преобразуется в `Delegate` или `Expression`. *LambdaExpression* (одно слово, PascalCase) относится к типу узла в Expression API, который представляет собой метод, который вы можете выполнить.

Деревья выражений и LINQ

Одним из наиболее распространенных применений деревьев выражений является запрос LINQ и базы данных. LINQ создает дерево выражений с поставщиком запросов для применения ваших инструкций к целевому удаленному запросу. Например, поставщик запросов LINQ to Entity Framework преобразует дерево выражений в SQL, который выполняется непосредственно из базы данных.

Объединяя все части, вы можете увидеть реальную силу LINQ.

1. Напишите запрос, используя выражение лямбда: `products.Where(x => x.Cost > 5)`
2. Компилятор преобразует это выражение в дерево выражений с инструкциями «проверьте, является ли свойство `Cost` параметра более пяти».
3. Поставщик запроса анализирует дерево выражений и создает корректный SQL-запрос `SELECT * FROM products WHERE Cost > 5`
4. ORM реализует все результаты в POCOs, и вы получаете список объектов назад

Заметки

- Деревья выражений неизменяемы. Если вы хотите изменить дерево выражений, вам нужно создать новый, скопировать существующий в новый (чтобы пересечь дерево выражений, вы можете использовать `ExpressionVisitor`) и внести нужные изменения.

Examples

Создание деревьев выражений с помощью API

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

Компиляция деревьев выражений

```
// Define an expression tree, taking an integer, returning a bool.
Expression<Func<int, bool>> expr = num => num < 5;

// Call the Compile method on the expression tree to return a delegate that can be called.
Func<int, bool> result = expr.Compile();

// Invoke the delegate and write the result to the console.
Console.WriteLine(result(4)); // Prints true

// Prints True.

// You can also combine the compile step with the call/invoke step as below:
Console.WriteLine(expr.Compile()(4));
```

Разбор выражений деревьев

```
using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);

// Decomposed expression: num => num LessThan 5
```

Создание деревьев выражений с выражением лямбда

Следующее - это самое основное дерево выражений, созданное лямбдой.

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

Чтобы создать деревья выражений «вручную», следует использовать класс `Expression`.

Вышеприведенное выражение эквивалентно:

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num argument
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 constant
BinaryExpression equality = Expression.Equals(parameter, constant); // equality of two
expressions (num == 42)
Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter);
```

Понимание выражений API

Мы будем использовать API дерева выражений для создания дерева `CalculateSalesTax`. На простом английском языке дайте краткое описание шагов, которые необходимо предпринять для создания дерева.

1. Проверьте, облагается ли товар
2. Если это так, умножьте общую сумму по применимой ставке налога и верните эту сумму
3. В противном случае верните 0

```
//For reference, we're using the API to build this lambda expression
orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//The orderLine parameter we pass in to the method. We specify it's type (OrderLine) and the
name of the parameter.
ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//Check if the parameter is taxable; First we need to access the is taxable property, then
check if it's true
 PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
 MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
 UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);

//Before creating the if, we need to create the branches
 //If the line is taxable, we'll return the total times the tax rate; get the total and tax
rate, then multiply
 //Get the total
 PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
 MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

 //Get the order
 PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
 MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

 //Get the tax rate - notice that we pass the getOrder expression directly to the member
access
```

```

PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

//Multiply the two - notice we pass the two operand expressions directly to multiply
BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

//If the line is not taxable, we'll return a constant value - 0.0 (decimal)
ConstantExpression zero = Expression.Constant(0M);

//Create the actual if check and branches
ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
multiplyTotalByRate, zero);

//Wrap the whole thing up in a "method" - a LambdaExpression
Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
decimal>>(ifTaxableTernary, orderLine);

```

Дерево выражений Basic

Деревья выражений представляют код в древовидной структуре данных, где каждый узел является выражением

Деревья выражений позволяют динамическую модификацию исполняемого кода, выполнение запросов LINQ в различных базах данных и создание динамических запросов. Вы можете скомпилировать и запустить код, представленный деревьями выражений.

Они также используются в динамической языковой версии (DLR) для обеспечения взаимодействия между динамическими языками и .NET Framework и позволяют разработчикам компилятора генерировать деревья выражений вместо промежуточного языка Microsoft (MSIL).

Деревья выражений могут быть созданы через

1. Анонимное выражение лямбда,
2. Вручную, используя пространство имен System.Linq.Expressions.

Деревья выражений из лямбда-выражений

Когда выражение lambda назначается переменной типа Expression, компилятор испускает код для построения дерева выражений, которое представляет собой выражение лямбда.

В следующих примерах кода показано, как компилятор C # создает дерево выражений, которое представляет выражение лямбда num => num < 5.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Деревья выражений с помощью API

Деревья выражений также создаются с использованием класса **Expression**. Этот класс содержит статические заводские методы, которые создают узлы дерева выражений

определенных типов.

Ниже несколько типов узлов дерева.

1. ParameterExpression
2. MethodCallExpression

В следующем примере кода показано, как создать дерево выражений, которое представляет выражение `lambda num => num <5` с помощью API.

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive, new
ParameterExpression[] { numParam });
```

Изучение структуры выражения с использованием посетителя

Определите новый класс посетителя, переопределив некоторые из методов `ExpressionVisitor`:

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("Constant: {0}", node);
        return base.VisitConstant(node);
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("Parameter: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("Binary with operator {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

Позвоните в `Visit` чтобы использовать этого посетителя в существующем выражении:

```
Expression<Func<int, bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

Прочтайте Деревья выражений онлайн: <https://riptutorial.com/ru/csharp/topic/75/деревья-выражений>

глава 53: Дженирики

Синтаксис

- `public void SomeMethod <T> () { }`
- `public void SomeMethod<T, V>() { }`
- `public T SomeMethod<T>(IEnumerable<T> sequence) { ... }`
- `public void SomeMethod<T>() where T : new() { }`
- `public void SomeMethod<T, V>() where T : new() where V : struct { }`
- `public void SomeMethod<T>() where T: IDisposable { }`
- `public void SomeMethod<T>() where T: Foo { }`
- `public class MyClass<T> { public T Data {get; set; } }`

параметры

Параметр (ы)	Описание
ТВ	Введите заполнители для общих деклараций

замечания

Дженирики в C # поддерживаются вплоть до среды выполнения: общие типы, созданные с помощью C #, будут иметь свою обобщенную семантику, даже после компиляции в [CIL](#) .

Это фактически означает, что в C # можно отражать общие типы и видеть их по мере их объявления или проверять, является ли объект экземпляром общего типа, например. Это контрастирует с [стиранием типа](#) , когда информация о родовом типе удаляется во время компиляции. Это также контрастирует с шаблоном подход к дженерикам, где несколько конкретных родовых типов становятся множественными не-генерическими типами во время выполнения, и любые метаданные, необходимые для последующего создания исходных описаний общего типа, теряются.

Однако будьте осторожны при отражении на общих типах: имена обобщенных типов будут изменены при компиляции, заменяя угловые скобки и имена типов параметров обратным ходом, а затем число параметров типового типа. Таким образом, `Dictionary< TKey, TValue >` будут переведены на `Dictionary`2` .

Examples

Параметры типа (классы)

Декларация:

```
class MyGenericClass<T1, T2, T3, ...>
{
    // Do something with the type parameters.
}
```

Инициализация:

```
var x = new MyGenericClass<int, char, bool>();
```

Использование (как тип параметра):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

Параметры типа (методы)

Декларация:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

Призвание:

Нет необходимости поставлять аргументы типа genru-методу, потому что компилятор может косвенно выводить тип.

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

Однако, если есть двусмысленность, общие методы нужно вызывать с аргументами типа, как

```
MyGenericMethod<int, int, string>(x,y,z);
```

Параметры типа (интерфейсы)

Декларация:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Использование (в наследовании):

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }
```

```
class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

Использование (как тип параметра):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

Неявный тип вывода (методы)

При передаче формальных аргументов в общий метод соответствующие аргументы общего типа обычно можно вывести неявно. Если все типичные типы можно вывести, то указание их в синтаксисе является необязательным.

Рассмотрим следующий общий метод. Он имеет один формальный параметр и один общий параметр типа. Между ними существует очень очевидная взаимосвязь: тип, переданный как аргумент параметру generic type, должен быть таким же, как тип времени компиляции аргумента, переданного формальному параметру.

```
void M<T>(T obj)
{
}
```

Эти два вызова эквивалентны:

```
M<object>(new object());
M(new object());
```

Эти два вызова также эквивалентны:

```
M<string>("");
M("");
```

И вот эти три звонка:

```
M<object>("");
M((object) "");
M("") as object;
```

Обратите внимание: если хотя бы один аргумент типа не может быть выведен, то все они должны быть указаны.

Рассмотрим следующий общий метод. Первый общий аргумент типа совпадает с типом формального аргумента. Но для второго аргумента типа типа нет таких отношений. Поэтому компилятор не может вывести второй аргумент типа общего типа при любом вызове этого метода.

```
void X<T1, T2>(T1 obj)
{
}
```

Это больше не работает:

```
X("");
```

Это тоже не работает, потому что компилятор не уверен, указываем ли мы первый или второй общий параметр (оба будут действительными как `object`):

```
X<object>("");
```

Мы должны вывести оба из них, например:

```
X<string, object>("");
```

Ограничения типа (классы и интерфейсы)

Ограничения типа могут принудительно вводить параметр типа для реализации определенного интерфейса или класса.

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{
}

// T must be a subtype of IType
class Generic<T>
    where T : IType
{
}

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }
```

```
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

Синтаксис для нескольких ограничений:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

Ограничения типа работают так же, как наследование, поскольку в качестве ограничений для общего типа можно указать несколько интерфейсов, но только один класс:

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{
}

class Generic2<T>
    where T : A, B //Compilation error
{
}
```

Другое правило состоит в том, что класс должен быть добавлен как первое ограничение, а затем интерфейсы:

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

Все объявленные ограничения должны выполняться одновременно для конкретного типичного экземпляра для работы. Нет способа указать два или более альтернативных набора ограничений.

Ограничения типа (класс и структура)

Можно указать, должен ли тип-тип быть ссылочным типом или типом значения с использованием соответствующего `class` ограничений или `struct`. Если эти ограничения

используются, они должны быть определены до того, как все остальные ограничения (например, родительский тип или `new()`) могут быть перечислены.

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.
// Interfaces are valid, as they are reference types
class AcceptsRefType<TRef>
    where TRef : class
{
    // TStruct must be a value type.
    public void AcceptStruct<TStruct>()
        where TStruct : struct
    {
    }

    // If multiple constraints are used along with class/struct
    // then the class or struct constraint MUST be specified first
    public void Foo<TComparableClass>()
        where TComparableClass : class, IComparable
    {
    }
}
```

Ограничения типа (новое ключевое слово)

Используя ограничение `new()`, можно задать параметры типа для определения пустого (по умолчанию) конструктора.

```
class Foo
{
    public Foo () { }
}

class Bar
{
    public Bar (string s) { ... }
}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty
constructor.
```

Второй вызов `Create()` даст ошибку времени компиляции со следующим сообщением:

«Бар» должен быть не абстрактным типом с открытым конструктором без параметров, чтобы использовать его как параметр «T» в родовом типе или методе «Фабрика»,

Для конструктора с параметрами нет ограничений, поддерживаются только конструкторы без параметров.

Тип вывода (классы)

Разработчики могут быть пойманы тем фактом, что вывод типа *не работает* для конструкторов:

```
class Tuple<T1, T2>
{
    public Tuple(T1 value1, T2 value2)
    {
    }
}

var x = new Tuple(2, "two");           // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

Первый способ создания экземпляра без явного указания параметров типа приведет к ошибке времени компиляции, которая скажет:

Использование типичного типа «`Tuple <T1, T2>`» требует 2 аргумента типа

Общим решением является добавление вспомогательного метода в статический класс:

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
        return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two"); // This WILL work...
```

Отражение по параметрам типа

Оператор `typeof` работает с параметрами типа.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

Явные параметры типа

Существуют разные случаи, когда необходимо явно указать параметры типа для общего метода. В обоих нижеприведенных случаях компилятор не может вывести все параметры

типа из указанных параметров метода.

Один случай - когда нет параметров:

```
public void SomeMethod<T, V>()
{
    // No code for simplicity
}

SomeMethod(); // doesn't compile
SomeMethod<int, bool>(); // compiles
```

Второй случай - когда один (или более) параметров типа не является частью параметров метода:

```
public K SomeMethod<K, V>(V input)
{
    return default(K);
}

int num1 = SomeMethod(3); // doesn't compile
int num2 = SomeMethod<int>("3"); // doesn't compile
int num3 = SomeMethod<int, string>("3"); // compiles.
```

Использование общего метода с интерфейсом в качестве типа ограничения.

Это пример того, как использовать общий тип TFood внутри метода Eat для класса Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }

    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
```

```

public Carnivore()
{
    Name = "Carnivore";
}
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}

```

Вы можете вызвать метод Eat следующим образом:

```

var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore

```

В этом случае, если вы попытаетесь позвонить:

```
sheep.Eat(lion);
```

Это будет невозможно, потому что объект-лев не реализует интерфейс IFood. Попытка выполнить вышеупомянутый вызов вызовет ошибку компилятора: «Тип «Carnivore» не может использоваться как параметр типа «TFood» в родовом типе или методе «Animal.Eat(TFood)». Нет никакого неявного преобразования ссылок из 'Carnivore' to 'IFood' . »

ковариации

Когда `IEnumerable<T>` является подтипов другого `IEnumerable<T1>`? Когда `T` является подтипов `T1`. `IEnumerable` является **ковариантным** в своем параметре `T`, что означает, что отношение подтипа `IEnumerable` идет в том же направлении, что и `T`

```

class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs; // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals; // Compilation error - IEnumerable<Animal> is not a subtype of
IEnumerable<Dog>

```

Экземпляр ковариантного родового типа с заданным параметром типа неявно конвертируется в один и тот же общий тип с параметром менее производного типа.

Это соотношение выполняется потому, что `IEnumerable` *производит* `T`, но не потребляет их. Объект, который создает `Dog` `S`, может использоваться так, как если бы он произвел `Animal` `S`.

Параметры типа `Covariant` объявляются с использованием ключевого слова `out`, потому что параметр должен использоваться только как *результат*.

```
interface IEnumerable<out T> { /* ... */ }
```

Параметр типа, объявленный как ковариантный, может не отображаться как вход.

```
interface Bad<out T>
{
    void SetT(T t); // type error
}
```

Вот полный пример:

```
using NUnit.Framework;

namespace ToyStore
{
    enum Taste { Bitter, Sweet };

    interface IWidget
    {
        int Weight { get; }
    }

    interface IFactory<out TWidget>
        where TWidget : IWidget
    {
        TWidget Create();
    }

    class Toy : IWidget
    {
        public int Weight { get; set; }
        public Taste Taste { get; set; }
    }

    class ToyFactory : IFactory<Toy>
    {
        public const int StandardWeight = 100;
        public const Taste StandardTaste = Taste.Sweet;

        public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste }; }
    }

    [TestFixture]
    public class GivenAToyFactory
```

```

{
    [Test]
    public static void WhenUsingToyFactoryToMakeWidgets()
    {
        var toyFactory = new ToyFactory();

        //// Without out keyword, note the verbose explicit cast:
        // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

        // covariance: concrete being assigned to abstract (shiny and new)
        IFactory<IWidget> widgetFactory = toyFactory;
        IWidget anotherToy = widgetFactory.Create();
        Assert.That(anotherToy.Weight, Is.EqualTo(ToyFactory.StandardWeight)); // abstract
contract
        Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(ToyFactory.StandardTaste)); // concrete
contract
    }
}
}
}

```

контравариация

Когда `IComparer<T>` является подтиповом другого `IComparer<T1>`? Когда `T1` является подтиповом `T` **контравариантен** по своему параметру `T`, что означает, что отношение подтипа `IComparer` идет в **противоположном направлении** как `T`

```

class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer; // IComparer<Animal> is a subtype of
IComparer<Dog>
// animalComparer = dogComparer; // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>

```

Экземпляр контравариантного типового типа с заданным параметром типа неявно конвертируется в один и тот же общий тип с параметром более производного типа.

Это соотношение выполняется потому, что `IComparer` **потребляет** `T` `s`, но не производит их. Объект, который может сравнивать любые два `Animal` `s`, можно использовать для сравнения двух `Dog` `s`.

Параметры контравариантного типа объявляются с использованием ключевого слова `in`, поскольку параметр должен использоваться только как **вход**.

```
interface IComparer<in T> { /* ... */ }
```

Параметр типа, объявленный как контравариантный, может не отображаться как **выходной**.

```
interface Bad<in T>
{
```

```
    T GetT(); // type error
}
```

НЕИЗМЕННОСТЬ

`IList<T>` никогда не является подтиром другого `IList<T1>`. `IList` **инвариантен** по своему параметру типа.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs; // type error
```

Для списков нет подтипов, потому что вы можете поместить значения в список и вывести значения из списка.

Если `IList` был ковариантным, вы могли бы добавить элементы *неправильного подтипа* в **данный список**.

```
IList<Animal> animals = new List<Dog>(); // supposing this were allowed...
animals.Add(new Giraffe()); // ... then this would also be allowed, which is bad!
```

Если `IList` был контравариантным, вы могли бы извлечь значения неправильного подтипа из **определенного списка**.

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() }; // if this were allowed...
Dog dog = dogs[1]; // ... then this would be allowed, which is bad!
```

Инвариантные параметры типа объявляются, опуская как **ключевые слова** `in` и `out`.

```
interface IList<T> { /* ... */ }
```

Варианты интерфейсов

Интерфейсы могут иметь параметры типа варианта.

```
interface IEnumerable<out T>
{
    // ...
}

interface IComparer<in T>
{
    // ...
}
```

но классы и структуры могут не

```
class BadClass<in T1, out T2> // not allowed
{
}

struct BadStruct<in T1, out T2> // not allowed
{
}
```

а также объявления общих методов

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1) // not allowed
    {
        // ...
    }
}
```

В приведенном ниже примере показаны несколько объявлений о различиях на одном интерфейсе

```
interface IFoo<in T1, out T2, T3>
// T1 : Contravariant type
// T2 : Covariant type
// T3 : Invariant type
{
    // ...
}

IFoo<Animal, Dog, int> fool = /* ... */;
IFoo<Dog, Animal, int> foo2 = fool;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

Варианты делегатов

У делегатов могут быть параметры типа варианта.

```
delegate void Action<in T>(T t); // T is an input
delegate T Func<out T>(); // T is an output
delegate T2 Func<in T1, out T2>(); // T1 is an input, T2 is an output
```

Это вытекает из [Принципа замещения Лискова](#), в котором говорится (среди прочего), что метод D можно считать более производным, чем метод B, если:

- D имеет равный или более производный тип возврата, чем B
- D имеет одинаковые или более общие типы параметров, чем B

Поэтому следующие присвоения являются безопасными по типу:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
```

```
Func<string, object> d3 = original;
```

Типы вариантов как параметры и возвращаемые значения

Если в качестве вывода появляется ковариантный тип, то содержащийся тип является ковариантным. Изготовление производителя `t` `s` похоже на производство `t` `s`.

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

Если контравариантный тип появляется как результат, то содержащийся тип контравариантен. Изготовление потребителя `t` `s` подобно потреблению `t` `s`.

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

Если в качестве ввода появляется ковариантный тип, содержащий тип контравариант. Потребление производителя `t` `s` подобно потреблению `t` `s`.

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

Если в качестве ввода появляется контравариантный тип, то содержащийся тип является ковариантным. Потребление потребителя `t` `s` похоже на производство `t` `s`.

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

Проверка равенства общих значений.

Если логика родового класса или метода требует проверки равенства значений, имеющих общий тип, используйте [свойство](#) `EqualityComparer<TType>.Default`:

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1, arg2)
    {
        ...
    }
}
```

Этот подход лучше, чем просто вызов метода `Object.Equals()`, поскольку реализация `TBar` умолчанию проверяет, реализует ли тип `IEquatable<TBar>` **интерфейс `IEquatable<TBar>`** и если да, вызывает `IEquatable<TBar>.Equals(TBar other)`. Это позволяет избежать боксирования / распаковки типов значений.

Литейное производство

```
/// <summary>
/// Converts a data type to another data type.
/// </summary>
public static class Cast
{
    /// <summary>
    /// Converts input to Type of default value or given as typeparam T
    /// </summary>
    /// <typeparam name="T">typeparam is the type in which value will be returned, it
    could be any type eg. int, string, bool, decimal etc.</typeparam>
    /// <param name="input">Input that need to be converted to specified type</param>
    /// <param name="defaultValue">defaultValue will be returned in case of value is null
    or any exception occurs</param>
    /// <returns>Input is converted in Type of default value or given as typeparam T and
    returned</returns>
    public static T To<T>(object input, T defaultValue)
    {
        var result = defaultValue;
        try
        {
            if (input == null || input == DBNull.Value) return result;
            if (typeof (T).IsEnum)
            {
                result = (T) Enum.ToObject(typeof (T), To(input,
Convert.ToInt32(defaultValue)));
            }
            else
            {
                result = (T) Convert.ChangeType(input, typeof (T));
            }
        }
        catch (Exception ex)
        {
            Tracer.Current.LogException(ex);
        }

        return result;
    }

    /// <summary>
    /// Converts input to Type of typeparam T
    /// </summary>
    /// <typeparam name="T">typeparam is the type in which value will be returned, it
    could be any type eg. int, string, bool, decimal etc.</typeparam>
    /// <param name="input">Input that need to be converted to specified type</param>
    /// <returns>Input is converted in Type of default value or given as typeparam T and
    returned</returns>
    public static T To<T>(object input)
    {
        return To(input, default(T));
    }
}
```

```
}
```

Обычай:

```
std.Name = Cast.To<string>(drConnection["Name"]);
std.Age = Cast.To<int>(drConnection["Age"]);
std.IsPassed = Cast.To<bool>(drConnection["IsPassed"]);

// Casting type using default value
//Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.To<string>(drConnection["Name"], "");
std.Marks = Cast.To<int>(drConnection["Marks"], 0);
```

Устройство чтения конфигурации с общим типом

```
/// <summary>
/// Read configuration values from app.config and convert to specified types
/// </summary>
public static class ConfigurationReader
{
    /// <summary>
    /// Get value from AppSettings by key, convert to Type of default value or typeparam T
    and return
    /// </summary>
    /// <typeparam name="T">typeparam is the type in which value will be returned, it
    could be any type eg. int, string, bool, decimal etc.</typeparam>
    /// <param name="strKey">key to find value from AppSettings</param>
    /// <param name="defaultValue">defaultValue will be returned in case of value is null
    or any exception occurs</param>
    /// <returns>AppSettings value against key is returned in Type of default value or
    given as typeparam T</returns>
    public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
    {
        var result = defaultValue;
        try
        {
            if (ConfigurationManager.AppSettings[strKey] != null)
                result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
        }
        catch (Exception ex)
        {
            Tracer.Current.LogException(ex);
        }

        return result;
    }
    /// <summary>
    /// Get value from AppSettings by key, convert to Type of default value or typeparam T
    and return
```

```
/// </summary>
/// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
/// <param name="strKey">key to find value from AppSettings</param>
/// <returns>AppSettings value against key is returned in Type given as typeparam
T</returns>
public static T GetConfigKeyValue<T>(string strKey)
{
    return GetConfigKeyValue(strKey, default(T));
}

}
```

Обычай:

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

Прочтайте Джениерики онлайн: <https://riptutorial.com/ru/csharp/topic/27/джениерики>

глава 54: диагностика

Examples

Debug.WriteLine

Записывает слушателям трассировки в коллекции `Listeners`, когда приложение компилируется в конфигурации отладки.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

В Visual Studio или Xamarin Studio это появится в окне вывода приложения. Это связано с наличием [прослушивателя трассировки по умолчанию](#) в `TraceListenerCollection`.

Перенаправление вывода журнала с помощью `TraceListeners`

Вы можете перенаправить вывод отладки в текстовый файл, добавив `TextWriterTraceListener` в коллекцию `Debug.Listeners`.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

Вы можете перенаправить вывод отладки в выходной поток консольного приложения с помощью `ConsoleTraceListener`.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Прочтайте диагностика онлайн: <https://riptutorial.com/ru/csharp/topic/2147/диагностика>

глава 55: Динамический тип

замечания

`dynamic` Ключевое слово объявляет переменную, тип которой неизвестен во время компиляции. `dynamic` переменная может содержать любое значение, а тип значения может изменяться во время выполнения.

Как отмечено в книге «Метапрограммирование в .NET», C # не имеет типа поддержки для `dynamic` ключевого слова:

Функциональность, включенная ключевым словом `dynamic` является умным набором действий компилятора, которые испускают и используют объекты `CallSite` в контейнере сайта области локального выполнения. Компилятор управляет тем, что программисты воспринимают как динамические ссылки на объекты через эти экземпляры `CallSite`. Параметры, типы возвращаемых данных, поля и свойства, которые получают динамическую обработку во время компиляции, могут быть отмечены некоторыми метаданными, чтобы указать, что они были созданы для динамического использования, но базовый тип данных для них всегда будет `System.Object`.

Examples

Создание динамической переменной

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357    Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper()):
// NOW A STRING
```

Возвращение динамического

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}
```

```
private static dynamic GetValue()
{
    return "dynamics are useful!";
}
```

Создание динамического объекта со свойствами

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

Обработка определенных типов, неизвестных во время компиляции

Следующие эквивалентные результаты:

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
        {
            return DebugToStringInternal(a as StringBuilder);
        }
        else if (a is List<string>)
        {
            return DebugToStringInternal(a as List<string>);
        }
        else
        {
            return a.ToString();
        }
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
```

```

        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())})";
    }
}

class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())})";
    }
}

```

Преимущество динамического, добавляя новый тип для обработки, просто требует добавления перегрузки DebugToStringInternal нового типа. Также устраняет необходимость вручную отбрасывать его на тип.

Прочтайте Динамический тип онлайн: <https://riptutorial.com/ru/csharp/topic/762/динамический-тип>

глава 56: Доступ к базам данных

Examples

Связи ADO.NET

Соединения ADO.NET - один из простейших способов подключения к базе данных из приложения C#. Они полагаются на использование провайдера и строку подключения, которая указывает на вашу базу данных для выполнения запросов.

Общие классы поставщиков данных

Многие из следующих относятся к классам, которые обычно используются для запросов к базам данных и связанным с ними пространствам имен:

- `SqlConnection` , `SqlCommand` , `SqlDataReader` ИЗ `System.Data.SqlClient`
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` ИЗ `System.Data.OleDb`
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` ИЗ `MySql.Data`

Все они обычно используются для доступа к данным через C# и обычно встречаются во всех приложениях, ориентированных на данные. Многие другие классы, которые не упомянуты, которые реализуют те же `FooConnection` , `FooCommand` , `FooDataReader` МОЖНО ожидать, что они будут вести себя одинаково.

Общий шаблон доступа для подключений ADO.NET

Общая схема, которая может использоваться при доступе к вашим данным через соединение ADO.NET, может выглядеть следующим образом:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}"))
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property";
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
        // Open your connection
        connection.Open();

        // Add your parameters here if necessary

        // Execute your query as a reader (again scoped with a using statement)
        using(var reader = command.ExecuteReader())
        {
            // Iterate through your results here
        }
    }
}
```

```
    }
}
```

Или, если вы просто выполняете простое обновление и не нуждаетесь в читателе, будет применяться одна и та же базовая концепция:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
    var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
    using(var command = new SqlCommand(query, connection))
    {
        connection.Open();

        // Add parameters here

        // Perform your update
        command.ExecuteNonQuery();
    }
}
```

Вы даже можете программировать против набора общих интерфейсов и не беспокоиться о конкретных классах провайдера. Основными интерфейсами, предоставляемыми ADO.NET, являются:

- **IDbConnection** - для управления соединениями с базой данных
- **IDbCommand** - для выполнения команд SQL
- **IDbTransaction** - для управления транзакциями
- **IDataReader** - для чтения данных, возвращаемых командой
- **IDataAdapter** - для направления данных в и из наборов данных

```
var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; //for Oracle use
"Oracle.ManagedDataAccess.Client"
//most likely you will get the above two from ConnectionStringSettings object

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}";      //this needs to be tailored for each database
        system

        using(var reader = command.ExecuteReader()) {
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

Подключения к платформе Entity Framework

Entity Framework предоставляет классы абстракции, которые используются для взаимодействия с базовыми базами данных в виде классов, таких как `DbContext`. Эти контексты обычно состоят из `DbSet<T>` которые раскрывают доступные коллекции, которые могут быть запрошены:

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

Сам `DbContext` будет обрабатывать соединения с базами данных и, как правило, считывает соответствующие данные строки соединения из конфигурации, чтобы определить, как установить соединения:

```
public class ExampleContext: DbContext
{
    // The parameter being passed in to the base constructor indicates the name of the
    // connection string
    public ExampleContext() : base("ExampleContextEntities")
    {

    }

    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

Выполнение запросов платформы Entity

Фактически выполнение запроса Entity Framework может быть довольно простым и просто требует, чтобы вы создавали экземпляр контекста, а затем использовали доступные ему свойства для вытягивания или доступа к своим данным

```
using(var context = new ExampleContext())
{
    // Retrieve all of the Widgets in your database
    var data = context.Widgets.ToList();
}
```

Entity Framework также предоставляет обширную систему отслеживания изменений, которая может использоваться для обработки обновлений записей в вашей базе данных путем простого вызова метода `SaveChanges()` для изменения изменений в базе данных:

```
using(var context = new ExampleContext())
{
    // Grab the widget you wish to update
    var widget = context.Widgets.Find(w => w.Id == id);
    // If it exists, update it
    if(widget != null)
    {
        // Update your widget and save your changes
        widget.Updated = DateTime.UtcNow;
        context.SaveChanges();
    }
}
```

```
    }  
}
```

Строки подключения

Строка подключения - это строка, которая указывает информацию о конкретном источнике данных и о том, как подключиться к нему, сохраняя учетные данные, местоположения и другую информацию.

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Сохранение строки подключения

Как правило, строка соединения будет храниться в файле конфигурации (например, `app.config` или `web.config` в приложениях ASP.NET). Ниже приведен пример того, как локальное соединение может выглядеть в одном из этих файлов:

```
<connectionStrings>  
    <add name="WidgetsContext" providerName="System.Data.SqlClient"  
        connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;" />  
</connectionStrings>  
  
<connectionStrings>  
    <add name="WidgetsContext" providerName="System.Data.SqlClient"  
        connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;" />  
</connectionStrings>
```

Это позволит вашему приложению получать доступ к строке подключения программно через `WidgetsContext`. Хотя и `Integrated Security=SSPI` и `Integrated Security=True` выполняют одну и ту же функцию, `Integrated Security=SSPI` предпочтительнее, поскольку работает как с поставщиком `SQLClient`, так и с `OleDb`, где в качестве `Integrated Security=true` генерирует исключение при использовании с поставщиком `OleDb`.

Различные подключения для разных поставщиков

Каждый поставщик данных (SQL Server, MySQL, Azure и т. д.) имеет свой собственный синтаксис для своих строк подключения и предоставляет различные доступные свойства. [ConnectionStrings.com](https://www.connectionstrings.com) - невероятно полезный ресурс, если вы не уверены в том, что должно выглядеть так.

Прочтите Доступ к базам данных онлайн: <https://riptutorial.com/ru/csharp/topic/4811/доступ-к-базам-данных>

глава 57: Доступ к общей общей папке с именем пользователя и паролем

Вступление

Доступ к файлу общего доступа к сети с помощью PInvoke.

Examples

Код для доступа к сетевому файлу

```
public class NetworkConnection : IDisposable
{
    string _networkName;

    public NetworkConnection(string networkName,
                           NetworkCredential credentials)
    {
        _networkName = networkName;

        var netResource = new NetResource()
        {
            Scope = ResourceScope.GlobalNetwork,
            ResourceType = ResourceType.Disk,
            DisplayType = ResourceDisplaytype.Share,
            RemoteName = networkName
        };

        var userName = string.IsNullOrEmpty(credentials.Domain)
            ? credentials.UserName
            : string.Format(@"{0}\{1}", credentials.Domain, credentials.UserName);

        var result = WNetAddConnection2(
            netResource,
            credentials.Password,
            userName,
            0);

        if (result != 0)
        {
            throw new Win32Exception(result);
        }
    }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
    }
}
```

```

        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);

    [DllImport("mpr.dll")]
    private static extern int WNetCancelConnection2(string name, int flags,
        bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope Scope;
    public ResourceType ResourceType;
    public ResourceDisplaytype DisplayType;
    public int Usage;
    public string LocalName;
    public string RemoteName;
    public string Comment;
    public string Provider;
}

public enum ResourceScope : int
{
    Connected = 1,
    GlobalNetwork,
    Remembered,
    Recent,
    Context
};

public enum ResourceType : int
{
    Any = 0,
    Disk = 1,
    Print = 2,
    Reserved = 8,
}

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
    Tree = 0x0a,
    Ndscontainer = 0x0b
}

```

}

Прочтайте Доступ к общей общей папке с именем пользователя и паролем онлайн:
<https://riptutorial.com/ru/csharp/topic/9627/доступ-к-общей-общей-папке-с-именем-пользователя-и-паролем>

глава 58: Заблокировать

Синтаксис

- `lock (obj) {}`

замечания

Используя оператор `lock` вы можете управлять доступом разных потоков к коду внутри блока кода. Он обычно используется для предотвращения условий гонки, например, для чтения и удаления элементов из коллекции. Поскольку блокировка заставляет потоки ждать, пока другие потоки покинут блок кода, это может вызвать задержки, которые могут быть решены другими методами синхронизации.

MSDN

Ключевое слово `lock` указывает блок оператора как критический раздел, получая блокировку взаимного исключения для данного объекта, выполняя инструкцию, а затем освобождая блокировку.

Ключевое слово `lock` гарантирует, что один поток не войдет в критический раздел кода, а другой поток находится в критическом разделе. Если другой поток пытается ввести заблокированный код, он будет ждать, блокировать, пока объект не будет выпущен.

Лучшая практика заключается в определении **частного** объекта для блокировки или **частной** переменной **статического** объекта для защиты данных, общих для всех экземпляров.

В С # 5.0 и более поздних версиях оператор `lock` эквивалентен:

```
bool lockTaken = false;
try
{
    System.Threading.Monitor.Enter(refObject, ref lockTaken);
    // code
}
finally
{
    if (lockTaken)
        System.Threading.Monitor.Exit(refObject);
}
```

Для С # 4.0 и ранее оператор `lock` эквивалентен:

```
System.Threading.Monitor.Enter(refObject);
try
{
    // code
}
finally
{
    System.Threading.Monitor.Exit(refObject);
}
```

Examples

Простое использование

Общее использование `lock` - критический раздел.

В следующем примере `ReserveRoom` должен быть вызван из разных потоков. Синхронизация с `lock` - это самый простой способ предотвратить состояние гонки здесь. Тело метода окружено `lock` который гарантирует, что два или более потока не могут выполнить его одновременно.

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

Если поток достигает `lock` блока, а другой поток работает внутри него, первый будет ждать другого, чтобы выйти из блока.

Лучшая практика заключается в определении частного объекта для блокировки или частной переменной статического объекта для защиты данных, общих для всех экземпляров.

Исключение исключения в инструкции блокировки

Следующий код освободит блокировку. Проблем не будет. За кулисами оператор блокировки работает как `try finally`

```
lock(locker)
{
```

```
        throw new Exception();
    }
```

Больше можно увидеть в [спецификации C # 5.0](#) :

Оператор `lock` формы

```
lock (x) ...
```

где `x` - выражение *ссыльного типа* , точно эквивалентно

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

за исключением того, что `x` оценивается только один раз.

Возвращение в операторе блокировки

Следующий код освободит блокировку.

```
lock(locker)
{
    return 5;
}
```

Для подробного объяснения рекомендуется использовать [этот SO-ответ](#) .

Использование экземпляров объекта для блокировки

При использовании встроенного оператора `lock` C # необходим экземпляр некоторого типа, но его состояние не имеет значения. Экземпляр `object` идеально подходит для этого:

```
public class ThreadSafe {
    private static readonly object locker = new object();

    public void SomeThreadSafeMethod() {
        lock (locker) {
            // Only one thread can be here at a time.
        }
    }
}
```

NB . экземпляры `Type` не должны использоваться для этого (в коде выше `typeof(ThreadSafe)`), потому что экземпляры `Type` совместно используются в `AppDomains`, и, следовательно,

степень блокировки может включать в себя код, который он не должен (например, если `ThreadSafe` загружается в два AppDomains в том же процессе, а затем блокировка на экземпляре `Type` будет взаимно блокироваться).

Anti-Patterns и gotchas

Блокировка на выделенную стеку / локальную переменную

Одной из ошибок при использовании `lock` является использование локальных объектов в качестве блокировки в функции. Поскольку эти локальные экземпляры объектов будут отличаться при каждом вызове функции, `lock` не будет выполняться так, как ожидалось.

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

Предполагая, что блокировка ограничивает доступ к самому синхронизирующему объекту

Если один поток вызывает: `lock(obj)` и другой поток вызывает `obj.ToString()` второй поток

не будет заблокирован.

```
object obj = new Object();

public void SomeMethod()
{
    lock(obj)
    {
        //do dangerous stuff
    }
}

//Meanwhile on other tread
public void SomeOtherMethod()
{
    var objInString = obj.ToString(); //this does not block
}
```

Ожидание подклассов, чтобы знать, когда блокировать

Иногда базовые классы создаются таким образом, что их подклассы должны использовать блокировку при доступе к определенным защищенным полям:

```
public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}
```

```
    }
}
```

Гораздо безопаснее инкапсулировать блокировку с помощью [метода шаблонов](#):

```
public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}
```

Блокировка на коробке ValueType переменной не синхронизирует

В следующем примере приватная переменная неявно помещается в ящик, поскольку она предоставляется в качестве аргумента `object` функции, ожидая, что ресурс монитора будет заблокирован. Бокс происходит непосредственно перед вызовом функции `IncInSync`, поэтому экземпляр в штучной упаковке соответствует разному кучному объекту при каждом вызове функции.

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}
```

```
private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

Бокс происходит в функции `Inc`:

```
BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld      UserQuery+BulemicCounter.counterLock
IL_0008:  box        System.Int32**
IL_000D:  call        UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret
```

Это не означает, что встроенный `ValueType` не может использоваться для блокировки монитора вообще:

```
private readonly object counterLock = 1;
```

Теперь бокс происходит в конструкторе, который отлично подходит для блокировки:

```
IL_0001:  ldc.i4.1
IL_0002:  box        System.Int32
IL_0007:  stfld      UserQuery+BulemicCounter.counterLock
```

Использование блокировок без необходимости, когда существует более безопасная альтернатива

Очень распространенная картина заключается в том, чтобы использовать закрытый `List` или `Dictionary` в потокобезопасном классе и блокировать каждый раз, когда к нему обращаются:

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
```

```

    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}

```

Если есть несколько методов доступа к словарям `values`, код может стать очень длинным и, что более важно, блокировать все время, затеняет его *намерение*. Блокировка также очень легко забыть, и отсутствие правильной блокировки может очень затруднить поиск ошибок.

Используя `ConcurrentDictionary`, мы можем полностью исключить блокировку:

```

public class Cache
{
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}

```

Использование параллельных коллекций также повышает производительность, потому что **все они** в некоторой степени *используют методы блокировки*.

Прочтайте Заблокировать онлайн: <https://riptutorial.com/ru/csharp/topic/1495/заблокировать>

глава 59: Запросы LINQ

Вступление

LINQ - это аббревиатура, обозначающая **L**anguage **I**ntegrated **Q** uery. Это концепция, которая объединяет язык запросов, предлагая согласованную модель для работы с данными в различных источниках и форматах данных; вы используете одни и те же базовые шаблоны кодирования для запроса и преобразования данных в документы XML, базы данных SQL, наборы данных ADO.NET, коллекции .NET и любой другой формат, для которого доступен поставщик LINQ.

Синтаксис

- Синтаксис запроса:
 - из <range variable> в <collection>
 - [from <range variable> в <collection>, ...]
 - <фильтр, объединение, группировка, агрегированные операторы, ...>
 - <выражение lambda>
 - <select или groupBy operator> <сформулировать результат>
- Синтаксис метода:
 - Enumerable.Aggregate (FUNC)
 - Enumerable.Aggregate (seed, func)
 - Enumerable.Aggregate (seed, func, resultSelector)
 - Enumerable.All (предикат)
 - Enumerable.Any ()
 - Enumerable.Any (предикат)
 - Enumerable.AsEnumerable ()
 - Enumerable.Average ()
 - Enumerable.Average (селектор)
 - Enumerable.Cast <результат> ()
 - Enumerable.Concat (второй)
 - Enumerable.Contains (значение)
 - Enumerable.Contains (значение, сравнение)
 - Enumerable.Count ()
 - Enumerable.Count (предикат)
 - Enumerable.DefaultIfEmpty ()
 - Enumerable.DefaultIfEmpty (DefaultValue)
 - Enumerable.Distinct ()
 - Enumerable.Distinct (Comparer)
 - Enumerable.ElementAt (индекс)

- `Enumerable.ElementAtOrDefault (индекс)`
- `Enumerable.Empty ()`
- `Enumerable.Except (второй)`
- `Enumerable.Except (второй, сравнительный)`
- `Enumerable.First ()`
- `Enumerable.First (предикат)`
- `Enumerable.FirstOrDefault ()`
- `Enumerable.FirstOrDefault (предикат)`
- `Enumerable.GroupBy (keySelector)`
- `Enumerable.GroupBy (keySelector, resultSelector)`
- `Enumerable.GroupBy (keySelector, elementSelector)`
- `Enumerable.GroupBy (keySelector, comparer)`
- `Enumerable.GroupBy (keySelector, resultSelector, comparer)`
- `Enumerable.GroupBy (keySelector, elementSelector, resultSelector, comparer)`
- `Enumerable.GroupBy (keySelector, elementSelector, resultSelector, comparer)`
- `Enumerable.Intersect (второй)`
- `Enumerable.Intersect (второй, сравнительный)`
- `Enumerable.Join (внутренний, внешнийKeySelector, innerKeySelector, resultSelector)`
- `Enumerable.Join (внутренний, внешнийKeySelector, innerKeySelector, resultSelector, comparer)`
- `Enumerable.Last ()`
- `Enumerable.Last (предикат)`
- `Enumerable.LastOrDefault ()`
- `Enumerable.LastOrDefault (предикат)`
- `Enumerable.LongCount ()`
- `Enumerable.LongCount (предикат)`
- `Enumerable.Max ()`
- `Enumerable.Max (селектор)`
- `Enumerable.Min ()`
- `Enumerable.Min (селектор)`
- `Enumerable.OfType <TResult> ()`
- `Enumerable.OrderBy (keySelector)`
- `Enumerable.OrderBy (keySelector, comparer)`
- `Enumerable.OrderByDescending (keySelector)`
- `Enumerable.OrderByDescending (keySelector, comparer)`
- `Enumerable.Range (начало, количество)`
- `Enumerable.Repeat (элемент, счетчик)`
- `Enumerable.Reverse ()`
- `Enumerable.Select (селектор)`
- `Enumerable.SelectMany (селектор)`
- `Enumerable.SelectMany (collectionSelector, resultSelector)`
- `Enumerable.SequenceEqual (второй)`

- `Enumerable.SequenceEqual` (второй, сравнительный)
- `Enumerable.Single` ()
- `Enumerable.Single` (предикат)
- `Enumerable.SingleOrDefault` ()
- `Enumerable.SingleOrDefault` (предикат)
- `Enumerable.Skip` (количество)
- `Enumerable.SkipWhile` (предикат)
- `Enumerable.Sum` ()
- `Enumerable.Sum` (селектор)
- `Enumerable.Take` (количество)
- `Enumerable.TakeWhile` (предикат)
- `orderedEnumerable.ThenBy` (keySelector)
- `orderedEnumerable.ThenBy` (keySelector, comparer)
- `orderedEnumerable.ThenByDescending` (keySelector)
- `orderedEnumerable.ThenByDescending` (keySelector, comparer)
- `Enumerable.ToArray` ()
- `Enumerable.ToDictionary` (keySelector)
- `Enumerable.ToDictionary` (keySelector, elementSelector)
- `Enumerable.ToDictionary` (keySelector, comparer)
- `Enumerable.ToDictionary` (keySelector, elementSelector, comparer)
- `Enumerable.ToList` ()
- `Enumerable.ToLookup` (keySelector)
- `Enumerable.ToLookup` (keySelector, elementSelector)
- `Enumerable.ToLookup` (keySelector, comparer)
- `Enumerable.ToLookup` (keySelector, elementSelector, comparer)
- `Enumerable.Union` (второй)
- `Enumerable.Union` (второй, сравнительный)
- `Enumerable.Where` (предикат)
- `Enumerable.Zip` (второй, resultSelector)

замечания

Чтобы использовать запросы LINQ, вам необходимо импортировать `System.Linq`.

Синтаксис метода более мощный и гибкий, но Синтаксис запроса может быть более простым и более знакомым. Все запросы, написанные в синтаксисе Query, транслируются в функциональный синтаксис компилятором, поэтому производительность одинакова.

Объекты запроса не оцениваются до тех пор, пока они не будут использованы, поэтому их можно изменить или добавить без штрафа за производительность.

Examples

куда

Возвращает подмножество элементов, для которых указанный предикат является истинным.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Синтаксис метода

```
// Select all trees with name of length 3
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

Синтаксис запроса

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

Выбрать - Преобразование элементов

Select позволяет применить преобразование к каждому элементу в любой структуре данных, реализующей `IEnumerable`.

Получение первого символа каждой строки в следующем списке:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Использование регулярного (лямбда) синтаксиса

```
//The below select stament transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

Выход:

О
Б
Б
Е
ЧАС
М

[Живая демонстрация на .NET скрипке](#)

Использование LINQ Query Syntax

```
initials = from tree in trees
    select tree.Substring(0, 1);
```

Цепочные методы

Многие функции LINQ работают на `IEnumerable<TSource>` а также возвращают `IEnumerable<TResult>`. Параметры типа `TSource` и `TResult` могут или не могут ссылаться на один и тот же тип в зависимости от рассматриваемого метода и любых переданных ему функций.

Вот несколько примеров этого:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

В то время как для некоторых цепочек методов может потребоваться весь набор, который должен быть обработан до перехода, LINQ использует преимущество [отложенного исполнения](#), используя [MSDN](#) [возврата доходности](#), который создает перечислимый и перечислитель за кулисами. Процесс привязки в LINQ по существу создает перечислимый (итератор) для исходного набора, который откладывается, до тех пор, пока не будет осуществлено [перечислимое перечисление](#).

Это позволяет этим функциям [свободно подключаться](#) [wiki](#), где одна функция может действовать непосредственно на результат другого. Этот стиль кода может использоваться для выполнения многих операций, основанных на последовательностях, в одном выражении.

Например, можно комбинировать `Select`, `Where` и `OrderBy` для преобразования, фильтрации и сортировки последовательности в одном выражении.

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
    .Select(n => n * 2)    // Multiply each number by 2
    .Where(n => n != 6)    // Keep all the results, except for 6
    .OrderBy(n => n);      // Sort in ascending order
```

Выход:

2
4
8

Живая демонстрация на .NET скрипке

Любые функции, которые расширяют и возвращают общий тип `IEnumerable<T>` могут быть использованы в виде цепочечных предложений в одном выражении. Этот стиль свободного программирования является мощным, и его следует учитывать при создании собственных [методов расширения](#).

Диапазон и повторение

Статические методы `Range` и `Repeat` для `Enumerable` могут использоваться для генерации простых последовательностей.

Спектр

`Enumerable.Range()` генерирует последовательность целых чисел, заданную начальным значением и счетчиком.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

Живая демонстрация на .NET скрипке

Повторение

`Enumerable.Repeat()` генерирует последовательность повторяющихся элементов с заданным элементом и количеством требуемых повторений.

```
// Generate a collection containing "a", three times ("a","a","a")
var repeatedValues = Enumerable.Repeat("a", 3);
```

Живая демонстрация на .NET скрипке

Пропустить и принять

Метод `Skip` возвращает коллекцию, исключая количество элементов из начала исходной коллекции. Количество исключенных элементов - это число, указанное в качестве аргумента. Если в коллекции меньше элементов, чем указано в аргументе, возвращается пустая коллекция.

Метод `Take` возвращает коллекцию, содержащую несколько элементов из начала исходной коллекции. Количество включенных элементов - это число, указанное в качестве

аргумента. Если в коллекции меньше элементов, чем указано в аргументе, то возвращенная коллекция будет содержать те же элементы, что и исходная коллекция.

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo      = values.Skip(2);           // { 3, 2, 1 }
var takeThree    = values.Take(3);           // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero     = values.Take(0);           // An IEnumerable<int> with 0 items
```

Живая демонстрация на .NET скрипке

Skip и **Take** обычно используются вместе для получения результатов с разбивкой по страницам, например:

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

Предупреждение: LINQ to Entities поддерживает только пропущенные [запросы](#). Если вы попытаетесь использовать Skip без заказа, вы получите **NotSupportedException** с сообщением «Метод« Пропустить »поддерживается только для отсортированного ввода в LINQ to Entities. Метод« OrderBy »должен быть вызван перед методом« Пропустить ».

Сначала **FirstOrDefault**, **Last**, **LastOrDefault**, **Single** и **SingleOrDefault**

Все шесть методов возвращают одно значение типа последовательности и могут быть вызваны с предикатом или без него.

В зависимости от количества элементов, соответствующих `predicate` или, если `predicate` не указан, количество элементов в исходной последовательности, они ведут себя следующим образом:

Первый()

- Возвращает первый элемент последовательности или первый элемент, соответствующий предоставленному `predicate`.
- Если последовательность не содержит элементов, сообщение `InvalidOperationException` с сообщением: «Последовательность не содержит элементов».
- Если последовательность не содержит элементов, соответствующих предоставленному `predicate`, `InvalidOperationException` с сообщением «Последовательность не содержит соответствующего элемента».

пример

```
// Returns "a":  
new[] { "a" }.First();  
  
// Returns "a":  
new[] { "a", "b" }.First();  
  
// Returns "b":  
new[] { "a", "b" }.First(x => x.Equals("b"));  
  
// Returns "ba":  
new[] { "ba", "be" }.First(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new[] { "ca", "ce" }.First(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new string[0].First();
```

Живая демонстрация на .NET скрипке

FirstOrDefault ()

- Возвращает первый элемент последовательности или первый элемент, соответствующий предоставленному `predicate`.
- Если последовательность не содержит элементов или не содержит элементов, соответствующих предоставленному `predicate`, возвращает значение по умолчанию для типа последовательности, используя `default (T)` по `default (T)`.

пример

```
// Returns "a":  
new[] { "a" }.FirstOrDefault();  
  
// Returns "a":  
new[] { "a", "b" }.FirstOrDefault();  
  
// Returns "b":  
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));  
  
// Returns "ba":  
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));  
  
// Returns null:  
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));  
  
// Returns null:  
new string[0].FirstOrDefault();
```

Живая демонстрация на .NET скрипке

Прошлой()

- Возвращает последний элемент последовательности или последний элемент, соответствующий предоставленному `predicate`.
- Если последовательность не содержит элементов, `InvalidOperationException` с сообщением «Последовательность не содержит элементов».
- Если последовательность не содержит элементов, соответствующих предоставленному `predicate`, `InvalidOperationException` с сообщением «Последовательность не содержит соответствующего элемента».

пример

```
// Returns "a":  
new[] { "a" }.Last();  
  
// Returns "b":  
new[] { "a", "b" }.Last();  
  
// Returns "a":  
new[] { "a", "b" }.Last(x => x.Equals("a"));  
  
// Returns "be":  
new[] { "ba", "be" }.Last(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new[] { "ca", "ce" }.Last(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new string[0].Last();
```

LastOrDefault ()

- Возвращает последний элемент последовательности или последний элемент, соответствующий предоставленному `predicate`.
- Если последовательность не содержит элементов или не содержит элементов, соответствующих предоставленному `predicate`, возвращает значение по умолчанию для типа последовательности, используя `default(T)` по `default(T)`.

пример

```
// Returns "a":  
new[] { "a" }.LastOrDefault();  
  
// Returns "b":  
new[] { "a", "b" }.LastOrDefault();  
  
// Returns "a":  
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));
```

```
// Returns "be":  
new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));  
  
// Returns null:  
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));  
  
// Returns null:  
new string[0].LastOrDefault();
```

Не замужем()

- Если последовательность содержит ровно один элемент или ровно один элемент, соответствующий предоставленному `predicate`, этот элемент возвращается.
- Если последовательность не содержит элементов или не содержит элементов, соответствующих предоставленному `predicate`, `InvalidOperationException` с сообщением «Последовательность не содержит элементов».
- Если последовательность содержит более одного элемента или более одного элемента, соответствующего предоставленному `predicate`, `InvalidOperationException` с сообщением «Последовательность содержит более одного элемента».
- **Примечание.** Чтобы оценить, содержит ли последовательность только один элемент, необходимо перечислить не более двух элементов.

пример

```
// Returns "a":  
new[] { "a" }.Single();  
  
// Throws InvalidOperationException because sequence contains more than one element:  
new[] { "a", "b" }.Single();  
  
// Returns "b":  
new[] { "a", "b" }.Single(x => x.Equals("b"));  
  
// Throws InvalidOperationException:  
new[] { "a", "b" }.Single(x => x.Equals("c"));  
  
// Throws InvalidOperationException:  
new string[0].Single();  
  
// Throws InvalidOperationException because sequence contains more than one element:  
new[] { "a", "a" }.Single();
```

SingleOrDefault ()

- Если последовательность содержит ровно один элемент или ровно один элемент, соответствующий предоставленному `predicate`, этот элемент возвращается.
- Если последовательность не содержит элементов или не содержит элементов, соответствующих предоставленному `predicate`, возвращается `default(T)`.

- Если последовательность содержит более одного элемента или более одного элемента, соответствующего предоставленному `predicate` , `InvalidOperationException` с сообщением «Последовательность содержит более одного элемента».
- Если последовательность не содержит элементов, соответствующих предоставленному `predicate` , возвращает значение по умолчанию для типа последовательности, используя `default(T)` ПО `default(T)` .
- **Примечание.** Чтобы оценить, содержит ли последовательность только один элемент, необходимо перечислить не более двух элементов.

пример

```
// Returns "a":  
new[] { "a" }.SingleOrDefault();  
  
// returns "a"  
new[] { "a", "b" }.SingleOrDefault(x => x == "a");  
  
// Returns null:  
new[] { "a", "b" }.SingleOrDefault(x => x == "c");  
  
// Throws InvalidOperationException:  
new[] { "a", "a" }.SingleOrDefault(x => x == "a");  
  
// Throws InvalidOperationException:  
new[] { "a", "b" }.SingleOrDefault();  
  
// Returns null:  
new string[0].SingleOrDefault();
```

рекомендации

- Хотя вы можете использовать `FirstOrDefault` , `LastOrDefault` ИЛИ `SingleOrDefault` , чтобы проверить , содержит ли последовательность какие - либо элементы, `Any` ИЛИ `Count` являются более надежными. Это связано с тем, что возвращаемое значение по `default(T)` из одного из этих трех методов не доказывает, что последовательность пуста, поскольку значение первого / последнего / единственного элемента последовательности может быть равно по `default(T)`
- Решите, какие методы больше всего подходят для вашего кода. Например, используйте `Single` только в том случае, если вы должны убедиться, что в коллекции есть один элемент, соответствующий вашему предикату, в противном случае используйте `First` ; как `Single` throw исключение, если последовательность имеет более одного соответствующего элемента. Это, конечно же, относится и к «* `OrDefault` » -конвертерам.
- Что касается эффективности: хотя часто бывает необходимо убедиться, что есть только один элемент (`Single`) или один или ноль (`SingleOrDefault`), возвращаемый

запросом, оба этих метода требуют больше, а часто и всей коллекции для проверки, чтобы гарантировать отсутствие второго ответа на запрос. Это не похоже на поведение, например, метода `First`, который может быть удовлетворен после нахождения первого совпадения.

Кроме

Метод `Except` возвращает набор элементов, которые содержатся в первой коллекции, но не содержатся во втором. По умолчанию `IEqualityComparer` используется для сравнения элементов в двух наборах. Существует перегрузка, которая воспринимает `IEqualityComparer` как аргумент.

Пример:

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

Выход:

```
1
4
```

Живая демонстрация на .NET скрипке

В этом случае `.Except(second)` исключает элементы, содержащиеся в массиве `second`, а именно 2 и 3 (0 и 5 не содержатся в `first` массиве и пропускаются).

Обратите внимание, что `Except` означает `Distinct` (т. е. Удаляет повторяющиеся элементы). Например:

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

Выход:

```
1
4
```

Живая демонстрация на .NET скрипке

В этом случае элементы 1 и 4 возвращаются только один раз.

Реализация `IEquatable` или предоставление функции `IEqualityComparer` позволит

использовать другой метод для сравнения элементов. Обратите внимание, что метод `GetHashCode` также должен быть переопределен, чтобы он возвращал идентичный хэш-код для `object` который идентичен в соответствии с реализацией `IEquatable`.

Пример с `IEquatable`:

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };

        holidayDifference = remoteHolidays
            .Except(localHolidays)
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
    }
}
```

Выход:

ханука

[Живая демонстрация на .NET скрипке](#)

SelectMany: Сглаживание последовательности последовательностей

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };
var sequence = sequenceOfSequences.SelectMany(x => x);
// returns { 1, 2, 3, 4, 5, 6 }
```

Используйте `SelectMany()` если у вас есть, или вы создаете последовательность последовательностей, но вы хотите, чтобы результат был как одна длинная последовательность.

В LINQ Query Синтаксис:

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

Если у вас есть коллекция коллекций и вы хотите одновременно работать с данными из родительской и дочерней коллекции, это также возможно с помощью `SelectMany`.

Давайте определим простые классы

```
public class BlogPost
{
    public int Id { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }
}

public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

Предположим, что мы имеем следующую коллекцию.

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
                Content = "Cool post!"
            }
        }
    },
    new BlogPost()
    {
```

```

        Id = 2,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 3,
                Content = "I don't think you're right",
            },
            new Comment()
            {
                Id = 4,
                Content = "This post is a complete nonsense"
            }
        }
    };
}

```

Теперь мы хотим выбрать комментарий Content вместе с Id BlogPost связанным с этим комментарием. Для этого мы можем использовать соответствующую перегрузку `SelectMany`.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId = post.Id, CommentContent = comment.Content });
```

Наши `commentsWithIds` выглядят так

```

{
    PostId = 1,
    CommentContent = "It's really great!"
},
{
    PostId = 1,
    CommentContent = "Cool post!"
},
{
    PostId = 2,
    CommentContent = "I don't think you're right"
},
{
    PostId = 2,
    CommentContent = "This post is a complete nonsense"
}

```

SelectMany

Метод `linq SelectMany «flattens» IEnumerable<IEnumerable<T>> в IEnumerable<T>`. Все T-элементы внутри экземпляров `IEnumerable` содержащиеся в исходном `IEnumerable` будут объединены в ОДИН `IEnumerable`.

```
var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

Если вы используете функцию селектора, которая превращает входные элементы в последовательности, результатом будут элементы тех последовательностей, которые

возвращаются один за другим.

Обратите внимание, что, в отличие от `Select()`, количество элементов на выходе не должно быть таким же, как на входе.

Более реальный пример

```
class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"} } },
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} } }
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}
```

Выход:

боб
Джек
Джим
Джон

[Живая демонстрация на .NET скрипке](#)

Все

`All` используется для проверки, если все элементы коллекции соответствуют условию или нет.

Смотри также: [.any](#)

1. Пустой параметр

`Все` : не разрешено использовать с пустыми параметрами.

2. Лямбда-выражение как параметр

All : Возвращает `true` если все элементы коллекции удовлетворяют лямбда-выражению и `false` противном случае:

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};  
bool result = numbers.All(i => i < 10); // true  
bool result = numbers.All(i => i >= 3); // false
```

3. Пустая коллекция

All : Возвращает `true` если коллекция пуста и предоставается выражение лямбда:

```
var numbers = new List<int>();  
bool result = numbers.All(i => i >= 0); // true
```

Примечание. `All` будут останавливать итерацию коллекции, как только она найдет элемент, **не** соответствующий условию. Это означает, что сбор не обязательно будет полностью перечислить; он будет только перечислить достаточно далеко, чтобы найти первый элемент, **не соответствующий** условию.

Сбор запросов по типу / литым элементам типа

```
interface IFoo { }  
class Foo : IFoo { }  
class Bar : IFoo { }
```

```
var item0 = new Foo();  
var item1 = new Foo();  
var item2 = new Bar();  
var item3 = new Bar();  
var collection = new IFoo[] { item0, item1, item2, item3 };
```

Использование `OfType`

```
var foos = collection.OfType<Foo>(); // result: IEnumarable<Foo> with item0 and item1  
var bars = collection.OfType<Bar>(); // result: IEnumarable<Bar> item item2 and item3  
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumarable<IFoo> with all four items
```

Использование `Where`

```
var foos = collection.Where(item => item is Foo); // result: IEnumarable<IFoo> with item0 and item1  
var bars = collection.Where(item => item is Bar); // result: IEnumarable<IFoo> with item2 and item3
```

Использование `Cast`

```
var bars = collection.Cast<Bar>(); // throws InvalidCastException on the 1st
```

```
item
var foos = collection.Cast<Foo>();           // throws InvalidCastException on the 3rd
item
var foosAndBars = collection.Cast<IFoo>();      // OK
```

СОЮЗ

Объединение двух коллекций для создания отдельной коллекции с использованием сопоставителя равенства по умолчанию

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);    // AllElement now contains 1,2,3,4,5
```

[Живая демонстрация на .NET скрипке](#)

JOINS

Соединения используются для объединения разных списков или таблиц, содержащих данные с помощью общего ключа.

Как и в SQL, в LINQ поддерживаются следующие типы соединений:

Внутренний, левый, правый, крест и полный внешний вид .

В приведенных ниже примерах используются следующие два списка:

```
var first = new List<string>() { "a", "b", "c" }; // Left data
var second = new List<string>() { "a", "c", "d" }; // Right data
```

(Внутреннее соединение

```
var result = from f in first
             join s in second on f equals s
             select new { f, s };

var result = first.Join(second,
                       f => f,
                       s => s,
                       (f, s) => new { f, s });

// Result: {"a", "a"}
//          {"c", "c"}
```

Левое внешнее соединение

```
var leftOuterJoin = from f in first
                     join s in second on f equals s into temp
```

```

        from t in temp.DefaultIfEmpty()
        select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
                     from s in second.Where(x => x == f).DefaultIfEmpty()
                     select new { First = f, Second = s};

// Result: {"a","a"}
//          {"b", null}
//          {"c", "c"}

// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
                                                 f => f,
                                                 s => s,
                                                 (f, s) => new { First = f, Second = s })
    .SelectMany(temp => temp.Second.DefaultIfEmpty(),
               (f, s) => new { First = f.First, Second = s });

```

Правостороннее соединение

```

var rightOuterJoin = from s in second
                      join f in first on s equals f into temp
                      from t in temp.DefaultIfEmpty()
                      select new {First=t,Second=s};

// Result: {"a","a"}
//          {"c", "c"}
//          {null,"d"}

```

Крест

```

var CrossJoin = from f in first
                 from s in second
                 select new { f, s };

// Result: {"a","a"}
//          {"a","c"}
//          {"a","d"}
//          {"b","a"}
//          {"b","c"}
//          {"b","d"}
//          {"c","a"}
//          {"c","c"}
//          {"c","d"}

```

Полная внешняя связь

```

var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}

```

```
//      {"b", null}
//      {"c", "c"}
//      {null, "d"}
```

Практический пример

Приведенные выше примеры имеют простую структуру данных, поэтому вы можете сосредоточиться на понимании различных LINQ-соединений технически, но в реальном мире у вас будут таблицы с столбцами, которые вам нужно присоединиться.

В следующем примере используется только один класс `Region`, на самом деле вы бы присоединились к двум или более различным таблицам, которые содержат один и тот же ключ (в этом примере `first` и `second` соединяются с помощью общего `ID` ключа).

Пример. Рассмотрим следующую структуру данных:

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

Теперь подготовьте данные (т.е. заполните данные):

```
// Left data
var first = new List<Region>()
    { new Region(1), new Region(3), new Region(4) };
// Right data
var second = new List<Region>()
{
    new Region(1, "Eastern"), new Region(2, "Western"),
    new Region(3, "Northern"), new Region(4, "Southern")
};
```

Вы можете видеть, что в этом примере `first` не содержит описания региона, поэтому вы хотите присоединиться к ним со `second`. Тогда внутреннее соединение будет выглядеть так:

```
// do the inner join
var result = from f in first
    join s in second on f.ID equals s.ID
    select new { f.ID, s.RegionDescription };

// Result: {1,"Eastern"}
//          {3, Northern}
//          {4, "Southern"}
```

Этот результат создал анонимные объекты «на лету», и это хорошо, но мы уже создали правильный класс, поэтому мы можем указать его: вместо `select new { f.ID, s.RegionDescription };` мы можем сказать, `select new Region(f.ID, s.RegionDescription);`, который вернет те же данные, но создаст объекты типа `Region` -, которые будут поддерживать совместимость с другими объектами.

Живая демонстрация на .NET скрипке

отчетливый

Возвращает уникальные значения из `IEnumerable`. Уникальность определяется с помощью сопоставления равенства по умолчанию.

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

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

Чтобы сравнить пользовательский тип данных, нам нужно реализовать интерфейс `IEquatable<T>` и предоставить методы `GetHashCode` и `Equals` для типа. Или сопоставитель равенства может быть переопределен:

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

GroupBy одно или несколько полей

Предположим, что у нас есть модель фильма:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Категория по категориям:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberofFilmsInCategory = grp.Count();
}
```

Группа по категориям и годам:

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberOfFilmsInCategory = grp.Count();
}
```

Использование Range с различными методами Linq

Вы можете использовать класс Enumerable вместе с запросами Linq, чтобы преобразовать для циклов в Linq один лайнер.

Выберите пример

Против этого:

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

Вы можете сделать это:

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

Где пример

В этом примере будет создано 100 номеров, и даже те будут извлечены

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

Запрос заказа - OrderBy () ThenBy () OrderByDescending () ThenByDescending ()

```
string[] names = { "mark", "steve", "adam" };
```

По возрастанию:

Синтаксис запроса

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

Синтаксис метода

```
var sortedNames = names.OrderBy(name => name);
```

sortedNames содержит имена в следующем порядке: «adam», «mark», «steve»

По убыванию:

Синтаксис запроса

```
var sortedNames =  
    from name in names  
    orderby name descending  
    select name;
```

Синтаксис метода

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNames содержит имена в следующем порядке: «steve», «mark», «adam»

Заказ по нескольким полям

```
Person[] people =  
{  
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},  
    new Person { FirstName = "Phil", LastName = "Collins", Age = 28},  
    new Person { FirstName = "Adam", LastName = "Ackerman", Age = 29},  
    new Person { FirstName = "Adam", LastName = "Ackerman", Age = 15}  
};
```

Синтаксис запроса

```
var sortedPeople = from person in people  
                   orderby person.LastName, person.FirstName, person.Age descending  
                   select person;
```

Синтаксис метода

```
sortedPeople = people.OrderBy(person => person.LastName)  
                  .ThenBy(person => person.FirstName)  
                  .ThenByDescending(person => person.Age);
```

Результат

1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins 28
4. Steve Collins 30

ОСНОВЫ

LINQ в значительной степени полезен для запросов к коллекциям (или массивам).

Например, учитывая следующие примеры данных:

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true },
    new Student { Name = "Bob", Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true },
    new Student { Name = "Greg", Grade = 90, HasSnack = false },
    new Student { Name = "Joe", Grade = 59, HasSnack = false }
}
```

Мы можем «запросить» эти данные с помощью синтаксиса LINQ. Например, чтобы получить всех студентов, которые сегодня перекусывают:

```
var studentsWithSnacks = from s in classroom.Students
                           where s.HasSnack
                           select s;
```

Или, чтобы получить студентов со степенью 90 или выше и только вернуть их имена, а не полный объект `Student`:

```
var topStudentNames = from s in classroom.Students
                           where s.Grade >= 90
                           select s.Name;
```

Функция LINQ состоит из двух синтаксисов, которые выполняют одни и те же функции, имеют почти идентичную производительность, но написаны очень по-разному. Синтаксис в приведенном выше примере называется **синтаксисом запроса**. Следующий пример, однако, иллюстрирует **синтаксис метода**. Те же данные будут возвращены, как в приведенном выше примере, но способ записи запроса отличается.

```
var topStudentNames = classroom.Students
                           .Where(s => s.Grade >= 90)
                           .Select(s => s.Name);
```

Группа по

`GroupBy` - это простой способ сортировки коллекции элементов `IEnumerable<T>` в отдельные группы.

Простой пример

В этом первом примере мы получаем две группы, нечетные и четные элементы.

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items
```

```
foreach(var group in grouped)
{
    foreach (int item in group)
    {
        Console.WriteLine(item); // 135792468 (first odd then even)
    }
}
```

Более сложный пример

Давайте возьмем в качестве примера список людей по возрасту. Во-первых, мы создадим объект Person, который имеет два свойства: имя и возраст.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Затем мы создаем наш образец списка людей с разными именами и возрастом.

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

Затем мы создаем запрос LINQ для группировки списка людей по возрасту.

```
var query = people.GroupBy(x => x.Age);
```

Поступая таким образом, мы можем видеть возраст для каждой группы и иметь список каждого человека в группе.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

Это приводит к следующему результату:

```
20
Mouse
30
Neo
Trinity
40
```

```
Morpheus
Dozer
Smith
```

Вы можете играть с [живой демонстрацией на .NET Fiddle](#)

любой

Any используется для проверки того, соответствует ли **какой-либо** элемент коллекции условию или нет.

Смотри также: [Все](#), [Любой](#) и [FirstOrDefault: лучшая практика](#)

1. Пустой параметр

Any : Возвращает `true` если коллекция имеет какие-либо элементы и `false` если коллекция пуста:

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); //true
```

2. Лямбда-выражение как параметр

Any : Возвращает `true` если коллекция имеет один или несколько элементов, удовлетворяющих условию в выражении лямбда:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a"); // true
arrayOfStrings.Any(item => item == "d"); // false
```

3. Пустая коллекция

Any : Возвращает `false` если коллекция пуста и предоставляется выражение лямбда:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

Примечание. `Any` остановит итерацию коллекции, как только найдет элемент, соответствующий условию. Это означает, что сбор не обязательно будет полностью перечислить; он будет только перечислить достаточно далеко, чтобы найти первый элемент, соответствующий условию.

[Живая демонстрация на .NET скрипке](#)

ToDictionary

Метод `ToDictionary()` LINQ может использоваться для создания коллекции `Dictionary<TKey, TElement>` на основе данного источника `IEnumerable<T>`.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

В этом примере единственный аргумент, переданный `ToDictionary` имеет тип `Func<TSource, TKey>`, который возвращает ключ для каждого элемента.

Это краткий способ выполнить следующую операцию:

```
Dictionary<int, User> usersById = new Dictionary<int, User>();
foreach (User u in users)
{
    usersById.Add(u.Id, u);
}
```

Вы также можете передать второй параметр методу `ToDictionary`, который имеет тип `Func<TSource, TElement>` и возвращает `Value` добавляемое для каждой записи.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

Также можно указать `IComparer` который используется для сравнения значений ключа. Это может быть полезно, когда ключ является строкой, и вы хотите, чтобы он соответствовал нечувствительности к регистру.

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsensitiveName = users.ToDictionary(x => x.Name,
StringComparer.InvariantCultureIgnoreCase);

var user1 = usersByCaseInsensitiveName["john"];
var user2 = usersByCaseInsensitiveName["JOHN"];
user1 == user2; // Returns true
```

Примечание. Метод `ToDictionary` требует, чтобы все ключи были уникальными, не должно быть дубликатов ключей. Если есть, то возникает исключение: `ArgumentException: An item with the same key has already been added.` Если у вас есть сценарий, в котором вы знаете, что у вас будет несколько элементов с одним и тем же ключом, тогда вам лучше использовать `ToLookup`.

заполнитель

`Aggregate` Применяет функцию аккумулятора по последовательности.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
int sum = intList.Aggregate((prevSum, current) => prevSum + current);  
// sum = 55
```

- На первом этапе `prevSum = 1`
- На втором `prevSum = prevSum(at the first step) + 2`
- На `i-м шаге` `prevSum = prevSum(at the (i-1) step) + i-th element of the array`

```
string[] stringList = { "Hello", "World", "!" };  
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);  
// joinedString = "Hello World!"
```

Вторая перегрузка `Aggregate` также получает параметр `seed` который является начальным значением аккумулятора. Это можно использовать для вычисления нескольких условий в коллекции без повторения его более одного раза.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Для сбора `items` мы хотим рассчитать

1. Общее количество `.Count`
2. Количество четных чисел
3. Соберите каждый четвертый пункт

Используя `Aggregate` это можно сделать следующим образом:

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },  
    (accumelative, item) =>  
    new {  
        Total = accumelative.Total + 1,  
        Even = accumelative.Even + (item % 2 == 0 ? 1 : 0),  
        FourthItems = (accumelative.Total + 1)%4 == 0 ?  
            new List<int>(accumelative.FourthItems) { item } :  
            accumelative.FourthItems  
    });  
// Result:  
// Total = 12  
// Even = 6  
// FourthItems = [4, 8, 12]
```

Обратите внимание, что использование анонимного типа в качестве семени должно создавать новый объект для каждого элемента, потому что свойства доступны только для чтения. Используя пользовательский класс, вы можете просто назначить информацию, а `new` не нужен (только при задании начального параметра `seed`)

Определение переменной внутри запроса Linq (путь ключевое слово)

Чтобы определить переменную внутри выражения `linq`, вы можете использовать ключевое слово `let`. Обычно это делается для хранения результатов промежуточных подзапросов,

например:

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var aboveAverages = from number in numbers
    let average = numbers.Average()
    let nSquared = Math.Pow(number,2)
    where nSquared > average
    select number;

Console.WriteLine("The average of the numbers is {0}.", numbers.Average());

foreach (int n in aboveAverages)
{
    Console.WriteLine("Query result includes number {0} with square of {1}.", n,
Math.Pow(n,2));
}
```

Выход:

Среднее число - 4,5.

Результат запроса включает номер 3 с квадратом 9.

Результат запроса включает номер 4 с квадратом 16.

Результат запроса включает номер 5 с квадратом 25.

Результат запроса включает номер 6 с квадратом 36.

Результат запроса включает номер 7 с квадратом 49.

Результат запроса включает в себя номер 8 с квадратом 64.

Результат запроса включает номер 9 с квадратом 81.

[Посмотреть демо](#)

SkipWhile

`SkipWhile()` используется для исключения элементов до первого несоответствия (это может быть интуитивно понятным для большинства)

```
int[] list = { 42, 42, 6, 6, 6, 42 };
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

DefaultIfEmpty

`DefaultIfEmpty` используется для возврата элемента по умолчанию, если последовательность не содержит элементов. Этот Элемент может быть По умолчанию Тип или пользовательского экземпляра этого Типа. Пример:

```
var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";
```

```

chars.Where(str => str.Length > 1)
    .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
    .DefaultIfEmpty().First(); // returns null;

```

Использование в левых соединениях :

С `DefaultIfEmpty` традиционная Linq `Join` может вернуть объект по умолчанию, если совпадение не найдено. Таким образом, выступая в качестве левого соединения SQL. Пример:

```

var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
                     select new
                     {
                         Number = l,
                         Character = result
                     };

foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

output:

```

```

Num = 99      Char = c
Num = 100     Char = d
Num = 5       Char = ?
Num = 20      Char = ?
Num = 102     Char = ?
Num = 105     Char = i

```

В случае, когда используется `DefaultIfEmpty` (без указания значения по умолчанию), и в результате не будет соответствующих элементов в правой последовательности, вы должны убедиться, что объект не имеет `null` до доступа к его свойствам. В противном случае это приведет к `NullReferenceException`. Пример:

```

var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
{
    new { Value = 1 },
    new { Value = 2 },
    new { Value = 3 },
    new { Value = 4 },
};

var numbersAsChars = (from l in leftSequence

```

```
join r in rightSequence
on l equals r.Value into leftJoin
from result in leftJoin.DefaultIfEmpty()
select new
{
    Left = l,
    // 5 will not have a matching object in the right so result
    // will be equal to null.
    // To avoid an error use:
    // - C# 6.0 or above - ?.
    // - Under           - result == null ? 0 : result.Value
    Right = result?.Value
}).ToList();
```

SequenceEqual

`SequenceEqual` используется для сравнения двух последовательностей `IEnumerable<T>` друг с другом.

```
int[] a = new int[] {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[] {1, 3, 2};

bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

Count и LongCount

`Count` возвращает количество элементов в `IEnumerable<T>`. `Count` также предоставляет необязательный параметр предиката, который позволяет вам фильтровать элементы, которые вы хотите подсчитать.

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

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

`LongCount` работает так же, как и `Count` но имеет тип возврата `long` и используется для подсчета последовательностей `IEnumerable<T>`, которые длиннее `int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater
than 100
```

Поэтапное построение запроса

Поскольку LINQ использует **отложенное выполнение**, мы можем иметь объект запроса, который фактически не содержит значений, но возвращает значения при оценке. Таким образом, мы можем динамически строить запрос на основе нашего потока управления и

оценивать его, как только мы закончим:

```
IEnumerable<VehicleModel> BuildQuery(int vehicleType, SearchModel search, int start = 1, int count = -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == vehicleType)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
}
```

Мы можем условно применять фильтры:

```
if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
    query = query.Where(v => search.Years.Contains(v.Year));

if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
    query = query.Where(v => search.Makes.Contains(v.Make));
}

if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
    query = query.Where(v => search.Models.Contains(v.Model));
}

if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
    decimal minCylinders = 0;
    decimal maxCylinders = 0;
    switch (search.Cylinders) {
        case "2-4":
            maxCylinders = 4;
            break;
        case "5-6":
            minCylinders = 5;
            maxCylinders = 6;
            break;
        case "8":
            minCylinders = 8;
            maxCylinders = 8;
            break;
        case "10+":
            minCylinders = 10;
            break;
    }
    if (minCylinders > 0) {
        query = query.Where(v => v.Cylinders >= minCylinders);
    }
    if (maxCylinders > 0) {
        query = query.Where(v => v.Cylinders <= maxCylinders);
    }
}
```

Мы можем добавить запрос сортировки к запросу на основе условия:

```
switch (search.SortingColumn.ToLower()) {
```

```

        case "make_model":
            query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
            break;
        case "year":
            query = query.OrderBy(v => v.Year);
            break;
        case "engine_size":
            query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
            break;
        default:
            query = query.OrderBy(v => v.Year); //The default sorting.
    }
}

```

Наш запрос может быть определен для начала из заданной точки:

```
query = query.Skip(start - 1);
```

и определен для возврата определенного количества записей:

```

if (count > -1) {
    query = query.Take(count);
}
return query;
}

```

Когда у нас есть объект запроса, мы можем оценить результаты с помощью цикла `foreach` или одного из методов LINQ, который возвращает набор значений, таких как `ToList` или `ToArray`:

```

searchModel sm;

// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();

```

застежка-молния

Метод расширения `Zip` действует на две коллекции. Он объединяет каждый элемент в двух сериях в зависимости от положения. С помощью экземпляра `Func` мы используем `Zip` для обработки элементов из двух коллекций С # в парах. Если серия отличается по размеру, дополнительные элементы большей серии будут игнорироваться.

Чтобы взять пример из книги «C # в двух словах»,

```

int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);

```

Выход:

3 = три

5 = пять

7 = семь

[Посмотреть демо](#)

GroupJoin с переменной внешней дальностью

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };

```

ElementAt и ElementAtOrDefault

`ElementAt` вернет элемент с индексом `n`. Если `n` не входит в диапазон перечислимого, генерируется `ArgumentOutOfRangeException`.

```
int[] numbers = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2); // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

`ElementAtOrDefault` вернет элемент с индексом `n`. Если `n` не входит в диапазон перечислимого, возвращает значение по `default(T)`.

```
int[] numbers = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2); // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

И `ElementAt` и `ElementAtOrDefault` оптимизированы для того, когда источником является `IList<T>` и нормальная индексация будет использоваться в этих случаях.

Обратите внимание, что для `ElementAt`, если предоставленный индекс больше, чем размер `IList<T>`, список должен (но технически не гарантирован) генерировать `ArgumentOutOfRangeException`.

Линк-квантификаторы

Операции квантора возвращают логическое значение, если некоторые или все элементы в последовательности удовлетворяют условию. В этой статье мы увидим некоторые общие сценарии LINQ to Objects, где мы можем использовать эти операторы. В LINQ можно

использовать 3 операции Quantifiers:

`All` - используется для определения того, удовлетворяют ли все элементы в последовательности условию. Например:

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

`Any` - используется для определения того, удовлетворяют ли какие-либо элементы в последовательности условию. Например:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

`Contains` - используется для определения того, содержит ли последовательность указанный элемент. Например:

```
//for int array
int[] query =new int[] { 1,2,3 };
query.Contains(1);

//for string array
string[] query={"Tom", "grey"};
query.Contains("Tom");

//for a string
var stringValue="hello";
stringValue.Contains("h");
```

Объединение нескольких последовательностей

Рассмотрим объекты `Customer` , `Purchase` and `PurchaseItem` следующим образом:

```
public class Customer
{
    public string Id { get; set } // A unique Id that identifies customer
    public string Name { get; set; }
}

public class Purchase
{
    public string Id { get; set }
    public string CustomerId { get; set; }
    public string Description { get; set; }
}
```

```

public class PurchaseItem
{
    public string Id { get; set; }
    public string PurchaseId { get; set; }
    public string Detail { get; set; }
}

```

Рассмотрим следующие данные выборки для вышеуказанных объектов:

```

var customers = new List<Customer>()
{
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
};

var purchases = new List<Purchase>()
{
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase1"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase2"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase1"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
};

var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId = purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId = purchases[1].Id,
        Detail = "Purchase1-PurchaseItem2"
    }
};

```

```

        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};

```

Теперь рассмотрим ниже запрос linq:

```

var result = from c in customers
             join p in purchases on c.Id equals p.CustomerId           // first join
             join pi in purchaseItems on p.Id equals pi.PurchaseId    // second join
             select new
             {
                 c.Name, p.Description, pi.Detail
             };

```

Чтобы вывести результат вышеуказанного запроса:

```

foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}

```

Результатом запроса будет:

Customer1, Customer1-Purchase1, Purchase1-PurchaseItem1

Customer1, Customer1-Purchase2, Purchase2-PurchaseItem1

Customer1, Customer1-Purchase2, Purchase2-PurchaseItem2

Customer2, Customer2-Purchase2, Purchase3-PurchaseItem1

[Живая демонстрация на .NET скрипке](#)

Присоединение к нескольким клавишам

```

 PropertyInfo[] stringProps = typeof (string).GetProperties(); //string properties
 PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties(); //stringbuilder properties

 var query =
    from s in stringProps

```

```

join b in builderProps
    on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
select new
{
    s.Name,
    s.PropertyType,
    StringTokenizer = s.MetadataToken,
    StringBuilderToken = b.MetadataToken
};

```

Обратите внимание, что анонимные типы в `join` выше должны содержать одинаковые свойства, поскольку объекты считаются равными, только если все их свойства равны. В противном случае запрос не будет компилироваться.

Выбрать с помощью Func selector - Использовать для ранжирования элементов

В случае перегрузок методов `Select` также передается `index` текущего элемента в `select` коллекции. Это несколько его применений.

Получите «номер строки» элементов

```

var rowNumbers = collection.OrderBy(item => item.Property1)
    .ThenBy(item => item.Property2)
    .ThenByDescending(item => item.Property3)
    .Select((item, index) => new { Item = item, RowNumber = index })
    .ToList();

```

Получить ранг элемента в своей группе

```

var rankInGroup = collection.GroupBy(item => item.Property1)
    .OrderBy(group => group.Key)
    .SelectMany(group => group.OrderBy(item => item.Property2)
        .ThenByDescending(item => item.Property3)
        .Select((item, index) => new
        {
            Item = item,
            RankInGroup = index
        })).ToList();

```

Получить рейтинг групп (также известный в Oracle как `dense_rank`)

```

var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
    .OrderBy(group => group.Key)
    .Select((group, index) => new
    {
        Items = group,
        Rank = index
    })
    .SelectMany(v => v.Items, (s, i) => new
    {
        Item = i,
        DenseRank = s.Rank
    });

```

```
}).ToList();
```

Для тестирования вы можете использовать:

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

И данные:

```
List<SomeObject> collection = new List<SomeObject>
{
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1 },
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1 },
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2 },
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1 },
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1 },
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1 },
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1 }
};
```

TakeWhile

TakeWhile возвращает элементы из последовательности, если условие истинно

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

сумма

Метод расширения `Enumerable.Sum` вычисляет сумму числовых значений.

В случае, если элементы коллекции сами являются числами, вы можете рассчитать сумму напрямую.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

Если тип элементов является сложным типом, вы можете использовать выражение лямбда для указания значения, которое должно быть рассчитано:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Метод расширения суммы может вычисляться со следующими типами:

- Int32
- Int64
- не замужем
- двойной
- Десятичный

Если ваша коллекция содержит типы с нулевым значением, вы можете использовать оператор null-coalescing для установки значения по умолчанию для нулевых элементов:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

ToLookup

ToLookup возвращает структуру данных, которая позволяет индексировать. Это метод расширения. Он создает экземпляр ILookup, который может быть проиндексирован или перечислен с использованием цикла foreach. Записи объединяются в группы по каждому ключу. - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(",", lookup[3]));
//output: one,two
```

Другой пример:

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(",", lookup[0]));
//output: 2,4,6,8

//print odd numbers after joining
Console.WriteLine(string.Join(",", lookup[1]));
//output: 1,3,5,7
```

Создайте собственные операторы Linq для IEnumerable

Одна из замечательных особенностей Linq заключается в том, что ее так легко расширить. Вам просто нужно создать [метод расширения](#), аргументом которого является `IEnumerable<T>`.

```

public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}

```

Этот пример разбивает элементы в `IEnumerable<T>` на списки фиксированного размера, последний список содержит оставшиеся элементы. Обратите внимание, как объект, к которому применяется метод расширения, передается в (`source` аргумента) в качестве исходного аргумента с использованием `this` ключевого слова. Затем ключевое слово `yield` используется для вывода следующего элемента в выходном `IEnumerable<T>` прежде чем продолжить выполнение с этой точки (см. [Ключевое слово yield](#)).

Этот пример будет использоваться в вашем коде следующим образом:

```

//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}

```

В первом цикле под список будет `{2, 3, 4}` и вторым `{5, 6}`.

Пользовательские методы Linq можно комбинировать со стандартными методами Linq. например:

```

//using MyNamespace;
var result = Enumerable.Range(0, 13)           // generate a list
    .Where(x => x%2 == 0) // filter the list or do something other
    .Batch(3)             // call our extension method
    .ToList()              // call other standard methods

```

Этот запрос вернет четные числа, сгруппированные партиями с размером 3: `{0, 2, 4}, {6, 8, 10}, {12}`

Помните, что вам нужно `using MyNamespace;` чтобы получить доступ к методу расширения.

Использование SelectMany вместо вложенных циклов

Учитывая 2 списка

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

если вы хотите вывести все перестановки, вы можете использовать вложенные циклы, например

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

Используя SelectMany, вы можете выполнить ту же операцию, что и

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Любая и первая (OrDefault) - лучшая практика

Я не буду объяснять, что делает Any и FirstOrDefault, потому что в них уже есть два хороших примера. Для получения дополнительной информации см. Раздел « [Все и первый](#)», « [FirstOrDefault](#)», « [Last](#)», « [LastOrDefault](#)», « [Single](#)» и « [SingleOrDefault](#) » .

Я часто вижу код, который **следует избегать** ,

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

Его можно было бы написать более эффективно

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //Do stuff
}
```

Используя второй пример, коллекция выполняется только один раз и дает тот же результат, что и первый. Та же идея может быть применена к Single .

GroupBy Sum и Count

Возьмем образец класса:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Теперь рассмотрим список транзакций:

```
var transactions = new List<Transaction>
{
    new Transaction { Category = "Saving Account", Amount = 56, Date =
DateTime.Today.AddDays(1) },
    new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-
10) },
    new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1) },
    new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
    new Transaction { Category = "Current Account", Amount = 100, Date =
DateTime.Today.AddDays(5) },
};
```

Если вы хотите рассчитать разумную сумму и количество баллов, вы можете использовать GroupBy следующим образом:

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
    .Select(t => new
    {
        Category = t.Key,
        Count = t.Count(),
        Amount = t.Sum(ta => ta.Amount),
    }).ToList();

Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
    row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

Кроме того, вы можете сделать это за один шаг:

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
    var transactionArray = t as Transaction[] ?? t.ToArray();
    return new
    {
        Category = key,
        Count = transactionArray.Length,
        Amount = transactionArray.Sum(ta => ta.Amount),
    };
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
    row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

Вывод для обоих вышеперечисленных запросов будет таким же:

Категория: Сберегательный счет, Сумма: 66, Количество: 2

Категория: Кредитная карточка, Сумма: 71, Количество: 2

Категория: Расчетный счет, Сумма: 100, Количество: 1

[Демо-версия в .NET Fiddle](#)

Задний ход

- Инвертирует порядок элементов в последовательности.
- Если нет элементов, вызывается `ArgumentNullException: source is null.`

Пример:

```
// Create an array.  
int[] array = { 1, 2, 3, 4 }; //Output:  
// Call reverse extension method on the array. //4  
var reverse = array.Reverse(); //3  
// Write contents of array to screen. //2  
foreach (int value in reverse) //1  
    Console.WriteLine(value);
```

Пример живого кода

Помните, что `Reverse()` может работать в зависимости от последовательности цепочек ваших операторов LINQ.

```
//Create List of chars  
List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };  
  
//Reversing the list then taking the two first elements  
IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);  
  
//Taking 2 elements and then reversing only thos two  
IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();  
  
//reverseFirst output: 6, 5  
//reverseLast output: 2, 1
```

Пример живого кода

`Reverse()` работает путем буферизации всего, а затем проходит через него назад, which не очень эффективен, но ни один `OrderBy` с этой точки зрения.

В LINQ-to-Objects выполняются операции буферизации (`Reverse`, `OrderBy`, `GroupBy` и т. д.) И операции без буферизации (`Where`, `Take`, `Skip` и т. д.).

Пример: Non-buffering Обратное расширение

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

Пример живого кода

Этот метод может столкнуться с проблемами, если и мутировать список во время итерации.

Перечисление Перечислимого

Интерфейс `IEnumerable <T>` является базовым интерфейсом для всех родовых счетчиков и является квинтэссенцией части понимания LINQ. По своей сути он представляет последовательность.

Этот базовый интерфейс наследуется всеми общими наборами, такими как `Collection <T>` , `Array` , `List <T>` , `Dictionary <TKey, TValue>` `Class` и `HashSet <T>` .

В дополнение к представлению последовательности любой класс, наследующий от `IEnumerable <T>`, должен предоставить `IEnumerator <T>`. Перечислитель предоставляет итератор для перечислимого, и эти два взаимосвязанных интерфейса и идеи являются источником высказывания «перечислять перечислимое».

«Перечисление перечислимого» - важная фраза. Перечислимый - это просто структура для итерации, она не содержит никаких материализованных объектов. Например, при сортировке перечислимый может содержать критерии поля для сортировки, но использование `.OrderBy()` само по себе возвращает `IEnumerable <T>`, который знает только, как сортировать. Использование вызова, который материализует объекты, как итерации набора, называется перечислением (например `.ToList()`). Процесс перечисления будет использовать перечисляемое определение того, как перемещаться по серии и возвращать соответствующие объекты (в порядке, отфильтровать, проецировать и т. д.).

Только после перечислимого перечисления он вызывает материализацию объектов, когда метрики, такие как [временная сложность](#) (как долго это должно относиться к размеру сериала) и пространственная сложность (объем пространства, который он должен использовать, связанный с размером рядов), могут быть измеренным.

Создание собственного класса, наследующего от `IEnumerable <T>`, может быть немного сложным в зависимости от базовой серии, которая должна быть перечислимой. В общем, лучше всего использовать одну из существующих коллекций. Тем не менее, также можно наследовать от интерфейса `IEnumerable <T>`, не имея определенного массива в качестве базовой структуры.

Например, использование серии Фибоначчи в качестве базовой последовательности. Обратите внимание, что вызов `Where` просто строит `IEnumerable` , и только до тех пор, пока не

будет вызван вызов для перечисления того, что перечисляемое сделано, чтобы было реализовано какое-либо из значений.

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 1000000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First()); //smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }

    //Enumerable called typically from linq
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

Выход

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

Сила во втором наборе (fibMod612) заключается в том, что, хотя мы сделали вызов упорядочить весь наш набор чисел Фибоначчи, поскольку только одно значение было

принято с использованием `.First()` сложность времени была $O(n)$, поскольку только 1 значение необходимо было сравнить при выполнении алгоритма упорядочения. Это связано с тем, что наш счетчик запрашивал только 1 значение, поэтому весь перечислимый материал не должен был быть реализован. Если бы мы использовали `.Take(5)` вместо `.First()` перечислитель запросил бы 5 значений, и не более 5 значений должны были быть реализованы. По сравнению с необходимостью заказа всего набора, *а затем* принимать первые 5 значений, принцип сохраняет много времени и пространства выполнения.

Сортировать по

Заказывает коллекцию по заданному значению.

Когда значение представляет собой **целое число**, **double** или **float** начинается с **минимального значения**, а это означает, что вы сначала получаете отрицательные значения, чем ноль, а послесловия - положительные значения (см. Пример 1).

Когда вы заказываете **char**, метод сравнивает **значения ascii** символов для сортировки коллекции (см. Пример 2).

Когда вы сортируете **строки**, метод `OrderBy` сравнивает их, просматривая их `CultureInfo`, но нормально, начиная с *первой буквы* в алфавите (a, b, c ...).

Такой порядок называется восходящим, если вы хотите, чтобы он был наоборот, вам нужно спуститься (см. `OrderByDescending`).

Пример 1:

```
int[] numbers = {2, 1, 0, -1, -2};  
IEnumerable<int> ascending = numbers.OrderBy(x => x);  
// returns {-2, -1, 0, 1, 2}
```

Пример 2:

```
char[] letters = { ' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',  
'Z'};  
IEnumerable<char> ascending = letters.OrderBy(x => x);  
// returns { ' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'z', '[', '{', 'a', 'b', 'y', 'z', '}' }
```

Пример:

```
class Person  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
}  
  
var people = new[]  
{  
    new Person {Name = "Alice", Age = 25},
```

```

    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};

var youngestPerson = people.OrderBy(x => x.Age).First();
var name = youngestPerson.Name; // Bob

```

OrderByDescending

Заказывает коллекцию по заданному значению.

Когда значение представляет собой **целое число** , **double** или **float** начинается с **максимального значения** , что означает, что вы сначала получаете положительные значения, а не 0, а послесловия - отрицательные значения (см. Пример 1).

Когда вы заказываете **char**, метод сравнивает **значения ascii** символов для сортировки коллекции (см. Пример 2).

Когда вы сортируете **строки**, метод OrderBy сравнивает их, рассматривая их [CultureInfo](#), но normally, начиная с *последней буквы* в алфавите (z, y, x, ...).

Такой порядок называется **нисходящим**, если вы хотите его наоборот, вам нужно подняться (см. OrderBy).

Пример 1:

```

int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}

```

Пример 2:

```

char[] letters = { ' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z' };
IEnumerable<char> descending = letters.OrderByDescending(x => x);
// returns { '{', 'z', 'y', 'b', 'a', '[', 'Z', 'Y', 'B', 'A', '?', '9', '1', '+', '!', ' ' }

```

Пример 3:

```

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol

```

Concat

Объединяет две коллекции (без удаления дубликатов)

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

Содержит

MSDN:

Определяет, содержит ли последовательность указанный элемент, используя указанный `IEqualityComparer<T>`

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
var result1 = numbers.Contains(4); // true
var result2 = numbers.Contains(8); // false

List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };
// Note that can use the Intersect method in this case
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true
only for 4,5
```

Использование пользовательского объекта:

```
public class Person
{
    public string Name { get; set; }
}

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki" },
    new Person { Name = "Gilad" },
    new Person { Name = "Phil" },
    new Person { Name = "John" }
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true
```

Использование перегрузки `Enumerable.Contains(value, comparer)`:

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
```

```
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

Умное использование Contains заключается в замене нескольких предложений if на вызов Contains .

Поэтому вместо этого:

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

Сделай это:

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

Прочтайте Запросы LINQ онлайн: <https://riptutorial.com/ru/csharp/topic/68/запросы-linq>

глава 60: Идентификация ASP.NET

Вступление

Учебные пособия, касающиеся asp.net Identity, такие как управление пользователями, управление ролью, создание токенов и многое другое.

Examples

Как реализовать токен сброса пароля в идентификаторе asp.net с помощью диспетчера пользователей.

1. Создайте новую папку под названием MyClasses и создайте и добавьте следующий класс

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
        Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
        ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. Настройте свой класс идентификации

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com",
    "(any subject here)"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. Добавьте свои учетные данные в web.config. Я не использовал gmail в этой части,

потому что использование gmail заблокировано на моем рабочем месте, и оно все еще отлично работает.

```
<add key="GmailUserName" value="youremail@yourdomain.com"/>
<add key="GmailPassword" value="yourPassword"/>
<add key="GmailHost" value="yourServer"/>
<add key="GmailPort" value="yourPort"/>
<add key="GmailSsl" value="chooseTrueOrFalse"/>
<!--Smtp Server (confirmations emails)-->
```

4. Внесите необходимые изменения в Контроллер учетной записи. Добавьте следующий выделенный код.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();

        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext())))
        {
            _context = new ApplicationDbContext();
        }

        public AccountController(UserManager<ApplicationUser> userManager, ApplicationSignInManager signInManager)
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                provider.Create("EmailConfirmation"));
        }

        private AccountController(UserManager<ApplicationUser> userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                provider.Create("EmailConfirmation"));
        }

        public UserManager<ApplicationUser> UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>();
            }
            private set
            {
                _signInManager = value;
            }
        }
    }
}
```

```
//  
// GET: /Account/ForgotPassword  
[AllowAnonymous]  
public ActionResult ForgotPassword()  
{  
    return View();  
}  
  
//  
// POST: /Account/ForgotPassword  
[HttpPost]  
[AllowAnonymous]  
[ValidateAntiForgeryToken]  
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)  
{  
    if (ModelState.IsValid)  
    {  
  
        var user = await UserManager.FindByNameAsync(model.UserName);  
  
        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))  
        {  
            // Don't reveal that the user does not exist or is not confirmed  
            return View("ForgotPasswordConfirmation");  
        }  
  
        // For more information on how to enable account confirmation and password reset please visit http://go.microsoft.com/fwlink/?LinkID=532715  
        // Send an email with this link  
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);  
        code = HttpUtility.UrlEncode(code);  
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility.UrlEncode(code) });  
        await EmailService.SendAsync(new IdentityMessage  
        {  
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",  
            Destination = user.Email,  
            Subject = "Reset Password"  
        });  
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicking  
        return RedirectToAction("ForgotPasswordConfirmation", "Account");  
    }  
  
    // If we got this far, something failed, redisplay form  
    return View(model);  
}
```

Затем выполните компиляцию. Ура!

Прочтайте Идентификация ASP.NET онлайн: <https://riptutorial.com/ru/csharp/topic/9577/идентификация-asp-net>

глава 61: Именованные аргументы

Examples

Именованные аргументы могут сделать ваш код более понятным

Рассмотрим этот простой класс:

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
    {
        // Some code
    }
}
```

До C # 3.0 это было:

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

вы можете сделать этот метод более понятным с помощью **именованных аргументов** :

```
var result = SmsUtil.SendMessage(
    from: "Mehran",
    to: "Maryam",
    message: "Hello there!",
    retryCount: 12,
    attachment: null);
```

Именованные аргументы и необязательные параметры

Вы можете комбинировать именованные аргументы с необязательными параметрами.

Давайте посмотрим на этот метод:

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
        // Some code
    }
}
```

Если вы хотите вызвать этот метод *без* аргумента `retryCount` :

```
var result = SmsUtil.SendMessage(
    from: "Cihan",
```

```
to      : "Yakar",
message : "Hello there!",
attachment : new object());
```

Аргумент не нужен

Вы можете разместить именованные аргументы в любом порядке.

Пример метода:

```
public static string Sample(string left, string right)
{
    return string.Join("-",left,right);
}
```

Образец вызова:

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

Результаты:

```
A-B
B-A
```

Именованные аргументы позволяют избежать ошибок по необязательным параметрам

Всегда используйте Named Arguments для дополнительных параметров, чтобы избежать возможных ошибок при изменении метода.

```
class Employee
{
    public string Name { get; private set; }

    public string Title { get; set; }

    public Employee(string name = "<No Name>", string title = "<No Title>")
    {
        this.Name = name;
        this.Title = title;
    }
}

var jack = new Employee("Jack", "Associate"); //bad practice in this line
```

Вышеприведенный код компилируется и работает normally, пока конструктор не изменится, как:

```
//Evil Code: add optional parameters between existing optional parameters
```

```
public Employee(string name = "<No Name>", string department = "intern", string title = "<No Title>")
{
    this.Name = name;
    this.Department = department;
    this.Title = title;
}

//the below code still compiles, but now "Associate" is an argument of "department"
var jack = new Employee("Jack", "Associate");
```

Лучшая практика, чтобы избежать ошибок, когда «КТО-ТО В КОМАНДЕ» допустил ошибки:

```
var jack = new Employee(name: "Jack", title: "Associate");
```

Прочтайте Именованные аргументы онлайн: <https://riptutorial.com/ru/csharp/topic/2076/именованные-аргументы>

глава 62: Именованные и необязательные аргументы

замечания

Именованные аргументы

Ссылка: аргументы с именами *MSDN* позволяют указать аргумент для определенного параметра, связав аргумент с именем параметра, а не с позицией параметра в списке параметров.

Как сказано в *MSDN*, именованный аргумент,

- Позволяет передавать аргумент функции, связывая имя параметра.
- Не нужно помнить о позиции параметров, о которых мы не знаем всегда.
- Не нужно искать порядок параметров в списке параметров вызываемой функции.
- Мы можем указать параметр для каждого аргумента по его имени.

Необязательные аргументы

Ссылка: *MSDN* Определение метода, конструктора, индексатора или делегата может указать, что его параметры требуются или что они являются необязательными. Любой вызов должен предоставлять аргументы для всех необходимых параметров, но может опускать аргументы для необязательных параметров.

Как сказано в *MSDN*, необязательный аргумент,

- Мы можем опустить аргумент в вызове, если этот аргумент является необязательным аргументом
- Каждое необязательное аргумент имеет собственное значение по умолчанию
- Он будет принимать значение по умолчанию, если мы не предоставим значение
- Значение по умолчанию необязательного аргумента должно быть
 - Постоянное выражение.
 - Должен быть тип значения, например enum или struct.
 - Должно быть выражение формы default (valueType)
- Он должен быть установлен в конце списка параметров

Examples

Именованные аргументы

Рассмотрим следующий вызов функции.

```
FindArea(120, 56);
```

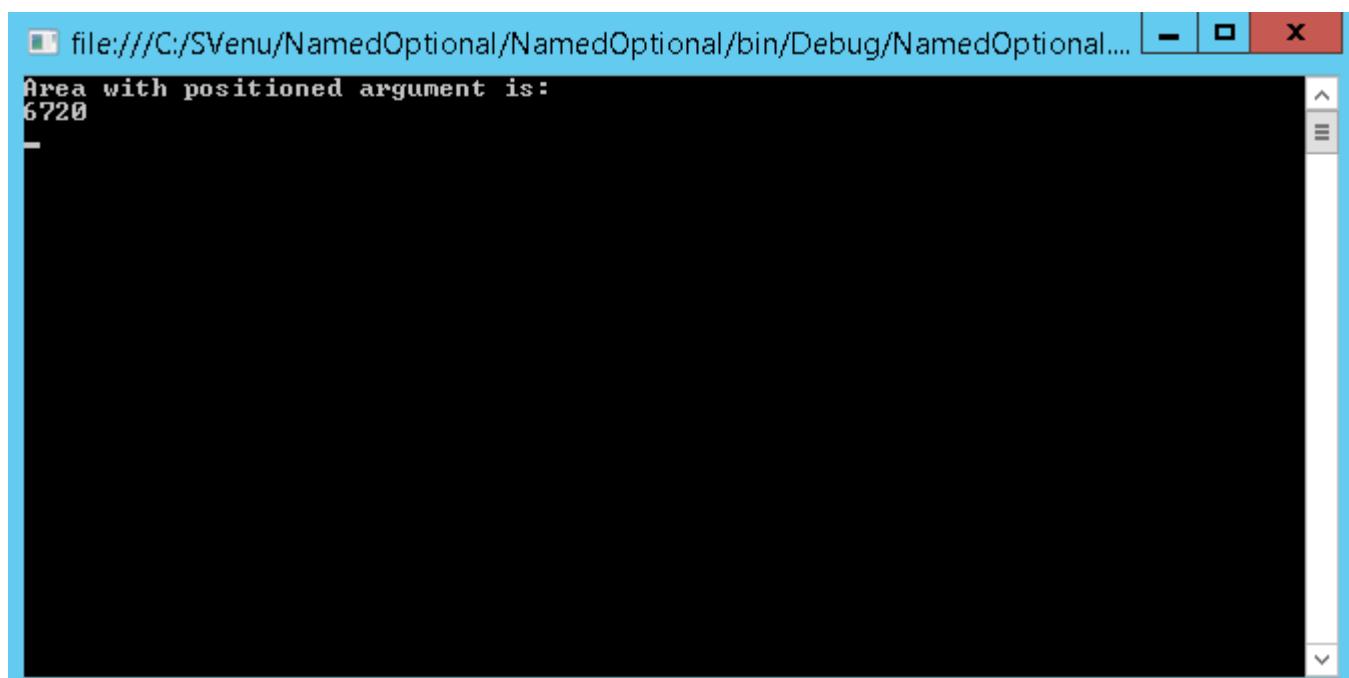
В этом наш первый аргумент - длина (т.е. 120), а второй аргумент - ширина (т.е. 56). И мы вычисляем область этой функцией. Следующее - определение функции.

```
private static double FindArea(int length, int width)
{
    try
    {
        return (length* width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
```

Поэтому в первом вызове функции мы просто передали аргументы по своей позиции.
Правильно?

```
double area;
Console.WriteLine("Area with positioned argument is: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

Если вы запустите это, вы получите результат следующим образом.



Теперь здесь появляются функции именованных аргументов. См. Предыдущий вызов функции.

```
Console.WriteLine("Area with Named argument is: ");
area = FindArea(length: 120, width: 56);
```

```
Console.WriteLine(area);
Console.Read();
```

Здесь мы приводим именованные аргументы в вызове метода.

```
area = FindArea(length: 120, width: 56);
```

Теперь, если вы запустите эту программу, вы получите тот же результат. Мы можем указать имена в обратном вызове метода, если мы используем именованные аргументы.

```
Console.WriteLine("Area with Named argument vice versa is: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

Одно из важных применений именованного аргумента заключается в том, что когда вы используете это в своей программе, это улучшает читаемость вашего кода. Он просто говорит, каков ваш аргумент, или что это такое?

Вы также можете указать позиционные аргументы. Это означает сочетание как позиционного аргумента, так и именованного аргумента.

```
Console.WriteLine("Area with Named argument Positional Argument : ");
area = FindArea(120, width: 56);
Console.WriteLine(area);
Console.Read();
```

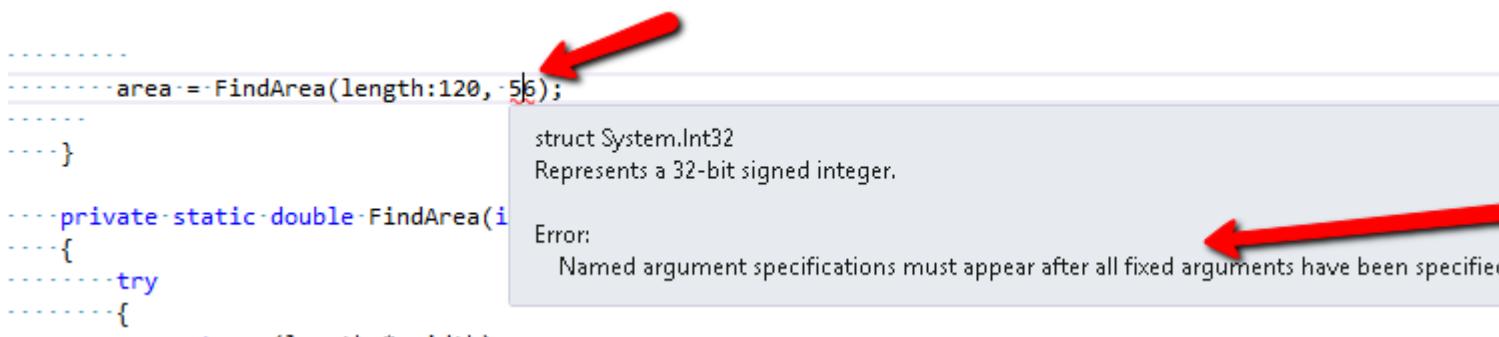
В приведенном выше примере мы передали 120 как длину и 56 в качестве именованного аргумента для ширины параметра.

Есть и некоторые ограничения. Теперь мы обсудим ограничение именных аргументов.

Ограничение использования именованного аргумента

Именованная спецификация аргумента должна появиться после указания всех фиксированных аргументов.

Если вы используете именованный аргумент перед фиксированным аргументом, вы получите ошибку времени компиляции следующим образом.



Именованная спецификация аргумента должна появляться после указания всех фиксированных аргументов

Необязательные аргументы

Рассмотрим предыдущее - это определение функции с необязательными аргументами.

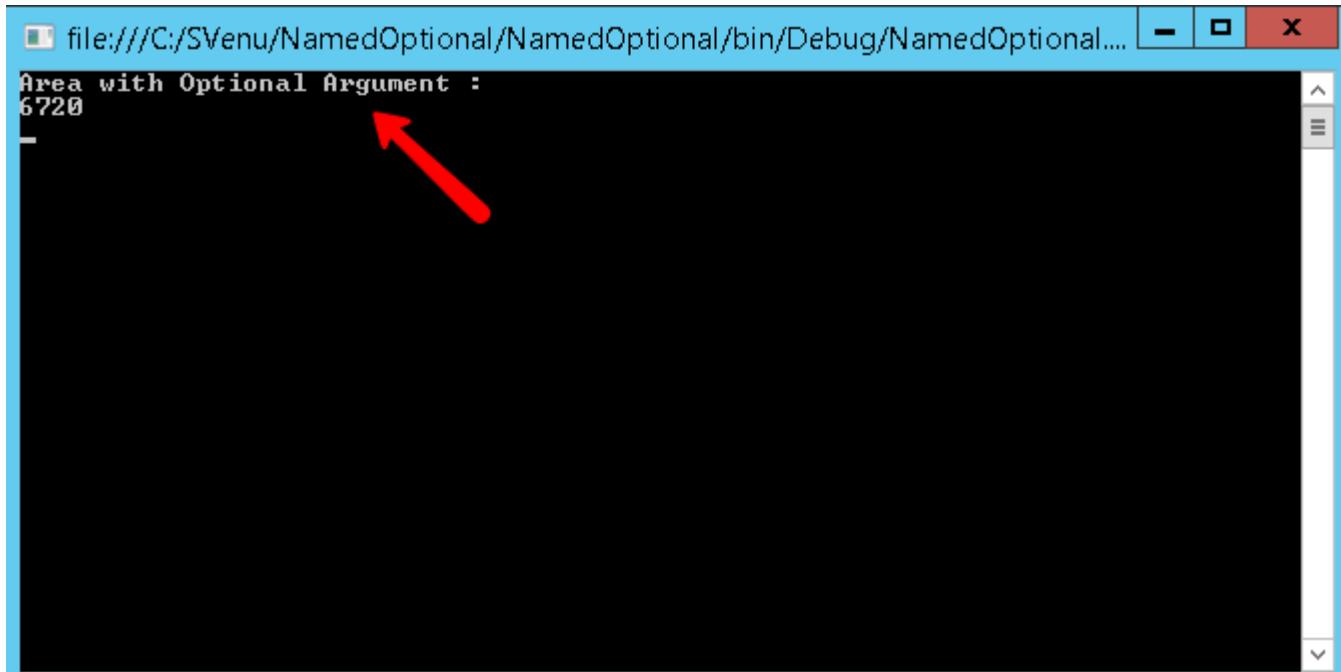
```
private static double FindAreaWithOptional(int length, int width=56)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
```

Здесь мы установили значение ширины как необязательное и дали значение 56. Если вы заметили, что IntelliSense сам показывает необязательный аргумент, как показано на рисунке ниже.

```
-area=FindAreaWithOptional(
    double Program.FindAreaWithOptional(int length, [int width = 56])
```

```
Console.WriteLine("Area with Optional Argument : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

Обратите внимание, что во время компиляции мы не получили никакой ошибки, и вы получите результат следующим образом.



Использование дополнительного атрибута.

Другим способом реализации необязательного аргумента является использование ключевого слова `[Optional]`. Если вы не передадите значение для необязательного аргумента, этому аргументу присваивается значение по умолчанию этого типа данных. Ключевое слово `Optional` присутствует в пространстве имен `Runtime.InteropServices`.

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
area = FindAreaWithOptional(120); //area=0
```

И когда мы вызываем функцию, мы получаем 0, потому что второй аргумент не передается, а значение по умолчанию для `int` равно 0, и поэтому произведение равно 0.

Прочтайте Именованные и необязательные аргументы онлайн:

<https://riptutorial.com/ru/csharp/topic/5220/именованные-и-необязательные-аргументы>

глава 63: Импорт контактов Google

замечания

Данные о контактах пользователей будут получены в формате JSON, мы его извлечем и, наконец, пропустим эти данные и, таким образом, получим контакты google.

Examples

Требования

Чтобы импортировать контакты Google (Gmail) в приложении ASP.NET MVC, сначала [загрузите «Настройка Google API»](#). Это предоставит следующие ссылки:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Добавьте их в соответствующее приложение.

Исходный код в контроллере

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = ""; // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
```

```

redirect action method NOTE: you need to configure same url in google console
    Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
redirectUrl + "&response_type=code&client_id=" + clientId +
"&scope=https://www.google.com/m8/feeds/&approval_prompt=force&access_type=offline");

    return View();
}

public ActionResult AddGoogleContacts()
{
    string code = Request.QueryString["code"];
    if (!string.IsNullOrEmpty(code))
    {
        var contacts = GetAccessToken().ToArray();
        if (contacts.Length > 0)
        {
            // You will get all contacts here
            return View("Index", contacts);
        }
        else
        {
            return RedirectToAction("Index", "Home");
        }
    }
    else
    {
        return RedirectToAction("Index", "Home");
    }
}

public List<GmailContacts> GetAccessToken()
{
    string code = Request.QueryString["code"];
    string google_client_id = ""; //your google client Id
    string google_client_sceret = ""; // your google secret key
    string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";

    HttpWebRequest webRequest =
    (HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
    webRequest.Method = "POST";
    string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
    byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
    webRequest.ContentType = "application/x-www-form-urlencoded";
    webRequest.ContentLength = byteArray.Length;
    Stream postStream = webRequest.GetRequestStream();
    // Add the post data to the web request
    postStream.Write(byteArray, 0, byteArray.Length);
    postStream.Close();
    WebResponse response = webRequest.GetResponse();
    postStream = response.GetResponseStream();
    StreamReader reader = new StreamReader(postStream);
    string responseFromServer = reader.ReadToEnd();
    GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
    /*End*/
    return GetContacts(serStatus);
}

public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)

```

```

{
    string google_client_id = ""; //client id
    string google_client_sceret = ""; //secret key
    /*Get Google Contacts From Access Token and Refresh Token*/
    // string refreshToken = serStatus.refresh_token;
    string accessToken = serStatus.access_token;
    string scopes = "https://www.google.com/m8/feeds/contacts/default/full/";
    OAuth2Parameters oAuthparameters = new OAuth2Parameters()
    {
        ClientId = google_client_id,
        ClientSecret = google_client_sceret,
        RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
        Scope = scopes,
        AccessToken = accessToken,
        // RefreshToken = refreshToken
    };

    RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
    ContactsRequest cr = new ContactsRequest(settings);
    ContactsQuery query = new
    ContactsQuery(ContactsQuery.CreateContactsUri("default"));
    query.NumberToRetrieve = 5000;
    Feed<Contact> ContactList = cr.GetContacts();

    List<GmailContacts> olist = new List<GmailContacts>();
    foreach (Contact contact in ContactList.Entries)
    {
        foreach (EMail email in contact.Emails)
        {
            GmailContacts gc = new GmailContacts();
            gc.EmailID = email.Address;
            var a = contact.Name.FullName;
            olist.Add(gc);
        }
    }
    return olist;
}

public class GmailContacts
{
    public string EmailID
    {
        get { return _EmailID; }
        set { _EmailID = value; }
    }
    private string _EmailID;
}

public class GooglePlusAccessToken
{
    public GooglePlusAccessToken()
    {
    }

    public string access_token
    {
        get { return _access_token; }
        set { _access_token = value; }
    }
    private string _access_token;
}

```

```
public string token_type
{
    get { return _token_type; }
    set { _token_type = value; }
}
private string _token_type;

public string expires_in
{
    get { return _expires_in; }
    set { _expires_in = value; }
}
private string _expires_in;

}
}
```

Исходный код в представлении.

Единственный способ действия, который вам нужно добавить, - добавить ссылку на действие, представленную ниже

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

Прочтайте Импорт контактов Google онлайн: <https://riptutorial.com/ru/csharp/topic/6744/импорт-контактов-google>

глава 64: имя оператора

Вступление

Оператор `nameof` позволяет вам получить имя **переменной**, **типа** или **члена** в строковой форме без жесткого кодирования в качестве литерала.

Операция оценивается во время компиляции, что означает, что вы можете переименовать ссылочный идентификатор, используя функцию переименования среды IDE, и строка имени будет обновляться вместе с ней.

Синтаксис

- `nameof` (выражение)

Examples

Основное использование: печать имени переменной

Оператор `nameof` позволяет вам получить имя переменной, типа или члена в строковой форме без жесткого кодирования в качестве литерала. Операция оценивается во время компиляции, что означает, что вы можете переименовать, используя функцию переименования среды IDE, ссылочный идентификатор, а строка имени будет обновляться вместе с ним.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Выпустит

MyString

потому что имя переменной «`myString`». Рефакторинг имени переменной изменит строку.

Если `nameof` в ссылочном типе, оператор `nameof` возвращает имя текущей ссылки, а *не* имя или имя типа базового объекта. Например:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Печать имени параметра

отрывок

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

Консольный выход

paramValue

Принятие события PropertyChanged

отрывок

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

Консольный выход

Адрес

Обработка событий PropertyChanged

отрывок

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...

private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

Консольный выход

Название изменено на «Все в огне»

Статус изменен на ShowStopper

Применяется к параметру общего типа

отрывок

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}
```

```
...  
var myClass = new SomeClass<int>();  
myClass.PrintTypeName();  
  
Console.WriteLine(nameof(SomeClass<int>));
```

Консольный выход

TItem

SomeClass

Применяется к квалифицированным идентификаторам

отрывок

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));  
Console.WriteLine(nameof(MyClass));  
Console.WriteLine(nameof(MyClass.MyNestedClass));  
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Консольный выход

MyNamespace

Мои занятия

MyNestedClass

MyStaticProperty

Аргументы проверки и защиты

предпочитать

```
public class Order  
{  
    public OrderLine AddOrderLine(OrderLine orderLine)  
    {  
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));  
        ...  
    }  
}
```

Над

```
public class Order  
{  
    public OrderLine AddOrderLine(OrderLine orderLine)  
    {
```

```
    if (orderLine == null) throw new ArgumentNullException("orderLine");
    ...
}
```

Использование функции `nameof` упрощает рефакторинг параметров метода.

Сильно набраны ссылки действий MVC

Вместо обычного навязчивого типа:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

Теперь вы можете четко набирать ссылки на действия:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Теперь, если вы хотите реорганизовать свой код и переименовать метод

`UserController.LogIn` в `UserController.SignIn`, вам не нужно беспокоиться о поиске всех строк.

Компилятор выполнит эту работу.

Прочтите имя оператора онлайн: <https://riptutorial.com/ru/csharp/topic/80/имя-оператора>

глава 65: Индексатор

Синтаксис

- `public ReturnType this [IndexType index] {get {...} set {...}}`

замечания

Indexer позволяет синтаксису типа массива получить доступ к свойству объекта с индексом.

- Может использоваться для класса, структуры или интерфейса.
- Может быть перегружен.
- Может использовать несколько параметров.
- Может использоваться для доступа и установки значений.
- Может использовать любой тип для своего индекса.

Examples

Простой индексатор

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

Использование:

```
var foo = new Foo();

// access a value
string berlin = foo[2];

// assign a value
foo[0] = "Rome";
```

[Посмотреть демо](#)

Indexer с двумя аргументами и интерфейсом

```
interface ITable {
    // an indexer can be declared in an interface
    object this[int x, int y] { get; set; }
}

class DataTable : ITable
{
    private object[,] cells = new object[10, 10];

    /// <summary>
    /// implementation of the indexer declared in the interface
    /// </summary>
    /// <param name="x">X-Index</param>
    /// <param name="y">Y-Index</param>
    /// <returns>Content of this cell</returns>
    public object this[int x, int y]
    {
        get
        {
            return cells[x, y];
        }
        set
        {
            cells[x, y] = value;
        }
    }
}
```

Перегрузка индексатора для создания SparseArray

Перегружая индексатор, вы можете создать класс, который выглядит и выглядит как массив, но это не так. Он будет иметь методы O (1) get и set, может получить доступ к элементу с индексом 100 и все же иметь размер элементов внутри него. Класс SparseArray

```
class SparseArray
{
    Dictionary<int, string> array = new Dictionary<int, string>();

    public string this[int i]
    {
        get
        {
            if(!array.ContainsKey(i))
            {
                return null;
            }
            return array[i];
        }
        set
        {
            if(!array.ContainsKey(i))
                array.Add(i, value);
        }
    }
}
```

Прочтайте Индексатор онлайн: <https://riptutorial.com/ru/csharp/topic/1660/индексатор>

глава 66: Инициализаторы коллекции

замечания

Единственное требование для инициализации объекта с использованием этого синтаксического сахара заключается в том, что тип реализует `System.Collections.IEnumerable` и метод `Add`. Хотя мы называем это инициализатором коллекции, объект *не* должен быть коллекцией.

Examples

Инициализаторы коллекции

Инициализировать тип коллекции со значениями:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Инициализаторы коллекции - это синтаксический сахар для вызовов `Add()`. Вышеуказанный код эквивалентен:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Обратите внимание, что инициализация выполняется атомарно с использованием временной переменной, чтобы избежать условий гонки.

Для типов, которые предлагают несколько параметров в методе `Add()`, заключают аргументы, разделенные запятыми, в фигурные скобки:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

Это эквивалентно:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
```

```
var numberDictionary = temp;
```

Инициализаторы индекса C # 6

Начиная с C # 6, коллекции с индексаторами могут быть инициализированы путем указания индекса для назначения в квадратных скобках, за которым следует знак равенства, а затем значение для назначения.

Инициализация словаря

Пример этого синтаксиса с использованием словаря:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

Это эквивалентно:

```
var dict = new Dictionary<string, int>();
dict["key1"] = 1;
dict["key2"] = 50
```

Синтаксис инициализатора коллекции, чтобы сделать это до C # 6:

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

Что соответствовало бы:

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

Таким образом, существует значительная разница в функциональности, поскольку новый синтаксис использует *индексатор инициализированного объекта* для назначения значений вместо использования метода `Add()`. Это означает, что новый синтаксис требует только общедоступного индексатора и работает для любого объекта, который имеет один.

```
public class IndexableClass
{
    public int this[int index]
    {
        set
```

```

        {
            Console.WriteLine("{0} was assigned to index {1}", value, index);
        }
    }

var foo = new IndexableClass
{
    [0] = 10,
    [1] = 20
}

```

Это приведет к выводу:

```

10 was assigned to index 0
20 was assigned to index 1

```

Инициализаторы коллекции в пользовательских классах

Чтобы создать инициализаторы коллекции поддержки класса, он должен реализовать интерфейс `IEnumerable` и иметь хотя бы один метод `Add`. Начиная с С # 6, любая коллекция, реализующая `IEnumerable` может быть дополнена пользовательскими методами `Add` с использованием методов расширения.

```

class Program
{
    static void Main()
    {
        var col = new MyCollection {
            "foo",
            { "bar", 3 },
            "baz",
            123.45d,
        };
    }
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item)
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

```

```
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
        @this.Add(value.ToString());
}
```

Инициализаторы коллекции с массивами параметров

Вы можете смешивать обычные параметры и массивы параметров:

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers) {
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

Этот синтаксис теперь возможен:

```
var Tickets = new List<LotteryTicket>{
    {"Mr Cool", 35663, 35732, 12312, 75685},
    {"Bruce", 26874, 66677, 24546, 36483, 46768, 24632, 24527},
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}
}
```

Использование инициализатора коллекции внутри инициализатора объекта

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms` - это свойство типа коллекции. Когда объект `Tag` создается с использованием синтаксиса инициализатора объекта, `Synonyms` также могут быть инициализированы синтаксисом инициализатора коллекции:

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

Свойство коллекции может быть только для чтения и по-прежнему поддерживать синтаксис инициализатора коллекции. Рассмотрим этот модифицированный пример (свойство `Synonyms` теперь имеет частный сеттер):

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }

    public IList<string> Synonyms { get; private set; }
}
```

Новый объект `Tag` может быть создан следующим образом:

```
Tag t = new Tag
{
    Synonyms = { "c#", "c-sharp" }
};
```

Это работает, потому что инициализаторы коллекции являются просто синтаксическим сахаром над вызовами `Add()`. Здесь нет нового списка, компилятор просто генерирует вызовы `Add()` на выходе из объекта.

Прочтайте Инициализаторы коллекции онлайн: <https://riptutorial.com/ru/csharp/topic/21/инициализаторы-коллекции>

глава 67: Инициализаторы объектов

Синтаксис

- `SomeClass sc = new SomeClass {Property1 = value1, Property2 = value2, ...};`
- `SomeClass sc = new SomeClass (param1, param2, ...) {Property1 = value1, Property2 = value2, ...}`

замечания

Консоли конструктора могут быть опущены только в том случае, если создаваемый тип имеет конструктор по умолчанию (без параметров).

Examples

Простое использование

Инициализаторы объектов удобны, когда вам нужно создать объект и сразу установить сразу несколько свойств, но доступных конструкторов недостаточно. Скажем, у вас есть класс

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

Чтобы инициализировать новый экземпляр класса с помощью инициализатора:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

Это эквивалентно

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

Использование анонимных типов

Инициализаторы объектов - единственный способ инициализировать анонимные типы, которые являются типами, сгенерированными компилятором.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

По этой причине инициализаторы объектов широко используются в запросах на выбор LINQ, поскольку они предоставляют удобный способ указать, какие части запрашиваемого объекта вам интересны.

```
var albumTitles = from a in albums
                  select new
                  {
                      Title = a.Title,
                      Artist = a.Band
                  };
```

Использование с нестандартными конструкторами

Вы можете комбинировать инициализаторы объектов с конструкторами для инициализации типов, если это необходимо. Возьмем, например, класс, определенный как таковой:

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        //do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

Сначала это создаст экземпляр `Book` с помощью конструктора `Book(int)`, а затем задает каждое свойство в инициализаторе. Это эквивалентно:

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Прочтайте Инициализаторы объектов онлайн: <https://riptutorial.com/ru/csharp/topic/738/инициализаторы-объектов>

глава 68: Инициализация свойств

замечания

Когда вы решаете, как создать свойство, начните с автоматически реализованного свойства для простоты и краткости.

Переключитесь на свойство с резервным полем только тогда, когда обстоятельства определяют. Если вам нужны другие манипуляции за пределами простого набора и получить, вам может потребоваться ввести фоновое поле.

Examples

C # 6.0: Инициализировать свойство с автоматической реализацией

Создайте свойство с помощью getter и / или setter и инициализируйте все в одной строке:

```
public string Foobar { get; set; } = "xyz";
```

Инициализация свойства с помощью поля поддержки

```
public string Foobar {  
    get { return _foobar; }  
    set { _foobar = value; }  
}  
private string _foobar = "xyz";
```

Инициализация свойства в конструкторе

```
class Example  
{  
    public string Foobar { get; set; }  
    public List<string> Names { get; set; }  
    public Example()  
    {  
        Foobar = "xyz";  
        Names = new List<string>() {"carrot", "fox", "ball"};  
    }  
}
```

Инициализация имущества при создании объекта

Свойства могут быть установлены при создании экземпляра объекта.

```
var redCar = new Car
```

```
{  
    Wheels = 2,  
    Year = 2016,  
    Color = Color.Red  
};
```

Прочтайте Инициализация свойств онлайн: <https://riptutorial.com/ru/csharp/topic/82/инициализация-свойств>

глава 69: Интерполяция строк

Синтаксис

- `$ "content {expression} content"`
- `$ "content {expression: format} content"`
- `$ "content {expression} {{content in braces}} content"`
- `$ "content {выражение: format} {{content in braces}} content"`

замечания

Строковая интерполяция является сокращением для `string.Format()` который упрощает `string.Format()` строк с переменными и значениями выражения внутри них.

```
var name = "World";
var oldWay = string.Format("Hello, {0}!", name);    // returns "Hello, World"
var newWay = $"Hello, {name}!";                      // returns "Hello, World"
```

Examples

Выражения

Полные выражения также могут использоваться в интерполированных строках.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

Живая демонстрация на .NET скрипке

Форматировать даты в строках

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

Вы также можете использовать метод `DateTime.ToString` для форматирования объекта `DateTime`. Это даст тот же результат, что и код выше.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

Выход:

Это 11 ноября 2015 года, сделайте желание!

Живая демонстрация на .NET скрипке

Демо-версия с использованием `DateTime.ToString`

Примечание: `мм` обозначает месяцы и `тттт` течение нескольких минут. Будьте очень осторожны, когда используете их, поскольку ошибки могут вводить ошибки, которые могут быть трудно обнаружить.

Простое использование

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

За кулисами

Внутри этого

```
 $"Hello, {name}!"
```

Будет скомпилирован примерно так:

```
string.Format("Hello, {0}!", name);
```

Заполнение вывода

Строка может быть отформатирована, чтобы принять параметр дополнения, который будет определять, сколько позиций символов будет использовать вставленная строка:

```
 ${value, padding}
```

ПРИМЕЧАНИЕ. Положительные значения заполнения означают, что левое заполнение и отрицательные значения прокладки указывают на правильное заполнение.

Левая прокладка

Левое заполнение 5 (добавляет 3 пробела перед значением числа, поэтому в итоговой строке оно занимает 5 символов).

```
var number = 42;
```

```
var str = $"The answer to life, the universe and everything is {number, 5}.";  
//str is "The answer to life, the universe and everything is 42."  
//  
System.Console.WriteLine(str);
```

Выход:

```
The answer to life, the universe and everything is 42.
```

Живая демонстрация на .NET скрипке

Правое заполнение

Правое заполнение, которое использует отрицательное значение заполнения, добавит пробелы в конец текущего значения.

```
var number = 42;  
var str = $"The answer to life, the universe and everything is ${number, -5}.";  
//str is "The answer to life, the universe and everything is 42    ."  
//  
System.Console.WriteLine(str);
```

Выход:

```
The answer to life, the universe and everything is 42    .
```

Живая демонстрация на .NET скрипке

Заполнение с помощью спецификаторов формата

Вы также можете использовать существующие спецификаторы форматирования в сочетании с дополнением.

```
var number = 42;  
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";  
//str is "The answer to life, the universe and everything is 42.1 "  
//  
System.Console.WriteLine(str);
```

Живая демонстрация на .NET скрипке

Форматирование чисел в строках

Вы можете использовать двоеточие и [синтаксис стандартного числового формата](#) для управления форматированием чисел.

```
var decimalValue = 120.5;
```

```
var asCurrency = $"It costs {decimalValue:C}";  
// String value is "It costs $120.50" (depending on your local currency settings)  
  
var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";  
// String value is "Exactly 120.500"  
  
var integerValue = 57;  
  
var prefixedIfNecessary = $"{integerValue:D5}";  
// String value is "00057"
```

Живая демонстрация на .NET скрипке

Прочтайте Интерполяция строк онлайн: <https://riptutorial.com/ru/csharp/topic/22/интерполяция-строк>

глава 70: Интерфейс `IDisposable`

замечания

- Клиенты класса, реализующие `IDisposable` могут быть уверены, что они вызовут метод `Dispose` когда они будут завершены с использованием объекта. В CLR ничего не найдено, что непосредственно ищет объекты для метода `Dispose` для вызова.
- Нет необходимости реализовывать финализатор, если ваш объект содержит только управляемые ресурсы. Обязательно вызывайте `Dispose` во всех объектах, которые использует ваш класс, когда вы реализуете свой собственный метод `Dispose`.
- Рекомендуется сделать класс безопасным для нескольких вызовов `Dispose`, хотя в идеале его следует вызывать только один раз. Этого можно добиться, добавив в свой класс `private bool` переменную `private bool` и установив значение `true` когда был запущен метод `Dispose`.

Examples

В классе, который содержит только управляемые ресурсы

Управляемые ресурсы - это ресурсы, которые сообщают сборщики мусора времени выполнения и находятся под контролем. Существует много классов, доступных в BCL, например, таких как `SqlConnection` который является классом-оболочкой для неуправляемого ресурса. Эти классы уже реализуют интерфейс `IDisposable` - это зависит от вашего кода, чтобы очистить их, когда вы закончите.

Нет необходимости реализовывать финализатор, если ваш класс содержит только управляемые ресурсы.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

В классе с управляемыми и неуправляемыми ресурсами

Важно, чтобы финализация игнорировала управляемые ресурсы. Финализатор работает в другом потоке - возможно, что управляемые объекты больше не существуют к моменту

завершения финализатора. Реализация защищенного метода `Dispose(bool)` является обычной практикой для обеспечения того, что управляемые ресурсы не имеют метода `Dispose` вызванного из финализатора.

```
public class ManagedAndUnmanagedObject : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }

            unmanagedHandle.Release();

            disposed = true;
        }
    }

    ~ManagedAndUnmanagedObject()
    {
        Dispose(false);
    }
}
```

IDisposable, Dispose

.NET Framework определяет интерфейс для типов, требующих метода сбросывания:

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

`Dispose()` в основном используется для очистки ресурсов, например, неуправляемых ссылок. Однако также может быть полезно принудительно распоряжаться другими ресурсами, даже если они управляются. Вместо того, чтобы ждать, пока GC в конечном итоге также очистит ваше соединение с базой данных, вы можете убедиться, что это сделано в вашей собственной реализации `Dispose()`.

```
public void Dispose()
{
    if (null != this.CurrentDatabaseConnection)
    {
        this.CurrentDatabaseConnection.Dispose();
        this.CurrentDatabaseConnection = null;
    }
}
```

Когда вам нужно напрямую обращаться к неуправляемым ресурсам, таким как неуправляемые указатели или ресурсы win32, создайте класс, наследующий от `SafeHandle` и используйте соглашения / инструменты этого класса для этого.

В унаследованном классе с управляемыми ресурсами

Довольно часто можно создать класс, который реализует `IDisposable`, а затем выводит классы, которые также содержат управляемые ресурсы. Рекомендуется пометить метод `Dispose` **Ключевым словом** `virtual` чтобы клиенты могли очищать любые ресурсы, которыми они могут владеть.

```
public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}
```

использование ключевого слова

Когда объект реализует интерфейс `IDisposable`, он может быть создан в синтаксисе `using`:

```
using (var foo = new Foo())
{
```

```
// do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
}
```

[Просмотреть демоверсию](#)

using [синтаксического сахара](#) для блока `try/finally`; приведенное выше использование будет примерно переводить:

```
{
    var foo = new Foo();
    try
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

Прочтайте Интерфейс `IDisposable` онлайн: <https://riptutorial.com/ru/csharp/topic/1795/интерфейс-idisposable>

глава 71: Интерфейс **INotifyPropertyChanged**

замечания

Интерфейс `INotifyPropertyChanged` необходим, когда вам нужно, чтобы ваш класс сообщал об изменениях, происходящих с его свойствами. Интерфейс определяет одно событие `PropertyChanged`.

С привязкой XAML событие `PropertyChanged` автоматически подключается, поэтому вам нужно реализовать интерфейс `INotifyPropertyChanged` для вашей модели представления или классов контекста данных для работы с привязкой XAML.

Examples

Внедрение `INotifyPropertyChanged` в C # 6

Реализация `INotifyPropertyChanged` может быть подвержена ошибкам, так как интерфейс требует указания имени свойства как строки. Чтобы сделать реализацию более надежной, можно использовать атрибут `CallerMemberName`.

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Если у вас несколько классов, реализующих `INotifyPropertyChanged`, вам может оказаться полезным реорганизовать реализацию интерфейса и вспомогательный метод для общего базового класса:

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

INotifyPropertyChanged с помощью метода общего набора

В `NotifyPropertyChangedBase` **ниже** классе `NotifyPropertyChangedBase` определяется общий метод `Set`, который можно вызвать из любого производного типа.

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName = null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

Чтобы использовать этот общий метод `Set`, вам просто нужно создать класс, который происходит из `NotifyPropertyChangedBase`.

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
```

```
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

Как показано выше, вы можете вызвать `Set(ref _fieldName, value);` в настройщике свойства, и он автоматически поднимет событие `PropertyChanged`, если это необходимо.

Затем вы можете зарегистрироваться в событие `PropertyChanged` из другого класса, который должен обрабатывать изменения свойств.

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
    }

    private readonly SomeViewModel _vm;
}
```

Прочтайте Интерфейс `INotifyPropertyChanged` онлайн:

<https://riptutorial.com/ru/csharp/topic/2990/интерфейс-inotifypropertychanged>

глава 72: Интерфейс `IQueryable`

Examples

Перевод запроса LINQ в SQL-запрос

Интерфейсы `IQueryable` и `IQueryable<T>` позволяют разработчикам преобразовывать запрос LINQ («интегрированный язык») в конкретный источник данных, например реляционную базу данных. Возьмите этот запрос LINQ, написанный на C #:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

Если переменные `books` имеют тип, реализующий `IQueryable<Book>` то указанный выше запрос передается провайдеру (установленному в `IQueryable.Provider`) в виде дерева выражений, структуры данных, которая отражает структуру кода ,

Поставщик может проверить дерево выражений во время выполнения, чтобы определить:

- что есть предикат для `Author` свойства `Book` класса;
- что используемый метод сравнения равен 'equals' (`==`);
- что ценность, которую она должна быть равна, - `"Stephen King"` .

С помощью этой информации поставщик может перевести запрос C # на SQL-запрос во время выполнения и передать этот запрос в реляционную базу данных для извлечения только тех книг, которые соответствуют предикату:

```
select *
from Books
where Author = 'Stephen King'
```

Поставщик получает вызов, когда переменная `query` перебирается (`IQueryable` реализует `IEnumerable`).

(Поставщик, используемый в этом примере, потребует некоторых дополнительных метаданных, чтобы узнать, какая таблица запрашивать и знать, как сопоставить свойства класса C # с столбцами таблицы, но такие метаданные находятся за пределами области интерфейса `IQueryable` .)

Прочтайте Интерфейс `IQueryable` онлайн: <https://riptutorial.com/ru/csharp/topic/3094/интерфейс-iqueryable>

глава 73: Интерфейсы

Examples

Внедрение интерфейса

Интерфейс используется для обеспечения наличия метода в любом классе, который «реализует» его. Интерфейс определяется `interface` ключевого слова, и класс может «реализовать» его, добавив `: InterfaceName` после имени класса. Класс может реализовывать несколько интерфейсов, разделяя каждый интерфейс запятой.

`: InterfaceName, ISecondInterface`

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

Поскольку они реализуют `INoiseMaker`, как `cat` и `dog` должны включать метод `string MakeNoise()` и не смогут скомпилироваться без него.

Реализация нескольких интерфейсов

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
```

```

        Name = "Cat";
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}

```

Явная реализация интерфейса

Явная реализация интерфейса необходима, когда вы реализуете несколько интерфейсов, которые определяют общий метод, но требуются разные реализации в зависимости от того, какой интерфейс используется для вызова метода (обратите внимание, что вам не нужны явные реализации, если несколько интерфейсов используют один и тот же метод и возможна общая реализация).

```

interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }

    string IGolfPlayer.Drive()
    {
        return "Took a swing...";
    }
}

GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;

Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...

```

Реализация не может быть вызвана нигде, кроме как с помощью интерфейса:

```

public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()

```

```
{  
    return "Swinging hard...";  
}  
public void Swing()  
{  
    Drive(); // Compiler error: No such method  
}  
}
```

В связи с этим может оказаться целесообразным поместить сложный код реализации явно реализованного интерфейса в отдельный частный метод.

Разумеется, явная реализация интерфейса может быть использована только для тех методов, которые существуют для этого интерфейса:

```
public class ProGolfer : IGolfPlayer  
{  
    string IGolfPlayer.Swear() // Error  
    {  
        return "The ball is in the pit";  
    }  
}
```

Аналогичным образом, использование явной реализации интерфейса без объявления этого интерфейса в классе также вызывает ошибку.

Подсказка:

Внедрение интерфейсов также может быть использовано для предотвращения мертвого кода. Когда метод больше не нужен и удаляется из интерфейса, компилятор будет жаловаться на каждую существующую реализацию.

Замечания:

Программисты ожидают, что контракт будет одинаковым независимо от контекста типа и явной реализации не должен выставлять другое поведение при вызове. Поэтому, в отличие от приведенного выше примера, `IGolfPlayer.Drive` и `Drive` должны делать то же самое, когда это возможно.

Почему мы используем интерфейсы

Интерфейс - это определение контракта между пользователем интерфейса и классом, который его реализует. Один из способов думать о интерфейсе - это объявление того, что объект может выполнять определенные функции.

Предположим, что мы определяем интерфейс `IShape` для представления различных типов

фигур, мы ожидаем, что у формы будет область, поэтому мы определим метод, чтобы заставить реализации интерфейса вернуть свою область:

```
public interface IShape
{
    double ComputeArea();
}
```

Давайте будем иметь следующие две формы: `Rectangle` и `Circle`

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}
```

У каждого из них есть собственное определение его области, но оба они являются формами. Поэтому логично рассматривать их как `IShape` в нашей программе:

```
private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
```

```
        Console.WriteLine("Area: {0:N}, shape.ComputeArea());  
    }  
}  
  
// Output:  
// Area : 50.00  
// Area : 78.54
```

Основы интерфейса

Функция интерфейса, известная как «контракт» функциональности. Это означает, что он объявляет свойства и методы, но не реализует их.

Так что в отличие от классов Интерфейсы:

- Невозможно создать экземпляр
- Не может быть никакой функциональности
- Может содержать только методы * (*Свойства и события являются методами внутри*)
- Наследование интерфейса называется «Реализация»,
- Вы можете наследовать от 1 класса, но вы можете «реализовать» несколько интерфейсов

```
public interface ICanDoThis{  
    void TheThingICanDo();  
    int SomeValueProperty { get; set; }  
}
```

Что следует заметить:

- Префикс «I» - это соглашение об именах, используемое для интерфейсов.
- Тело функции заменяется точкой с запятой «;».
- Свойства также разрешены, потому что внутренне они также являются методами

```
public class MyClass : ICanDoThis {  
    public void TheThingICanDo(){  
        // do the thing  
    }  
  
    public int SomeValueProperty { get; set; }  
    public int SomeValueNotImplementingAnything { get; set; }  
}
```

,

```
ICanDoThis obj = new MyClass();  
  
// ok  
obj.TheThingICanDo();  
  
// ok  
obj.SomeValueProperty = 5;
```

```
// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
(MyClass)obj.SomeValueNotImplemtingAnything = 5; // ok
```

Это особенно полезно, когда вы работаете с инфраструктурами пользовательского интерфейса, такими как WinForms или WPF, потому что для создания пользовательского элемента управления необходимо унаследовать от базового класса, и вы теряете способность создавать абстракции по различным типам управления. Пример? Прибытие:

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

Предложенная проблема состоит в том, что оба содержат некоторое понятие «Текст», но имена свойств различаются. И вы не можете создать создание *абстрактного базового класса*, потому что у них есть обязательное наследование для двух разных классов. Интерфейс может облегчить

```
public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }

    public int Clicks { get; set; }
}
```

Теперь MyButton и MyTextBlock взаимозаменяемы.

```
var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};
```

```

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");

    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    (MyButton)ctrl.Clicks = 0;

    /* the solution is to check the type first.
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors

}

```

«Скрытие» членов с явной реализацией

Разве вы не ненавидите его, когда интерфейсы загрязняют вас классом со слишком многими членами, которых вам даже не волнует? Ну, я получил решение! Явные реализации

```

public interface IMessageservice {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}

```

Обычно вы реализуете класс следующим образом.

```

public class MyObjectWithMessages : IMessageservice {
    public void OnMessageRecieve() {

    }

    public void SendMessage() {

    }

    public string Result { get; set; }
    public int Encoding { get; set; }
}

```

Каждый участник является публичным.

```

var obj = new MyObjectWithMessages();

// why would i want to call this function?

```

```
obj.OnMessageRecieve();
```

Ответ: Не знаю. Таким образом, ни один из них не должен быть объявлен открытым, но просто объявляя, что члены как частные, заставят компилятор выбросить ошибку

Решение заключается в использовании явной реализации:

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {
    }

    void IMessageService.SendMessage() {
    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

Итак, теперь вы внедрили участников по мере необходимости, и они не будут публиковать публичных участников.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've successfully made it "private" */
obj.OnMessageRecieve();
```

Если вы серьезно хотите получить доступ к члену, хотя явным образом реализую все, что вам нужно сделать, это передать объект в интерфейс, и вам хорошо идти.

```
((IMessageService) obj).OnMessageRecieve();
```

Comparable как пример реализации интерфейса

Интерфейсы могут казаться абстрактными, пока вы не увидите их на практике. `Comparable` и `Comparable<T>` - отличные примеры того, почему интерфейсы могут быть полезны для нас.

Предположим, что в программе для интернет-магазина у нас есть множество предметов, которые вы можете купить. Каждый элемент имеет имя, идентификационный номер и цену.

```
public class Item {

    public string name; // though public variables are generally bad practice,
    public int idNumber; // to keep this example simple we will use them instead
    public decimal price; // of a property.

    // body omitted for brevity
```

```
}
```

У нас есть наш `Item` хранящийся внутри `List<Item>`, и в нашей программе где-то мы хотим отсортировать наш список по ID-номеру от наименьшего до самого большого. Вместо написания нашего собственного алгоритма сортировки мы можем вместо этого использовать метод `Sort()` который уже имеет `List<T>`. Однако, поскольку наш класс `Item` находится прямо сейчас, для `List<T>` нет способа понять, какой порядок сортировать список. Здесь находится интерфейс `IComparable`.

Чтобы правильно реализовать метод `CompareTo`, `CompareTo` должен возвращать положительное число, если параметр «меньше» текущего, ноль, если они равны, и отрицательное число, если параметр «больше».

```
Item apple = new Item();
apple.idNumber = 15;
Item banana = new Item();
banana.idNumber = 4;
Item cow = new Item();
cow.idNumber = 15;
Item diamond = new Item();
diamond.idNumber = 18;

Console.WriteLine(apple.CompareTo(banana)); // 11
Console.WriteLine(apple.CompareTo(cow)); // 0
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Вот на примере `Item` реализации «сек интерфейса»:

```
public class Item : IComparable<Item> {

    private string name;
    private int idNumber;
    private decimal price;

    public int CompareTo(Item otherItem) {

        return (this.idNumber - otherItem.idNumber);

    }

    // rest of code omitted for brevity
}
```

На поверхностном уровне метод `CompareTo` в нашем элементе просто возвращает разницу в их идентификационных номерах, но что делает это на практике?

Теперь, когда мы вызываем `Sort()` в объекте `List<Item>`, `List` будет автоматически вызывать метод `CompareTo` `Item` когда ему нужно определить, в какой порядок помещать объекты. Кроме того, помимо `List<T>`, любые другие объекты которые нуждаются в способности сравнивать два объекта, будут работать с `Item` потому что мы определили

способность двух разных `Item` с сравнивать друг с другом.

Прочтайте Интерфейсы онлайн: <https://riptutorial.com/ru/csharp/topic/2208/интерфейсы>

глава 74: Использование json.net

Вступление

Использование класса [JSON.net JsonConverter](#) .

Examples

Использование JsonConverter по простым значениям

Пример использования JsonConverter для десериализации свойства времени выполнения из ответа api в объект [Timespan](#) в модели Movies

JSON (

[http://www.omdbapi.com/?i=tt1663662\)](http://www.omdbapi.com/?i=tt1663662)

```
{  
    Title: "Pacific Rim",  
    Year: "2013",  
    Rated: "PG-13",  
    Released: "12 Jul 2013",  
    Runtime: "131 min",  
    Genre: "Action, Adventure, Sci-Fi",  
    Director: "Guillermo del Toro",  
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham (story)",  
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",  
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and a trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to save the world from the apocalypse.",  
    Language: "English, Japanese, Cantonese, Mandarin",  
    Country: "USA",  
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",  
    Poster: "https://images-na.ssl-images-amazon.com/images/M/MV5BMTY3MTI5NjQ4N15BM15BanBnXkFtZTcwOTU1OTU0OQ@._V1_SX300.jpg",  
    Ratings: [ {  
        Source: "Internet Movie Database",  
        Value: "7.0/10"  
    },  
    {  
        Source: "Rotten Tomatoes",  
        Value: "71%"  
    },  
    {  
        Source: "Metacritic",  
        Value: "64/100"  
    }  
],  
    Metascore: "64",
```

```
        imdbRating: "7.0",
        imdbVotes: "398,198",
        imdbID: "tt1663662",
        Type: "movie",
        DVD: "15 Oct 2013",
        BoxOffice: "$101,785,482.00",
        Production: "Warner Bros. Pictures",
        Website: "http://pacificrimmovie.com",
        Response: "True"
    }
```

Модель фильма

```
using Project.Serializers;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }

    }
}
```

RuntimeSerializer

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

```

using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if (!Int32.TryParse(value, out timespanMin))
            {
                throw new NotSupportedException();
            }

            return new TimeSpan(0, timespanMin, 0);
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
        {
            serializer.Serialize(writer, value);
        }
    }
}

```

Вызов

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

Соберите все поля объекта JSON

```

using Newtonsoft.Json.Linq;
using System.Collections.Generic;

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {

```

```

        fields = new Dictionary<string, JValue>();
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue) jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}

```

Использование:

```

var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
    Console.WriteLine($"{field.Key}: '{field.Value}'");

```

демонстрация

Для этого объекта JSON

```
{
    "User": "John",
    "Workdays": {
        "Monday": true,
        "Tuesday": true,
        "Friday": false
    },
    "Age": 42
}
```

ожидаемый результат будет:

```

User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'

```

Прочтайте Использование json.net онлайн: <https://riptutorial.com/ru/csharp/topic/9879/использование-json-net>

глава 75: Использование SQLite в C

Examples

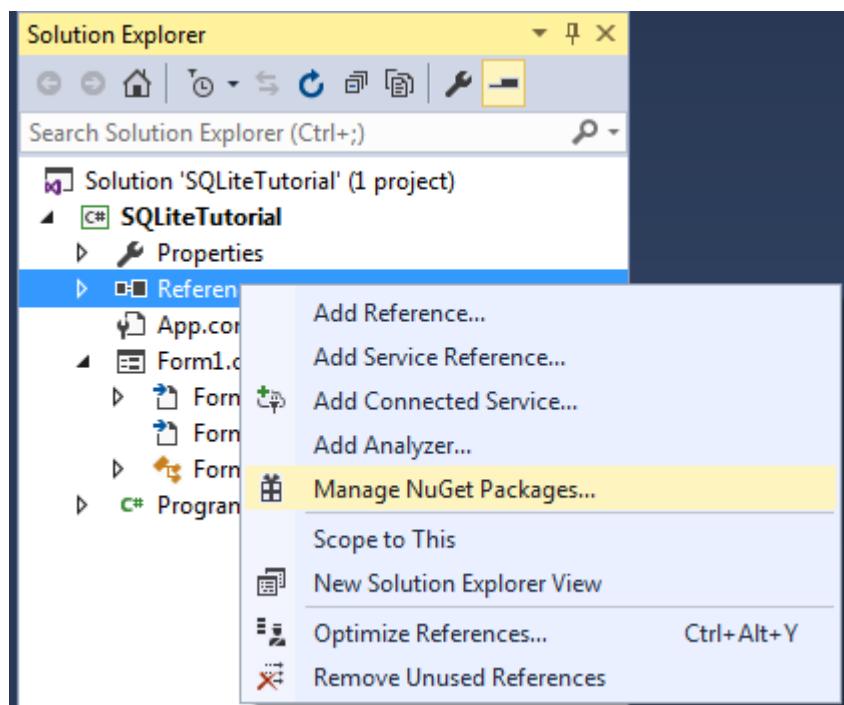
Создание простого CRUD с использованием SQLite в C

Прежде всего нам нужно добавить поддержку SQLite в наше приложение. Есть два способа сделать это

- Загрузите DLL, подходящую для вашей системы, на [странице загрузки SQLite](#), а затем добавьте в проект вручную
- Добавить зависимость от SQLite через NuGet

Мы сделаем это вторым способом

Сначала откройте меню NuGet



и найдите **System.Data.SQLite** , выберите его и нажмите **Установить**.

NuGet: SQLiteTutorial

Browse Installed Updates

SQLite Include prerelease

 **System.Data.SQLite** by SQLite Development Team, 776K downloads v1.0.102
The official SQLite database engine for both x86 and x64 along with the ADO.NET provider. This package includes support for LINQ and Entity Framework 6.

 **System.Data.SQLite.Core** by SQLite Development Team, 813K downloads v1.0.102
The official SQLite database engine for both x86 and x64 along with the ADO.NET provider.

 **System.Data.SQLite.EF6** by SQLite Development Team, 519K downloads v1.0.102
Support for Entity Framework 6 using System.Data.SQLite.

Установка также может быть выполнена из [консоли диспетчера пакетов](#) с

```
PM> Install-Package System.Data.SQLite
```

Или только для основных функций

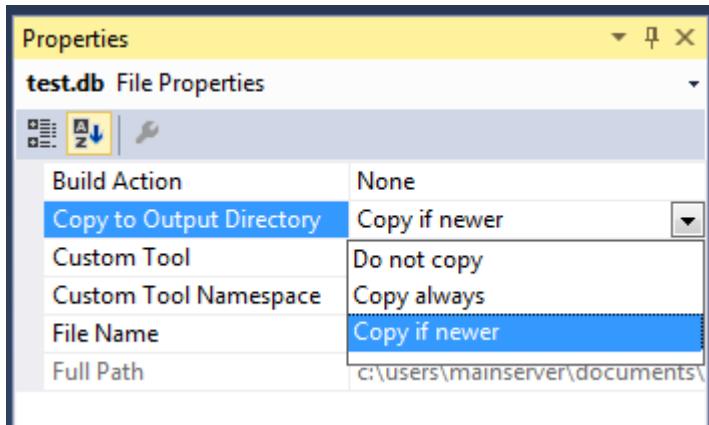
```
PM> Install-Package System.Data.SQLite.Core
```

Это все для загрузки, поэтому мы можем перейти прямо к кодированию.

Сначала создайте простую базу данных SQLite с этой таблицей и добавьте ее как файл в проект

```
CREATE TABLE User (
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL
);
```

Также не забудьте установить для параметра «**Копировать в каталог вывода**» файл «**Копировать**», если новый экземпляр «**Копировать**» всегда, исходя из ваших потребностей



Создайте класс User, который будет базовым объектом для нашей базы данных.

```
private class User
{
    public string FirstName { get; set; }
    public string Lastname { get; set; }
}
```

Мы напишем два метода для выполнения запроса, сначала для вставки, обновления или удаления из базы данных

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }
    }

    return numberOfRowsAffected;
}
```

и второй для чтения из базы данных

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;
```

```

using (var con = new SQLiteConnection("Data Source=test.db"))
{
    con.Open();
    using (var cmd = new SQLiteCommand(query, con))
    {
        foreach (KeyValuePair<string, object> entry in args)
        {
            cmd.Parameters.AddWithValue(entry.Key, entry.Value);
        }

        var da = new SQLiteDataAdapter(cmd);

        var dt = new DataTable();
        da.Fill(dt);

        da.Dispose();
        return dt;
    }
}
}

```

Теперь давайте перейдем к нашим методам **CRUD**

Добавление пользователя

```

private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName,
@lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.LastName}
    };

    return ExecuteWrite(query, args);
}

```

Редактирование пользователя

```

private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE
Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.LastName}
    };
}

```

```
        return ExecuteWrite(query, args);
    }
```

Удаление пользователя

```
private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}
```

Получение пользователя по Id

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

Выполнение запроса

```
using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))
    {
        cmd.CommandText = "query";
```

```
using (SqlDataReader dr = cmd.ExecuteReader())
{
    while(dr.Read())
    {
        //do stuff
    }
}
```

Примечание. Установка `FailIfMissing` в `true` создает файл `data.db` если отсутствует. Однако файл будет пустым. Таким образом, все необходимые таблицы необходимо воссоздать.

Прочтайте Использование SQLite в C # онлайн: <https://riptutorial.com/ru/csharp/topic/4960/использование-sqlite-в-c-sharp>

глава 76: Использование директивы

замечания

Ключевое слово `using` - это директива (этот раздел) и оператор.

Для оператора `using` (т. Е. Для инкапсуляции области объекта `IDisposable`, гарантирующего, что вне этой области объект будет очищен) см. [Использование Statement](#)

Examples

Основное использование

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console; //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

Ссылка на пространство имен

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace. i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Связывать псевдонимом с пространством имен

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
//prefixing them with only the defined alias and not the full namespace. i.e:

//...
```

```
var sb = new st.StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Доступ к статическим членам класса

6,0

Позволяет вам импортировать определенный тип и использовать статические элементы типа, не присваивая им имя типа. Это показывает пример использования статических методов:

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

И это показывает пример использования статических свойств и методов:

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

Связать псевдоним для разрешения конфликтов

Если вы используете несколько пространств имен, которые могут иметь классы с одинаковым именем (например, `System.Random` и `UnityEngine.Random`), вы можете использовать псевдоним, чтобы указать, что `Random` приходит от того или другого без использования всего пространства имен в вызове ,

Например:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

Это заставит компилятор быть неуверенным, какой `Random` для оценки новой переменной ас.

Вместо этого вы можете:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
```

Это не мешает вам позвонить другому по его полностью квалифицированному пространству имен, например:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd` будет переменной `System.Random` а `unityRandom` будет переменной `UnityEngine.Random`.

Использование псевдонимов

Вы можете использовать `using` для того, чтобы установить псевдоним для пространства имен или типа. Более подробную информацию можно найти [здесь](#).

Синтаксис:

```
using <identifier> = <namespace-or-type-name>;
```

Пример:

```
using NewType = Dictionary<string, Dictionary<string,int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

Прочтайте Использование директивы онлайн: <https://riptutorial.com/ru/csharp/topic/52/использование-директивы>

глава 77: Использование заявления

Вступление

Обеспечивает удобный синтаксис, который обеспечивает правильное использование объектов [IDisposable](#).

Синтаксис

- используя (одноразовое) {}
- используя (IDisposable disposable = new MyDisposable ()) {}

замечания

Объект в инструкции `using` должен реализовывать интерфейс `IDisposable`.

```
using(var obj = new MyObject())
{
}

class MyObject : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}
```

Более полные примеры реализации `IDisposable` можно найти в [документах MSDN](#).

Examples

Использование основов ведения

`using` - это синтаксический сахар, который позволяет гарантировать очистку ресурса без необходимости использования явного блока `try-finally`. Это означает, что ваш код будет намного чище, и вы не будете пропускать неконтролируемые ресурсы.

Стандартный `Dispose` шаблон очистки, для объектов, реализующих интерфейс `IDisposable` (который имеет базовый класс `FileStream` в .NET):

```
int Foo()
{
    var fileName = "file.txt";
```

```

{
    FileStream disposable = null;

    try
    {
        disposable = File.Open(fileName, FileMode.Open);

        return disposable.ReadByte();
    }
    finally
    {
        // finally blocks are always run
        if (disposable != null) disposable.Dispose();
    }
}
}

```

using упрощает ваш синтаксис, скрывая явное try-finally:

```

int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}

```

Подобно тому, как блоки finally всегда выполняются независимо от ошибок или возвратов, using всегда вызывает Dispose() даже в случае ошибки:

```

int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
}

```

Примечание. Так как Dispose гарантированно вызывается независимо от потока кода, рекомендуется убедиться, что Dispose никогда не генерирует исключение при реализации IDisposable. В противном случае фактическое исключение будет переопределено новым исключением, результатом чего станет кошмар отладки.

Возврат из блока

```

using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}

```

```
}
```

Из-за семантики `try..finally` к которой выполняется блок `using`, оператор `return` работает как ожидалось - возвращаемое значение вычисляется до того, как будет выполнен `finally` блок, и значение будет выбрано. Порядок оценки выглядит следующим образом:

1. Оцените тело `try`
2. Вычислить и кешировать возвращаемое значение
3. Выполнить окончательный блок
4. Возвращает значение кэшированного возврата

Тем не менее, вы не можете возвращать переменную `disposable` самостоятельно, так как она будет содержать недопустимую, удаленную ссылку - см. [Соответствующий пример](#).

Несколько операторов с одним блоком

Можно использовать несколько вложенных операторов `using` без добавления нескольких уровней вложенных фигурных скобок. Например:

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

Альтернативой является запись:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Это в точности эквивалентно первому примеру.

Примечание: Вложенный с `using` заявления может вызвать Microsoft Code Analysis правило [CS2002](#) (см [этот ответ](#) для освещения) и генерирует предупреждение. Как поясняется в связанном ответе, обычно безопасно вставлять `using` операторов.

Когда типы внутри оператора `using` имеют один и тот же тип, вы можете их разграничить и указать тип только один раз (хотя это редкость):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

Это также можно использовать, когда типы имеют общую иерархию:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

Ключевое слово `var` не может использоваться в приведенном выше примере. Возникла ошибка компиляции. Даже декларация, разделенная запятыми, не будет работать, если объявленные переменные имеют типы из разных иерархий.

Gotcha: возврат ресурса, который вы утилизируете

Ниже приведена плохая идея, потому что она должна была бы выставить переменную `db` перед ее возвратом.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

Это также может создавать более тонкие ошибки:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

Это выглядит нормально, но `catch` заключается в том, что оценка выражения LINQ является ленивой и, возможно, будет выполнена только позже, когда базовый `DBContext` уже был удален.

Короче говоря, выражение не оценивается перед тем, как покинуть `using`. Одно из возможных решений этой проблемы, которое все еще использует `using`, заключается в том, чтобы заставить выражение немедленно оценить, вызвав метод, который будет перечислять результат. Например, `ToList()`, `ToArray()` и т. д. Если вы используете новейшую версию Entity Framework, вы можете использовать `async` аналоги, такие как `ToListAsync()` ИЛИ `ToArrayAsync()`.

Ниже вы найдете пример в действии:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

```
}
```

Важно отметить, однако, что, вызывая `ToList()` или `ToArray()`, выражение будет с нетерпением оценено, что означает, что все лица с указанным возрастом будут загружены в память, даже если вы не будете их перебирать.

Использование операторов является нулевым

Вам не нужно проверять объект `IDisposable` на `null`. `using` не будет вызывать исключение, а `Dispose()` не будет вызываться:

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

Gotcha: исключение в методе `Dispose` маскирование других ошибок в использовании блоков

Рассмотрим следующий блок кода.

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

Вы можете ожидать увидеть «Не удалось выполнить операцию», напечатанную на консоли, но на самом деле вы увидите «Не удалось успешно утилизировать». поскольку метод `Dispose` все еще вызывается даже после того, как будет выбрано первое исключение.

Стоит осознавать эту тонкость, поскольку она может маскировать реальную ошибку, которая препятствовала тому, чтобы объект был удален, и затруднить его отладку.

Использование утверждений и подключений к базам данных

Ключевое слово `using` гарантирует, что ресурс, определенный внутри оператора, существует только в пределах самой инструкции. Любые ресурсы, определенные в инструкции, должны реализовывать интерфейс `IDisposable`.

Они невероятно важны при работе с любыми соединениями, реализующими интерфейс `IDisposable` поскольку он может гарантировать, что соединения не только закрыты должным образом, но и освобождаются из-за того, что их ресурсы освобождены после того, как оператор `using` выходит за рамки.

Общие классы данных `IDisposable`

Многие из следующих относятся к классам, связанным с данными, которые реализуют интерфейс `IDisposable` и являются идеальными кандидатами для `using` оператора:

- `SqlConnection`, `SqlCommand`, `SqlDataReader` и т. д.
- `OleDbConnection`, `OleDbCommand`, `OleDbDataReader` и т. д.
- `MySqlConnection`, `MySqlCommand`, `MySqlDbDataReader` и т. д.
- `DbContext`

Все они обычно используются для доступа к данным через C# и обычно встречаются во всех приложениях, ориентированных на данные. Многие другие классы, которые не упомянуты, которые реализуют те же `FooConnection`, `FooCommand`, `FooDataReader` можно ожидать, что они будут вести себя одинаково.

Общий шаблон доступа для подключений ADO.NET

Общая схема, которая может использоваться при доступе к вашим данным через соединение ADO.NET, может выглядеть следующим образом:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property";
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
```

```

{
    // Open your connection
    connection.Open();

    // Add your parameters here if necessary

    // Execute your query as a reader (again scoped with a using statement)
    using(var reader = command.ExecuteReader())
    {
        // Iterate through your results here
    }
}
}

```

Или, если вы просто выполняете простое обновление и не нуждаетесь в читателе, будет применяться одна и та же базовая концепция:

```

using(var connection = new SqlConnection("{your-connection-string}"))
{
    var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
    using(var command = new SqlCommand(query, connection))
    {
        connection.Open();

        // Add parameters here

        // Perform your update
        command.ExecuteNonQuery();
    }
}

```

Использование выражений с DataContexts

Многие ORM, такие как Entity Framework, выставляют классы абстракции, которые используются для взаимодействия с базовыми базами данных в виде классов, таких как `DbContext`. Эти контексты, как правило, также реализуют интерфейс `IDisposable` и должны воспользоваться этим, `using` возможные возможности:

```

using(var context = new YourDbContext())
{
    // Access your context and perform your query
    var data = context.Widgets.ToList();
}

```

Использование Dispose Syntax для определения настраиваемой области

Для некоторых случаев использования вы можете использовать синтаксис `using` чтобы определить пользовательскую область. Например, вы можете определить следующий класс для выполнения кода в определенной культуре.

```

public class CultureContext : IDisposable
{
}

```

```

private readonly CultureInfo originalCulture;

public CultureContext(string culture)
{
    originalCulture = CultureInfo.CurrentCulture;
    Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
}

public void Dispose()
{
    Thread.CurrentThread.CurrentCulture = originalCulture;
}
}

```

Затем вы можете использовать этот класс для определения блоков кода, которые выполняются в определенной культуре.

```

Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");

using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM

```

Примечание: поскольку мы не используем экземпляр `CultureContext` мы создаем, мы не назначаем ему переменную.

Этот метод используется [помощником](#) `BeginForm` в ASP.NET MVC.

Выполнение кода в контексте ограничений

Если у вас есть код (*подпрограмма*), который вы хотите выполнить в определенном (ограничение) контексте, вы можете использовать инъекцию зависимостей.

В следующем примере показано ограничение выполнения при открытом SSL-соединении. Эта первая часть будет в вашей библиотеке или структуре, которую вы не будете подвергать клиентскому коду.

```

public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
}

```

```

public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
{
    using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
    {
        sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

        if (!sslStream.IsAuthenticated)
        {
            throw new SecurityException("SSL tunnel not authenticated");
        }

        if (!sslStream.IsEncrypted)
        {
            throw new SecurityException("SSL tunnel not encrypted");
        }

        using (BinaryReader sslReader = new BinaryReader(sslStream))
        using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
        {
            routine(sslReader, sslWriter);
        }
    }
}

```

Теперь клиентский код, который хочет что-то сделать под SSL, но не хочет обрабатывать все данные SSL. Теперь вы можете делать все, что хотите, в туннеле SSL, например, обмен симметричным ключом:

```

public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToInt64(bytes, 0));
}

```

Вы выполняете эту процедуру следующим образом:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

Для этого вам понадобится предложение `using()`, потому что это единственный способ (кроме блока `try..finally`) вы можете гарантировать, что клиентский код (`ExchangeSymmetricKey`) никогда не выходит из строя, не располагая надлежащим образом располагаемых ресурсов. Без `using()` вы никогда не узнаете, может ли подпрограмма нарушить ограничение контекста, чтобы избавиться от этих ресурсов.

Прочтайте Использование заявления онлайн: <https://riptutorial.com/ru/csharp/topic/38/использование-заявления>

глава 78: итераторы

замечания

Итератором является метод, `get accessor` или оператор, который выполняет пользовательскую итерацию по массиву или классу коллекции, используя ключевое слово `yield`

Examples

Пример простого числового итератора

Обычным вариантом использования итераторов является выполнение некоторой операции над набором чисел. В приведенном ниже примере показано, как каждый элемент в массиве чисел можно индивидуально распечатать на консоли.

Это возможно, потому что массивы реализуют интерфейс `IEnumerable`, позволяя клиентам получать итератор для массива с использованием метода `GetEnumerator()`. Этот метод возвращает *перечислитель*, который является курсором только для чтения, только для каждого числа в массиве.

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

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

Выход

```
1
2
3
4
5
```

Также можно добиться тех же результатов, используя оператор `foreach`:

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Создание итераторов с использованием урожая

Итераторы *продуцируют* счетчики. В C # перечисления производятся путем определения методов, свойств или индексаторов, содержащих операторы `yield`.

Большинство методов возвращают управление своим вызывающим абонентам через обычные операторы `return`, которые располагают всем локальным данным для этого метода. Напротив, методы, которые используют операторы `yield` позволяют им возвращать несколько значений вызывающему по запросу, *сохраняя* при этом локальное состояние между этими возвращаемыми значениями. Эти возвращаемые значения представляют собой последовательность. В итераторах используются два типа операторов `yield`:

- `yield return`, который возвращает управление вызывающему, но сохраняет состояние. Вызов будет продолжать выполнение из этой строки, когда управление будет возвращено к нему.
- `yield break`, который функционирует аналогично нормальному оператору `return` - это означает конец последовательности. Нормальные операторы `return` сами являются незаконными в блоке итератора.

В приведенном ниже примере показан метод итератора, который может быть использован для генерации [последовательности Фибоначчи](#):

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

Этот итератор затем может быть использован для создания перечислителя последовательности Фибоначчи, который может быть использован вызывающим методом. В приведенном ниже коде показано, как можно перечислять первые десять членов в последовательности Фибоначчи:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

Выход

```
1
1
2
3
5
8
13
21
34
55
```

Прочтайте итераторы онлайн: <https://riptutorial.com/ru/csharp/topic/4243/итераторы>

глава 79: Как использовать C # Structs для создания типа Union (аналогично C Unions)

замечания

Типы Union используются на нескольких языках, особенно на С-языке, для хранения нескольких разных типов, которые могут «перекрываться» в одном и том же пространстве памяти. Другими словами, они могут содержать разные поля, все из которых начинаются с одного и того же смещения памяти, даже если они могут иметь разные длины и типы. Это имеет преимущество как для экономии памяти, так и для автоматического преобразования.

Обратите внимание на комментарии в конструкторе Struct. Порядок, в котором инициализируются поля, чрезвычайно важен. Вы хотите сначала инициализировать все остальные поля, а затем установить значение, которое вы намерены изменить в качестве последнего оператора. Поскольку поля перекрываются, последняя настройка значения - та, которая учитывается.

Examples

C-Style Unions в C

Типы Union используются на нескольких языках, таких как С-язык, для хранения нескольких разных типов, которые могут «перекрываться». Другими словами, они могут содержать разные поля, все из которых начинаются с одного и того же смещения памяти, даже если они могут иметь разные длины и типы. Это имеет преимущество как для экономии памяти, так и для автоматического преобразования. В качестве примера рассмотрим IP-адрес. Внутренне IP-адрес представлен как целое число, но иногда мы хотим получить доступ к другому байт-компоненту, как в Byte1.Byte2.Byte3.Byte4. Это работает для любых типов значений, будь то примитивы, такие как Int32 или long, или для других структур, которые вы сами определяете.

Мы можем добиться такого же эффекта в C #, используя Явные структуры компоновки.

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
```

```

[FieldOffset(1)] public byte Byte2;
[FieldOffset(2)] public byte Byte3;
[FieldOffset(3)] public byte Byte4;

public IpAddress(int address) : this()
{
    // When we init the Int, the Bytes will change too.
    Address = address;
}

// Now we can use the explicit layout to access the
// bytes separately, without doing any conversion.
public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}

```

Определив Struct таким образом, мы можем использовать его, поскольку мы будем использовать Union в C. Например, давайте создадим IP-адрес как Random Integer, а затем изменим первый токен в адресе «100», изменив его от «ABCD» до «100.BCD»:

```

var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");

```

Выход:

```

75.49.5.32 = 537211211
100.49.5.32 = 537211236

```

[Посмотреть демо](#)

Типы Union в C # также могут содержать поля Struct

Помимо примитивов, Структуры (Unions) Явного Макета в C # также могут содержать другие Структуры. Пока поле является типом значения, а не ссылкой, оно может содержаться в Союзе:

```

using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // Same definition of IpAddress, from the example above
}

// Now let's see if we can fit a whole URL into a long
// Let's define a short enum to hold protocols
enum Protocol : short { Http,Https,Ftp,Sftp,Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]

```

```

struct Service
{
    [FieldOffset(0)] public IPAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

    public Service(IPAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IPAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}//{Address}:{Port}/";
}

```

Теперь мы можем проверить, что весь Service Union вписывается в размер длинного (8 байтов).

```

var ip = new IPAddress(new Random().Next());
Console.WriteLine($"Size: {Marshal.SizeOf(ip)} bytes. Value: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;

Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");

```

[Посмотреть демо](#)

Прочтайте [Как использовать C# Structs для создания типа Union \(аналогично C Unions\)](#) онлайн: <https://riptutorial.com/ru/csharp/topic/5626/как-использовать-c-sharp-structs-для-создания-типа-union--аналогично-c-unions->

глава 80: Кастинг

замечания

Кастинг - это не то же самое, что *конвертировать*. Можно преобразовать строковое значение "-1" в целочисленное значение (-1), но это необходимо сделать с помощью таких методов библиотеки, как `Convert.ToInt32()` или `Int32.Parse()`. Это невозможно сделать, используя синтаксис casting напрямую.

Examples

Передача объекта базовому типу

Учитывая следующие определения:

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

Приведение объекта к примеру базового типа:

```
MyClass obj = new MyClass();

IMyInterface1 myClass1 = (IMyInterface1)obj;
IMyInterface2 myClass2 = (IMyInterface2)obj;

Console.WriteLine("I am : {0}", myClass1.GetName());
Console.WriteLine("I am : {0}", myClass2.GetName());

// Outputs :
// I am : IMyInterface1
// I am : IMyInterface2
```

Явное литье

Если вы знаете, что значение имеет определенный тип, вы можете явно применить его к этому типу, чтобы использовать его в контексте, где этот тип необходим.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

Если мы попытались передать `value` непосредственно в `Math.Abs()`, мы получили бы исключение для компиляции, потому что `Math.Abs()` не имеет перегрузки, которая принимает `object` в качестве параметра.

Если `value` не может быть передано в `int`, тогда вторая строка в этом примере вызовет `InvalidOperationException`.

Безопасное явное литье (оператор `as`)

Если вы не уверены, имеет ли значение тип, который, по вашему мнению, он есть, вы можете безопасно использовать его с помощью оператора `as`. Если значение не относится к этому типу, результирующее значение будет равно `null`.

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Обратите внимание, что `null` значения не имеют типа, поэтому ключевое слово `as` будет безопасно выдавать `null` при литье любого `null` значения.

Неявное литье

Значение будет автоматически передано соответствующему типу, если компилятор знает, что он всегда может быть преобразован в этот тип.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

В этом примере нам не нужно было использовать типичный синтаксис синтаксиса, поскольку компилятор знает, что все `int` могут быть переданы `object`s. На самом деле мы могли бы избежать создания переменных и передать `-1` непосредственно в качестве аргумента `Console.WriteLine()` который ожидает `object`.

```
Console.WriteLine(-1);
```

Проверка совместимости без литья

Если вам нужно знать, распространяется ли тип значения или реализует заданный тип, но вы не хотите, чтобы он действительно использовал его как этот тип, вы можете использовать оператор `is`.

```
if(value is int)
{
    Console.WriteLine(value + "is an int");
}
```

Явные числовые преобразования

Явные операторы литья могут использоваться для выполнения преобразований числовых типов, даже если они не расширяют или не реализуют друг друга.

```
double value = -1.1;
int number = (int) value;
```

Обратите внимание, что в случаях, когда тип назначения имеет меньшую точность, чем исходный тип, точность будет потеряна. Например, `-1.1` как двойное значение в приведенном выше примере становится `-1` в качестве целочисленного значения.

Кроме того, числовые преобразования основаны на типах времени компиляции, поэтому они не будут работать, если числовые типы были помещены в объекты.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

Операторы преобразования

В C# типы могут определять пользовательские *операторы преобразования*, которые позволяют преобразовывать значения в другие типы и из других типов с использованием явных или неявных отбросов. Например, рассмотрим класс, предназначенный для представления выражения JavaScript:

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
```

```
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

Если бы мы хотели создать JsExpression, представляющее сравнение двух значений JavaScript, мы могли бы сделать что-то вроде этого:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.AreEqual(doubleExpression)); // (-1 == -1.0)
```

Но мы можем добавить некоторые явные операторы преобразования в JsExpression, чтобы обеспечить простое преобразование при использовании явного литья.

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.AreEqual(doubleExpression)); // (-1 == -1.0)
```

Или мы могли бы сменить эти операторы на неявные, чтобы сделать синтаксис намного проще.

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.AreEqual(-1.0)); // (-1 == -1.0)
```

LINQ Операции литья

Предположим, у вас есть такие типы, как:

```
interface IThing { }
class Thing : IThing { }
```

LINQ позволяет вам создать проекцию, которая изменяет общий тип времени компиляции

IEnumerable<> ПОМОЩЬЮ МЕТОДОВ РАСШИРЕНИЯ `Enumerable.Cast<>()` И `Enumerable.OfType<>()` .

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

Когда `things2` оценивается, метод `Cast<>()` попытается `things2` все значения в `things` в `Thing`s. Если он встречает значение, которое невозможно выполнить, будет `InvalidOperationException` .

Когда `things3` оцениваются, метод `OfType<>()` будет делать то же самое, за исключением того, что, если он встречает значение, которое невозможно выполнить, он просто опустит это значение, а не выбросит исключение.

Из-за общего типа этих методов они не могут вызывать Операторы преобразования или выполнять числовые преобразования.

```
double[] doubles = new[] {1,2,3}.Cast<double>().ToArray(); // Throws InvalidOperationException
```

Вы можете просто выполнить бросок внутри `Select()` в качестве обходного пути:

```
double[] doubles = new[] {1,2,3}.Select(i => (double)i).ToArray();
```

Прочтайте Кастиг онлайн: <https://riptutorial.com/ru/csharp/topic/2690/кастинг>

глава 81: Ключевое слово доходности

Вступление

Когда вы используете ключевое слово `yield` в инструкции, вы указываете, что метод, оператор или `get accessor`, в котором он отображается, является итератором.

Использование `yield` для определения итератора устраняет необходимость в явном дополнительном классе (класс, который содержит состояние для перечисления) при реализации шаблона `IEnumerable` и `IEnumerator` для пользовательского типа коллекции.

Синтаксис

- `return return [TYPE]`
- разрыв дохода

замечания

Полагая ключевое слово `yield` в методе с типом возвращаемого значения `IEnumerable` , `IEnumerable<T>` , `IEnumerator` ИЛИ `IEnumerator<T>` сообщает компилятору сгенерировать реализацию возвращаемого типа (`IEnumerable` ИЛИ `IEnumerator`), который при циклическом запуске метод до каждого «урожая», чтобы получить каждый результат.

Ключевое слово `yield` полезно, когда вы хотите вернуть «следующий» элемент теоретически неограниченной последовательности, поэтому вычисление всей последовательности заблаговременно было бы невозможно или при вычислении полной последовательности значений перед возвратом приводило бы к нежелательной паузе для пользователя ,

`yield break` также может быть использован для прекращения последовательности в любое время.

Поскольку ключевому слову `yield` требуется тип интерфейса итератора в качестве возвращаемого типа, например `IEnumerable<T>` , вы не можете использовать его в методе `async`, так как это возвращает объект `Task<IEnumerable<T>>` .

дальнейшее чтение

- <https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx>

Examples

Простое использование

Ключевое слово `yield` используется для определения функции, которая возвращает `IEnumerable` или `IEnumerator` (а также их производные обобщенные варианты), значения которых генерируются лениво, когда вызывающий выполняет итерацию по возвращенной коллекции. Подробнее о цели в разделе [замечаний](#).

В следующем примере есть оператор возврата доходности, который находится внутри цикла `for`.

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Тогда вы можете назвать это:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

Консольный выход

```
4
5
6
...
14
```

Живая демонстрация на .NET скрипке

Каждая итерация тела оператора `foreach` создает вызов функции итератора `Count`.

Каждый вызов функции итератора переходит к следующему исполнению оператора `yield return`, который возникает во время следующей итерации цикла `for`.

Более подходящее использование

```
public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name
FROM Users"))
    {
        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            yield return new User(id, name);
        }
    }
}
```

```
}
```

Конечно, есть другие способы получить `IEnumerable<User>` из базы данных SQL - это просто демонстрирует, что вы можете использовать `yield` чтобы превратить все, что имеет семантику «последовательность элементов» в `IEnumerable<T>` которую кто-то может перебирать через ,

Раннее прекращение

Вы можете расширить функциональность существующих методов `yield` , передав одно или несколько значений или элементов, которые могли бы определить условие завершения в функции, вызвав `yield break` чтобы остановить выполнение внутреннего цикла.

```
public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}
```

Вышеуказанный метод будет перебирать из заданной `start` позиции до тех пор, пока не будет `earlyTerminationSet` **одно из значений** в `earlyTerminationSet` .

```
// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1,terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}
```

Выход:

1

2
3
4
5
6

Живая демонстрация на .NET скрипке

Правильно проверяющие аргументы

Метод итератора не выполняется до тех пор, пока не будет перечислено возвращаемое значение. Поэтому выгодно утверждать предпосылки за пределами итератора.

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));

    return CountCore(start, count);
}

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

Вызов бокового кода (использование):

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

Выход:

1
2
3
4
5
6
7

8
9
10

Живая демонстрация на .NET скрипке

Когда метод использует `yield` для генерации перечислимого, компилятор создает конечный автомат, который при повторном запуске будет работать с кодом до `yield`. Затем он возвращает полученный элемент и сохраняет его состояние.

Это означает, что вы не узнаете о недопустимых аргументах (передаете `null` и т. д.) При первом вызове метода (поскольку это создает конечный автомат), только когда вы пытаетесь получить доступ к первому элементу (потому что только тогда код внутри метода `get` запускается конечным автоматом). Обернув его обычным методом, который сначала проверяет аргументы, вы можете проверить их при вызове метода. Это пример неудачи.

При использовании C # 7+ функция `CountCore` может быть удобно скрыта в функции `Count` как *локальная функция*. См. Пример [здесь](#).

Возвращает другой Перечислимый в методе, возвращающем `Enumerable`

```
public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}
```

Ленивая оценка

Только когда оператор `foreach` переходит к следующему элементу, блок итератора оценивает до следующего оператора `yield`.

Рассмотрим следующий пример:

```
private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}
```

```

        }
    }

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
    Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
}

```

Это приведет к выводу:

```

Запуск итерации
Внутри итератора: 0
Внутри foreach: 0
Внутри итератора: 1
Внутри foreach: 1
Внутри итератора: 2
Внутри foreach: 2

```

[Посмотреть демо](#)

Как следствие:

- «Начальная итерация» сначала печатается, даже если метод итератора был вызван до того, как строка напечатала его, потому что строка `Integers().Take(3);` на самом деле не запускает итерацию (не был `IEnumerator.MoveNext()` **вызован** `IEnumerator.MoveNext()`)
- Линии, печатающие на консоли, чередуются между тем, который находится внутри метода итератора, и внутри `foreach`, а не все внутри метода итератора, оценивая сначала
- Эта программа завершается из-за метода `.Take()`, хотя метод итератора имеет значение `while true` которое никогда не прерывается.

Попробуйте ... наконец-то

Если метод итератора имеет выход внутри `try...finally`, возвращаемый `IEnumerator` выполнит оператор `finally` когда `Dispose` вызывается на нем, если текущая точка оценки находится внутри блока `try`.

С учетом функции:

```

private IEnumerable<int> Numbers()
{
    yield return 1;
}

```

```
try
{
    yield return 2;
    yield return 3;
}
finally
{
    Console.WriteLine("Finally executed");
}
}
```

При звонке:

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Затем он печатает:

1

[Посмотреть демо](#)

При звонке:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Затем он печатает:

1

2

Наконец выполнен

[Посмотреть демо](#)

Использование yield для создания IEnumator при внедрении IEnumarable

Интерфейс `IEnumerable<T>` имеет единственный метод `GetEnumerator()`, который возвращает

```
IEnumerable<T> .
```

Хотя ключевое слово `yield` можно использовать для непосредственного создания `IEnumerable<T>`, его также можно использовать точно так же, чтобы создать `IEnumerable<T>`. Единственное, что изменяется, это тип возвращаемого метода.

Это может быть полезно, если мы хотим создать собственный класс, который реализует `IEnumerable<T>`:

```
public class PrintingEnumerable<T> : IEnumerable<T>
{
    private IEnumerable<T> _wrapped;

    public PrintingEnumerable(IEnumerable<T> wrapped)
    {
        _wrapped = wrapped;
    }

    // This method returns an IEnumerable<T>, rather than an IEnumerable<T>
    // But the yield syntax and usage is identical.
    public IEnumerator<T> GetEnumerator()
    {
        foreach(var item in _wrapped)
        {
            Console.WriteLine("Yielding: " + item);
            yield return item;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

(Обратите внимание, что этот конкретный пример является просто иллюстративным и может быть более чисто реализован с помощью одного метода итератора, возвращающего `IEnumerable<T>`.)

Простая оценка

Ключевое слово `yield` позволяет ленивую оценку коллекции. Принудительная загрузка всей коллекции в память называется **непрерывной оценкой**.

Следующий код показывает это:

```
IEnumerable<int> myMethod()
{
    for(int i=0; i <= 8675309; i++)
    {
        yield return i;
    }
}
```

```
// define the iterator
var it = myMethod.Take(3);
// force its immediate evaluation
// list will contain 0, 1, 2
var list = it.ToList();
```

Вызов `ToList` , `ToDictionary` **ИЛИ** `ToArray` **заставит немедленную оценку перечисления, извлекая все элементы в коллекцию.**

Ленивый пример оценки: номера Фибоначчи

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numerics

namespace ConsoleApplication33
{
    class Program
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

Как это работает под капотом (я рекомендую декомпилировать полученный файл .exe в инструменте IL Disassembler):

1. Компилятор C# генерирует класс, реализующий `IEnumerable<BigInteger>` и `IEnumerator<BigInteger>` (`<Fibonacci>d__0` in ildasm).
2. Этот класс реализует конечный автомат. Состояние состоит из текущей позиции в методе и значениях локальных переменных.
3. Самый интересный код - в методе `bool IEnumerator.MoveNext()`. В основном, что `MoveNext()` делает:
 - Восстанавливает текущее состояние. Переменные типа `prev` и `current` становятся полями в нашем классе (`<current>5__2` и `<prev>5__1` in ildasm). В нашем методе мы

- имеем две позиции (`<>1__state`): сначала в открывающей фигурной скобке, второй - при `yield return`.
- Выполняет код до следующего `yield return` **или** `yield break / }`.
 - Для `yield return` результата `yield return` значение сохраняется, поэтому `Current` свойство может вернуть его. `true` возвращается. В этот момент текущее состояние сохраняется снова для следующего вызова `MoveNext`.
 - Для метода `yield break / }` метод возвращает `false` итерация выполняется.

Также обратите внимание, что 10001-е число составляет 468 байт. State machine сохраняет только `current` и `prev` переменные как поля. Хотя если мы хотим сохранить все числа в последовательности от первой до 10000-й, объем потребляемой памяти будет превышать 4 мегабайта. Такая ленивая оценка, если она правильно используется, в некоторых случаях может уменьшить объем памяти.

Разница между разрывом и выходом

Использование `yield break` в отличие от `break` может быть не столь очевидным, как можно подумать. В Интернете много плохих примеров, где использование этих двух взаимозаменяемых и на самом деле не демонстрирует разницы.

Запутанная часть состоит в том, что оба ключевых слова (или ключевые фразы) имеют смысл только внутри циклов (`foreach`, `while` ...). Когда выбирать один за другим?

Важно понимать, что как только вы используете ключевое слово `yield` в методе, вы эффективно превращаете этот метод в [итератор](#). Единственная цель такого метода состоит в том, чтобы затем перебирать конечный или бесконечный набор и выводить (выводить) его элементы. Как только цель будет выполнена, нет причин продолжать выполнение метода `}`. Но иногда вы хотите досрочно завершить метод. В обычном (без повторения) методе вы должны использовать ключевое слово `return`. Но вы не можете использовать `return` в итераторе, вы должны использовать `yield break`. Другими словами, `yield break` для итератора совпадает с `return` стандартного метода. Принимая во внимание, что оператор `break` просто завершает ближайший цикл.

Давайте посмотрим несколько примеров:

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
    }
}
```

```
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}
```

```
/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9,10}</returns>
public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Terminates just the loop
            break;
        }
    }
    // Execution continues
    yield return 10;
}
```

Прочтайте Ключевое слово доходности онлайн: <https://riptutorial.com/ru/csharp/topic/61/ключевое-слово-доходности>

глава 82: Ключевые слова

Вступление

Ключевые слова предопределены, зарезервированные идентификаторы со специальным значением для компилятора. Они не могут использоваться в качестве идентификаторов в вашей программе без префикса @ . Например, @if является юридическим идентификатором, но не ключевым словом if .

замечания

С # имеет предопределенную коллекцию «ключевых слов» (или зарезервированных слов), каждая из которых имеет специальную функцию. Эти слова нельзя использовать в качестве идентификаторов (имена для переменных, методов, классов и т. Д.), Если только не префикс @ .

- abstract
- as
- base
- bool
- break
- byte
- case
- catch
- char
- checked
- class
- const
- continue
- decimal
- default
- delegate
- do
- double
- else
- enum
- event
- explicit
- extern
- false
- finally
- fixed
- float
- for
- foreach
- goto
- if
- implicit
- in

- `int`
- `interface`
- `internal`
- `is`
- `lock`
- `long`
- `namespace`
- `new`
- `null`
- `object`
- `operator`
- `out`
- `override`
- `params`
- `private`
- `protected`
- `public`
- `readonly`
- `ref`
- `return`
- `sbyte`
- `sealed`
- `short`
- `sizeof`
- `stackalloc`
- `static`
- `string`
- `struct`
- `switch`
- `this`
- `throw`
- `true`
- `try`
- `typeof`
- `uint`
- `ulong`
- `unchecked`
- `unsafe`
- `ushort`
- `using (директива)`
- `using (заявление)`
- `virtual`
- `void`
- `volatile`
- `when`
- `while`

Помимо них, C # также использует некоторые ключевые слова, чтобы обеспечить конкретное значение кода. Они называются контекстуальными ключевыми словами. Контекстные ключевые слова могут использоваться как идентификаторы и не должны иметь префикс `@` при использовании в качестве идентификаторов.

- `add`
- `alias`

- `ascending`
- `async`
- `await`
- `descending`
- `dynamic`
- `from`
- `get`
- `global`
- `group`
- `into`
- `join`
- `let`
- `nameof`
- `orderby`
- `partial`
- `remove`
- `select`
- `set`
- `value`
- `var`
- `where`
- `yield`

Examples

stackalloc

`stackalloc` СЛОВО `stackalloc` создает область памяти в стеке и возвращает указатель на начало этой памяти. Выделенная память стека автоматически удаляется, когда открывается область, в которой она была создана.

```
//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;
ptr[2] = 232;
...
```

Используется в небезопасном контексте.

Как и во всех указателях на C #, проверки чтения и присваивания не проверяются. Чтение за пределами выделенной памяти будет иметь непредсказуемые результаты - он может получить доступ к произвольному местоположению в памяти или может вызвать исключение нарушения доступа.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
```

```
ptr[-1] = 2;
```

Выделенная память стека автоматически удаляется, когда открывается область, в которой она была создана. Это означает, что вы никогда не должны возвращать память, созданную с помощью `stackalloc`, или хранить ее за пределами срока действия области.

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
    //ptr is now an invalid pointer, using it in any way will have
    //unpredictable results. This is exactly the same as accessing beyond
    //the bounds of the pointer.
    var ptr = Leak();
}
```

`stackalloc` может использоваться только при объявлении и инициализации переменных. Ниже не действует:

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

Примечания:

`stackalloc` следует использовать только для оптимизации производительности (либо для вычисления, либо для взаимодействия). Это связано с тем, что:

- Сборщик мусора не требуется, поскольку память распределяется в стеке, а не в куче - память освобождается, как только переменная выходит из области видимости
- Быстрее выделять память в стеке, а не кучу
- Увеличьте вероятность попадания кеша на процессор из-за локальности данных

летучий

Добавление ключевого слова `volatile` в поле указывает компилятору, что значение поля может быть изменено несколькими отдельными потоками. Основной целью ключевого слова `volatile` является предотвращение оптимизации компилятора, предполагающего только однопоточный доступ. Использование `volatile` гарантирует, что значение поля является самым последним значением, которое доступно, и значение не подлежит кешированию, которое является энергонезависимым значением.

Рекомендуется отметить каждую переменную, которая может использоваться

несколькими потоками как `volatile` чтобы предотвратить непредвиденное поведение из-за закулисных оптимизаций. Рассмотрим следующий блок кода:

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

В приведенном выше кодовом блоке компилятор читает операторы `x = 5` и `y = x + 10` и определяет, что значение `y` всегда будет равным 15. Таким образом, он будет оптимизировать последний оператор как `y = 15`. Однако переменная `x` фактически является `public` полем, а значение `x` может быть изменено во время выполнения через другой поток, действующий на это поле отдельно. Теперь рассмотрим этот модифицированный блок кода. Обратите внимание, что поле `x` теперь объявлено `volatile`.

```
public class Example
{
    public volatile int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler no longer optimizes this statement
        var y = x + 10;

        /* the value of x and y will always be the correct values */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Теперь компилятор ищет поля **чтения** поля `x` и гарантирует, что текущее значение поля всегда извлекается. Это гарантирует, что даже если несколько потоков считывают и записывают в это поле, текущее значение `x` всегда извлекается.

`volatile` может использоваться только для полей в `class` es или `struct` s. Ниже **не действует**:

```
public void MyMethod()
{
    volatile int x;
}
```

`volatile` может применяться только к полям следующих типов:

- ссылочные типы или типовые параметры, известные как ссылочные типы
- примитивные типы, такие как `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` И `bool`
- типы перечислений на основе `byte`, `sbyte`, `short`, `ushort`, `int` ИЛИ `uint`
- `IntPtr` И `UIntPtr`

Примечания:

- Модификатор `volatile` обычно используется для поля, к которому обращаются несколько потоков, без использования оператора блокировки для сериализации доступа.
- Ключевое слово `volatile` может применяться к полям ссылочных типов
- Ключевое слово `volatile` не будет работать на 64-битных примитивах на 32-битной платформе Atom. Блокированные операции, такие как `Interlocked.Read` И `Interlocked.Exchange` все равно должны использоваться для безопасного многопоточного доступа на этих платформах.

фиксированный

Фиксированный оператор фиксирует память в одном месте. Объекты в памяти обычно движутся вокруг, что делает сборку мусора возможным. Но когда мы используем небезопасные указатели на адреса памяти, эта память не должна перемещаться.

- Мы используем фиксированный оператор, чтобы убедиться, что сборщик мусора не перемещает строковые данные.

Фиксированные переменные

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

Используется в небезопасном контексте.

Размер фиксированного массива

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed` может использоваться только для полей в `struct` (также должен использоваться в небезопасном контексте).

дефолт

Для классов, интерфейсов, делегатов, массивов, с нулевым значением (например, `int?`) и типов указателей по `default(TheType)` возвращается значение `null`:

```
class MyClass {}  
Debug.Assert(default(MyClass) == null);  
Debug.Assert(default(string) == null);
```

Для `structs` и `default(TheType)` возвращают то же, что и `new TheType()`:

```
struct Coordinates  
{  
    public int X { get; set; }  
    public int Y { get; set; }  
}  
  
struct MyStruct  
{  
    public string Name { get; set; }  
    public Coordinates Location { get; set; }  
    public Coordinates? SecondLocation { get; set; }  
    public TimeSpan Duration { get; set; }  
}  
  
var defaultStruct = default(MyStruct);  
Debug.Assert(defaultStruct.Equals(new MyStruct()));  
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));  
Debug.Assert(defaultStruct.Location.X == 0);  
Debug.Assert(defaultStruct.Location.Y == 0);  
Debug.Assert(defaultStruct.SecondLocation == null);  
Debug.Assert(defaultStruct.Name == null);  
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)` может быть особенно полезно, когда `T` является общим параметром, для которого не существует ограничений, чтобы определить, является ли `T` ссылочным типом или типом значения, например:

```
public T GetResourceOrDefault<T>(string resourceName)  
{  
    if (ResourceExists(resourceName))  
    {  
        return (T)GetResource(resourceName);  
    }  
    else  
    {  
        return default(T);  
    }  
}
```

ТОЛЬКО ДЛЯ ЧТЕНИЯ

Ключевое слово `readonly` является модификатором поля. Когда объявление поля включает модификатор `readonly`, присваивания этому полю могут выполняться только как часть объявления или в конструкторе того же класса.

Ключевое слово `readonly` отличается от ключевого слова `const`. Поле `const` может быть инициализировано только при объявлении поля. Поле `readonly` может быть инициализировано либо в объявлении, либо в конструкторе. Поэтому поля `readonly` могут иметь разные значения в зависимости от используемого конструктора.

Ключевое слово `readonly` часто используется при инъекции зависимостей.

```
class Person
{
    readonly string _name;
    readonly string _surname = "Surname";

    Person(string name)
    {
        _name = name;
    }
    void ChangeName()
    {
        _name = "another name"; // Compile error
        _surname = "another surname"; // Compile error
    }
}
```

Примечание. Объявление поля `readonly` не подразумевает *неизменность*. Если поле является *ссылочным типом*, то **содержимое** объекта может быть изменено. *Readonly* обычно используется для предотвращения **перезаписи** и назначения объекта только во время **создания экземпляра** этого объекта.

Примечание. Внутри конструктора поле `readonly` можно переназначить

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

КАК

Ключевое слово `as` - это оператор, похожий на `приведение`. Если бросок невозможен, использование создает `as null` а не приводит к `InvalidOperationException`.

`expression as type` ЭКВИВАЛЕНТНО `expression is type ? (type)expression : (type)null` с оговоркой, что, `as` действует только на ссылочные преобразования, обнуляемого преобразования и преобразования бокса. Определенные пользователем преобразования не поддерживаются; вместо этого следует использовать обычный литой состав.

Для расширения выше компилятор генерирует код таким образом, что `expression` будет оцениваться только один раз и использовать проверку одного динамического типа (в отличие от двух в примере выше).

`as` может быть полезно при ожидании аргументации для облегчения нескольких типов. В частности, она предоставляет многочисленные пользовательские опции - вместо того, чтобы проверять каждую возможность с `is` перед заливкой, или просто литья и отлова исключения. Лучше всего использовать «как» при кастинге / проверке объекта, который вызовет только одно неуправляемое наказание. Использование `is` в проверке, тогда литье приведет к двум неуправляемым штрафам.

Если ожидается, что аргумент будет экземпляром определенного типа, регулярный листинг предпочтителен, так как его цель более понятна читателю.

Так как вызов `as` может привести к `null`, всегда проверить результат, чтобы избежать `NullReferenceException`.

Пример использования

```
object something = "Hello";
Console.WriteLine(something as string);           //Hello
Console.WriteLine(something as Nullable<int>); //null
Console.WriteLine(something as int?);             //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

Живая демонстрация на .NET скрипке

Эквивалентный пример без использования `as`:

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

Это полезно при переопределении функции `Equals` в пользовательских классах.

```
class MyCustomClass
{
    public override bool Equals(object obj)
    {
```

```

MyCustomClass customObject = obj as MyCustomClass;

// if it is null it may be really null
// or it may be of a different type
if (Object.ReferenceEquals(null, customObject))
{
    // If it is null then it is not equal to this instance.
    return false;
}

// Other equality controls specific to class
}

}

```

является

Проверяет, совместим ли объект с данным типом, то есть, если объект является экземпляром типа `BaseInterface` или типом, который происходит от `BaseInterface`:

```

interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass); // True
Console.WriteLine(d is BaseClass); // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object); // True
Console.WriteLine(d is string); // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass); // False
Console.WriteLine(b is BaseClass); // True
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object); // True
Console.WriteLine(b is string); // False

```

Если целью броска является использование объекта, лучше всего использовать ключевое слово « `as` »,

```

interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass); // True - valid use of 'is'
Console.WriteLine(d is BaseClass); // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

```

```
if(asD!=null){  
    asD.Method(); //prefered method since you incur only one unboxing penalty  
}
```

Но из функции `pattern matching` C # 7 расширяет оператор `is` для проверки типа и объявления новой переменной в одно и то же время. Эта же часть кода с C # 7:

7,0

```
if(d is BaseClass asD ){  
    asD.Method();  
}
```

ТИП

Возвращает `Type` объекта без необходимости его экземпляра.

```
Type type = typeof(string);  
Console.WriteLine(type.FullName); //System.String  
Console.WriteLine("Hello".GetType() == type); //True  
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

Const

`const` используется для представления значений, **которые никогда не будут меняться в течение всего жизненного цикла программы**. Его значение является постоянным от времени **компиляции**, в отличие от ключевого слова `readonly`, значение которого постоянно от времени выполнения.

Например, поскольку скорость света никогда не изменится, мы можем сохранить его в константе.

```
const double c = 299792458; // Speed of light  
  
double CalculateEnergy(double mass)  
{  
    return mass * c * c;  
}
```

Это по существу то же самое, что и `return mass * 299792458 * 299792458`, поскольку компилятор будет напрямую подставлять `c` своим постоянным значением.

В результате `c` не может быть изменен после объявления. Ниже приведена ошибка времени компиляции:

```
const double c = 299792458; // Speed of light  
  
c = 500; //compile-time error
```

Константа может иметь префикс с теми же модификаторами доступа, что и методы:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const` элементы `static` по своей природе. Однако использование `static` явно запрещено.

Вы также можете определить методы-локальные константы:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

Они не могут иметь префикса `c` `private` или `public` ключевым словом, поскольку они неявно локальны для метода, в котором они определены.

Не все типы могут использоваться в объявлении `const`. Допустимыми типами значений являются предопределенные типы `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` и все типы `enum`. Пытаться объявить `const` члены с другими типами значений (например, `TimeSpan` или `Guid`) не удастся во время компиляции.

Для специальной заданной `string` ссылочного типа константы могут быть объявлены с любым значением. Для всех других ссылочных типов константы могут быть объявлены, но всегда должны иметь значение `null`.

Поскольку значения `const` известны во время компиляции, они разрешены как ярлыки `case` в инструкции `switch`, в качестве стандартных аргументов для необязательных параметров, в качестве аргументов спецификаций атрибутов и т. д.

Если значения `const` используются для разных сборок, следует соблюдать осторожность при управлении версиями. Например, если сборка А определяет `public const int MaxRetries = 3;`, а сборка В использует эту константу, то, если значение `MaxRetries` позже будет изменено на 5 в сборке А (которое затем будет скомпилировано), это изменение не будет эффективным в сборке В, если сборка В также не будет скомпилирована (с ссылка на новую версию А).

По этой причине, если значение может измениться в будущих версиях программы, и если значение должно быть общедоступным, не объявляйте это значение `const` если не знаете, что все зависимые сборки будут перекомпилированы всякий раз, когда что-то изменится. Альтернативой является использование `static readonly` вместо `const`, которое разрешено во время выполнения.

Пространство имен

Ключевое слово `namespace` - это организационная конструкция, которая помогает нам понять, как устроена кодовая база. Пространства имен в C# являются виртуальными пространствами, а не находятся в физической папке.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
    }
}
```

Пространства имен в C# также могут быть записаны в цепочке синтаксиса. Ниже приведено следующее:

```
namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}
```

Попытаться, поймать, наконец, бросить

`try` , `catch` , `finally` , и `throw` разрешите вам обрабатывать исключения в вашем коде.

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block
// will be executed.
catch (FormatException ex)
```

```

{
    // Throw is a keyword that will manually throw an exception, triggering any catch block
    // that is
    // waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,
// with no type specified, catches any exception that hasn't already been caught
// in a prior catch block.
catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try
has
// succeeded in running all commands or after all exceptions have been caught.
finally
{
    processor.Dispose();
}

```

Примечание. Ключевое слово `return` можно использовать в блоке `try`, и блок `finally` все равно будет выполнен (как раз перед возвратом). Например:

```

try
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}

```

Заявление `connection.Close()` **будет выполняться до результата** `connection.Get(query)`
Возвращается `connection.Get(query)`.

Продолжить

Сразу же передайте управление следующей итерации конструкции замкнутого контура (`for`, `foreach`, `do`, `while`):

```

for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}

```

Выход:

6
7
8
9

[Живая демонстрация на .NET скрипке](#)

```
var stuff = new [] {"a", "b", null, "c", "d"};  
  
foreach (var s in stuff)  
{  
    if (s == null)  
    {  
        continue;  
    }  
    Console.WriteLine(s);  
}
```

Выход:

б
с
д

[Живая демонстрация на .NET скрипке](#)

ref, out

Ключевые слова `ref` и `out` заставляют аргумент передаваться по ссылке, а не по значению. Для типов значений это означает, что значение переменной может быть изменено вызываемым пользователем.

```
int x = 5;  
ChangeX(ref x);  
// The value of x could be different now
```

Для ссылочных типов экземпляр в переменной может быть изменен не только (как в случае без `ref`), но также может быть полностью заменен:

```
Address a = new Address();  
ChangeFieldInAddress(a);  
// a will be the same instance as before, even if it is modified  
CreateANewInstance(ref a);  
// a could be an entirely new instance now
```

Основное различие между `out` и `ref` ключевым словом, что `ref` требует переменной для инициализации вызывающего, в то время как `out` передает, что ответственность перед вызываемым.

Чтобы использовать параметр `out` , и определение метода, и метод вызова должны явно использовать ключевое слово `out` .

```
int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number); // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number); // number = 34

void AddOneByRef(ref int value)
{
    value++;
}

void SetByOut(out int value)
{
    value = 34;
}
```

[Живая демонстрация на .NET скрипке](#)

Ниже *не* компилируется, так как `out` параметров должен иметь значение , присвоенное до возврата метода (он будет компилировать с помощью `ref` , а):

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

использование ключевого слова как Generic Modifier

ключевое слово `out` также может использоваться в параметрах родового типа при определении общих интерфейсов и делегатов. В этом случае ключевое слово `out` указывает, что параметр типа является ковариантным.

Ковариация позволяет использовать более производный тип, чем тот, который задан общим параметром. Это позволяет неявное преобразование классов, которые реализуют варианты интерфейсов и неявное преобразование типов делегатов. Ковариантность и контравариантность поддерживаются для ссылочных типов, но они не поддерживаются для типов значений. - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

checked, unchecked

checked и unchecked ключевые слова определяют, как операции обрабатывают математическое переполнение. «Переполнение» в контексте checked и unchecked ключевых слов - это когда целочисленная арифметическая операция приводит к значению, которое больше по величине, чем может представлять целевой тип данных.

Когда переполнение происходит в пределах checked блока (или когда компилятор настроен на глобальное использование проверенной арифметики), исключение выдается для предупреждения о нежелательном поведении. Между тем, в unchecked блоке, переполнение не работает: никаких исключений не выбрасывается, и значение просто переносится на противоположную границу. Это может привести к тонким, трудно найти ошибкам.

Поскольку большинство арифметических операций выполняются по значениям, которые не являются большими или достаточно малыми для переполнения, большую часть времени нет необходимости явно определять блок как checked . Необходимо проявлять осторожность при выполнении арифметики на неограниченном входе, что может вызвать переполнение, например, при выполнении арифметики в рекурсивных функциях или при вводе пользователя.

Не checked и unchecked checked влияние арифметических операций с плавающей запятой.

Когда блок или выражение объявляются как unchecked , любые арифметические операции внутри него допускают переполнение без возникновения ошибки. Примером, когда это поведение является желательным , является вычисление контрольной суммы, когда значение разрешено «обертывать» во время вычисления:

```
byte Checksum(byte[] data) {  
    byte result = 0;  
    for (int i = 0; i < data.Length; i++) {  
        result = unchecked(result + data[i]); // unchecked expression  
    }  
    return result;  
}
```

Одним из наиболее распространенных применений для unchecked является реализация настраиваемого переопределения для object.GetHashCode() , типа контрольной суммы. Вы можете увидеть использование ключевого слова в ответах на этот вопрос: [Каков наилучший алгоритм для переопределенного System.Object.GetHashCode?](#) ,

Когда блок или выражение объявляются как checked , любая арифметическая операция, которая вызывает переполнение, приводит к тому, что генерируется OverflowException .

```
int SafeSum(int x, int y) {  
    checked { // checked block  
        return x + y;  
    }  
}
```

Как проверенные, так и непроверенные могут быть в форме блока и выражения.

Проверенные и непроверенные блоки не влияют на вызываемые методы, только операторы, вызываемые непосредственно в текущем методе. Например, `Enum.ToObject()`, `Convert.ToInt32()` и определяемые пользователем операторы не зависят от настраиваемых проверенных / непроверенных контекстов.

Примечание. По умолчанию поведение по умолчанию переполнения (проверено или не отмечено) может быть изменено в **свойствах проекта** или с помощью переключателя командной строки **[+ | -]**. Обычно по умолчанию проверяются операции для отладочных сборников и неконтролируемые для выпуска сборок. `checked` и `unchecked` ключевые слова будут использоваться только тогда, когда подход по умолчанию не применяется, и для обеспечения правильности требуется четкое поведение.

ИДТИ К

`goto` может использоваться для перехода к определенной строке внутри кода, указанной меткой.

`goto` **КАК:**

Этикетка:

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}
```

[Живая демонстрация на .NET скрипке](#)

Дело:

```
enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;

    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
}
```

Живая демонстрация на .NET скрипке

Это особенно полезно при выполнении множественных действий в инструкции `switch`, поскольку C # не поддерживает [распашные блоки `case`](#).

Исправление исключений

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
        goto retry;
    }
    throw;
}
```

Живая демонстрация на .NET скрипке

Подобно многим языкам, использование ключевого слова `goto` не рекомендуется, за исключением случаев, описанных ниже.

[Допустимые действия `goto`](#) которые применяются к C #:

- Провал в инструкции `switch`.
- Многоуровневый разрыв. LINQ часто может использоваться вместо этого, но обычно он имеет худшую производительность.
- Освобождение ресурсов при работе с развернутыми объектами низкого уровня. В C # объекты низкого уровня обычно должны быть обернуты отдельными классами.
- Конечные автоматы, например парсеры; используемый внутренне компилятором, созданным машинами состояния `async / wait`.

перечисление

Ключевое слово `enum` сообщает компилятору, что этот класс наследуется от абстрактного класса `Enum` без необходимости его явного наследования. `Enum` является потомком `ValueType`, который предназначен для использования с определенным набором именованных констант.

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

Вы можете указать конкретное значение для каждого из них (или некоторых из них):

```
public enum NotableYear
{
    EndOfWwI = 1918,
    EndOfWwII = 1945,
}
```

В этом примере я опустил значение для 0, это, как правило, плохая практика. `enum` всегда будет иметь значение по умолчанию, выраженное явным преобразованием (`YourEnumType`) 0, где `YourEnumType` - ваш объявленный тип `enum`. Без значения 0 определяется, что `enum` не будет иметь определенного значения при инициализации.

Основной тип `enum` умолчанию - `int`, вы можете изменить базовый тип на любой интегральный тип, включая `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`. Ниже перечисление с `byte` базового типа:

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Также обратите внимание, что вы можете конвертировать в / из базового типа просто с помощью броска:

```
int value = (int)NotableYear.EndOfWwI;
```

По этим причинам вам лучше всегда проверять правильность `enum` когда вы просматриваете библиотечные функции:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

база

Ключевое слово `base` используется для доступа к элементам из базового класса. Он обычно используется для вызова базовых реализаций виртуальных методов или для указания того, какой базовый конструктор следует вызывать.

Выбор конструктора

```
public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in
        Child's constructor
        Console.WriteLine(message);
    }
}
```

Вызов базовой реализации виртуального метода

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

Можно использовать ключевое слово `base` для вызова базовой реализации из любого метода. Это связывает вызов метода непосредственно с базовой реализацией, а это означает, что даже если новые дочерние классы переопределяют виртуальный метод, базовая реализация все равно будет вызвана, поэтому это нужно использовать с осторожностью.

```
public class Parent
{
    public virtual int VirtualMethod()
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }
}
```

```

    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());

        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());

        // Notice that the call to NormalMethod below still returns the value
        // from the extreme base class even though the method has been overridden
        // in the child class.
        Assert.AreEqual(1, NormalMethod());
    }
}

```

для каждого

`foreach` используется для итерации по элементам массива или элементам внутри коллекции, которая реализует `IEnumerable` `T`.

```

var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}

```

Это приведет к выводу

"Привет, мир!"
 "Как у тебя сегодня дела?"
 "Прощай"

Живая демонстрация на .NET скрипке

Вы можете выйти из цикла `foreach` в любой момент, используя ключевое слово `break` или

перейти к следующей итерации с помощью ключевого слова `continue`.

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.Write(number + ", ");
}

// Prints: 1, 3, 4,
```

Живая демонстрация на .NET скрипке

Обратите внимание, что порядок итераций гарантируется *только* для определенных коллекций, таких как массивы и `List`, но **не** гарантируется для многих других коллекций.

† Хотя `IEnumerable` обычно используется для указания перечислимых коллекций, `foreach` требует, чтобы сбор публично публиковал `object GetEnumerator()`, который должен возвращать объект, который предоставляет метод `bool MoveNext()` и `object Current { get; }`.

Титулы

`params` позволяет параметру метода принимать переменное количество аргументов, т.е. ноль, один или несколько аргументов разрешены для этого параметра.

```
static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

Этот метод теперь можно вызывать с помощью типичного списка аргументов `int` или массива `int`.

```
AddAll(5, 10, 15, 20);           // 50
AddAll(new int[] { 5, 10, 15, 20 }); // 50
```

`params` должны появляться не более одного раза, и если они используются, они должны

быть **последними** в списке аргументов, даже если следующий тип отличается от следующего.

Будьте осторожны при перегрузке функций при использовании ключевого слова `params`. С # предпочитает сопоставлять более конкретные перегрузки, прежде чем прибегать к использованию перегрузок с `params`. Например, если у вас есть два метода:

```
static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}
```

Тогда перед перегрузкой `params` приоритет будет иметь приоритет 2 аргументов.

```
Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)
```

перерыв

В цикле (foreach, do, while, while) оператор `break` прерывает выполнение самого внутреннего цикла и возвращает его после кода. Также он может использоваться с `yield` в котором он указывает, что итератор подходит к концу.

```
for (var i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break;
    }
    Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}
```

Живая демонстрация на .NET скрипке

```
foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
```

```
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}
```

Оператор `break` также используется в конструкциях `case-case` для выхода из сегмента или сегмента по умолчанию.

```
switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

В операторах `switch` ключевое слово `break` используется в конце каждого оператора `case`. Это противоречит некоторым языкам, которые позволяют «проваиваться» к следующему описанию случая в серии. Обходные пути для этого включают в себя операторы «`goto`» или стекирование операторов «`case`» последовательно.

Следующий код даст числа `0, 1, 2, ..., 9` и последняя строка не будет выполнена. `yield break` означает конец функции (а не только цикл).

```
public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}
```

Живая демонстрация на .NET скрипке

Обратите внимание, что в отличие от некоторых других языков, нет способа маркировать определенный разрыв в C#. Это означает, что в случае вложенных циклов будет остановлен только самый внутренний цикл:

```
foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
    {
        if (innerItem.ShouldBreakForWhateverReason)
            // This will only break out of the inner loop, the outer will continue:
            break;
}
```

```
    }  
}
```

Если вы хотите выйти из *внешнего* цикла здесь, вы можете использовать одну из нескольких различных стратегий, таких как:

- Оператор **goto** выпрыгивает из всей структуры цикла.
- Специфическая переменная флага (`shouldBreak` в следующем примере), которая может быть проверена в конце каждой итерации внешнего цикла.
- Рефакторинг кода для использования оператора `return` в самом внутреннем цикле цикла или вообще полностью исключить всю структуру вложенных циклов.

```
bool shouldBreak = false;  
while(comeCondition)  
{  
    while(otherCondition)  
    {  
        if (conditionToBreak)  
        {  
            // Either tranfer control flow to the label below...  
            goto endAllLooping;  
  
            // OR use a flag, which can be checked in the outer loop:  
            shouldBreak = true;  
        }  
    }  
  
    if(shouldBreakNow)  
    {  
        break; // Break out of outer loop if flag was set to true  
    }  
}  
  
endAllLooping: // label from where control flow will continue
```

Аннотация

Класс, отмеченные ключевыми словами `abstract` не может быть создан.

Класс должен быть помечен как абстрактный, если он содержит абстрактные члены или если он наследует абстрактные члены, которых он не реализует. Класс может быть отмечен как абстрактный, даже если не задействованы абстрактные члены.

Абстрактные классы обычно используются в качестве базовых классов, когда некоторая часть реализации должна быть указана другим компонентом.

```
abstract class Animal  
{  
    string Name { get; set; }  
    public abstract void MakeSound();  
}  
  
public class Cat : Animal
```

```

{
    public override void MakeSound()
    {
        Console.WriteLine("Meov meov");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat();           // Allowed due to Cat deriving from Animal
cat.MakeSound();                 // will print out "Meov meov"

Animal dog = new Dog();          // Allowed due to Dog deriving from Animal
dog.MakeSound();                 // will print out "Bark bark"

Animal animal = new Animal();    // Not allowed due to being an abstract class

```

Способ, свойство или событие , отмеченные ключевыми словами `abstract` указывает на то, что реализация для этого элемента , как ожидается , должны быть предоставлены в подклассе. Как упоминалось выше, абстрактные члены могут появляться только в абстрактных классах.

```

abstract class Animal
{
    public abstract string Name { get; set; }
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}

```

float, double, decimal

ПОПЛАВОК

`float` - это псевдоним для типа данных .NET `System.Single` . Он позволяет хранить данные с плавающей точкой с одинарной точностью IEEE 754. Этот тип данных присутствует в `mscorlib.dll` который неявно ссылается на каждый проект С # при их создании.

Ориентировочный диапазон: от $-3,4 \times 10^{38}$ до $3,4 \times 10^{38}$

Десятичная точность: 6-9 значащих цифр

Обозначение :

```
float f = 0.1259;  
var f1 = 0.7895f; // f is literal suffix to represent float values
```

Следует отметить, что тип `float` часто приводит к значительным ошибкам округления. В приложениях, где важна точность, следует учитывать другие типы данных.

двойной

`double` - это псевдоним для типа данных `System.Double`. Он представляет собой 64-битное число с плавающей запятой двойной точности. Этот тип данных присутствует в `mscorlib.dll`, который неявно упоминается в любом проекте C #.

Диапазон: $\pm 5,0 \times 10^{-324}$ $\pm 1,7 \times 10^{308}$

Десятичная точность: 15-16 значащих цифр

Обозначение :

```
double distance = 200.34; // a double value  
double salary = 245; // an integer implicitly type-casted to double value  
var marks = 123.764D; // D is literal suffix to represent double values
```

десятичный

`decimal` - это псевдоним для типа данных .NET. `System.Decimal`. Он представляет ключевое слово, указывающее 128-битный тип данных. По сравнению с типами с плавающей запятой десятичный тип имеет более высокую точность и меньший диапазон, что делает его подходящим для финансовых и денежных расчетов. Этот тип данных присутствует в `mscorlib.dll`, который неявно упоминается в любом проекте C #.

Диапазон: $-7,9 \times 10^{-28}$ до $7,9 \times 10^{28}$

Десятичная точность: 28-29 значащих цифр

Обозначение :

```
decimal payable = 152.25m; // a decimal value  
var marks = 754.24m; // m is literal suffix to represent decimal values
```

UINT

Целое число без знака, или **uint**, представляет собой числовой тип данных, который может содержать только целые положительные числа. Как и в случае с названием, он представляет собой 32-битное целое число без знака. Ключевое слово **uint** является псевдонимом для типа `Common Type System.UInt32`. Этот тип данных присутствует в `mscorlib.dll`, который неявно ссылается на каждый проект C# при их создании. Он занимает четыре байта памяти.

Целочисленные значения без знака могут содержать любое значение от 0 до 4 294 967 295.

Примеры того, как и теперь не объявлять целые числа без знака

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

Обратите внимание: согласно [Microsoft](#) рекомендуется использовать тип данных **int**, где это возможно, поскольку тип данных **uint** не соответствует CLS.

ЭТОТ

`this` Ключевое слово относится к текущему экземпляру класса (объект). Таким образом, можно различать две переменные с одним и тем же именем, один на уровне класса (поле) и один, являющийся параметром (или локальной переменной) метода.

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

Другими способами использования ключевого слова являются [цепочки нестатических перегрузок конструктора](#):

```
public MyClass(int arg) : this(arg, null)
{
}
```

и написание [индексаторов](#):

```
public string this[int idx1, string idx2]
```

```
{  
    get { /* ... */ }  
    set { /* ... */ }  
}
```

и объявление методов расширения :

```
public static int Count<TItem>(this IEnumerable<TItem> source)  
{  
    // ...  
}
```

Если нет конфликта с локальной переменной или параметром, то вопрос в том, следует ли использовать `this` ИЛИ нет, так это `this.MemberOfType` И `MemberOfType` были бы эквивалентны в этом случае. Также см. Ключевое слово `base`.

Обратите внимание, что если в текущем экземпляре должен быть вызван метод расширения, `this` необходимо. Например, если ваш внутри не-статический метод класса, который реализует `IEnumerable<>`, и вы хотите, чтобы вызвать расширение `Count` из ранее, вы должны использовать:

```
this.Count() // works like StaticClassForExtensionMethod.Count(this)
```

и `this` не может быть опущено там.

за

Синтаксис: `for (initializer; condition; iterator)`

- Цикл `for` обычно используется, когда число итераций известно.
- Операторы в секции `initializer` выполняются только один раз, прежде чем вводить цикл.
- Раздел « `condition` » содержит логическое выражение, которое оценивается в конце каждой итерации цикла, чтобы определить, должен ли цикл выйти или должен работать снова.
- Секция `iterator` определяет, что происходит после каждой итерации тела цикла.

Этот пример показывает, как `for` можно использовать для перебора символов строки:

```
string str = "Hello";  
for (int i = 0; i < str.Length; i++)  
{  
    Console.WriteLine(str[i]);  
}
```

Выход:

ЧАС

e
L
L
o

Живая демонстрация на .NET скрипке

Все выражения, определяющие оператор `for` являются необязательными; например, для создания бесконечного цикла используется следующий оператор:

```
for( ; ; )
{
    // Your code here
}
```

Секция `initializer` может содержать несколько переменных, если они одного типа. Раздел `condition` может состоять из любого выражения, которое может быть оценено в `bool`. И секция `iterator` может выполнять несколько действий, разделенных запятой:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Выход:

Привет
hello1
hello12

Живая демонстрация на .NET скрипке

В то время как

Оператор `while` выполняет итерацию над блоком кода до тех пор, пока условный запрос не будет равен `false` или код будет прерван `goto`, `return`, `break` или `throw`.

Синтаксис для ключевого слова `while`:

```
while ( condition ) { code block; }
```

Пример:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Выход:

«Пока находится цикл №1».
«Пока находится петля номер 2.»
«Пока находится петля номер 3.»
«Пока находится петля номер 4.»
«Пока находится петля номер 5.»

[Живая демонстрация на .NET скрипке](#)

Цикл `while` - **Entry Controlled**, поскольку условие проверяется **перед** выполнением заключенного кодового блока. Это означает, что цикл `while` не будет выполнять свои инструкции, если условие ложно.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Давать `while` состояние без инициализации, чтобы он стал ложными в каком - то момент приведет к бесконечному или бесконечному циклу. Насколько это возможно, этого следует избегать, однако, если вам это нужно, могут возникнуть некоторые исключительные обстоятельства.

Вы можете создать такой цикл следующим образом:

```
while (true)
{
// ...
}
```

Обратите внимание, что компилятор C # будет преобразовывать циклы, такие как

```
while (true)
{
// ...
}
```

или же

```
for(;;)
{
// ...
}
```

в

```
{
:label
// ...
}
```

```
    goto label;  
}
```

Обратите внимание, что цикл `while` может иметь любое условие, независимо от того, насколько он сложный, если он оценивает (или возвращает) логическое значение (`bool`). Он также может содержать функцию, которая возвращает логическое значение (поскольку такая функция оценивается с тем же типом, что и выражение, такое как `'a == x'`).

Например,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))  
{  
    myFarm.PickCorn();  
}
```

вернуть

MSDN: оператор `return` завершает выполнение метода, в котором он появляется, и возвращает управление вызывающему методу. Он также может вернуть необязательное значение. Если метод является типом `void`, оператор `return` может быть опущен.

```
public int Sum(int valueA, int valueB)  
{  
    return valueA + valueB;  
}  
  
public void Terminate(bool terminateEarly)  
{  
    if (terminateEarly) return; // method returns to caller if true was passed in  
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false  
}
```

В

Ключевое слово `in` умолчанию имеет три варианта использования:

а) Как часть синтаксиса в инструкции `foreach` или как часть синтаксиса в запросе LINQ

```
foreach (var member in sequence)  
{  
    // ...  
}
```

б) В контексте общих интерфейсов и общих типов делегатов означает *контравариантность* для рассматриваемого параметра типа:

```
public interface IComparer<in T>  
{
```

```
// ...  
}
```

с) В контексте запроса LINQ относится к коллекции, которая запрашивается

```
var query = from x in source select new { x.Name, x.ID, };
```

С ПОМОЩЬЮ

Существует два типа `using` ключевых слов, `using` statement и `using` directive:

1. используя инструкцию :

Ключевое слово `using` гарантирует, что объекты, реализующие интерфейс `IDisposable` будут правильно удалены после использования. Существует отдельная тема для оператора `using`

2. с использованием директивы

Директива `using` имеет три варианта использования, см. [Страницу msdn для директивы use](#). Существует отдельная тема для [директивы use](#).

запечатанный

При применении к классу `sealed` модификатор предотвращает наследование других классов.

```
class A {}  
sealed class B : A {}  
class C : B {} //error : Cannot derive from the sealed class
```

При применении к `virtual` методу (или виртуальному свойству) `sealed` модификатор предотвращает *переопределение* этого метода (свойства) в производных классах.

```
public class A  
{  
    public sealed override string ToString() // Virtual method inherited from class Object  
    {  
        return "Do not override me!";  
    }  
}  
  
public class B: A  
{  
    public override string ToString() // Compile time error  
    {  
        return "An attempt to override";  
    }  
}
```

размер

Используется для получения размера в байтах для неуправляемого типа

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2 (Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

статический

`static` Модификатор используется для объявления статического члена, который не нужно создавать, чтобы получить доступ, но вместо этого открывается просто через его имя, то есть `DateTime.Now`.

`static` может использоваться с классами, полями, методами, свойствами, операторами, событиями и конструкторами.

Хотя экземпляр класса содержит отдельную копию всех полей экземпляра класса, существует только одна копия каждого статического поля.

```
class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        A b = new A();
        A c = new A();

        Console.WriteLine(A.count); // 3
    }
}
```

`count` равно общему числу экземпляров класса `A`

Статический модификатор также может использоваться для объявления статического конструктора для класса, для инициализации статических данных или для запуска кода, который нужно только один раз вызывать. Статические конструкторы вызываются до того, как класс ссылается в первый раз.

```
class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

static class Помечен static ключевым словом и может использоваться в качестве полезного контейнера для набора методов, которые работают с параметрами, но необязательно требуют привязки к экземпляру. Из-за static природы класса он не может быть создан, но может содержать static constructor. Некоторые функции static class включают:

- Не может быть унаследован
- Не может наследовать ничего, кроме Object
- Может содержать статический конструктор, но не конструктор экземпляра
- Может содержать только статические элементы
- Запечатан

Компилятор также дружелюбен и позволит разработчику узнать, существуют ли какие-либо члены экземпляра в классе. Примером может служить статический класс, который преобразует между метриками США и Канады:

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

Когда классы объявляются static:

```
public static class Functions
{
    public static int Double(int value)
    {
        return value + value;
    }
}
```

все функции, свойства или члены внутри класса также должны быть объявлены статическими. Ни один экземпляр класса не может быть создан. По существу статический класс позволяет создавать пулы функций, которые логически сгруппированы.

Поскольку C # 6 static также можно использовать вместе using импортом статических элементов и методов. Они могут использоваться без имени класса.

Старый способ, без using static :

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
        Console.WriteLine("Hello World!"); //Writeline is method belonging to static class
    }
}
```

Пример using static

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
        WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }
}
```

Недостатки

Хотя статические классы могут быть невероятно полезными, у них есть свои собственные оговорки:

- Как только статический класс был вызван, класс загружается в память и не может быть запущен через сборщик мусора до тех пор, пока AppDomain не разместит статический класс.
- Статический класс не может реализовать интерфейс.

ИНТ

int - это псевдоним для System.Int32 , который является типом данных для подписанных 32-битных целых чисел. Этот тип данных можно найти в mscorelib.dll который неявно ссылается на каждый проект C # при их создании.

Диапазон: -2,147,483,648 до 2,147,483,647

```
int int1 = -10007;
var int2 = 2132012521;
```

ДОЛГО

Ключевое слово **long** используется для представления подписаных 64-битных целых чисел. Это псевдоним для типа `System.Int64` присутствующего в `mscorlib.dll`, который неявно ссылается на каждый проект C# при их создании.

Любая длинная переменная может быть объявлена как явно, так и неявно:

```
long long1 = 9223372036854775806; // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

Длинная переменная может содержать любое значение от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807 и может быть полезна в ситуациях, когда переменная должна содержать значение, превышающее границы других переменных (таких как переменная `int`).

ULONG

Ключевое слово, используемое для 64-битных целых чисел без знака. Он представляет тип данных `System.UInt64` найденный в `mscorlib.dll` который неявно ссылается на каждый проект C# при их создании.

Диапазон: от 0 до 18 446 744 073 709 551 615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 15446744063609451315UL;
```

динамический

`dynamic` Ключевое слово используется с [динамически типизированными объектами](#). Объекты, объявленные как `dynamic` статические проверки перед компиляцией, и затем оцениваются во время выполнения.

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456
```

```
Console.WriteLine(info.DoesNotExist);
// Throws RuntimeBinderException
```

Следующий пример использует `dynamic` с библиотекой `Newtonsoft.Json.NET`, чтобы легко считывать данные из десериализованного файла JSON.

```
try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic serializedJson = JsonConvert.DeserializeObject(json);
    int x = serializedJson.x;
    string y = serializedJson.y;
    // int z = serializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}
```

Существуют некоторые ограничения, связанные с ключевым словом `dynamic`. Одним из них является использование методов расширения. В следующем примере добавлен метод расширения для строки: `SayHello`.

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

Первый подход будет состоять в том, чтобы назвать его как обычно (как для строки):

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

Ошибка компиляции, но во время выполнения вы получаете `RuntimeBinderException`. Обходным путем для этого будет вызов метода расширения через статический класс:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

виртуальный, переопределить, новый

виртуальный и переопределить

Ключевое слово `virtual` позволяет переопределять метод, свойство, индексор или событие с помощью производных классов и текущего полиморфного поведения. (Члены по

умолчанию не являются виртуальными в C #)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

Для переопределения члена ключевое слово `override` используется в производных классах.
(Обратите внимание, что подпись членов должна быть одинаковой)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

Полиморфное поведение виртуальных членов означает, что при вызове фактический исполняемый элемент определяется во время выполнения, а не во время компиляции. Переопределяющим элементом в самом производном классе является конкретный объект, экземпляр которого будет выполнен.

Короче говоря, объект может быть объявлен типа `BaseClass` во время компиляции, но если во время выполнения это экземпляр `DerivedClass` тогда будет выполнен переопределенный элемент:

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

Переопределение метода необязательно:

```
public class SecondDerivedClass: DerivedClass { }

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

НОВЫЙ

Поскольку только члены, определенные как `virtual` являются переопределяемыми и полиморфными, производный класс, переопределяющий не виртуального участника, может привести к неожиданным результатам.

```

public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!

```

Когда это происходит, выполняемый элемент всегда определяется во время компиляции в зависимости от типа объекта.

- Если объект объявлен типа `BaseClass` (даже если во время выполнения имеет производный класс), то выполняется метод `BaseClass`
- Если объект объявлен типа `DerivedClass` тогда выполняется метод `DerivedClass`.

Обычно это случайность (когда член добавляется к базовому типу после того, как идентичный был добавлен к производному типу), и в этих сценариях генерируется предупреждение **CS0108** компилятора.

Если это было намеренно, то `new` ключевое слово используется для подавления предупреждения компилятора (и сообщите другим разработчикам о ваших намерениях!). поведение остается неизменным, `new` ключевое слово просто подавляет предупреждение компилятора.

```

public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();

```

```
obj1.Foo(); //Outputs "Foo from BaseClass"  
  
obj1 = new DerivedClass();  
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

Использование переопределения *не является необязательным*

В отличие от C++, использование ключевого слова `override` *не является необязательным*:

```
public class A  
{  
    public virtual void Foo()  
    {  
    }  
}  
  
public class B : A  
{  
    public void Foo() // Generates CS0108  
    {  
    }  
}
```

Приведенный выше пример также вызывает предупреждение **CS0108**, потому что `B.Foo()` не автоматически переопределяет `A.Foo()`. Добавить `override` когда намерение переопределить базовый класс и вызвать полиморфное поведение, добавить `new` когда вы хотите неполиморфное поведение и разрешить вызов с использованием статического типа. Последнее следует использовать с осторожностью, так как это может вызвать серьезную путаницу.

Следующий код даже приводит к ошибке:

```
public class A  
{  
    public void Foo()  
    {  
    }  
}  
  
public class B : A  
{  
    public override void Foo() // Error: Nothing to override  
    {  
    }  
}
```

Производные классы могут вводить

ПОЛИМОРФИЗМ

Следующий код является абсолютно допустимым (хотя и редким):

```
public class A
{
    public void Foo()
    {
        Console.WriteLine("A");
    }
}

public class B : A
{
    public new virtual void Foo()
    {
        Console.WriteLine("B");
    }
}
```

Теперь все объекты со статической ссылкой B (и ее производных) используют полиморфизм для разрешения `Foo()`, а ссылки A используют `A.Foo()`.

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

Виртуальные методы не могут быть частными

Компилятор C# строг в предотвращении бессмысленных конструкций. Методы, помеченные как `virtual` не могут быть частными. Поскольку частный метод не может быть замечен из производного типа, он не может быть перезаписан. Это не скомпилируется:

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

асинхронный, ждут

Ключевое слово `await` было добавлено как часть выпуска C# 5.0, которое поддерживается

в Visual Studio 2012 и далее. Он использует параллельную библиотеку задач (TPL), которая упрощает многопоточность. `async` И `await` Ключевые слова используются в паре в той же функции, что показано ниже. Ключевое слово `await` используется для приостановки выполнения текущего асинхронного метода до тех пор, пока ожидаемая асинхронная задача не будет завершена и / или ее результаты не будут возвращены. Чтобы использовать ключевое слово `await`, метод, который его использует, должен быть помечен как **ключевое слово** `async`.

Использование `async` С `void` сильно обескуражено. Для получения дополнительной информации вы можете посмотреть [здесь](#).

Пример:

```
public async Task DoSomethingAsync()
{
    Console.WriteLine("Starting a useless process...");
    Stopwatch stopwatch = Stopwatch.StartNew();
    int delay = await UselessProcessAsync(1000);
    stopwatch.Stop();
    Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}
```

Выход:

«Начать бесполезный процесс ...»

** ... 1 секунда задержка ... **

«Бесполезный процесс занял 1000 миллисекунд».

`async` Ключевых слов `async` И `await` могут быть опущены, если метод возврата `Task` ИЛИ `Task<T>` возвращает только одну асинхронную операцию.

Вместо этого:

```
public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

Это предпочтительнее:

```
public Task PrintAndDelayAsync(string message, int delay)
```

```
{  
    Debug.WriteLine(message);  
    return Task.Delay(x);  
}
```

5.0

В С # 5.0 `await` не может быть использовано в `catch` и `finally`.

6,0

С С # 6.0 `await` может быть использованы в `catch` и в `finally`.

Голец

Символ - это одна буква, хранящаяся внутри переменной. Это встроенный тип значения, который занимает два байта памяти. Он представляет собой тип данных `System.Char` найденный в `mscorlib.dll` который неявно ссылается на каждый проект С # при их создании.

Существует несколько способов сделать это.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99; //Integral`

`ushort, int, uint, long, ulong, float, double, может быть неявно преобразован в ushort, int, uint, long, ulong, float, double, ИЛИ decimal` и он вернет целочисленное значение этого символа.

```
ushort u = c;
```

возвращает 99 и т.д.

Однако нет никаких неявных преобразований из других типов в `char`. Вместо этого вы должны бросить их.

```
ushort u = 99;  
char c = (char)u;
```

замок

`lock` обеспечивает безопасность потока для блока кода, так что к нему можно получить доступ только одним потоком в рамках одного и того же процесса. Пример:

```
private static object _lockObj = new object();  
static void Main(string[] args)  
{
```

```

    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock (_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}

Output:
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving

```

Случаи применения:

Всякий раз, когда у вас есть блок кода, который может создавать побочные эффекты, если он выполняется несколькими потоками одновременно. Ключевое слово блокировки вместе с **общим объектом синхронизации** (`_objLock` в примере) можно использовать для предотвращения этого.

Обратите внимание, что `_objLock` не может быть `null` а несколько потоков, выполняющих код, должны использовать один и тот же экземпляр объекта (либо путем `static` поля, либо с помощью одного и того же экземпляра класса для обоих потоков)

Со стороны компилятора ключевое слово блокировки является синтаксическим сахаром, который заменяется на `Monitor.Enter(_lockObj);` и `Monitor.Exit(_lockObj);`, Поэтому, если вы замените блокировку, окружив блок кода этими двумя методами, вы получите те же результаты. Вы можете увидеть фактический код в [синтаксическом сахаре в примере C# - lock](#)

Ноль

Переменная ссылочного типа может содержать либо действительную ссылку на

экземпляр, либо нулевую ссылку. Нулевой ссылкой является значение по умолчанию для ссылочных типов переменных, а также типы значений с нулевым значением.

`null` - это ключевое слово, которое представляет собой нулевую ссылку.

В качестве выражения его можно использовать для назначения нулевой ссылки на переменные вышеупомянутых типов:

```
object a = null;
string b = null;
int? c = null;
List<int> d = null;
```

Недействительным типам значений не может быть присвоена нулевая ссылка. Все следующие присвоения недействительны:

```
int a = null;
float b = null;
decimal c = null;
```

Нулевую ссылку *не* следует путать с действительными экземплярами различных типов, таких как:

- пустой список (`new List<int>()`)
- пустая строка (`""`)
- число 0 (`0, 0f, 0m`)
- нулевой символ (`'\0'`)

Иногда имеет смысл проверить, является ли что-либо нулевым или пустым / стандартным объектом. Для проверки этого может использоваться метод `System.String.IsNullOrEmpty(String)`, или вы можете реализовать свой собственный эквивалентный метод.

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null
        //reference. Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

внутренний

`internal` ключевое слово - это модификатор доступа для типов и типов. Внутренние типы

или элементы **доступны только в файлах одной и той же сборки**

использование:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

Разница между различными модификаторами доступа уточняется [здесь](#).

Модификаторы доступа

общественности

Доступ к типу или члену может получить любой другой код в той же сборке или другой сборке, которая ссылается на него.

частный

Доступ к типу или члену может получить только код в том же классе или структуре.

защищенный

Доступ к типу или члену может получить только код в том же классе или структуре или в производном классе.

внутренний

Доступ к типу или члену возможен с помощью любого кода в той же сборке, но не с другой сборки.

защищенный внутренний

Доступ к типу или члену может получить любой код в той же сборке или любой производный класс в другой сборке.

Когда **не установлен модификатор доступа**, используется модификатор доступа по умолчанию. Таким образом, всегда есть модификатор доступа, даже если он не установлен.

где

where может служить две цели в C #: ограничение типа в общем аргументе и фильтрация запросов LINQ.

В обобщенном классе давайте рассмотрим

```
public class Cup<T>
{
    // ...
}
```

Т называется параметром типа. Определение класса может налагать ограничения на фактические типы, которые могут быть предоставлены для Т.

Могут применяться следующие виды ограничений:

- тип ценности
- ссылочный тип
- конструктор по умолчанию
- наследование и реализация

тип ценности

В этом случае может быть предоставлена только `struct` **S** (сюда входят «примитивные» типы данных, такие как `int`, `boolean` т. д.)

```
public class Cup<T> where T : struct
{
    // ...
}
```

ссылочный тип

В этом случае могут быть предоставлены только типы классов

```
public class Cup<T> where T : class
{
    // ...
}
```

гибридное значение / ссылочный тип

Иногда требуется ограничить аргументы типа доступными в базе данных, и они обычно будут отображаться для типов значений и строк. Поскольку все ограничения типов должны быть выполнены, невозможно указать, `where T : struct or string` (это недопустимый синтаксис). Обходной путь заключается в ограничении аргументов типа `IConvertible` который имеет встроенные типы «... Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, Char и String. » Возможно, другие объекты будут реализовывать `IConvertible`, хотя на практике это редко.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

конструктор по умолчанию

Разрешены только типы, содержащие конструктор по умолчанию. Это включает типы значений и классы, которые содержат конструктор по умолчанию (без параметров)

```
public class Cup<T> where T : new
{
    // ...
}
```

наследование и реализация

Могут поставляться только типы, которые наследуются от определенного базового класса или реализуют определенный интерфейс.

```
public class Cup<T> where T : Beverage
{
    // ...
}

public class Cup<T> where T : IBeer
{
    // ...
}
```

Ограничение может даже ссылаться на другой параметр типа:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

Для аргумента типа можно указать несколько ограничений:

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

В предыдущих примерах показаны общие ограничения для определения класса, но ограничения могут использоваться везде, где предоставляется аргумент типа: классы, структуры, интерфейсы, методы и т. д.

where также может быть предложение LINQ. В этом случае он аналогичен WHERE в SQL:

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };
```

```
var query =
    from num in nums
    where num < 5
    select num;

foreach (var n in query)
{
    Console.WriteLine(n + " ");
}
// prints 2 1 3 2 0
```

внешний

Ключевое слово `extern` используется для объявления методов, которые реализуются извне. Это можно использовать вместе с атрибутом `DllImport` для вызова неуправляемого кода с использованием службы `Interop`, который в этом случае будет иметь `static` модификатор

Например:

```
using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);

    public void ActivateProcessWindow(Process p)
    {
        SetForegroundWindow(p.MainWindowHandle);
    }
}
```

Это использует метод `SetForegroundWindow`, импортированный из библиотеки `User32.dll`

Это также можно использовать для определения внешнего ассемблера сборки, которые позволяют нам ссылаться на разные версии одних и тех же компонентов из одной сборки.

Чтобы ссылаться на две сборки с одинаковыми именами с полным типом, в командной строке должен быть указан псевдоним следующим образом:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

Это создает внешние псевдонимы `GridV1` и `GridV2`. Чтобы использовать эти псевдонимы из программы, обратитесь к ним с помощью ключевого слова `extern`. Например:

```
extern alias GridV1;
extern alias GridV2;
```

BOOL

Ключевое слово для хранения логических значений `true` и `false`. `bool` - это псевдоним `System.Boolean`.

Значение по умолчанию `bool` равно `false`.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

Для того чтобы `bool` допускал нулевые значения, он должен быть инициализирован как `bool?`.

Значение по умолчанию для `bool?` нулевой.

```
bool? a // default value is null
```

когда

`when` Ключевое слово добавлено в **C # 6**, оно используется для фильтрации исключений.

До появления ключевого слова `when` вас могло быть одно предложение `catch` для каждого типа исключения; с добавлением ключевого слова теперь возможен более мелкозернистый контроль.

А, `when` выражение присоединяется к ветви `catch`, и только если условие `when` равно `true`, будет выполняться предложение `catch`. Можно иметь несколько предложений `catch` с одинаковыми типами классов исключений и разными, `when` условия.

```
private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}

private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }

CatchException(Method1);
```

```
CatchException(Method2);
```

непроверенный

`unchecked` ключевое слово запрещает компилятору проверять наличие переполнений / недочетов.

Например:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Без ключевого слова `unchecked` ни одна из двух операций добавления не будет скомпилирована.

Когда это полезно?

Это полезно, так как это может ускорить вычисления, которые, безусловно, не будут переполняться, поскольку проверка переполнения требует времени или когда требуется переполнение / недополнение (например, при генерации хэш-кода).

недействительным

Зарезервированное слово `"void"` является псевдонимом типа `System.Void` и имеет два применения:

1. Объявить метод, который не имеет возвращаемого значения:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

Метод с возвращаемым типом `void` все равно может иметь ключевое слово `return` в своем теле. Это полезно, когда вы хотите выйти из выполнения метода и вернуть поток вызывающему:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;
```

```
// Do some more work if the condition evaluated to false.  
}
```

2. Объявите указатель на неизвестный тип в небезопасном контексте.

В небезопасном контексте тип может быть типом указателя, типом значения или ссылочным типом. Объявление типа указателя обычно является `type* identifier`, где тип является известным типом, то есть `int* myInt`, но также может быть `void* identifier`, где тип неизвестен.

Обратите внимание, что объявление типа указателя `void` не рекомендуется Microsoft.

`if, if ... else, if ... else if`

Оператор `if` используется для управления потоком программы. Оператор `if` определяет, какой оператор запускаться на основе значения Boolean выражения.

Для одного оператора `braces {}` являются необязательными, но рекомендуется.

```
int a = 4;  
if(a % 2 == 0)  
{  
    Console.WriteLine("a contains an even number");  
}  
// output: "a contains an even number"
```

`if` также может иметь предложение `else`, которое будет выполнено в случае, если условие принимает значение `false`:

```
int a = 5;  
if(a % 2 == 0)  
{  
    Console.WriteLine("a contains an even number");  
}  
else  
{  
    Console.WriteLine("a contains an odd number");  
}  
// output: "a contains an odd number"
```

Конструкция `if ... else if` позволяет задать несколько условий:

```
int a = 9;  
if(a % 2 == 0)  
{  
    Console.WriteLine("a contains an even number");  
}  
else if(a % 3 == 0)  
{
```

```
        Console.WriteLine("a contains an odd number that is a multiple of 3");
    }
else
{
    Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"
```

Важно отметить, что если в приведенном выше примере выполняется условие, элемент управления пропускает другие тесты и переходит к концу этой конкретной, если else конструирует. Таким образом, порядок тестов важен, если вы используете if .. else if construct

С # Булевые выражения используют [оценку короткого замыкания](#). Это важно в тех случаях, когда оценка условий может иметь побочные эффекты:

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
    //...
}
```

Нет гарантии, что `someOtherBooleanMethodWithSideEffects` будут работать.

Это также важно в тех случаях, когда более ранние условия гарантируют, что он «безопасен» для оценки более поздних. Например:

```
if (someCollection != null && someCollection.Count > 0) {
    // ..
}
```

Порядок очень важен в этом случае, потому что, если мы отменим порядок:

```
if (someCollection.Count > 0 && someCollection != null) {
```

он выкинет `NullReferenceException` если `someCollection` равен `null`.

делать

Оператор `do` выполняет итерацию по блоку кода до тех пор, пока условный запрос не будет равен `false`. Цикл `do-while` также может быть прерван `goto`, `return`, `break` или `throw`.

Синтаксис ключевого слова `do`:

```
do { code block; } while ( условие );
```

Пример:

```
int i = 0;  
  
do  
{  
    Console.WriteLine("Do is on loop number {0}.", i);  
} while (i++ < 5);
```

Выход:

```
«Do is on loop number 1.»  
«Do is on loop number 2.»  
«Do on on loop number 3.»  
«Do is on loop number 4.»  
«Do is on loop number 5.»
```

В отличие от цикла `while` цикл `do-while` является **Exit Controlled**. Это означает, что цикл `do-while` будет выполнять свои инструкции хотя бы один раз, даже если условие не выполняется в первый раз.

```
bool a = false;  
  
do  
{  
    Console.WriteLine("This will be printed once, even if a is false.");  
} while (a == true);
```

оператор

Большинство **встроенных операторов** (включая операторы преобразования) могут быть перегружены с помощью **ключевого слова** `operator` **вместе с** `public` и `static` **модификаторами**.

Операторы представлены в трех формах: унарные операторы, двоичные операторы и операторы преобразования.

Унарные и двоичные операторы требуют, по крайней мере, одного параметра того же типа, что и содержащий тип, а для некоторых требуется дополнительный оператор сопоставления.

Операторы преобразования должны конвертироваться в закрытый тип или из него.

```
public struct Vector32  
{  
  
    public Vector32(int x, int y)  
    {  
        X = x;  
        Y = y;  
    }
```

```

}

public int X { get; }
public int Y { get; }

public static bool operator ==(Vector32 left, Vector32 right)
=> left.X == right.X && left.Y == right.Y;

public static bool operator !=(Vector32 left, Vector32 right)
=> !(left == right);

public static Vector32 operator +(Vector32 left, Vector32 right)
=> new Vector32(left.X + right.X, left.Y + right.Y);

public static Vector32 operator +(Vector32 left, int right)
=> new Vector32(left.X + right, left.Y + right);

public static Vector32 operator +(int left, Vector32 right)
=> right + left;

public static Vector32 operator -(Vector32 left, Vector32 right)
=> new Vector32(left.X - right.X, left.Y - right.Y);

public static Vector32 operator -(Vector32 left, int right)
=> new Vector32(left.X - right, left.Y - right);

public static Vector32 operator -(int left, Vector32 right)
=> right - left;

public static implicit operator Vector64(Vector32 vector)
=> new Vector64(vector.X, vector.Y);

public override string ToString() => $"{{{X}}, {{Y}}}";

}

public struct Vector64
{

    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{{X}}, {{Y}}}";

}

```

пример

```

var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2); // {102, 103}

```

```
Console.WriteLine(vector1 - vector2); // {-72, -25}
```

структура

Тип `struct` тип значения, который обычно используется для инкапсуляции небольших групп связанных переменных, таких как координаты прямоугольника или характеристики элемента в инвентаре.

[Классы](#) - это ссылочные типы, `structs` - типы значений.

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }

        public void Display(string name)
        {
            WriteLine(name + " : " + ToString());
        }
    }

    class Program
    {
        static void Main()
        {
            var point1 = new Point {X = 10, Y = 20};
            // it's not a reference but value type
            var point2 = point1;
            point2.X = 777;
            point2.Y = 888;
            point1.Display(nameof(point1)); // point1: X = 10, Y = 20
            point2.Display(nameof(point2)); // point2: X = 777, Y = 888

            ReadKey();
        }
    }
}
```

Структуры также могут содержать конструкторы, константы, поля, методы, свойства, индексы, операторы, события и вложенные типы, хотя, если требуется несколько таких членов, вам следует подумать о том, чтобы вместо этого создать класс.

Некоторые [предложения](#) от MS о том, когда использовать `struct` и когда использовать `класс`:

РАССМАТРИВАТЬ

определение структуры вместо класса, если экземпляры этого типа являются малыми и обычно недолговечны или обычно внедряются в другие объекты.

ИЗБЕЖАТЬ

определяя структуру, если тип имеет все следующие характеристики:

- Он логически представляет одно значение, подобное примитивным типам (int, double и т. д.),
- Он имеет размер экземпляра до 16 байт.
- Это неизменно.
- Его не нужно часто вставлять в бокс.

переключатель

Оператор `switch` - это оператор управления, который выбирает раздел переключателя для выполнения из списка кандидатов. Оператор `switch` включает в себя один или несколько разделов коммутатора. Каждая секция переключателя содержит один или несколько `case` меток следуют один или более операторов. Если ни одна метка случая не содержит соответствующего значения, управление передается в раздел по `default`, если таковой имеется. Случайное падение не поддерживается в C #, строго говоря. Однако, если 1 или более ярлыков `case` пусты, выполнение будет следовать за кодом следующего блока `case` который содержит код. Это позволяет группировку из нескольких `case` наклеек с одной и той же реализацией. В следующем примере, если `month` равен 12, код в `case 2` будет выполнен, так как `case` этикеткой 12 1 и 2 сгруппированы. Если `case` блок не пуст, то `break` должен присутствовать до следующего `case` наклейки, в противном случае компилятор будет флаг ошибки.

```
int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
    case 6:
    case 7:
    case 8:
        Console.WriteLine("Summer");
        break;
    case 9:
```

```

    case 10:
    case 11:
        Console.WriteLine("Autumn");
        break;
    default:
        Console.WriteLine("Incorrect month index");
        break;
}

```

`case` может быть помечен только значением, известным во время компиляции (например, `1`, `"str"`, `Enum.A`), поэтому `variable` не является допустимой меткой `case`, но значение `const` или `Enum` (а также любое буквальное значение).

интерфейс

`interface` содержит [подписи](#) методов, свойств и событий. Производные классы определяют члены, поскольку интерфейс содержит только объявление членов.

Интерфейс объявляется с использованием ключевого слова `interface`.

```

interface IProduct
{
    decimal Price { get; }
}

class Product : IProduct
{
    const decimal vat = 0.2M;

    public Product(decimal price)
    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}

```

небезопасный

`unsafe` ключевое слово может использоваться в объявлениях типов или методов или для объявления встроенного блока.

Цель этого ключевого слова - включить использование *небезопасного подмножества C #* для рассматриваемого блока. Небезопасное подмножество включает в себя такие функции, как указатели, распределение стека, массивы типа С и т. Д.

Небезопасный код не поддается проверке, поэтому его использование не рекомендуется. Компиляция небезопасного кода требует передачи коммутатора на компилятор C #. Кроме того, CLR требует, чтобы работающая сборка имела полное доверие.

Несмотря на эти ограничения, небезопасный код имеет действующие правила, позволяющие сделать некоторые операции более эффективными (например, индексирование массива) или проще (например, взаимодействовать с некоторыми неуправляемыми библиотеками).

В качестве очень простого примера

```
// compile with /unsafe
class UnsafeTest
{
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p; // the '*' dereferences the pointer.
        //Since we passed in "the address of i", this becomes "i *= i"
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&):
        SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
        Console.WriteLine(i); // Output: 25
    }
}
```

При работе с указателями мы можем напрямую изменять значения ячеек памяти, а не обращаться к ним по имени. Обратите внимание, что для этого часто требуется использование ключевого слова **fixed**, чтобы предотвратить возможное повреждение памяти, поскольку сборщик мусора перемещает вещи (в противном случае вы можете получить ошибку [CS0212](#)). Поскольку переменную, которая была «фиксированной», не может быть записана, нам также часто приходится иметь второй указатель, который начинается с того же места, что и первый.

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;
```

```

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++; //move the pointer to the next memory space.
            // NOTE that the pointer will move 4 bytes since "p" is an
            // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;" 
        }
    }
}

```

Выход:

```

1
4
9
16
25
36
49
64
81
100

```

`unsafe` также позволяет использовать `stackalloc`, который будет выделять память в стеке, например `_alloca`, в библиотеке времени выполнения С. Мы можем изменить приведенный выше пример, чтобы использовать `stackalloc` следующим образом:

```

unsafe void Main()
{
    const int len=10;
    int* seedArray = stackalloc int[len];

    //We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
    // We have at least 2 options to populate the array. The end result of either
    // option will be the same (doing both will also be the same here).

    //FIRST OPTION:
    int* p = seedArray; // we don't want to lose where the array starts, so we
                        // create a shadow copy of the pointer
    for(int i=1; i<=len; i++)
        *p++ = i;
    //end of first option

    //SECOND OPTION:
    for(int i=0; i<len; i++)
        seedArray[i] = i+1;
    //end of second option

    UnsafeSquareArray(seedArray, len);
    for(int i=0; i< len; i++)
        Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code

```

```
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}
```

(Выход такой же, как и выше)

неявный

`implicit` ключевое слово используется для перегрузки оператора преобразования. Например, вы можете объявить класс `Fraction` который должен быть автоматически преобразован в `double` при необходимости и который может быть автоматически преобразован из `int`:

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

true, false

`true` и `false` ключевые слова имеют два применения:

1. Как буквальные логические значения

```
var myTrueBool = true;
var myFalseBool = false;
```

2. Как операторы, которые могут быть перегружены

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

Перегрузка фальшивого оператора была полезной до С # 2.0 перед введением типов

Nullable .

Тип, который перегружает `true` оператор, также должен перегружать `false` оператор.

строка

`string` - это псевдоним типа данных .NET `System.String`, который позволяет сохранять текст (последовательности символов).

Обозначения:

```
string a = "Hello";
var b = "world";
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Каждый символ в строке кодируется в UTF-16, что означает, что для каждого символа требуется минимум 2 байта пространства для хранения.

USHORT

Числовой тип, используемый для хранения 16-битных положительных целых чисел. `ushort` - это псевдоним для `System.UInt16` и занимает 2 байта памяти.

Допустимый диапазон от 0 до 65535 .

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

SByte

Числовой тип , используемый для хранения 8-битные целые числа. `sbyte` - это псевдоним для `System.SByte` и занимает 1 байт памяти. Для беззнакового эквивалента используйте `byte` .

Допустимый диапазон - от -127 до 127 (остаточный используется для хранения знака).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

var

Неявно типизированная локальная переменная, которая строго типизирована так же, как если бы пользователь объявил тип. В отличие от других объявлений переменных, компилятор определяет тип переменной, которую он представляет, на основе значения, которое ему присвоено.

```
var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).
```

В отличие от других типов переменных, определения переменных с этим ключевым словом должны быть инициализированы при объявлении. Это связано с тем, что ключевое слово **var** представляет собой неявно типизированную переменную.

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

Ключевое слово **var** также можно использовать для создания новых типов данных «на лету». Эти новые типы данных называются *анонимными типами*. Они весьма полезны, так как они позволяют пользователю определять набор свойств без необходимости явно объявлять тип какого-либо типа.

Обычный анонимный тип

```
var a = new { number = 1, text = "hi" };
```

Запрос LINQ, возвращающий анонимный тип

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName { get; set; }
}

public void GetDogsWithBreedNames()
{
    var db = new DogDataContext(ConnectionString);
    var result = from d in db.Dogs
                 join b in db.Breeds on d.BreedId equals b.BreedId
                 select new
                 {
                     DogName = d.Name,
                     BreedName = b.BreedName
                 };
    DoStuff(result);
}
```

Вы можете использовать ключевое слово **var** в инструкции `foreach`

```

public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( (string)item .equals(stringToSearch) )
            return true;
    }

    return false;
}

```

делегат

Делегаты - это типы, которые представляют собой ссылку на метод. Они используются для передачи методов в качестве аргументов другим методам.

Делегаты могут хранить статические методы, методы экземпляра, анонимные методы или лямбда-выражения.

```

class DelegateExample
{
    public void Run()
    {
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0} ", input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}", ret ) );
    }

    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'", input ) );
        return true;
    }
}

```

При назначении метода делегату важно отметить, что метод должен иметь одинаковый тип возвращаемого значения, а также параметры. Это отличается от «нормального» перегрузки метода, где только параметры определяют сигнатуру метода.

События создаются поверх делегатов.

событие

event позволяет разработчику внедрить шаблон уведомления.

Простой пример

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}

public class Client
{
    public void Client(Server server)
    {
        // client subscribes to the server's DataChangeEvent
        server.DataChangeEvent += server_DataChanged;
    }

    private void server_DataChanged(object sender, EventArgs args)
    {
        // notified when the server raises the DataChangeEvent
    }
}
```

Ссылка на MSDN

частичный

partial Ключевое слово может использоваться при определении типа класса, структуры или интерфейса, чтобы разрешить определение типа в несколько файлов. Это полезно для включения новых функций в автоматически сгенерированный код.

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

Примечание. Класс можно разделить на любое количество файлов. Тем не менее, все объявления должны быть в том же пространстве имен и в той же сборке.

Методы также могут быть объявлены частично с использованием `partial` ключевого слова. В этом случае один файл будет содержать только определение метода, а другой файл будет содержать реализацию.

Частичный метод имеет свою подпись, определенную в одной части частичного типа, а ее реализация определена в другой части типа. Частичные методы позволяют разработчикам классов предлагать крючки методов, аналогичные обработчикам событий, которые разработчики могут решить реализовать или нет. Если разработчик не предоставляет реализацию, компилятор удаляет подпись во время компиляции. Для частичных методов применяются следующие условия:

- Подписи в обеих частях частичного типа должны совпадать.
- Метод должен возвращать `void`.
- Модификаторы доступа не допускаются. Частичные методы неявно закрыты.

- MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
    }
}
```

```
        {
            Console.WriteLine(str);
        }
    }
}
```

Примечание . Тип, содержащий частичный метод, также должен быть объявлен частичным.

Прочтайте Ключевые слова онлайн: <https://riptutorial.com/ru/csharp/topic/26/ключевые-слова>

глава 83: Кодовые контракты

Синтаксис

1. Contract.Requires (условие, userMessage)

Contract.Requires (условие, userMessage)

Contract.Result <T>

Contract.Ensures ()

Contract.Invariants ()

замечания

.NET поддерживает идею Design by Contract через свой класс Contracts, найденный в пространстве имен System.Diagnostics и представленный в .NET 4.0. API Code Contracts API включает в себя классы для проверки статичности и времени выполнения кода и позволяет определять предварительные условия, постусловия и инварианты внутри метода. Предварительные условия определяют условия, которые должны выполнять параметры перед тем, как метод может выполнить, постусловия, которые проверяются по завершении метода, а инварианты определяют условия, которые не изменяются во время выполнения метода.

Зачем нужны кодовые контракты?

Отслеживание проблем приложения, когда приложение работает, является одной из главных проблем всех разработчиков и администраторов. Отслеживание может выполняться многими способами. Например -

- Вы можете применить трассировку в нашем приложении и получить информацию о приложении, когда приложение запущено
- Вы можете использовать механизм регистрации событий при запуске приложения. Сообщения можно увидеть с помощью средства просмотра событий
- Вы можете применять мониторинг производительности через определенный промежуток времени и записывать данные в реальном времени из своего приложения.

Кодовые контракты используют другой подход для отслеживания и управления проблемами в приложении. Вместо того, чтобы проверять все, что возвращается из вызова метода, Кодовые контракты с помощью предварительных условий, постусловий и

инвариантов в методах, убедитесь, что все входящие и выходящие из ваших методов правильны.

Examples

Предпосылками

```
namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
        {
            Contract.Requires(payment != null);
            Contract.Requires(!string.IsNullOrEmpty(payment.Name));
            Contract.Requires(payment.Date <= DateTime.Now);
            Contract.Requires(payment.Amount > 0);

            this._payments.Add(payment);
        }
    }
}
```

Постусловия

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

Инварианты

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

    public class Point
```

```

{
    public int X { get; set; }
    public int Y { get; set; }

    public Point()
    {
    }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public void Set(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public void Test(int x, int y)
    {
        for (int dx = -x; dx <= x; dx++) {
            this.X = dx;
            Console.WriteLine("Current X = {0}", this.X);
        }

        for (int dy = -y; dy <= y; dy++) {
            this.Y = dy;
            Console.WriteLine("Current Y = {0}", this.Y);
        }

        Console.WriteLine("X = {0}", this.X);
        Console.WriteLine("Y = {0}", this.Y);
    }

    [ContractInvariantMethod]
    private void ValidateCoordinates()
    {
        Contract.Invariant(this.X >= 0);
        Contract.Invariant(this.Y >= 0);
    }
}
}

```

Определение контрактов по интерфейсу

```

[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {

```

```

[Pure]
get
{
    return Contract.Result<string>();
}
set
{
    Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer
ID cannot be null!!");
}
}

string IValidation.Password
{
    [Pure]
    get
    {
        return Contract.Result<string>();
    }
    set
    {
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password
cannot be null!!");
    }
}
}

class Validation:IValidation
{
    public string GetCustomerPassword(string customerID)
    {
        Contract.Requires(!string.IsNullOrEmpty(customerID), "Customer ID cannot be Null");
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID),
"Exception!!");
        Contract.Ensures(Contract.Result<string>() != null);
        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }
    }
}

private string m_custID, m_PWD;

public string CustomerID
{
    get
    {
        return m_custID;
    }
    set
    {
        m_custID = value;
    }
}

```

```
public string Password
{
    get
    {
        return m_PWD;
    }
    set
    {
        m_PWD = value;
    }
}
```

В приведенном выше коде мы определили интерфейс под названием `IValidation` с атрибутом `[ContractClass]`. Этот атрибут принимает адрес класса, в котором мы реализовали контракт для интерфейса. Класс `ValidationContract` использует свойства, определенные в интерфейсе, и проверяет нулевые значения с использованием `Contract.Requires<T>`. `T` - класс исключения.

Мы также отметили `get accessor` с атрибутом `[Pure]`. Чистый атрибут гарантирует, что метод или свойство не изменяет состояние экземпляра класса, в котором `IValidation` интерфейс `IValidation`.

Прочтайте Кодовые контракты онлайн: <https://riptutorial.com/ru/csharp/topic/4241/кодовые-контракты>

глава 84: Кодовые контракты и утверждения

Examples

Утверждения для проверки логики всегда должны быть верны

Утверждения используются не для тестирования входных параметров, а для проверки того, что поток программы является ключевым, то есть вы можете сделать определенные предположения о своем коде в определенный момент времени. Другими словами: тест, выполненный с помощью `Debug.Assert` должен *всегда* предполагать, что проверенное значение `true`.

`Debug.Assert` выполняется только в сборках DEBUG; он отфильтровывается из сборников RELEASE. Он должен считаться инструментом отладки в дополнение к модульному тестированию, а не заменой кодовых контрактов или методов проверки ввода.

Например, это хорошее утверждение:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Здесь `assert` - хороший выбор, потому что мы можем предположить, что `RetrieveSystemConfiguration ()` вернет действительное значение и никогда не вернет `null`.

Вот еще один хороший пример:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

Во-первых, мы можем предположить, что `RetrieveUserData ()` вернет действительное значение. Затем, прежде чем использовать свойство `Age`, мы проверим предположение (которое всегда должно быть правдой), что возраст пользователя строго положителен.

Это плохой пример утверждения:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Утверждение не для проверки ввода, потому что неверно предполагать, что это

утверждение всегда будет истинным. Для этого вы должны использовать методы проверки ввода. В приведенном выше случае вы также должны убедиться, что входное значение - это номер в первую очередь.

Прочтайте Кодовые контракты и утверждения онлайн:

<https://riptutorial.com/ru/csharp/topic/4349/кодовые-контракты-и-утверждения>

глава 85: Комментарии и регионы

Examples

Комментарии

Использование комментариев в ваших проектах - это удобный способ оставить объяснения ваших вариантов дизайна и постараться облегчить вашу (или чужую) жизнь при сохранении или добавлении кода.

Есть два способа добавить комментарий к вашему коду.

Комментарии к одной строке

Любой текст, помещенный после `//` будет рассматриваться как комментарий.

```
public class Program
{
    // This is the entry point of my program.
    public static void Main()
    {
        // Prints a message to the console. - This is a comment!
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // You can even comment out code.
        System.Console.ReadLine();
    }
}
```

Многострочные или разделительные комментарии

Любой текст между `/*` и `*/` будет рассматриваться как комментарий.

```
public class Program
{
    public static void Main()
    {
        /*
         * This is a multi line comment
         * it will be ignored by the compiler.
        */
        System.Console.WriteLine("Hello, World!");

        // It's also possible to make an inline comment with /* */
    }
}
```

```
// although it's rarely used in practice
System.Console.WriteLine(/* Inline comment */ "Hello, World!");

System.Console.ReadLine();
}

}
```

районы

Область - это разборчивый блок кода, который может помочь в удобочитаемости и организации вашего кода.

ПРИМЕЧАНИЕ. Правило StyleCop SA1124 DoNotUseRegions препятствует использованию регионов. Они обычно являются признаком плохо организованного кода, поскольку C # включает частичные классы и другие функции, которые делают регионы устаревшими.

Вы можете использовать регионы следующим образом:

```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

Когда вышеуказанный код отображается в среде IDE, вы сможете свернуть и развернуть код с помощью символов + и -.

расширенный

```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

рухнул

```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

Комментарии к документации

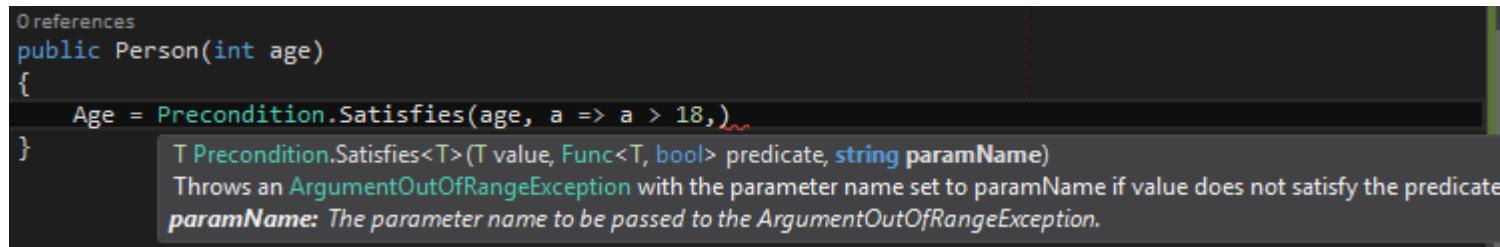
Комментарии к документации XML могут использоваться для предоставления документации API, которая может быть легко обработана инструментами:

```
/// <summary>
/// A helper class for validating method arguments.
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     Throws an <see cref="ArgumentOutOfRangeException"/> with the parameter
    ///     name set to <c>paramName</c> if <c>value</c> does not satisfy the
    ///     <c>predicate</c> specified.
    /// </summary>
    /// <typeparam name="T">
    ///     The type of the argument checked
    /// </typeparam>
    /// <param name="value">
    ///     The argument to be checked
    /// </param>
    /// <param name="predicate">
    ///     The predicate the value is required to satisfy
    /// </param>
    /// <param name="paramName">
    ///     The parameter name to be passed to the
    ///     <see cref="ArgumentOutOfRangeException"/>.
    /// </param>
    /// <returns>The value specified</returns>
    public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
```

```
    {
        if (!predicate(value))
            throw new ArgumentOutOfRangeException(paramName);

        return value;
    }
}
```

Документация мгновенно подхвачена IntelliSense:



0 references

```
public Person(int age)
{
    Age = Precondition.Satisfies(age, a => a > 18, )
```

T Precondition.Satisfies<T>(T value, Func<T, bool> predicate, **string** paramName)
Throws an **ArgumentOutOfRangeException** with the parameter name set to paramName if value does not satisfy the predicate
paramName: The parameter name to be passed to the **ArgumentOutOfRangeException**.

Прочтайте Комментарии и регионы онлайн: <https://riptutorial.com/ru/csharp/topic/5346/комментарии-и-регионы>

глава 86: Комментарии к документации XML

замечания

Несколько раз вам нужно **создать расширенную текстовую документацию** из ваших комментариев xml. К несчастью, для **него нет стандартного способа**.

Но есть несколько отдельных проектов, которые вы можете использовать для этого случая:

- [замок из песка](#)
- [Docu](#)
- [NDoc](#)
- [DocFX](#)

Examples

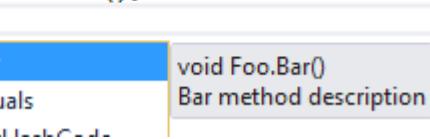
Простая аннотация метода

Комментарии к документации помещаются непосредственно над методом или классом, который они описывают. Они начинаются с трех косых черт `///` и позволяют хранить метаинформацию через XML.

```
/// <summary>
/// Bar method description
/// </summary>
public void Bar()
{
}
```

Информация внутри тегов может использоваться Visual Studio и другими инструментами для предоставления таких услуг, как IntelliSense:

```
private static void Main()
{
    Foo foo = new Foo();
    foo.|
```



См. Также [список общих тегов документации Microsoft](#).

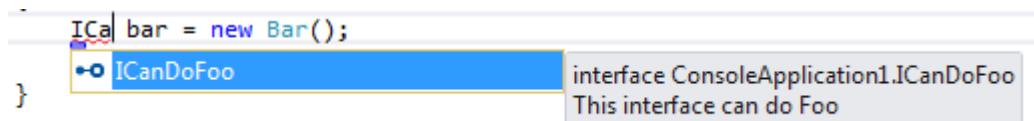
Комментарии к документации по интерфейсу и классу

```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

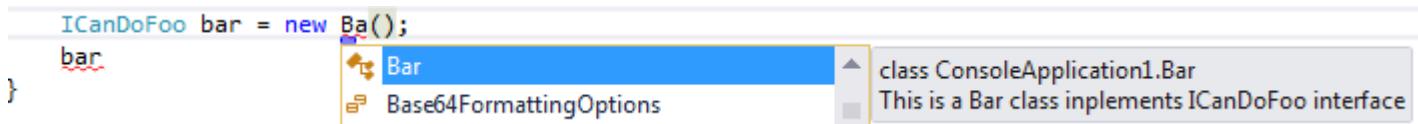
Результат

Сводка интерфейса



ICanDoFoo bar = new Bar();
ICanDoFoo interface ConsoleApplication1.ICanDoFoo
This interface can do Foo

Сводка классов



ICanDoFoo bar = new Bar();
Bar class ConsoleApplication1.Bar
This is a Bar class implements ICanDoFoo interface

Комментарий к документации по методу с элементами param и return

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
    // ...
}
```

IntelliSense показывает вам описание для каждого параметра:

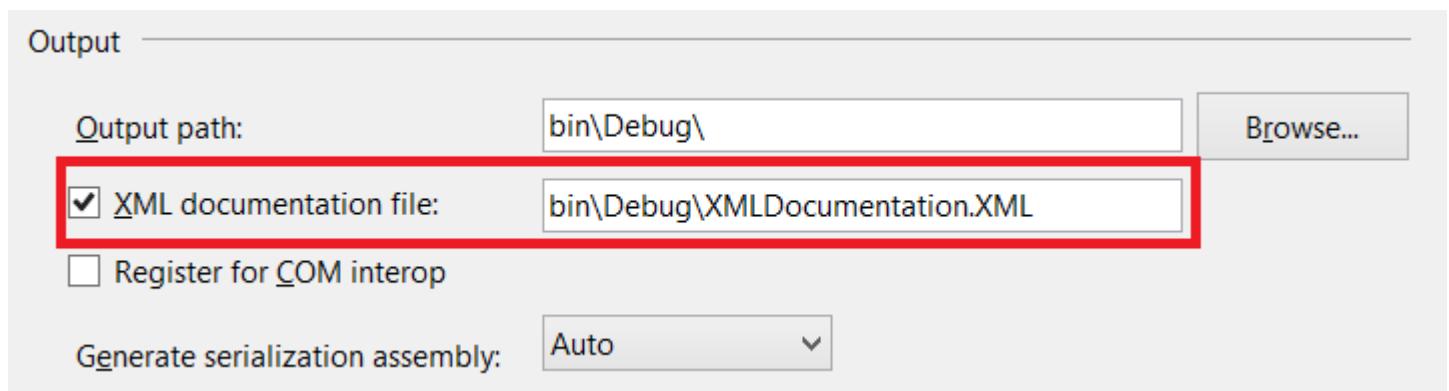
```
obj.GetData(3, DateTime.Now);  
DataClass Foo.GetData(int id, DateTime time)  
This method returning some data  
id: Id parameter
```

Совет. Если Intellisense не отображается в Visual Studio, удалите первую скобку или запятую, а затем введите ее еще раз.

Генерация XML из комментариев к документации

Чтобы создать файл документации XML из комментариев к документации в коде, используйте параметр `/doc` с компилятором `csc.exe` C #.

В Visual Studio 2013/2015, в **проекте -> Свойства -> Создать -> Вывод**, установите флагок **в XML documentation file**:



Когда вы создадите проект, компилятор будет XML-файл с именем, соответствующим имени проекта (например, `XMLDocumentation.dll -> XMLDocumentation.xml`).

Когда вы используете сборку в другом проекте, убедитесь, что файл XML находится в том же каталоге, что и ссылка на DLL.

Этот пример:

```
/// <summary>  
/// Data class description  
/// </summary>  
public class DataClass  
{  
    /// <summary>  
    /// Name property description  
    /// </summary>  
    public string Name { get; set; }  
}  
  
/// <summary>  
/// Foo function  
/// </summary>
```

```

public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}

```

Производит этот xml для сборки:

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
                Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
                Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
                Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32, System.DateTime)">
            <summary>
                This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
        </member>
    </members>
</doc>

```

Ссылка на другой класс в документации

Тег `<see>` можно использовать для связи с другим классом. Он содержит член `cref` который должен содержать имя класса, на который нужно ссылаться. Visual Studio предоставит Intellisense при написании этого тега, и такие ссылки будут обработаны при переименовании ссылочного класса.

```

    /// <summary>

```

```
/// You might also want to check out <see cref="SomeOtherClass"/>.  
/// </summary>  
public class SomeClass  
{  
}
```

В всплывающих окнах Visual Studio Intellisense такие ссылки также будут отображаться по цвету в тексте.

Чтобы ссылаться на общий класс, используйте что-то похожее на следующее:

```
/// <summary>  
/// An enhanced version of <see cref="List{T}"/>.  
/// </summary>  
public class SomeGenericClass<T>  
{  
}
```

Прочтайте [Комментарии к документации XML](#) онлайн:

<https://riptutorial.com/ru/csharp/topic/740/комментарии-к-документации-xml>

глава 87: Компиляция времени выполнения

Examples

RoslynScript

Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript - это новый механизм сценариев C #.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

Вы можете компилировать и запускать любые операторы, переменные, методы, классы или любые сегменты кода.

CSharpCodeProvider

Microsoft.CSharp.CSharpCodeProvider может использоваться для компиляции классов C #.

```
var code = @"
    public class Abc {
        public string Get() { return ""abc""; }
    }
";

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

Прочтайте Компиляция времени выполнения онлайн:

<https://riptutorial.com/ru/csharp/topic/3139/компиляция-времени-выполнения>

глава 88: Конкатенация строк

замечания

Если вы создаете динамическую строку, это хорошая практика , чтобы выбрать `StringBuilder` Класса , а не вступающие строки с помощью + или `Concat` метода, каждый + / `Concat` создает новую строку объект каждый раз она выполняется.

Examples

+ Оператор

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

Объединить строки, используя `System.Text.StringBuilder`

Конкатенация строк с помощью `StringBuilder` может обеспечить преимущества производительности перед простой конкатенацией строк с помощью + . Это связано с тем, как выделяется память. Строки перераспределяются с каждой конкатенацией, `StringBuilders` выделяют память в блоках, только перераспределяя, когда текущий блок исчерпан. Это может иметь огромное значение при выполнении множества небольших конкатенаций.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

Вызовы в `Append()` могут быть подключены последовательно, потому что он возвращает ссылку на `StringBuilder` :

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
    .Append("another string");
```

Элементы массива Concat string с использованием `String.Join`

Метод `String.Join` может использоваться для конкатенации нескольких элементов из массива строк.

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

Производит следующий выход: «оранжевый, виноградный»

В этом примере используется `String.Join(String, String[], Int32, Int32)`, которая указывает начальный индекс и счетчик сверху разделителя и значения.

Если вы не хотите использовать перегрузку `startIndex` и `count`, вы можете присоединиться ко всем указанным строкам. Как это:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

который будет производить;

яблоко, апельсин, виноград, груша

Конкатенация двух строк с использованием \$

\$ обеспечивает простой и сжатый способ конкатенации нескольких строк.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; // "text1 text3"
```

Прочтайте Конкатенация строк онлайн: <https://riptutorial.com/ru/csharp/topic/3616/конкатенация-строк>

глава 89: Конструкторы и финализаторы

Вступление

Конструкторы - это методы в классе, которые вызывается при создании экземпляра этого класса. Их основная ответственность заключается в том, чтобы оставить новый объект в полезном и согласованном состоянии.

Деструкторы / Финализаторы - это методы в классе, которые вызывается, когда экземпляр этого объекта уничтожается. В C # они редко явно пишутся / используются.

замечания

На C # фактически нет деструкторов, а скорее финализаторы, которые используют синтаксис деструктора стиля C ++. Указание деструктора переопределяет метод `Object.Finalize()` который нельзя вызвать напрямую.

В отличие от других языков с похожим синтаксисом, эти методы *не* вызывают, когда объекты выходят из области действия, но вызывается, когда запускается сборщик мусора, который происходит [при определенных условиях](#). Таким образом, они *не* гарантируются в каком-либо конкретном порядке.

Финализаторы должны нести ответственность за очистку неуправляемых ресурсов **только** (указатели , полученные с помощью класса Marshal, полученные через `p / Invoke` (системные вызовы) или сырье указатели , используемые в небезопасных блоках). Чтобы очистить управляемые ресурсы, просмотрите `IDisposable`, шаблон `Dispose` и инструкцию `using` .

(Дальнейшее чтение: [Когда я должен создать деструктор?](#))

Examples

Конструктор по умолчанию

Когда тип определен без конструктора:

```
public class Animal
{
}
```

то компилятор генерирует конструктор по умолчанию, эквивалентный следующему:

```
public class Animal
```

```
{  
    public Animal() {}  
}
```

Определение любого конструктора для типа будет подавлять генерацию конструктора по умолчанию. Если тип был определен следующим образом:

```
public class Animal  
{  
    public Animal(string name) {}  
}
```

То `Animal` может быть создано только путем вызова объявленного конструктора.

```
// This is valid  
var myAnimal = new Animal("Fluffy");  
// This fails to compile  
var unnamedAnimal = new Animal();
```

Во втором примере компилятор отобразит сообщение об ошибке:

«`Animal`» не содержит конструктор, который принимает 0 аргументов

Если вы хотите, чтобы класс имел как конструктор без параметров, так и конструктор, который принимает параметр, вы можете сделать это, явно реализуя оба конструктора.

```
public class Animal  
{  
  
    public Animal() {} //Equivalent to a default constructor.  
    public Animal(string name) {}  
}
```

Компилятор не сможет создать конструктор по умолчанию, если класс расширяет другой класс, который не имеет конструктора без параметров. Например, если бы у нас было классное `Creature`:

```
public class Creature  
{  
    public Creature(Genus genus) {}  
}
```

То `Animal` определяется как `class Animal : Creature {}` не будет компилироваться.

Вызов конструктора из другого конструктора

```
public class Animal  
{  
    public string Name { get; set; }
```

```

public Animal() : this("Dog")
{
}

public Animal(string name)
{
    Name = name;
}

var dog = new Animal();           // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat");    // cat.Name is "Cat", the empty constructor is not called.

```

Статический конструктор

Статический конструктор называется первым, когда инициализируется любой член типа, вызывается статический член класса или статический метод. Статический конструктор является потокобезопасным. Статический конструктор обычно используется для:

- Инициализируйте статическое состояние, то есть состояние, которое используется для разных экземпляров одного и того же класса.
- Создать синглтон

Пример:

```

class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();

```

Выход:

Инициализировано

Животное создано

Животное создано

[Посмотреть демо](#)

Если первый вызов относится к статическому методу, статический конструктор вызывается без конструктора экземпляра. Это нормально, потому что статический метод никак не может получить доступ к состоянию экземпляра.

```
Animal.Yawn();
```

Это приведет к выводу:

Инициализировано
Зевать!

См. Также [Исключения в статических конструкторах](#) и [Generic Static Constructors](#).

Пример Singleton:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

Вызов конструктора базового класса

Вызывается конструктор базового класса перед выполнением конструктора производного класса. Например, если `Mammal` расширяет `Animal`, тогда код, содержащийся в конструкторе `Animal` вызывается первым при создании экземпляра `Mammal`.

Если производный класс явно не указывает, какой конструктор базового класса следует вызывать, компилятор принимает конструктор без параметров.

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

В этом случае создание экземпляра `Mammal` путем вызова `new Mammal("George the Cat")` напечатает

Появляется неизвестное животное.
Джордж Кошка - млекопитающее.

[Посмотреть демо](#)

Вызов другого конструктора базового класса выполняется путем размещения `: base(args)` между сигнатурой конструктора и его телом:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Вызов `new Mammal("George the Cat")` теперь напечатает:

Джордж Кот рождается.
Джордж Кошка - млекопитающее.

[Посмотреть демо](#)

Финализаторы на производных классах

Когда графический объект завершен, порядок является обратной конструкцией. Например, супертип завершается до базового типа, как показывает следующий код:

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
```

```
GC.Collect();  
  
//Derived class finalized!  
//Base class finalized!
```

Шаблон конструктора Singleton

```
public class SingletonClass  
{  
    public static SingletonClass Instance { get; } = new SingletonClass();  
  
    private SingletonClass()  
    {  
        // Put custom constructor code here  
    }  
}
```

Поскольку конструктор является закрытым, никакие новые экземпляры `SingletonClass` могут быть сделаны путем использования кода. Единственный способ доступа к единственному экземпляру `SingletonClass` - использование статического свойства `SingletonClass.Instance`.

`Instance` присваивается статический конструктор, создаваемый компилятором C#. Среда выполнения .NET гарантирует, что статический конструктор запускается не более одного раза и запускается до того, как `Instance` будет сначала прочитан. Поэтому все проблемы синхронизации и инициализации выполняются во время выполнения.

Обратите внимание: если статический конструктор не работает, класс `Singleton` становится непригодным для жизни `AppDomain`.

Кроме того, статический конструктор не гарантированно запускается во время первого доступа `Instance`. Скорее, он будет работать в какой-то момент до этого. Это делает время, при котором инициализация происходит без детерминизма. В практических случаях JIT часто вызывает статический конструктор во время компиляции (а не выполнения) метода, ссылающегося на `Instance`. Это оптимизация производительности.

См. Страницу « [Режимы использования Singleton](#)» для других способов реализации одноэлементного шаблона.

Принудительный вызов статического конструктора

В то время как статические конструкторы всегда вызывают перед первым использованием типа, иногда полезно иметь возможность заставить их быть вызванными, а класс `RuntimeHelpers` предоставляет для этого помощника:

```
using System.Runtime.CompilerServices;  
// ...  
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

Примечание : вся статическая инициализация (инициализаторы полей, например) будет выполняться не только самим конструктором.

Потенциальные способы использования : принудительная инициализация во время заставки в приложении пользовательского интерфейса или обеспечение того, что статический конструктор не сбой в модульном teste.

Вызов виртуальных методов в конструкторе

В отличие от C ++ в C # вы можете вызвать виртуальный метод из конструктора классов (OK, вы также можете на C ++, но поведение сначала удивительно). Например:

```
abstract class Base
{
    protected Base()
    {
        _obj = CreateAnother();
    }

    protected virtual AnotherBase CreateAnother()
    {
        return new AnotherBase();
    }

    private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }

    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived
```

Если вы исходите из фона C ++, это удивительно, конструктор базового класса уже видит таблицу виртуальных методов производного класса!

Будьте осторожны : производный класс еще не полностью инициализирован (его конструктор будет выполняться после конструктора базового класса), и этот метод опасен (для этого также существует предупреждение StyleCop). Обычно это считается плохой практикой.

Общие статические конструкторы

Если тип, на котором объявлен статический конструктор, является общим, статический конструктор будет вызываться один раз для каждой уникальной комбинации общих

аргументов.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }

}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

Это приведет к выводу:

```
System.Object
System.String
```

См. Также [Как работают статические конструкторы для общих типов?](#)

Исключения в статических конструкторах

Если статический конструктор генерирует исключение, он никогда не будет повторен. Тип непригоден для жизни AppDomain. Любые дополнительные применения этого типа будут вызывать `TypeInitializationException` обернутое вокруг исходного исключения.

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }

    public static void Yawn() { }

}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

Это приведет к выводу:

Статический калькулятор

System.TypeInitializationException: инициализатор типа для «Animal» сделал исключение. ---> System.Exception: выбрано исключение типа «System.Exception».

[...]

System.TypeInitializationException: инициализатор типа для «Animal» сделал исключение. ---> System.Exception: выбрано исключение типа «System.Exception».

где вы можете видеть, что фактический конструктор выполняется только один раз, и исключение повторно используется.

Инициализация конструктора и объектов

Устанавливается ли присвоение значения свойства *до* или *после* конструктора класса?

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

var testInstance = new TestClass() { TestProperty = 1 };
```

В приведенном выше `TestProperty` значение `TestProperty` равно 1 в конструкторе класса или после конструктора класса?

Присвоение значений свойств в создании экземпляра следующим образом:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Будет выполнен **после** запуска конструктора. Однако инициализировать значение свойства в свойстве класса в C # 6.0 следующим образом:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

будет выполняться **до** запуска конструктора.

Объединение двух концепций выше в одном примере:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }

    static void Main(string[] args)
    {
        var testInstance = new TestClass() { TestProperty = 1 };
        Console.WriteLine(testInstance.TestProperty); //resulting in 1
    }
}
```

Конечный результат:

```
"Or shall this be executed"
"1"
```

Объяснение:

Значение `TestProperty` сначала будет назначено как `2`, тогда будет `TestClass` конструктор `TestClass`, в результате чего будет напечатана

```
"Or shall this be executed"
```

И тогда `TestProperty` будет назначен как `1` из-за `new TestClass() { TestProperty = 1 }`, в результате чего окончательное значение для `TestProperty` напечатанное `Console.WriteLine(testInstance.TestProperty)` будет

"1"

Прочтайте Конструкторы и финализаторы онлайн: <https://riptutorial.com/ru/csharp/topic/25/конструкторы-и-финализаторы>

глава 90: Конструкции потока данных параллельной библиотеки задач (TPL)

Examples

JoinBlock

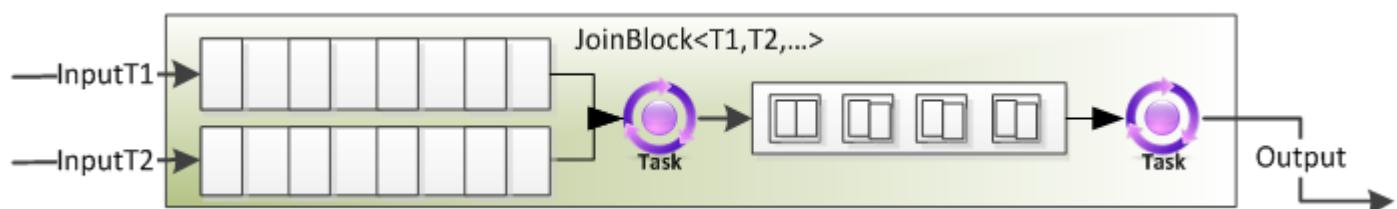
(Собирает 2-3 входа и объединяет их в Tuple)

Подобно BatchBlock, JoinBlock $\langle T1, T2, \dots \rangle$ способен группировать данные из нескольких источников данных. На самом деле, это основная задача JoinBlock $\langle T1, T2, \dots \rangle$.

Например, JoinBlock $\langle \text{string}, \text{double}, \text{int} \rangle$ - это $\text{ISourceBlock} \langle \text{Tuple} \langle \text{string}, \text{double}, \text{int} \rangle \rangle$.

Как и в случае с BatchBlock, JoinBlock $\langle T1, T2, \dots \rangle$ способен работать как в жадном, так и в не жадном режиме.

- В жадном режиме по умолчанию все данные, предлагаемые для целей, принимаются, даже если у другой цели нет необходимых данных, чтобы сформировать кортеж.
- В нежадном режиме целевые объекты блока будут откладывать данные до тех пор, пока всем целям не будут предложены необходимые данные для создания кортежа, после чего блок будет участвовать в двухфазном протоколе фиксации для атомарного извлечения всех необходимых элементов из источников. Эта отсрочка позволяет другому субъекту потреблять данные тем временем, чтобы позволить общей системе продвигаться вперед.



Запросы обработки с ограниченным количеством объединенных объектов

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);
})
```

```

        return resource;
    });

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);

```

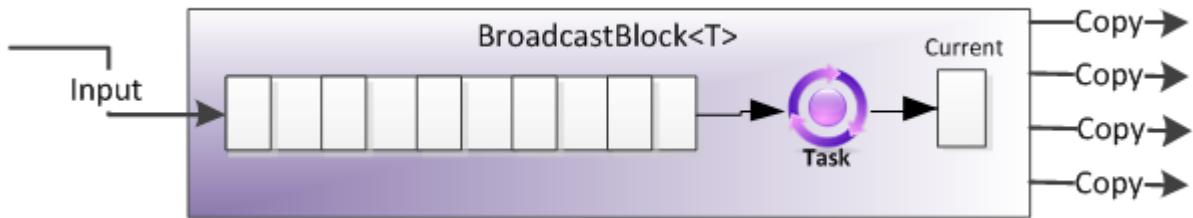
Введение в поток данных TPL Стивена Туба

BroadcastBlock

(Скопируйте элемент и отправьте копии в каждый блок, к которому он привязан)

В отличие от BufferBlock, миссия BroadcastBlock в жизни - включить все цели, связанные с блоком, чтобы получить копию каждого опубликованного элемента, постоянно переписывая «текущее» значение с теми, которые были распространены на него.

Кроме того, в отличие от BufferBlock, BroadcastBlock не требует лишних данных. После того как конкретная data была предложена всем целям, этот элемент будет перезаписан любым фрагментом данных в строке (как и во всех блоках потока данных, сообщения обрабатываются в порядке FIFO). Этот элемент будет предлагаться всем целям и т. д.



Асинхронный производитель / потребитель с дроссельной заслонкой

```

var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);

```

Предоставление статуса от агента

```

public class MyAgent
{

```

```

public ISourceBlock<string> Status { get; private set; }

public MyAgent()
{
    Status = new BroadcastBlock<string>();
    Run();
}

private void Run()
{
    Status.Post("Starting");
    Status.Post("Doing cool stuff");
    ...
    Status.Post("Done");
}
}

```

Введение в поток данных TPL Стивена Туба

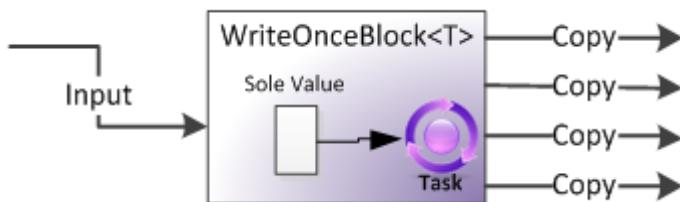
WriteOnceBlock

(Readonly variable: запоминает свой первый элемент данных и выдает копии его в качестве вывода. Игнорирует все другие элементы данных)

Если BufferBlock является самым фундаментальным блоком в потоке данных TPL, WriteOnceBlock является самым простым.

Он хранит не более одного значения, и как только это значение будет установлено, оно никогда не будет заменено или перезаписано.

Вы можете думать о WriteOnceBlock как о сходстве с переменной-членом только для чтения в C #, за исключением того, что вместо того, чтобы настраиваться только в конструкторе, а затем быть неизменяемым, он устанавливается только один раз и неизменен.



Разделение потенциальных результатов задачи

```

public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);

    try
    {
        result.Post(await task);
    }
}

```

```
        }  
        catch (Exception ex)  
        {  
            exception.Post (ex)  
        }  
    }  
}
```

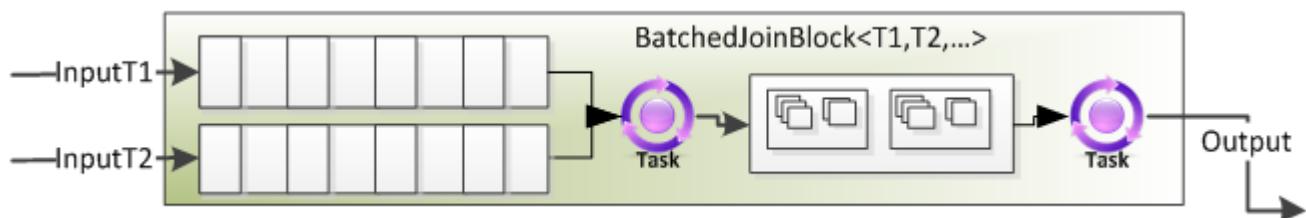
Введение в поток данных TPL Стивена Туба

BatchedJoinBlock

(Собирает определенное количество общих элементов из 2-3 входов и группирует их в набор наборов элементов данных)

BatchedJoinBlock $\langle T_1, T_2, \dots \rangle$ является в некотором смысле комбинацией BatchBlock и JoinBlock $\langle T_1, T_2, \dots \rangle$.

В то время как `JoinBlock <T1, T2, ...>` используется для объединения одного входа от каждой цели в кортеж, а `BatchBlock` используется для объединения N входов в коллекцию, `BatchedJoinBlock <T1, T2, ...>` используется для сбора N входов из всех целей в кортежей коллекций.



Scatter / Gather

Рассмотрим проблему рассеяния / сбора, в которой запущены операции N, некоторые из которых могут преуспеть и вывести строковые выходы, а другие из них могут завершиться неудачей и создать исключения.

```
var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
```

```
{  
    Console.WriteLine(e);  
}
```

Введение в поток данных TPL Стивена Туба

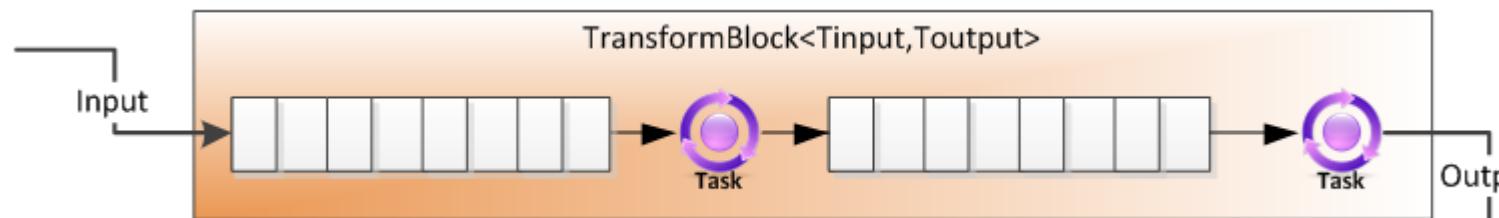
TransformBlock

(Выбрать, один к одному)

Как и в случае с ActionBlock, TransformBlock `<TInput, TOutput>` позволяет выполнить делегат для выполнения некоторых действий для каждой входной базы данных; **в отличие от ActionBlock, эта обработка имеет выход**. Этот делегат может быть `Func <TInput, TOutput>`, и в этом случае обработка этого элемента считается завершенной при возврате делегата или может быть `Func <TInput, Task>`, в этом случае обработка этого элемента считается завершенной не когда делегат возвращается, но когда возвращенная Задача завершается. Для тех, кто знаком с LINQ, он несколько похож на `Select ()`, поскольку он принимает вход, каким-то образом преобразует этот вход, а затем производит вывод.

По умолчанию TransformBlock `<TInput, TOutput>` последовательно обрабатывает свои данные с помощью `MaxDegreeOfParallelism`, равного 1. В дополнение к приему буферизованного ввода и его обработке этот блок будет также обрабатывать все обработанные выходные данные и буфер (данные, которые не были обработаны и обработаны данные).

Он имеет 2 задачи: один для обработки данных, а другой - для передачи данных в следующий блок.



Параллельный трубопровод

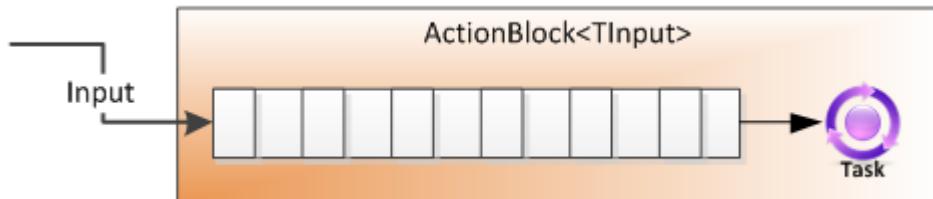
```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));  
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));  
  
compressor.LinkTo(Encryptor);
```

Введение в поток данных TPL Стивена Туба

ActionBlock

(для каждого)

Этот класс можно логически мыслить в качестве буфера для обрабатываемых данных в сочетании с задачами обработки этих данных, причем «блок потока данных» управляет обоими. В своем самом основном использовании мы можем создавать экземпляры ActionBlock и «post»; делегат, предоставленный при построении ActionBlock, будет выполняться асинхронно для каждой части данных.



Синхронные вычисления

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
...
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Дросселирование асинхронных загрузок не более 5 одновременно

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

Введение в поток данных TPL Стивена Туба

TransformManyBlock

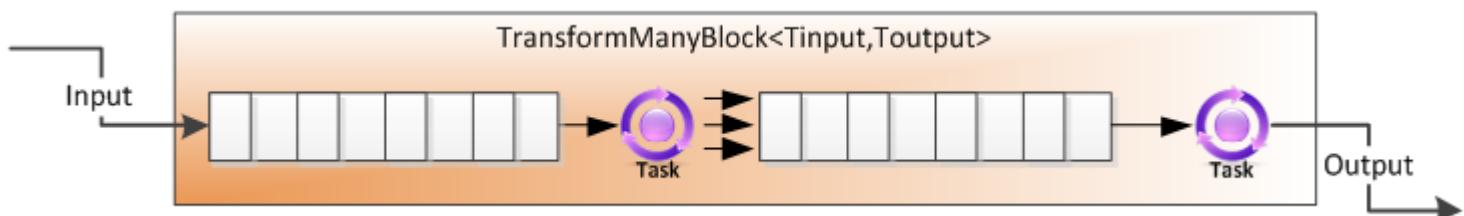
(SelectMany, 1-м: результаты этого сопоставления «сплющены», как и LINQ SelectMany)

TransformManyBlock <TInput, TOutput> очень похож на TransformBlock <TInput, TOutput>. Главное отличие состоит в том, что, если TransformBlock <TInput, TOutput> создает один и только один вывод для каждого входа, TransformManyBlock <TInput, TOutput> производит любое количество (ноль или более) выходов для каждого входа. Как и в ActionBlock и TransformBlock <TInput, TOutput>, эта обработка может быть задана с использованием делегатов, как для синхронной, так и для асинхронной обработки.

Для синхронного используется Func <TInput, IEnumerable>, а для асинхронных используется Func <TInput, Task <IEnumerable>>. Как и для ActionBlock, так и для

TransformBlock `<TInput, TOutput>`, TransformManyBlock `<TInput, TOutput>` по умолчанию используется последовательная обработка, но может быть настроена иначе.

Делегат сопоставления перенастраивает набор элементов, которые вставляются индивидуально в выходной буфер.



Асинхронный веб-сканер

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{ }

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Расширение перечислимого в его составные элементы

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

Фильтрация путем перехода от 1 до 0 или 1 элементов

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Введение в поток данных TPL Стивена Туба

BatchBlock

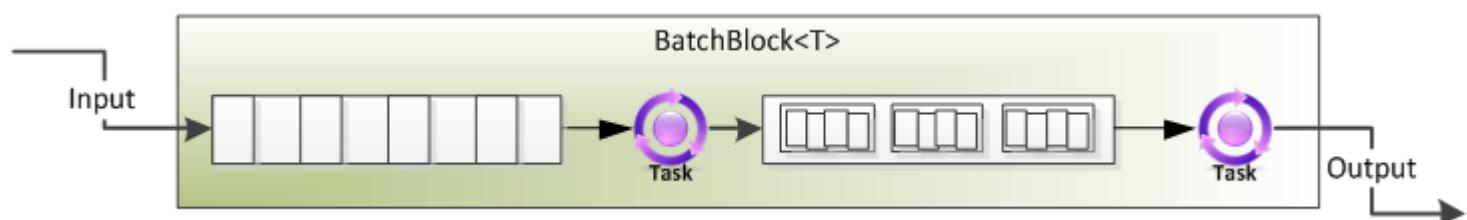
(Группирует определенное количество последовательных элементов данных в коллекции элементов данных)

BatchBlock объединяет N отдельных элементов в один пакетный элемент, представленный как массив элементов. Экземпляр создается с определенным размером партии, и блок затем создает пакет, как только он получает это количество элементов, асинхронно

выводит пакет в выходной буфер.

BatchBlock способен выполнять как жадные, так и неживые режимы.

- В жадном режиме по умолчанию все сообщения, предлагаемые блоку из любого количества источников, принимаются и буферизуются для преобразования в партии.
- - В не жадном режиме все сообщения откладываются из источников, пока достаточное количество источников не предложит блок сообщения для создания пакета. Таким образом, BatchBlock может использоваться для приема 1 элемента из каждого из N источников, N элементов из 1 источника и множества параметров между ними.



Группировка запросов в группы по 100 для отправки в базу данных

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request []>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

Создание партии один раз в секунду

```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

Введение в поток данных TPL Стивена Туба

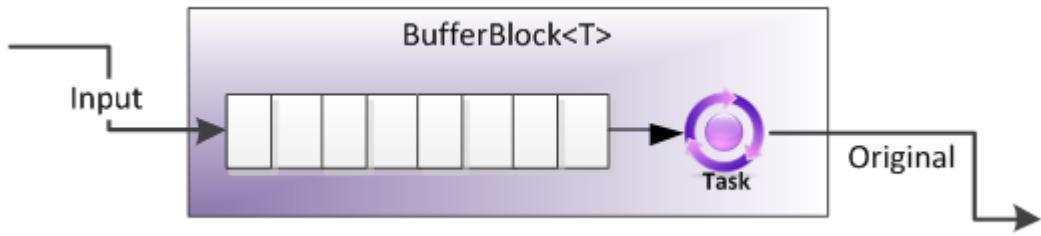
BufferBlock

(Очередь FIFO: данные, которые поступают, - это данные, которые исчезают)

Короче говоря, BufferBlock предоставляет неограниченный или ограниченный буфер для хранения экземпляров T.

Вы можете «отправить» экземпляры T в блок, которые заставляют данные, которые будут размещаться, быть сохранены в порядке первого входа в первый (FIFO) блоком.

Вы можете «получать» из блока, что позволяет синхронно или асинхронно получать экземпляры T, ранее сохраненные или доступные в будущем (опять же, FIFO).



Асинхронный производитель / потребитель с дроссельной заслонкой

```

// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}

```

Введение в поток данных TPL Стивена Туба

Прочтайте Конструкции потока данных параллельной библиотеки задач (TPL) онлайн:
<https://riptutorial.com/ru/csharp/topic/3110/конструкции-потока-данных-параллельной-библиотеки-задач--tpl->

глава 91: Контекст синхронизации в Async-Await

Examples

Псевдокод для ключевых слов async / await

Рассмотрим простой асинхронный метод:

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

Упрощение, мы можем сказать, что этот код на самом деле означает следующее:

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

Это означает, что `async` слова `async / await` используют текущий контекст синхронизации, если он существует. Т.е. вы можете написать код библиотеки, который будет корректно работать в пользовательских интерфейсах, веб-и консольных приложениях.

[Исходная статья](#).

Отключение синхронизации

Чтобы отключить контекст синхронизации, вы должны вызвать метод `ConfigureAwait`:

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}
```

```
    . . .
    Foo().ConfigureAwait(false);
```

ConfigureAwait предоставляет средства для предотвращения поведения по умолчанию SynchronizationContext; передача false для параметра flowContext запрещает использование SynchronizationContext для возобновления выполнения после ожидания.

Цитата из [It's All About SynchronizationContext](#) .

Почему SynchronizationContext так важен?

Рассмотрим этот пример:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

Этот метод заморозит приложение пользовательского интерфейса до тех пор, пока RunTooLong не будет завершен. Приложение будет неактуальным.

Вы можете попробовать выполнить внутренний код асинхронно:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

Но этот код не будет выполняться, потому что внутреннее тело может быть запущено на не-UI-потоке и [не должно напрямую изменять свойства пользовательского интерфейса](#) :

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            label1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

Теперь не забудьте всегда использовать этот шаблон. Или попробуйте [SynchronizationContext.Post](#) , который сделает это для вас:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1.Text;
        }, null);
    });
}
```

Прочтайте Контекст синхронизации в Async-Await онлайн:

<https://riptutorial.com/ru/csharp/topic/7381/контекст-синхронизации-в-async-await>

глава 92: Кортеж

Examples

Создание кортежей

Кортежи создаются с использованием общих типов `Tuple<T1> - Tuple<T1, T2, T3, T4, T5, T6, T7, T8>`

. Каждый из типов представляет собой кортеж, содержащий от 1 до 8 элементов.

Элементы могут быть разных типов.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

Кортежи также могут быть созданы с использованием статических методов `Tuple.Create` . В этом случае типы элементов выводятся компилятором С #.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

7,0

Начиная с С # 7.0, Tuples можно легко создать с помощью [ValueTuple](#) .

```
var tuple = ("foo", 123, true, new MyClass());
```

Элементы можно назвать проще для разложения.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

Доступ к элементам кортежа

Для доступа кортежей элементы используют `Item1 - Item8` свойства. Будут доступны только свойства с номером индекса, меньшим или равным размеру кортежа (т. `Item3` Нельзя получить доступ к свойству `Item3` в `Tuple<T1, T2>`).

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new MyClass()
```

Сравнение и сортировка кортежей

Кортежи можно сравнивать по их элементам.

В качестве примера, перечисляемый, чьи элементы имеют тип Tuple может быть отсортирован на основе операторов сравнения, определенных на указанном элементе:

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
// (1, bar)
// (2, foo)
// (3, qux)
```

Или изменить порядок сортировки:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

Возвращать несколько значений из метода

Кортежи могут использоваться для возврата нескольких значений из метода без использования параметров. В следующем примере AddMultiply используется для возврата двух значений (sum, product).

```
void Write()
{
    var result = AddMultiply(25, 28);
    Console.WriteLine(result.Item1);
    Console.WriteLine(result.Item2);
}

Tuple<int, int> AddMultiply(int a, int b)
{
    return new Tuple<int, int>(a + b, a * b);
}
```

Выход:

53

700

Теперь C # 7.0 предлагает альтернативный способ возврата нескольких значений из методов с использованием кортежей значений. [Дополнительная информация о ValueTuple struct](#).

Прочтите Кортеж онлайн: <https://riptutorial.com/ru/csharp/topic/838/кортеж>

глава 93: Криптография (System.Security.Cryptography)

Examples

Современные примеры симметричного аутентифицированного шифрования строки

Криптография - это что-то очень трудное, и, потратив много времени на чтение разных примеров и выясняя, как легко ввести некоторую форму уязвимости, я нашел ответ, первоначально написанный @jbtule, который, я думаю, очень хорош. Наслаждайся чтением:

«Общей передовой практикой симметричного шифрования является использование аутентифицированного шифрования с ассоциированными данными (AEAD), однако это не является частью стандартных библиотек [cnet .net](#). Таким образом, в первом примере используется [AES256](#), а затем [HMAC256](#), двухэтапный [шифрование, тогда MAC](#), для чего требуется больше накладных расходов и больше ключей.

Во втором примере используется более простая практика AES256- [GCM](#) с использованием Bugcy Castle с открытым исходным кодом (через nuget).

Оба примера имеют основную функцию, которая берет секретную строку сообщения, ключ (ы) и необязательную несекретную полезную нагрузку, а также возвращаемую и аутентифицированную зашифрованную строку, необязательно добавленную к несекретным данным. В идеале вы использовали бы их с 256-битным ключом (-ами), случайно сгенерированным в `NewKey()` .

Оба примера также имеют вспомогательные методы, которые используют строковый пароль для генерации ключей. Эти вспомогательные методы предоставляются в качестве удобства для сопоставления с другими примерами, однако они гораздо менее безопасны, поскольку сила пароля будет намного слабее, чем 256-битный ключ .

Обновление: добавлены `byte[]` перегрузки, и только [Gist](#) имеет полное форматирование с 4-мя отступом и api docs из-за ограничений ответа StackOverflow. "

.NET Встроенный шифрование (AES) -Тен-MAC (HMAC) [[Gist](#)]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
```

```

*/
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
{
    public static class AESThenHMAC
    {
        private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

        //Preconfigured Encryption Parameters
        public static readonly int BlockBitSize = 128;
        public static readonly int KeyBitSize = 256;

        //Preconfigured Password Key Derivation Parameters
        public static readonly int SaltBitSize = 64;
        public static readonly int Iterations = 10000;
        public static readonly int MinPasswordLength = 12;

        /// <summary>
        /// Helper that generates a random key on each call.
        /// </summary>
        /// <returns></returns>
        public static byte[] NewKey()
        {
            var key = new byte[KeyBitSize / 8];
            Random.GetBytes(key);
            return key;
        }

        /// <summary>
        /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
        /// </summary>
        /// <param name="secretMessage">The secret message.</param>
        /// <param name="cryptKey">The crypt key.</param>
        /// <param name="authKey">The auth key.</param>
        /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
        /// <returns>
        /// Encrypted Message
        /// </returns>
        /// <exception cref="System.ArgumentException">Secret Message Required!;secretMessage</exception>
        /// <remarks>
        /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +
        HMac-Tag(32)) * 1.33 Base64
        /// </remarks>
        public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                                           byte[] nonSecretPayload = null)
        {
            if (string.IsNullOrEmpty(secretMessage))
                throw new ArgumentException("Secret Message Required!", "secretMessage");

            var plainText = Encoding.UTF8.GetBytes(secretMessage);
            var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
            return Convert.ToString(cipherText);
        }

        /// <summary>

```

```

/// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="cryptKey">The crypt key.</param>
/// <param name="authKey">The auth key.</param>
/// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
/// <returns>
/// Decrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[]
authKey,
int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

/// <summary>
/// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
/// using Keys derived from a Password (PBKDF2).
/// </summary>
/// <param name="secretMessage">The secret message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayload">The non secret payload.</param>
/// <returns>
/// Encrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">password</exception>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// Adds additional non secret payload for key generation parameters.
/// </remarks>
public static string SimpleEncryptWithPassword(string secretMessage, string password,
                                              byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
    return Convert.ToBase64String(cipherText);
}

/// <summary>
/// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
/// using keys derived from a password (PBKDF2).
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
/// <returns>
/// Decrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>

```

```

/// <remarks>
/// Significantly less secure than using random binary keys.
/// </remarks>
public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                                              int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
{
    //User Error Checks
    if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

    if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");

    if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    //non-secret payload optional
    nonSecretPayload = nonSecretPayload ?? new byte[] { };

    byte[] cipherText;
    byte[] iv;

    using (var aes = new AesManaged
    {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
    })
    {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
            using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(cryptoStream))
            {
                //Encrypt Data
                binaryWriter.Write(secretMessage);
            }

            cipherText = cipherStream.ToArray();
        }
    }
}

```

```

    }

    //Assemble encrypted message and add authentication
    using (var hmac = new HMACSHA256(authKey))
    using (var encryptedStream = new MemoryStream())
    {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
        {
            //Prepend non-secret payload if any
            binaryWriter.Write(nonSecretPayload);
            //Prepend IV
            binaryWriter.Write(iv);
            //Write Ciphertext
            binaryWriter.Write(cipherText);
            binaryWriter.Flush();

            //Authenticate all data
            var tag = hmac.ComputeHash(encryptedStream.ToArray());
            //Postpend tag
            binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
    }

}

public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[]
authKey, int nonSecretPayloadLength = 0)
{
    //Basic Usage Error Checks
    if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!", KeyBitSize), "cryptKey");

    if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize), "authKey");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    using (var hmac = new HMACSHA256(authKey))
    {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length - sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
            return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0, sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
    }
}

```

```

        for (var i = 0; i < sentTag.Length; i++)
            compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
            return null;

        using (var aes = new AesManaged
        {
            KeySize = KeyBitSize,
            BlockSize = BlockBitSize,
            Mode = CipherMode.CBC,
            Padding = PaddingMode.PKCS7
        })
        {

            //Grab IV from message
            var iv = new byte[ivLength];
            Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

            using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
            using (var plainTextStream = new MemoryStream())
            {
                using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
                    using (var binaryWriter = new BinaryWriter(decrypterStream))
                {
                    //Decrypt Cipher Text from Message
                    binaryWriter.Write(
                        encryptedMessage,
                        nonSecretPayloadLength + iv.Length,
                        encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
                    );
                }
                //Return Plain Text
                return plainTextStream.ToArray();
            }
        }
    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
{
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    int payloadIndex = nonSecretPayload.Length;

    byte[] cryptKey;
    byte[] authKey;
}

```

```

//Use Random Salt to prevent pre-generated weak password attacks.
using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
{
    var salt = generator.Salt;

    //Generate Keys
    cryptKey = generator.GetBytes(KeyBitSize / 8);

    //Create Non Secret Payload
    Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
    payloadIndex += salt.Length;
}

//Deriving separate key, might be less efficient than using HKDF,
//but now compatible with RNEncryptor which had a very similar wireformat and requires
less code than HKDF.
using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
{
    var salt = generator.Salt;

    //Generate Keys
    authKey = generator.GetBytes(KeyBitSize / 8);

    //Create Rest of Non Secret Payload
    Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
}

return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
}

public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
{
    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cryptSalt = new byte[SaltBitSize / 8];
    var authSalt = new byte[SaltBitSize / 8];

    //Grab Salt from Non-Secret Payload
    Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
    Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

    byte[] cryptKey;
    byte[] authKey;

    //Generate crypt key
    using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
    {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
    }
    //Generate auth key
    using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
    {
        authKey = generator.GetBytes(KeyBitSize / 8);
    }
}

```

```
        }

        return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length +
authSalt.Length + nonSecretPayloadLength);
    }
}
}
```

Bouncy Castle AES-GCM [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{
    public static class AESGCM
    {
        private static readonly SecureRandom Random = new SecureRandom();

        //Preconfigured Encryption Parameters
        public static readonly int NonceBitSize = 128;
        public static readonly int MacBitSize = 128;
        public static readonly int KeyBitSize = 256;

        //Preconfigured Password Key Derivation Parameters
        public static readonly int SaltBitSize = 128;
        public static readonly int Iterations = 10000;
        public static readonly int MinPasswordLength = 12;

        /// <summary>
        /// Helper that generates a random new key on each call.
        /// </summary>
        /// <returns></returns>
        public static byte[] NewKey()
        {
            var key = new byte[KeyBitSize / 8];
            Random.NextBytes(key);
            return key;
        }

        /// <summary>
        /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
        /// </summary>
        /// <param name="secretMessage">The secret message.</param>
    }
}
```

```

/// <param name="key">The key.</param>
/// <param name="nonSecretPayload">Optional non-secret payload.</param>
/// <returns>
/// Encrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
/// <remarks>
/// Adds overhead of (Optional-Payload + BlockSize(16) + Message + HMac-Tag(16)) * 1.33
Base64
/// </remarks>
public static string SimpleEncrypt(string secretMessage, byte[] key, byte[]
nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
    return Convert.ToBase64String(cipherText);
}

/// <summary>
/// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="key">The key.</param>
/// <param name="nonSecretPayloadLength">Length of the optional non-secret
payload.</param>
/// <returns>Decrypted Message</returns>
public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
{
    if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

/// <summary>
/// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
/// using key derived from a password (PBKDF2).
/// </summary>
/// <param name="secretMessage">The secret message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayload">The non secret payload.</param>
/// <returns>
/// Encrypted Message
/// </returns>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// Adds additional non secret payload for key generation parameters.
/// </remarks>
public static string SimpleEncryptWithPassword(string secretMessage, string password,
                                                byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");
}

```

```

var plainText = Encoding.UTF8.GetBytes(secretMessage);
var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
return Convert.ToBase64String(cipherText);
}

/// <summary>
/// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
/// using a key derived from a password (PBKDF2)
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
/// <returns>
/// Decrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// </remarks>
public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                                                int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[]
nonSecretPayload = null)
{
    //User Error Checks
    if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    //Non-secret Payload Optional
    nonSecretPayload = nonSecretPayload ?? new byte[] { };

    //Using random nonce large enough not to repeat
    var nonce = new byte[NonceBitSize / 8];
    Random.NextBytes(nonce, 0, nonce.Length);

    var cipher = new GcmBlockCipher(new AesFastEngine());
    var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
    cipher.Init(true, parameters);

    //Generate Cipher Text With Auth Tag
    var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
    var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
    cipher.DoFinal(cipherText, len);
}

```

```

//Assemble Message
using (var combinedStream = new MemoryStream())
{
    using (var binaryWriter = new BinaryWriter(combinedStream))
    {
        //Prepend Authenticated Payload
        binaryWriter.Write(nonSecretPayload);
        //Prepend Nonce
        binaryWriter.Write(nonce);
        //Write Cipher Text
        binaryWriter.Write(cipherText);
    }
    return combinedStream.ToArray();
}

public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
{
    //User Error Checks
    if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    using (var cipherStream = new MemoryStream(encryptedMessage))
    using (var cipherReader = new BinaryReader(cipherStream))
    {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);

        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length -
nonSecretPayloadLength - nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
            var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
            cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
            //Return null if it doesn't authenticate
            return null;
        }

        return plainText;
    }
}

```

```

}

public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
{
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0} characters!", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var generator = new Pkcs5S2ParametersGenerator();

    //Use Random Salt to minimize pre-generated weak password attacks.
    var salt = new byte[SaltBitSize / 8];
    Random.NextBytes(salt);

    generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

    //Generate Key
    var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

    //Create Full Non Secret Payload
    var payload = new byte[salt.Length + nonSecretPayload.Length];
    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    Array.Copy(salt, 0, payload, nonSecretPayload.Length, salt.Length);

    return SimpleEncrypt(secretMessage, key.GetKey(), payload);
}

public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
{
    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0} characters!", MinPasswordLength), "password");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var generator = new Pkcs5S2ParametersGenerator();

    //Grab Salt from Payload
    var salt = new byte[SaltBitSize / 8];
    Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

    generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

    //Generate Key
    var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);
}

```

```
        return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length +  
nonSecretPayloadLength);  
    }  
}  
}
```

Введение в симметричное и асимметричное шифрование

Вы можете улучшить безопасность для транзита или хранения данных, применяя методы шифрования. В основном существует два подхода при использовании `System.Security.Cryptography` : **симметричный и асимметричный**.

Симметричное шифрование

Этот метод использует закрытый ключ для выполнения преобразования данных.

Плюсы:

- Симметричные алгоритмы потребляют меньше ресурсов и быстрее асимметричных.
- Объем данных, которые вы можете зашифровать, неограничен.

Минусы:

- Шифрование и дешифрование используют один и тот же ключ. Кто-то сможет расшифровать ваши данные, если ключ скомпрометирован.
- У вас может получиться множество разных секретных ключей для управления, если вы решите использовать другой секретный ключ для разных данных.

В `System.Security.Cryptography` у вас есть разные классы, которые выполняют симметричное шифрование, они известны как **блок-шифры** :

- `AesManaged` (алгоритм [AES](#)).
 - `AesCryptoServiceProvider` (алгоритм [AES FIPS 140-2](#)).
 - `DESCryptoServiceProvider` (алгоритм [DES](#)).
 - `RC2CryptoServiceProvider` (алгоритм [Rivest Cipher 2](#)).
 - `RijndaelManaged` (алгоритм [AES](#)). *Примечание* : `RijndaelManaged` не является жалобой FIPS-197 .
 - `TripleDES` (алгоритм [TripleDES](#)).
-

Асимметричное шифрование

Этот метод использует комбинацию открытых и закрытых ключей для выполнения преобразования данных.

Плюсы:

- Он использует большие ключи, чем симметричные алгоритмы, поэтому они менее подвержены взлому с помощью грубой силы.
- Легче гарантировать, кто способен шифровать и расшифровывать данные, поскольку он использует два ключа (общедоступные и частные).

Минусы:

- Существует ограничение на количество данных, которые вы можете зашифровать. Предел для каждого алгоритма различен и обычно пропорционален размеру ключа алгоритма. Например, объект RSACryptoServiceProvider с длиной ключа 1024 байта может шифровать только сообщение размером менее 128 байт.
- Асимметричные алгоритмы очень медленны по сравнению с симметричными алгоритмами.

В System.Security.Cryptography у вас есть доступ к различным классам, которые выполняют асимметричное шифрование:

- [DSACryptoServiceProvider](#) (алгоритм алгоритма цифровой подписи)
- [RSACryptoServiceProvider](#) (алгоритм алгоритма RSA)

Хеширование паролей

Пароли никогда не должны храниться как обычный текст! Они должны быть хэшированы со случайно генерируемой солью (для защиты от атак радужного стола) с использованием медленного алгоритма хэширования пароля. Большое количество итераций (> 10k) может быть использовано для замедления атаки грубой силы. Задержка ~ 100 мс приемлема для входа пользователя в систему, но затрудняет нарушение длинного пароля. При выборе количества итераций вы должны использовать максимально допустимое значение для своего приложения и увеличивать его по мере повышения производительности компьютера. Вам также необходимо будет рассмотреть возможность повторения повторяющихся запросов, которые могут быть использованы как DoS-атака.

При хешировании в первый раз соль может быть сгенерирована для вас, полученный хеш и соль могут быть затем сохранены в файле.

```
private void firstHash(string userName, string userPassword, int numberofIterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberofIterations);
    //Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);      //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberofIterations); //Store the hashed
    password with the salt and number of iterations to check against future password entries
}
```

Проверка существующего пароля пользователя, чтение хеша и соли из файла и сравнение с хешем введенного пароля

```
private bool checkPassword(string userName, string userPassword, int numberOfIterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
    numberOfIterations); //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20); //Returns a 20 byte hash
    bool passwordsMatch = comparePasswords(usersHash, hashedPassword); //Compares byte
    arrays
    return passwordsMatch;
}
```

Простое симметричное шифрование файлов

Следующий пример кода демонстрирует быстрое и простое средство шифрования и дешифрования файлов с использованием симметричного алгоритма AES.

Код случайным образом генерирует векторы соли и инициализации каждый раз, когда файл зашифрован, а это означает, что шифрование одного и того же файла с одним и тем же паролем всегда приведет к разному выходу. Соль и IV записываются в выходной файл, поэтому для его расшифровки требуется только пароль.

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string
outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
            fsIn.Read(salt, 0, saltLength);
            fsIn.Read(iv, 0, iv.Length);
        }

        // Generate a secure password, based on the password and salt provided
    }
}
```

```
var pdb = new Rfc2898DeriveBytes(password, salt);
var key = pdb.GetBytes(cypher.KeySize / 8);

// Encrypt or decrypt the file
using (var cryptoTransform = encryptMode
    ? cypher.CreateEncryptor(key, iv)
    : cypher.CreateDecryptor(key, iv))
using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
{
    fsIn.CopyTo(cs);
}
}
```

Криптографически безопасные случайные данные

Бывают случаи, когда класс Random () Framework не может считаться случайным, учитывая, что он основан на генераторе псевдослучайных чисел. Однако классы Crypto структуры обеспечивают нечто более надежное в виде RNGCryptoServiceProvider.

В следующих примерах кода показано, как создавать криптографически безопасные байтовые массивы, строки и номера.

Случайный байт-массив

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

Случайное целое число (с четным распределением)

```
public static int GenerateRandomInt(int minValue=0, int maxValue=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxValue - minValue + 1) + minValue);
}
```

Случайная строка

```
public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
```

```

rng.GetBytes(rnd);

// Generate the output string
var allowable = allowableChars.ToCharArray();
var l = allowable.Length;
var chars = new char[length];
for (var i = 0; i < length; i++)
    chars[i] = allowable[rnd[i] % l];

return new string(chars);
}

```

Быстрое асимметричное шифрование файлов

Асимметричное шифрование часто считается предпочтительным для симметричного шифрования для передачи сообщений другим сторонам. Это связано главным образом с тем, что оно отрицает многие риски, связанные с обменом общим ключом, и гарантирует, что, хотя любой, у кого есть открытый ключ, может зашифровать сообщение для предполагаемого получателя, только этот получатель может его расшифровать. К сожалению, основным алгоритмом асимметричного шифрования является то, что они значительно медленнее, чем их симметричные кузены. Таким образом, асимметричное шифрование файлов, особенно больших, часто может быть очень интенсивным с вычислительной точки зрения процессом.

Чтобы обеспечить как безопасность, так и производительность, можно использовать гибридный подход. Это влечет за собой криптографически случайное генерирование ключа и вектора инициализации для *симметричного* шифрования. Эти значения затем зашифровываются с использованием *асимметричного* алгоритма и записываются в выходной файл перед использованием для шифрования исходных данных *симметрично* и добавления его к выходу.

Этот подход обеспечивает высокую степень как производительности, так и безопасности, поскольку данные шифруются с использованием симметричного алгоритма (быстро), а ключ и iv, как произвольно сгенерированные (защищенные), зашифрованы асимметричным алгоритмом (безопасным). Это также имеет дополнительное преимущество в том, что одна и та же полезная нагрузка, зашифрованная в разное время, будет иметь очень различный ciphertext, поскольку симметричные ключи генерируются случайным образом каждый раз.

Следующий класс демонстрирует асимметричное шифрование строк и байтовых массивов, а также гибридное шифрование файлов.

```

public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }
}

```

```

public static KeyPair GenerateNewKeyPair(int keySize = 4096)
{
    // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more
    robust but takes a fair bit longer to generate.
    using (var rsa = new RSACryptoServiceProvider(keySize))
    {
        return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
    }
}

#endregion

#region Asymmetric Data Encryption and Decryption

public static byte[] EncryptData(byte[] data, string publicKey)
{
    using (var asymmetricProvider = new RSACryptoServiceProvider())
    {
        asymmetricProvider.FromXmlString(publicKey);
        return asymmetricProvider.Encrypt(data, true);
    }
}

public static byte[] DecryptData(byte[] data, string publicKey)
{
    using (var asymmetricProvider = new RSACryptoServiceProvider())
    {
        asymmetricProvider.FromXmlString(publicKey);
        if (asymmetricProvider.PublicOnly)
            throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
        return asymmetricProvider.Decrypt(data, true);
    }
}

public static string EncryptString(string value, string publicKey)
{
    return Convert.ToBase64String(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
}

public static string DecryptString(string value, string privateKey)
{
    return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value),
privateKey));
}

#endregion

#region Hybrid File Encryption and Decryption

public static void EncryptFile(string inputFilePath, string outputPath, string
publicKey)
{
    using (var symmetricCypher = new AesManaged())
    {
        // Generate random key and IV for symmetric encryption
        var key = new byte[symmetricCypher.KeySize / 8];
        var iv = new byte[symmetricCypher.BlockSize / 8];
        using (var rng = new RNGCryptoServiceProvider())

```

```

    {
        rng.GetBytes(key);
        rng.GetBytes(iv);
    }

    // Encrypt the symmetric key and IV
    var buf = new byte[key.Length + iv.Length];
    Array.Copy(key, buf, key.Length);
    Array.Copy(iv, 0, buf, key.Length, iv.Length);
    buf = EncryptData(buf, publicKey);

    var bufLen = BitConverter.GetBytes(buf.Length);

    // Symmetrically encrypt the data and write it to the file, along with the
    encrypted key and iv
    using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
    using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
    using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
    using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
    {
        fsOut.Write(bufLen, 0, bufLen.Length);
        fsOut.Write(buf, 0, buf.Length);
        fsIn.CopyTo(cs);
    }
}

public static void DecryptFile(string inputFilePath, string outputPath, string
privateKey)
{
    using (var symmetricCypher = new AesManaged())
    using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
    {
        // Determine the length of the encrypted key and IV
        var buf = new byte[sizeof(int)];
        fsIn.Read(buf, 0, buf.Length);
        var bufLen = BitConverter.ToInt32(buf, 0);

        // Read the encrypted key and IV data from the file and decrypt using the
        asymmetric algorithm
        buf = new byte[bufLen];
        fsIn.Read(buf, 0, buf.Length);
        buf = DecryptData(buf, privateKey);

        var key = new byte[symmetricCypher.KeySize / 8];
        var iv = new byte[symmetricCypher.BlockSize / 8];
        Array.Copy(buf, key, key.Length);
        Array.Copy(buf, key.Length, iv, 0, iv.Length);

        // Decrypt the file data using the symmetric algorithm
        using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
        using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
        using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
        {
            fsIn.CopyTo(cs);
        }
    }
}

#endif

```

```

#region Key Storage

public static void WritePublicKey(string publicKeyFilePath, string publicKey)
{
    File.WriteAllText(publicKeyFilePath, publicKey);
}
public static string ReadPublicKey(string publicKeyFilePath)
{
    return File.ReadAllText(publicKeyFilePath);
}

private const string SymmetricSalt = "Stack_Overflow!"; // Change me!

public static string ReadPrivateKey(string privateKeyFilePath, string password)
{
    var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
    var cypherText = File.ReadAllBytes(privateKeyFilePath);

    using (var cypher = new AesManaged())
    {
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);
        var iv = pdb.GetBytes(cypher.BlockSize / 8);

        using (var decryptor = cypher.CreateDecryptor(key, iv))
        using (var msDecrypt = new MemoryStream(cypherText))
        using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read))
            using (var srDecrypt = new StreamReader(csDecrypt))
            {
                return srDecrypt.ReadToEnd();
            }
    }
}

public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string password)
{
    var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
    using (var cypher = new AesManaged())
    {
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);
        var iv = pdb.GetBytes(cypher.BlockSize / 8);

        using (var encryptor = cypher.CreateEncryptor(key, iv))
        using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
        using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor,
CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.WriteLine(privateKey);
            }
    }
}

#endregion
}

```

Пример использования:

```

private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey,
privateKeyPassword);
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}

```

Прочтайте Криптография (System.Security.Cryptography) онлайн:

<https://riptutorial.com/ru/csharp/topic/2988/криптография--system-security-cryptography->

глава 94: Кэширование

Examples

MemoryCache

```
//Get instance of cache
using System.Runtime.Caching;

var cache = MemoryCache.Default;

//Check if cache contains an item with
cache.Contains("CacheKey");

//get item from cache
var item = cache.Get("CacheKey");

//get item from cache or add item if not existing
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",
DateTime.Now.AddHours(12));

//note if item not existing the item is added by this method
//but the method returns null
```

Прочтайте Кэширование онлайн: <https://riptutorial.com/ru/csharp/topic/4383/кэширование>

глава 95: литералы

Синтаксис

- **bool**: true или false
- **byte**: None, целочисленный литерал, неявно преобразованный из int
- **sbyte**: None, целочисленный литерал, неявно преобразованный из int
- **char**: обернуть значение с помощью одиночных кавычек
- **десятичный**: M или m
- **double**: D, d или действительное число
- **float**: F или f
- **int**: Нет, значение по умолчанию для интегральных значений в пределах диапазона int
- **uint**: U, u или интегральные значения в диапазоне uint
- **long**: L, l или интегральные значения в диапазоне длинного
- **ulong**: UL, ul, UI, uL, LU, lu, Lu, IU или интегральные значения в пределах улунга
- **short**: None, целочисленный литерал, неявно преобразованный из int
- **ushort**: None, целочисленный литерал, неявно преобразованный из int
- **string**: оберните значение двойными кавычками, необязательно добавив с помощью @
- **null** : Литеральный null

Examples

int литералы

int литералы определяются простым использованием интегральных значений в диапазоне int :

```
int i = 5;
```

uint литералы

uint литералы определяются с использованием суффикса U или u или с использованием интегральных значений в диапазоне uint :

```
uint ui = 5U;
```

строковые литералы

string литералы определяются оберточным значение с двойными кавычками " :

```
string s = "hello, this is a string literal";
```

Строковые литералы могут содержать escape-последовательности. См. «[Последовательности последовательности строк](#)»

Кроме того, С # поддерживает стенографические литералы (см. Строку [Verbatim Strings](#)). Они определяются путем об包围ивания значения с двойными кавычками " , и предваряя его с @ Escape - последовательности игнорируются в стенографических строковых литералов, и все пробельные символы включены.:

```
string s = @"The path is:  
C:\Windows\System32";  
//The backslashes and newline are included in the string
```

литералы

`char` литералы определяются путем об包围ивания значения с одинарными кавычками ' :

```
char c = 'h';
```

Символьные литералы могут содержать escape-последовательности. См. «[Последовательности последовательности строк](#)»

Литерал символов должен быть ровно на один символ (после того, как все escape-последовательности были оценены). Недопустимые пустые символы. Символ по умолчанию (возвращаемый по `default(char)` или `new char()`) равен '\0' или символу NULL (чтобы не путать с `null` литералом и нулевыми ссылками).

байт-литералы

`byte` тип не имеет буквального суффикса. Целочисленные литералы неявно преобразуются из `int` :

```
byte b = 127;
```

шбайтские литералы

`sbyte` type не имеет буквального суффикса. Целочисленные литералы неявно преобразуются из `int` :

```
sbyte sb = 127;
```

десятичные литералы

`decimal` литералы определяются с использованием суффикса M или m на действительном числе:

```
decimal m = 30.5M;
```

двойные литералы

`double` литералы определяются с использованием суффикса `D` или `d` или с использованием реального числа:

```
double d = 30.5D;
```

плавающие литералы

`float` литералы определяются с использованием суффикса `F` или `f` или с использованием реального числа:

```
float f = 30.5F;
```

длинные литералы

`long` литералы определяются с использованием суффикса `L` или `l` или с использованием интегральных значений в диапазоне `long`:

```
long l = 5L;
```

ulong literal

`ulong` литералы определяются с использованием суффикса `UL`, `ul`, `U1`, `uL`, `LU`, `lu`, `Lu` ИЛИ `lu` или с использованием интегральных значений в диапазоне `ulong`:

```
ulong ul = 5UL;
```

короткий литерал

`short` тип не имеет литерала. Целочисленные литералы неявно преобразуются из `int`:

```
short s = 127;
```

ushort literal

Тип `ushort` не имеет буквального суффикса. Целочисленные литералы неявно преобразуются из `int`:

```
ushort us = 127;
```

букв `bool`

`bool` **литеры являются либо** `true` **либо** `false` ;

```
bool b = true;
```

Прочтайте литералы онлайн: <https://riptutorial.com/ru/csharp/topic/2655/литералы>

глава 96: Лямбда-выражения

замечания

Выражение лямбда является синтаксисом для создания анонимных функций `inline`. Более формально, из руководства по [программированию на C #](#) :

Выражение лямбда является анонимной функцией, которую можно использовать для создания делегатов или типов дерева выражений. Используя лямбда-выражения, вы можете записывать локальные функции, которые могут передаваться в качестве аргументов или возвращаться в качестве значений вызовов функций.

Выражение лямбда создается с помощью оператора `=>`. Поместите любые параметры на левую сторону оператора. С правой стороны поставьте выражение, которое может использовать эти параметры; это выражение будет использоваться как возвращаемое значение функции. Реже, если необходимо, с правой стороны можно использовать целый `{code block}`. Если тип возврата не является недействительным, блок будет содержать оператор `return`.

Examples

Передача выражения Лямбды в качестве параметра метода

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Здесь `x => x > 6` - это лямбда-выражение, действующее как предикат, который гарантирует, что возвращаются только элементы выше 6.

Лямбда-выражения как сокращение для инициализации делегата

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

Вышеупомянутый синтаксис выражения Lambda эквивалентен следующему подробному коду:

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = delegate(int x) {
    return x * 2;
};
```

Lambdas для `Func` и `Action`

Обычно lambdas используются для определения простых функций (как правило, в контексте выражения linq):

```
var incremented = myEnumerable.Select(x => x + 1);
```

Здесь `return` неявно.

Тем не менее, также можно передавать действия как лямбда:

```
myObservable.Do(x => Console.WriteLine(x));
```

Лямбда-выражения с несколькими параметрами или без параметров

Используйте круглые скобки вокруг выражения слева от оператора `=>` чтобы указать несколько параметров.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

Аналогично, пустой набор круглых скобок указывает, что функция не принимает параметры.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.;"
```

Положить несколько выражений в заявление лямбда

В отличие от выражения `lambda`, оператор `lambda` может содержать несколько операторов, разделенных точкой с запятой.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Обратите внимание, что инструкции заключены в фигурные скобки `{}`.

Помните, что оператор `lambdas` не может использоваться для создания деревьев выражений.

Lambdas может излучаться как `Func`, так и `Expression`

Предполагая следующий класс `Person`:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Следующая лямбда:

```
p => p.Age > 18
```

Может передаваться в качестве аргумента для обоих методов:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Поскольку компилятор способен преобразовывать lambdas как для делегатов, так и для Expression .

Очевидно, что провайдеры LINQ в значительной степени полагаются на Expression s (IQueryable<T> основном через интерфейс IQueryable<T>), чтобы иметь возможность анализировать запросы и переводить их для хранения запросов.

Выражение Lambda как обработчик события

Лямбда-выражения могут использоваться для обработки событий, что полезно, когда:

- Обработчик короток.
- Обработчик никогда не должен быть отписано.

Ниже приведена хорошая ситуация, в которой может использоваться обработчик события лямбда:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

Если необходимо отказаться от подписывания зарегистрированного обработчика событий в какой-то будущей точке кода, выражение обработчика события должно быть сохранено в переменной, а регистрация / регистрация осуществляется через эту переменную:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

Причина, по которой это делается, а не просто повторное наложение выражения лямбда дословно, чтобы отменить подписку (-=), заключается в том, что компилятор C # не обязательно будет считать два выражения равными:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Обратите внимание, что если к выражению лямбда добавляются дополнительные операторы, тогда требуемые окружающие фигурные скобки могут быть случайно опущены, не вызывая ошибки времени компиляции. Например:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

Это скомпилируется, но приведет к добавлению лямбда-выражения `(sender, args) => Console.WriteLine("Email sent");` как обработчик события, и выполнение инструкции `emailSendButton.Enabled = true;` немедленно. Чтобы исправить это, содержимое лямбда должно быть окружено фигурными фигурными скобками. Этого можно избежать, используя фигурные скобки с самого начала, будучи осторожными при добавлении дополнительных инструкций в обработчик лямбда-события или окружающих лямбда в круглых скобках с самого начала:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Прочтайте Лямбда-выражения онлайн: <https://riptutorial.com/ru/csharp/topic/46/лямбда-выражения>

глава 97: Лямбда-выражения

замечания

Затворы

Лямбда-выражения будут неявно **захватывать используемые переменные и создавать закрытие**. Закрытие - это функция, а также некоторый контекст состояния. Компилятор будет генерировать замыкание всякий раз, когда выражение лямбда «окружает» значение из его окружающего контекста.

Например, когда выполняется следующее

```
Func<object, bool> safeApplyFiltererPredicate = o => (o != null) && filterer.Predicate(i);
```

safeApplyFilterPredicate относится к вновь созданному объекту, который имеет частную ссылку на текущее значение filterer и метод Invoke которого ведет себя как

```
o => (o != null) && filterer.Predicate(i);
```

Это может быть важно, потому что до тех пор, как ссылка на значение в настоящее время в safeApplyFilterPredicate поддерживается, будет ссылка на объект, который filterer в настоящее время относится к. Это оказывает влияние на сборке мусора, и может привести к непредсказуемым результатам, если объект, который filterer в настоящее время относится к муттирует.

С другой стороны, замыкания могут использоваться для преднамеренного эффекта, чтобы инкапсулировать поведение, которое включает ссылки на другие объекты.

Например

```
var logger = new Logger();
Func<int, int> Add1AndLog = i => {
    logger.Log("adding 1 to " + i);
    return (i + 1);
};
```

Закрытие также может использоваться для моделирования состояний машин:

```
Func<int, int> MyAddingMachine() {
    var i = 0;
    return x => i += x;
};
```

Examples

Основные лямбда-выражения

```
Func<int, int> add1 = i => i + 1;

Func<int, int, int> add = (i, j) => i + j;

// Behaviourally equivalent to:

int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...

Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

Основные лямбда-выражения с LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

Использование синтаксиса лямбда для создания замыкания

См. Замечания для обсуждения замыканий. Предположим, что у нас есть интерфейс:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

и затем выполняется следующее:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
```

```
};
```

Теперь `machineClosure` ссылается на функцию от `int` до `int`, которая за кулисами использует экземпляр `IMachine` который ссылается `machine`, чтобы выполнить вычисление. Даже если ссылка `machine` выходит из области видимости, до тех пор, как `machineClosure` объект сохраняется, оригинальный `IMachine` экземпляр будет сохранена как часть «закрытия», автоматически определяется компилятором.

Предупреждение: это может означать, что один и тот же вызов функции возвращает разные значения в разное время (например, в этом примере, если машина хранит сумму своих входов). Во многих случаях это может быть неожиданным, и его следует избегать для любого кода в функциональном стиле - случайные и неожиданные закрытия могут быть источником ошибок.

Синтаксис Lambda с телом блока оператора

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

Лямбда-выражения с `System.Linq.Expressions`

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

Прочтите **Лямбда-выражения** онлайн: <https://riptutorial.com/ru/csharp/topic/7057/лямбда-выражения>

глава 98: Манипуляция строк

Examples

Изменение случая символов внутри строки

Класс `System.String` поддерживает ряд методов для преобразования между строчными и строчными символами в строке.

- `System.String.ToLowerInvariant` используется для возврата объекта `String`, преобразованного в нижний регистр.
- `System.String.ToUpperInvariant` используется для возврата объекта `String`, преобразованного в верхний регистр.

Примечание . Причина использования *инвариантных* версий этих методов заключается в предотвращении создания неожиданных букв, специфичных для конкретной культуры. Это объясняется [здесь подробно](#) .

Пример:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Обратите внимание, что вы можете указать конкретную **Культуру** при преобразовании в строчные и прописные буквы, используя [методы `String.ToLower \(CultureInfo\)`](#) и [`String.ToUpper \(CultureInfo\)`](#) соответственно.

Поиск строки внутри строки

Используя `System.String.Contains` вы можете узнать, существует ли определенная строка внутри строки. Метод возвращает логическое значение, `true`, если строка существует `false`.

```
string s = "Hello World";
bool stringExists = s.Contains("ello"); //stringExists =true as the string contains the
substring
```

Используя метод `System.String.IndexOf`, вы можете найти начальную позицию подстроки в пределах существующей строки.

Обратите внимание, что возвращенная позиция основана на нуле, возвращается значение - 1, если подстрока не найдена.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

Чтобы найти первое местоположение из **конца** строки, используйте метод `System.String.LastIndexOf`:

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

Удаление (обрезка) белого пространства из строки

Метод `System.String.Trim` можно использовать для удаления всех символов верхнего и нижнего пробелов из строки:

```
string s = "      String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

К тому же:

- Чтобы удалить пустое пространство только с *начала* строки, используйте: `System.String.TrimStart`
- Чтобы удалить пустое пространство только с *конца* строки, используйте: `System.String.TrimEnd`

Подстрока для извлечения части строки.

Метод `System.String.Substring` может использоваться для извлечения части строки.

```
string s ="A portion of word that is retained";
s=str.Substring(26); //s="retained"

s1 = s.Substring(0,5); //s="A por"
```

Замена строки в строке

Используя метод `System.String.Replace`, вы можете заменить часть строки другой строкой.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Все вхождения строки поиска заменяются:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

`String.Replace` также можно использовать для удаления части строки, указав пустую строку в качестве значения замены:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Разделение строки с использованием разделителя

Используйте метод `System.String.Split` для возврата строкового массива, который содержит подстроки исходной строки, разделенные на определенный разделитель:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Выход:

Один
Два
Три
четыре

Объединение массива строк в одну строку

Метод `System.String.Join` позволяет объединить все элементы в массив строк, используя указанный разделитель между каждым элементом:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(", ", words); // singleString = "One, Two, Three, Four"
```

Конкатенация строк

Конкатенация строк может быть выполнена с помощью метода `System.String.Concat` или (намного проще) с помощью оператора `+`:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

Прочтайте Манипуляция строк онлайн: <https://riptutorial.com/ru/csharp/topic/3599/манипуляция-строк>

глава 99: Массивы

Синтаксис

- **Объявление массива:**

`<тип> [] <имя>;`

- **Объявление двумерного массива:**

`<тип> [,] <имя> = новый <тип> [<значение>, <значение>];`

- **Объявление Jagged Array:**

`<тип> [] <имя> = новый <тип> [<значение>];`

- **Объявление subarray для Jagged Array:**

`<имя> [<значение>] = новый <тип> [<значение>];`

- **Инициализация массива без значений:**

`<имя> = new <тип> [<длина>];`

- **Инициализация массива со значениями:**

`<имя> = new <тип> [] {<значение>, <значение>, <значение>, ...};`

- **Инициализация двумерного массива со значениями:**

`<имя> = new <тип> [,] {{<значение>, <значение>}, {<значение>, <значение>}, ...};`

- **Доступ к элементу с индексом i:**

`<имя> [я]`

- **Получение длины массива:**

`<имя> .Length`

замечания

В С # массив является ссылочным типом, что означает, что он имеет значение *NULL* .

Массив имеет фиксированную длину, что означает, что вы не можете его использовать `.Add()` или `.Remove()` . Чтобы использовать их, вам понадобится динамический массив - `List`

или `ArrayList`.

Examples

Ковариантность массива

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

Это преобразование не является безопасным для типов. Следующий код вызовет исключение во время выполнения:

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0]; // would have been bad if above assignment had succeeded
```

Получение и установка значений массива

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

Объявление массива

Массив может быть объявлен и заполнен значением по умолчанию с использованием синтаксиса инициализации с квадратной скобкой (`[]`). Например, создавая массив из 10 целых чисел:

```
int[] arr = new int[10];
```

Индексы в C# основаны на нуле. Индексы массива выше будут 0-9. Например:

```
int[] arr = new int[3] {7, 9, 4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

Это означает, что система начинает подсчет индекса элемента из 0. Более того, доступ к элементам массивов выполняется в **постоянное время**. Это означает, что доступ к первому элементу массива имеет одинаковую стоимость (по времени) доступа к второму

элементу, третьему элементу и так далее.

Вы также можете объявить ссылку на массив без создания экземпляра массива.

```
int[] arr = null;    // OK, declares a null reference to an array.  
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

Массив также может быть создан и инициализирован с помощью пользовательских значений с использованием синтаксиса инициализации коллекции:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

`new int[]` может быть опущена при объявлении переменной массива. Это не самодостаточное выражение, поэтому использование его как части другого вызова не работает (для этого используйте версию с `new`):

```
int[] arr = { 24, 2, 13, 47, 45 }; // OK  
int[] arr1;  
arr1 = { 24, 2, 13, 47, 45 };      // Won't compile
```

Неявно типизированные массивы

В качестве альтернативы, в сочетании с ключевым словом `var`, конкретный тип может быть опущен так, чтобы был выведен тип массива:

```
// same as int[]  
var arr = new [] { 1, 2, 3 };  
// same as string[]  
var arr = new [] { "one", "two", "three" };  
// same as double[]  
var arr = new [] { 1.0, 2.0, 3.0 };
```

Итерация по массиву

```
int[] arr = new int[] {1, 6, 3, 3, 9};  
  
for (int i = 0; i < arr.Length; i++)  
{  
    Console.WriteLine(arr[i]);  
}
```

используя `foreach`:

```
foreach (int element in arr)  
{  
    Console.WriteLine(element);  
}
```

использование небезопасного доступа с указателями <https://msdn.microsoft.com/en>

```
unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
            pInt++; // move pointer to next element
        }
    }
}
```

Выход:

```
1
6
3
3
9
```

Многомерные массивы

Массивы могут иметь более одного измерения. В следующем примере создается двумерный массив из десяти строк и десяти столбцов:

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

Массив из трех измерений:

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

Вы также можете инициализировать массив после объявления:

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

// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]); // 4
```

Жесткие массивы

Жесткие массивы - это массивы, которые вместо примитивных типов содержат массивы (или другие коллекции). Это похоже на массив массивов - каждый элемент массива содержит другой массив.

Они похожи на многомерные массивы, но имеют небольшую разницу - поскольку

многомерные массивы ограничены фиксированным числом строк и столбцов, с зубчатыми массивами, каждая строка может иметь различное количество столбцов.

Объявление зубчатого массива

Например, объявив зубчатый массив с 8 столбцами:

```
int[][] a = new int[8][];
```

Второй [] инициализируется без номера. Чтобы инициализировать вспомогательные массивы, вам нужно будет сделать это отдельно:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

Получение / установка значений

Теперь получить один из подмассивов легко. Давайте печатать все цифры 3 - го столбца :

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Получение определенного значения:

```
a[<row_number>][<column_number>]
```

Установка определенного значения:

```
a[<row_number>][<column_number>] = <value>
```

Помните : всегда рекомендуется использовать зубчатые массивы (массивы массивов), а не многомерные массивы (матрицы). Это быстрее и безопаснее.

Обратите внимание на порядок скобок

Рассмотрим трехмерный массив пятимерных массивов одномерных массивов `int` . Это написано на C # как:

```
int[,,][,,,][] arr = new int[8, 10, 12][,,,][];
```

В системе типа CLR соглашение для упорядочения скобок отменяется, поэтому с приведенным выше экземпляром `arr` мы имеем:

```
arr.GetType().ToString() == "System.Int32[][,,,][,,]"
```

и аналогичным образом:

```
typeof(int[,][,,][,]).ToString() == "System.Int32[][,,,][,,]"
```

Проверка того, содержит ли один массив другой массив

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

/// Образец

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

Инициализация массива, заполненного повторным значением, отличным

от значения по умолчанию

Как мы знаем, мы можем объявить массив со значениями по умолчанию:

```
int[] arr = new int[10];
```

Это создаст массив из 10 целых чисел с каждым элементом массива, имеющим значение 0 (значение по умолчанию для типа `int`).

Чтобы создать массив, инициализированный значением, отличным от значения по умолчанию, мы можем использовать `Enumerable.Repeat` из пространства имен `System.Linq`:

1. Чтобы создать массив `bool` размером 10, заполненный «истинным»,

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. Чтобы создать массив `int` размером 5, заполненный «100»,

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. Чтобы создать массив `string` размером 5, заполненный «C #»,

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

Копирование массивов

Копирование частичного массива со статическим `Array.Copy()`, начинающимся с индекса 0 в обоих источниках и источниках:

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, destinationArray, 3);

// destinationArray will have 11,12 and 3
```

Копирование всего массива с `CopyTo()` экземпляра экземпляра `CopyTo()`, начинающегося с индекса 0 источника и указанного индекса в получателе:

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);

// destinationArray will have 0, 0, 11, 12, 7 and 0
```

`Clone` используется для создания копии объекта массива.

```
var sourceArray = new int[] { 11, 12, 7 };
```

```
var destinationArray = (int)sourceArray.Clone();  
//destinationArray will be created and will have 11,12,17.
```

Оба `CopyTo` и `Clone` выполняют мелкую копию, что означает, что содержимое содержит ссылки на тот же объект, что и элементы в исходном массиве.

Создание массива последовательных чисел

LINQ предоставляет метод, который упрощает создание коллекции, заполненной последовательными номерами. Например, вы можете объявить массив, который содержит целые числа от 1 до 100.

Метод `Enumerable.Range` позволяет нам создавать последовательность целых чисел из заданной начальной позиции и нескольких элементов.

Метод принимает два аргумента: начальное значение и количество генерируемых элементов.

```
Enumerable.Range(int start, int count)
```

Обратите внимание, что `count` не может быть отрицательным.

Использование:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

Это создаст массив, содержащий числа от 1 до 100 (`[1, 2, 3, ..., 98, 99, 100]`).

Поскольку метод `Range` возвращает `IEnumerable<int>`, мы можем использовать другие методы LINQ на нем:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

Это создаст массив, содержащий 10 целых квадратов, начиная с `4 : [4, 9, 16, ..., 100, 121]`.

Сравнение массивов для равенства

LINQ предоставляет встроенную функцию для проверки равенства двух `IEnumerable` `S`, и эта функция может использоваться на массивах.

Функция `SequenceEqual` вернет `true` если массивы имеют одинаковую длину, а значения в соответствующих индексах равны, а `false` противном случае.

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

Это напечатает:

```
Arrays equal? True
```

Массивы как `IEnumerable` <> экземпляры

Все массивы реализуют неосновный интерфейс `IList` (и, следовательно, не общие базовые интерфейсы `ICollection` и `IEnumerable`).

Что еще более важно, одномерные массивы реализуют `IList` и `IReadOnlyList` общие интерфейсы (и их базовые интерфейсы) для типа данных, которые они содержат. Это означает, что их можно рассматривать как общие перечислимые типы и передавать в различные методы без необходимости сначала преобразовывать их в форму без массива.

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

После запуска этого кода список `listOfIntegers` будет содержать `List<int>` содержащий значения 3, 5 и 7.

Массивы средств поддержки `IEnumerable` могут быть запрошены с помощью LINQ, например `arr1.Select(i => 10 * i)`.

Прочтите Массивы онлайн: <https://riptutorial.com/ru/csharp/topic/1429/массивы>

глава 100: методы

Examples

Объявление метода

Каждый метод имеет уникальную подпись, состоящую из доступа (`public` , `private` , ...), необязательного модификатора (`abstract`), имени и при необходимости параметров метода. Обратите внимание, что тип возврата не является частью подписи. Прототип метода выглядит следующим образом:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier` может быть `public` , `protected` , `private` или по умолчанию `internal` .

`OptionalModifier` может быть `static` `abstract` `virtual` `override` `new` или `sealed` .

`ReturnType` может быть `void` без возврата или может быть любым типом от базовых, как `int` до сложных классов.

Метод может иметь некоторые или никакие входные параметры. чтобы задать параметры для метода, вы должны объявить каждый из них, как обычные объявления переменных (например, `int a`), и для более чем одного параметра вы должны использовать запятую между ними (например, `int a, int b`).

Параметры могут иметь значения по умолчанию. для этого вы должны установить значение для параметра (например, `int a = 0`). если параметр имеет значение по умолчанию, установка входного значения является необязательной.

Следующий пример метода возвращает сумму двух целых чисел:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

Вызов метода

Вызов статического метода:

```
// Single argument
System.Console.WriteLine("Hello World");
```

```
// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

Вызов статического метода и сохранение его возвращаемого значения:

```
string input = System.Console.ReadLine();
```

Вызов метода экземпляра:

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

Вызов универсального метода

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

Параметры и аргументы

Метод может объявлять любое количество параметров (в этом примере `i`, `s` и `o` являются параметрами):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

Параметры могут использоваться для передачи значений в метод, так что метод может работать с ними. Это может быть любая работа, например, печать значений или внесение изменений в объект, на который ссылается параметр, или сохранение значений.

Когда вы вызываете метод, вам нужно передать фактическое значение для каждого параметра. На данный момент значения, которые вы фактически передаете вызову метода, называются Аргументами:

```
DoSomething(x, "hello", new object());
```

Типы возврата

Метод может возвращать либо ничего (`void`), либо значение определенного типа:

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
```

```
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

Если ваш метод указывает возвращаемое значение, метод **должен** вернуть значение. Вы делаете это с помощью оператора `return`. Как только оператор `return` будет достигнут, он вернет указанное значение и любой код после того, как он больше не будет запущен (исключения - это, `finally` блоки, которые все равно будут выполняться до возвращения метода).

Если ваш метод ничего не возвращает (`void`), вы все равно можете использовать оператор `return` без значения, если хотите немедленно вернуться из метода. В конце такого метода оператор `return` был бы лишним.

Примеры действительных операторов `return`:

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Выброс исключения может прекратить выполнение метода без возврата значения. Кроме того, существуют итераторные блоки, где значения возвращаемого значения генерируются с использованием ключевого слова `yield`, но это специальные случаи, которые не будут объясняться на данном этапе.

Параметры по умолчанию

Вы можете использовать параметры по умолчанию, если хотите предоставить возможность оставить параметры:

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

Когда вы вызываете такой метод и опускаете параметр, для которого предоставляется значение по умолчанию, компилятор вставляет это значение по умолчанию для вас.

Имейте в виду, что параметры со значениями по умолчанию должны быть записаны **после** параметров без значений по умолчанию.

```
static void SaySomething(string say, string what = "ehh") {
```

```

    //Correct
    Console.WriteLine(say + what);
}

static void SaySomethingElse(string what = "ehh", string say) {
    //Incorrect
    Console.WriteLine(say + what);
}

```

ПРЕДУПРЕЖДЕНИЕ. Поскольку это работает так, значения по умолчанию могут быть проблематичными в некоторых случаях. Если вы измените значение по умолчанию параметра метода и не перекомпилируете всех вызывающих лиц этого метода, эти вызывающие абоненты будут по-прежнему использовать значение по умолчанию, которое присутствовало, когда они были скомпилированы, что может привести к несогласованности.

Перегрузка метода

Определение. Когда несколько методов с тем же именем объявляются с разными параметрами, это называется перегрузкой метода. Перегрузка метода обычно представляет собой функции, которые идентичны по своей назначению, но записываются для приема разных типов данных в качестве их параметров.

Факторы, влияющие

- Количество аргументов
- Тип аргументов
- Тип возврата **

Рассмотрим метод с именем `Area` который будет выполнять функции вычисления, которые будут принимать различные аргументы и возвращать результат.

пример

```

public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}

```

Этот метод принимает один аргумент и возвращает строку, если мы вызываем метод с целым числом (скажем, 5), выход будет "Area of Square is 25".

```

public double Area(double value1, double value2)
{
    return value1 * value2;
}

```

Аналогично, если мы передадим два двойных значения этому методу, результат будет произведением двух значений и будет иметь тип `double`. Это можно использовать для

умножения, а также для поиска области прямоугольников

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1, 2);
}
```

Его можно использовать специально для поиска области круга, которая будет принимать двойное значение (`radius`) и вернуть другое двойное значение в качестве своего Района.

Каждый из этих методов можно вызвать normally без конфликтов - компилятор будет проверять параметры каждого вызова метода, чтобы определить, какая версия `Area` должна использоваться.

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

** Обратите внимание, что тип возвращаемого значения в одиночку не может различать между двумя методами. Например, если бы у нас было два определения для Области, которые имели одинаковые параметры, например:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

Если нам нужно, чтобы наш класс использовал те же имена методов, которые возвращают разные значения, мы можем устранить проблемы двусмыслинности, реализуя интерфейс и явно определяя его использование.

```
public interface IAreaCalculatorString {
    public string Area(double width, double height);
}

public class AreaCalculator : IAreaCalculatorString {
    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
}
```

Анонимный метод

Анонимные методы предоставляют метод передачи кода в качестве параметра делегата. Это методы с телом, но не именем.

```
delegate int IntOp(int lhs, int rhs);
```

```

class Program
{
    static void Main(string[] args)
    {
        // C# 2.0 definition
        IntOp add = delegate(int lhs, int rhs)
        {
            return lhs + rhs;
        };

        // C# 3.0 definition
        IntOp mul = (lhs, rhs) =>
        {
            return lhs * rhs;
        };

        // C# 3.0 definition - shorthand
        IntOp sub = (lhs, rhs) => lhs - rhs;

        // Calling each method
        Console.WriteLine("2 + 3 = " + add(2, 3));
        Console.WriteLine("2 * 3 = " + mul(2, 3));
        Console.WriteLine("2 - 3 = " + sub(2, 3));
    }
}

```

Права доступа

```

// static: is callable on a class even when no instance of the class has been created
public static void MyMethod()

// virtual: can be called or overridden in an inherited class
public virtual void MyMethod()

// internal: access is limited within the current assembly
internal void MyMethod()

//private: access is limited only within the same class
private void MyMethod()

//public: access right from every class / assembly
public void MyMethod()

//protected: access is limited to the containing class or types derived from it
protected void MyMethod()

//protected internal: access is limited to the current assembly or types derived from the
containing class.
protected internal void MyMethod()

```

Прочтайте методы онлайн: <https://riptutorial.com/ru/csharp/topic/60/методы>

глава 101: Методы DateTime

Examples

DateTime.Add (TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.  
System.DateTime today = System.DateTime.Now;  
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);  
System.DateTime answer = today.Add(duration);  
System.Console.WriteLine("{0:dddd}", answer);
```

DateTime.AddDays (Двухместный)

Добавьте дни в объект dateTime.

```
DateTime today = DateTime.Now;  
DateTime answer = today.AddDays(36);  
Console.WriteLine("Today: {0:dddd}", today);  
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

Вы также можете вычесть дни, минуя отрицательное значение:

```
DateTime today = DateTime.Now;  
DateTime answer = today.AddDays(-3);  
Console.WriteLine("Today: {0:dddd}", today);  
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

DateTime.AddHours (Двухместный)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,  
                  29, 30, 31, 90, 365};  
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);  
  
foreach (double hour in hours)  
    Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,  
                     dateValue.AddHours(hour));
```

DateTime.AddMilliseconds (Двухместный)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";  
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);  
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",  
                 date1.ToString(dateFormat), date1.Ticks);  
  
DateTime date2 = date1.AddMilliseconds(1);  
Console.WriteLine("Second date: {0} ({1:N0} ticks)",  
                 date2.ToString(dateFormat), date2.Ticks);
```

```

Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date: {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);

```

DateTime.Compare (DateTime t1, DateTime t2)

```

DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);

```

DateTime.DaysInMonth (Int32, Int32)

```

const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);

```

DateTime.AddYears (Int32)

Добавьте годы в объект `dateTime`:

```

DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
    Console.WriteLine("{0,2} year(s) ago:{1:d}",
                     Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.

```

```
for (int ctr = 1; ctr <= 15; ctr++)
    Console.WriteLine("{0,2} year(s) from now: {1:d}",
                      ctr, baseDate.AddYears(ctr));
```

Предупреждение о чистых функциях при работе с DateTime

Википедия в настоящее время определяет чистую функцию следующим образом:

1. Функция всегда оценивает одно и то же значение результата с учетом того же значения (-ов) аргументов. Значение результата функции не может зависеть от какой-либо скрытой информации или состояния, которое может измениться во время выполнения программы или между различными исполнениями программы, а также не может зависеть от любого внешнего входа от устройств ввода / вывода.
2. Оценка результата не вызывает каких-либо семантически наблюдаемых побочных эффектов или результатов, таких как мутация изменяемых объектов или выход на устройства ввода / вывода

Как разработчик вам нужно знать о чистых методах, и вы наткнетесь на них во многих областях. Один из тех, что я видел, что укусы многих младших разработчиков работают с методами класса DateTime. Многие из них чисты, и если вы не знаете об этом, вы можете быть в восторге. Пример:

```
DateTime sample = new DateTime(2016, 12, 25);
sample.AddDays(1);
Console.WriteLine(sample.ToString());
```

В приведенном выше примере можно ожидать, что результат будет напечатан на консоли «26/12/2016», но на самом деле вы получите ту же дату. Это связано с тем, что AddDays является чистым методом и не влияет на исходную дату. Чтобы получить ожидаемый результат, вам придется изменить вызов AddDays на следующее:

```
sample = sample.AddDays(1);
```

DateTime.Parse (String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

DateTime.TryParse (String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent
// and returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
```

```

    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings) {

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}

```

Parse и TryParse с информацией о культуре

Возможно, вы захотите использовать его при анализе DateTime из [разных культур \(языки\)](#), в следующем примере проанализируйте голландскую дату.

```

DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}

```

Пример анализа:

```

DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}

```

DateTime как инициализатор в for-loop

```

// This iterates through a range between two DateTime's
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
        (dt - start).Days, dt.ToString());
}

```

Итерация на `TimeSpan` работает одинаково.

DateTime.ToString, ToShortDateString, ToLongDateString и ToString отформатированы

```
using System;

public class Program
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

        Console.WriteLine(date.ToString());           //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToString());           //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy")); //Outputs: 31/12/2016
    }
}
```

Текущая дата

Чтобы получить текущую дату, вы используете свойство `DateTime.Today`. Это возвращает объект `DateTime` с сегодняшней датой. Когда это преобразуется `.ToString()` это делается по умолчанию в вашей системе.

Например:

```
Console.WriteLine(DateTime.Today);
```

Записывает сегодняшнюю дату в вашем локальном формате на консоль.

Формирование даты

Стандартное форматирование даты

`DateTimeFormatInfo` задает набор спецификаторов для простого форматирования даты и времени. Каждый спецификатор соответствует определенному шаблону формата `DateTimeFormatInfo`.

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);

var t = String.Format("{0:t}", dt); // "6:50 PM"           ShortTime
var d = String.Format("{0:d}", dt); // "8/1/2016"          ShortDate
var T = String.Format("{0:T}", dt); // "6:50:23 PM"        LongTime
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016" LongDate
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM" FullDateTime
LongDate+ShortTime
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM" FullDateTime
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"   ShortDate+ShortTime
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
```

```

ShortDate+LongTime
var m = String.Format("{0:m}", dt); // "August 1"                                MonthDay
var y = String.Format("{0:y}", dt); // "August 2016"                             YearMonth
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"      RFC1123
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"                  SortableDateTime
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"                  UniversalSortableDateTime

```

Пользовательское форматирование даты

Существуют следующие спецификаторы специальных форматов:

- `y` (год)
- `M` (месяц)
- `d` (день)
- `h` (час 12)
- `H` (час 24)
- `m` (минута)
- `s` (второй)
- `f` (вторая фракция)
- `F` (вторая фракция, конечные нули обрезаны)
- `t` (PM или AM)
- `z` (часовой пояс).

```

var year =           String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016"   year
var month =          String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August" month
var day =            String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday"  day
var hour =           String.Format("{0:h hh H HH}", dt); // "6 06 18 18"        hour 12/24
var minute =         String.Format("{0:m mm}", dt); // "50 50"               minute
var secound =         String.Format("{0:s ss}", dt); // "23 23"                second
var fraction =       String.Format("{0:f ff ffff ffff}", dt); // "2 23 230 2300"     sec.fraction
var fraction2 =      String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23"        without zeroes
var period =         String.Format("{0:t tt}", dt); // "P PM"                A.M. or P.M.
var zone =           String.Format("{0:z zz zzz}", dt); // "+0 +00 +00:00"      time zone

```

Вы также можете использовать разделитель даты / (косая черта) и временный разделитель : (двоеточие).

Пример кода

Для получения дополнительной информации [MSDN](#) .

DateTime.ParseExact (String, String, IFormatProvider)

Преобразует указанное строковое представление даты и времени в эквивалент DateTime, используя указанный формат и информацию о формате для конкретной культуры. Формат строкового представления должен точно соответствовать указанному формату.

Преобразование определенной строки формата в эквивалентную дату DateTime

Предположим, у нас есть 08-07-2016 11:30:12 PM к культуре строки DateTime 08-07-2016 11:30:12 PM в 08-07-2016 11:30:12 PM MM-dd-yyyy hh:mm:ss tt, и мы хотим, чтобы она конвертировалась в эквивалентный объект DateTime

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt",
CultureInfo.CurrentCulture);
```

Преобразование строки времени даты в эквивалентный объект DateTime без какого-либо определенного формата культуры

Допустим, у нас есть строка DateTime в формате dd-MM-yy hh:mm:ss tt, и мы хотим, чтобы она конвертировалась в эквивалентный объект DateTime без какой-либо конкретной информации о культуре

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt",
CultureInfo.InvariantCulture);
```

Преобразование строки времени даты в эквивалентный объект DateTime без какого-либо определенного формата культуры в другом формате

Допустим, у нас есть строка Date, например «23-12-2016» или «12/23/2016», и мы хотим, чтобы она конвертировалась в эквивалентный объект DateTime без какой-либо конкретной информации о культуре

```
string date = '23-12-2016' or date = 12/23/2016';
string[] formats = new string[] {"dd-MM-yyyy", "MM/dd/yyyy"}; // even can add more possible
formats.
DateTime date = DateTime.ParseExact(date, formats,
CultureInfo.InvariantCulture, DateTimeStyles.None);
```

Примечание: `System.Globalization` должна быть добавлена для `CultureInfo` класса

`DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime)`

Преобразует указанное строковое представление даты и времени в эквивалент DateTime, используя указанный формат, информацию о формате для конкретной культуры и стиль. Формат строкового представления должен точно соответствовать указанному формату. Метод возвращает значение, указывающее, удалось ли преобразование.

Например

```
CultureInfo enUS = new CultureInfo("en-US");
string dateString;
System.DateTime dateValue;
```

Параметр даты без флагов стиля.

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Используйте пользовательские форматы с M и MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Разбор строки с информацией о часовом поясе.

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
```

```

}

// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.AdjustToUniversal, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

```

Разберите строку, представляющую UTC.

```

dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None,
out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

```

Выходы

```

' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).

```

Прочтайте Методы DateTime онлайн: <https://riptutorial.com/ru/csharp/topic/1587/методы-datetime>

глава 102: Методы расширения

Синтаксис

- public static ReturnType MyExtensionMethod (этот целевой объект TargetType)
- public static ReturnType MyExtensionMethod (этот целевой объект TargetType, TArg1 arg1, ...)

параметры

параметр	подробности
этот	Первому параметру метода расширения всегда должно предшествовать <code>this</code> ключевое слово, за которым следует идентификатор, с помощью которого можно ссылаться на «текущий» экземпляр объекта, который вы распространяете

замечания

Методы расширения - это синтаксический сахар, который позволяет статическим методам быть вызванными в экземплярах объектов, как если бы они были членами самого типа.

Для методов расширения требуется явный целевой объект. Вам нужно будет использовать `this` ключевое слово для доступа к методу из самого расширенного типа.

Методы расширений должны быть объявлены статическими и должны находиться в статическом классе.

Какое пространство имен?

Выбор пространства имен для вашего класса методов расширения - это компромисс между видимостью и возможностью обнаружения.

Наиболее часто упоминаемый [вариант](#) состоит в том, чтобы иметь собственное пространство имен для ваших методов расширения. Однако это будет связано с коммуникационными усилиями, чтобы пользователи вашего кода знали, что существуют методы расширения и где их искать.

Альтернативой является выбор пространства имен, чтобы разработчики открывали ваши методы расширения через Intellisense. Поэтому, если вы хотите расширить класс `Foo`, логично поставить методы расширения в том же пространстве имен, что и `Foo`.

Важно понимать, что **ничто не мешает вам использовать «чужое» пространство имен** : Таким образом, если вы хотите расширить `IEnumerable`, вы можете добавить свой метод расширения в пространство имен `System.Linq`.

Это не всегда хорошая идея. Например, в одном конкретном случае вам может понадобиться расширить общий тип (например, `bool IsApproxEqualTo(this double value, double other)`), но не имеет, что «загрязняет» всю `System`. В этом случае предпочтительнее выбрать локальное, конкретное пространство имен.

Наконец, также можно поместить методы расширения в **пространство имен вообще** !

Хороший справочный вопрос: [как вы управляете пространствами имен ваших методов расширения?](#)

применимость

Следует проявлять осторожность при создании методов расширения, чтобы гарантировать, что они подходят для всех возможных входных данных и не только релевантны конкретным ситуациям. Например, можно расширить системные классы, такие как `string`, что делает ваш новый код доступным для **любой** строки. Если вашему коду необходимо выполнить логику домена для определенного формата в домене, метод расширения не будет подходящим, поскольку его присутствие будет путать вызывающих абонентов, работающих с другими строками в системе.

Следующий список содержит основные функции и свойства методов расширения

1. Это должен быть статический метод.
2. Он должен находиться в статическом классе.
3. Он использует ключевое слово «`this`» в качестве первого параметра с типом в .NET, и этот метод будет вызываться экземпляром определенного типа на стороне клиента.
4. Это также показано VS intellisense. Когда мы нажимаем точку . после экземпляра типа, то он приходит в VS intellisense.
5. Метод расширения должен находиться в том же пространстве имен, в котором он используется, или вам нужно импортировать пространство имен класса с помощью инструкции `using`.
6. Вы можете указать любое имя для класса с методом расширения, но класс должен быть статическим.
7. Если вы хотите добавить новые методы к типу, и у вас нет исходного кода для него, тогда решение должно использовать и применять методы расширения этого типа.
8. Если вы создаете методы расширения, которые имеют те же методы подписи, что и тип, который вы распространяете, методы расширения никогда не будут вызываться.

Examples

Методы расширения - обзор

Методы расширения были введены в C # 3.0. Методы расширения расширяют и добавляют поведение к существующим типам без создания нового производного типа, перекомпиляции или иного изменения исходного типа. *Они особенно полезны, когда вы не можете изменить источник типа, который вы хотите улучшить.* Методы расширения могут быть созданы для типов систем, типов, определенных третьими лицами, и типов, которые вы определили сами. Метод расширения может быть вызван, как если бы это был метод-член исходного типа. Это позволяет использовать **метод Chaining** для реализации **Fluent Interface**.

Метод расширения создается путем добавления **статического метода к статическому классу**, который отличается от исходного типа. Статический класс, содержащий метод расширения, часто создается с единственной целью хранения методов расширения.

В методах расширения используется специальный первый параметр, который указывает, что исходный тип будет расширен. Этот первый параметр увенчан ключевым словом `this` (что представляет собой особое и четкое использование `this` в C # - это следует понимать как отличающееся от использования `this` которое позволяет ссылаться на членов экземпляра текущего объекта).

В следующем примере расширением исходного типа является `string` класса. `String` была расширена с помощью метода `Shorten()`, который обеспечивает дополнительную функциональность сокращения. Статический класс `StringExtensions` был создан для хранения метода расширения. Метод расширения `Shorten()` показывает, что это расширение `string` через специально выделенный первый параметр. Чтобы показать, что метод `Shorten()` расширяет `string`, первый параметр отмечен `this`. Таким образом, полная подпись первого параметра - `this string text`, где `string` является расширением исходного типа, а `text` - выбранным именем параметра.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
    }
}
```

```
    // and the assignment below is functionally equivalent
    var newString2 = StringExtensions.Shorten(myString, 5);
}
}
```

Живая демонстрация на .NET скрипке

Объектом, переданным в качестве *первого аргумента метода расширения* (который сопровождается `this` ключевым словом), является экземпляр, на который вызывается метод расширения.

Например, когда этот код выполняется:

```
"some string".Shorten(5);
```

Значения аргументов таковы:

```
text: "some string"
length: 5
```

Обратите внимание, что методы расширения доступны только в том случае, если они находятся в том же пространстве имен, что и их определение, если пространство имен явно импортировано кодом с использованием метода расширения или если класс расширения меньше пространства имен. В директивах .NET Framework рекомендуется помещать классы расширения в собственное пространство имен. Однако это может привести к обнаружению проблем.

Это не приводит к конфликтам между методами расширения и используемыми библиотеками, если явно не затронуты пространства имен, которые могут конфликтовать. Например, [расширения LINQ](#) :

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace

class Program
{
    static void Main()
    {
        var ints = new int[] {1, 2, 3, 4};

        // Call Where() extension method from the System.Linq namespace
        var even = ints.Where(x => x % 2 == 0);
    }
}
```

Живая демонстрация на .NET скрипке

Начиная с C # 6.0, также можно поместить `using static` директиву в класс, содержащий методы расширения. Например, `using static System.Linq.Enumerable;`, Это делает методы

расширения из этого конкретного класса доступными без привлечения других типов из одного и того же пространства имен в область видимости.

Когда метод класса с той же сигнатурой доступен, компилятор приостанавливает его по вызову метода расширения. Например:

```
class Test
{
    public void Hello()
    {
        Console.WriteLine("From Test");
    }
}

static class TestExtensions
{
    public static void Hello(this Test test)
    {
        Console.WriteLine("From extension method");
    }
}

class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}
```

Живая демонстрация на .NET Fiddle

Обратите внимание: если есть две функции расширения с одной и той же сигнатурой, и одна из них находится в одном и том же пространстве имен, то это будет приоритетным. С другой стороны, если оба из них будут доступны с `using`, то с сообщением будет записана ошибка времени компиляции:

Вызов неоднозначен между следующими способами или свойствами

Обратите внимание, что синтаксическое удобство вызова метода расширения через `originalTypeInstance.ExtensionMethod()` является дополнительным удобством. Этот метод также можно вызывать традиционным способом, так что в качестве параметра к методу используется специальный первый параметр.

То есть, обе следующие работы:

```
//Calling as though method belongs to string--it seamlessly extends string
String s = "Hello World";
s.Shorten(5);
```

```
//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);
```

Явно использую метод расширения

Методы расширения можно также использовать как обычные методы статического класса. Этот способ вызова метода расширения более подробен, но необходим в некоторых случаях.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

Использование:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

Когда вызывать методы расширения как статические методы

Есть еще сценарии, где вам нужно использовать метод расширения как статический метод:

- Разрешение конфликта с методом участника. Это может произойти, если новая версия библиотеки представляет новый метод-член с той же сигнатурой. В этом случае метод-член будет предпочтительнее компилятором.
- Разрешение конфликтов с другим методом расширения с одной и той же сигнатурой. Это может произойти, если две библиотеки включают аналогичные методы расширения, а пространства имен обоих классов с методами расширения используются в одном файле.
- Передача метода расширения в качестве группы методов в параметр делегата.
- Выполнение собственной привязки через `Reflection`.
- Использование метода расширения в окне `Immediate` в `Visual Studio`.

Использование статических

Если `using static` директиву для переноса статических членов статического класса в глобальную область, методы расширения пропускаются. Пример:

```

using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);

```

Если вы удалите `this` модификатор из первого аргумента метода `Shorten`, последняя строка будет скомпилирована.

Проверка нулей

Методы расширения - это статические методы, которые ведут себя как методы экземпляра. Однако, в отличие от того, что происходит при вызове метода экземпляра на `null` ссылке, когда метод расширения вызывается с `null` ссылкой, он не бросает `NullReferenceException`. Это может быть весьма полезно в некоторых сценариях.

Например, рассмотрим следующий статический класс:

```

public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}

string nullString = null;
string emptyString = nullString.EmptyIfNull(); // will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null

```

Живая демонстрация на .NET скрипке

Методы расширения могут видеть только общедоступные (или внутренние) члены расширенного класса

```

public class SomeClass
{
    public void DoStuff()
    {

    }

    protected void DoMagic()
    {

```

```

    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
}

```

Методы расширения - это просто синтаксический сахар и на самом деле не являются членами класса, который они расширяют. Это означает, что они не могут разрушить инкапсуляцию - они имеют доступ только к `public` (или реализованным в тех же сборках, `internal`) полям, свойствам и методам.

Общие методы расширения

Как и другие методы, методы расширения могут использовать дженерики. Например:

```

static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}

```

и называть это было бы так:

```

IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();

```

Посмотреть демо

Аналогично для нескольких аргументов типа:

```

public static TU GenericExt<T, TU>(this T obj)
{
    TU ret = default(TU);
    // do some stuff with obj
    return ret;
}

```

Вызвать это было бы так:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};  
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

[Посмотреть демо](#)

Вы также можете создавать методы расширения для частично связанных типов в нескольких общих типах:

```
class MyType<T1, T2>  
{  
}  
  
static class Extensions  
{  
    public static void Example<T>(this MyType<int, T> test)  
    {  
    }  
}
```

Вызвать это было бы так:

```
MyType<int, string> t = new MyType<int, string>();  
t.Example();
```

[Посмотреть демо](#)

Кроме того, можно указать ограничения типа с `where`:

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>  
{  
    return EqualityComparer<T>.Default.Equals(obj, default(T));  
}
```

Код звонка:

```
int number = 5;  
var IsDefault = number.IsDefault();
```

[Посмотреть демо](#)

Распространение методов расширения на основе статического типа

Для соответствия параметрам используется статический (время компиляции), а не динамический (тип времени выполнения).

```
public class Base  
{  
    public virtual string GetName()  
    {  
        return "Base";  
    }  
}
```

```

}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)
    {
        return "Derived";
    }
}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;

        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"

        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}

```

Живая демонстрация на .NET скрипке

Также отправка на основе статического типа не позволяет вызывать метод расширения для `dynamic` объекта:

```

public class Person
{
    public string Name { get; set; }
}

public static class ExtensionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };

```

```
var name = person.GetPersonName(); // RuntimeBinderException is thrown
```

Методы расширения не поддерживаются динамическим кодом.

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsThisAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsThisAwesome(dynamicObject.StringValue));

        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsThisAwesome());
    }
}

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}
```

Причина, по которой [вызов методов расширения из динамического кода] не работает, заключается в том, что в обычных, нединамических методах расширения кода работают путем полного поиска всех классов, известных компилятору для статического класса, который имеет метод расширения, который соответствует . Поиск выполняется в порядке, основанном на вложенности пространства имен и доступном `using` директив в каждом пространстве имен.

Это означает, что для того, чтобы вызов динамического метода расширения был разрешен правильно, DLL должна знать *во время выполнения*, что все вложенные пространства имен и `using` директивы были *в вашем исходном коде* . У нас нет механизма для кодирования всей этой информации на сайте вызова. Мы считали, что изобретаем такой механизм, но решили, что он слишком дорогостоящий, и он слишком много рискует, чтобы его стоило.

[Источник](#)

Методы расширения как строго типизированные обертки

Методы расширения можно использовать для написания сильно типизированных оболочек для словарных объектов. Например, кеш, `HttpContext.Items` в `cetera` ...

```
public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}
```

Этот подход устраняет необходимость использования строковых литералов в качестве ключей по всей кодовой базе, а также необходимость литья требуемого типа во время операции чтения. В целом он создает более безопасный, строго типизированный способ взаимодействия с такими свободно типизированными объектами, как словари.

Методы расширения для цепочки

Когда метод расширения возвращает значение, которое имеет тот же тип, как `this` аргумент, он может быть использован для «цепи» один или более вызовов метода совместимой подписи. Это может быть полезно для запечатанных и / или примитивных типов и позволяет создавать так называемые «свободные» API, если имена методов читаются как естественный человеческий язык.

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

Или как это

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6 };
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}
```

```

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}

```

Методы расширения в сочетании с интерфейсами

Очень удобно использовать методы расширения с интерфейсами, поскольку реализация может храниться вне класса, и все, что требуется для добавления некоторой функциональности в класс, - это украсить класс интерфейсом.

```

public interface IIInterface
{
    string Do();
}

public static class ExtensionMethods{
    public static string DoWith(this IIInterface obj){
        //does something with IIInterface instance
    }
}

public class Classy : IIInterface
{
    // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
    // provided you import the namespace containing the extension method
    public Do(){
        return this.DoWith();
    }
}

```

используйте как:

```

var classy = new Classy();
classy.Do(); // will call the extension
classy.DoWith(); // Classy implements IIInterface so it can also be called this way

```

IList Пример метода расширения: сравнение 2 списков

Вы можете использовать следующий метод расширения для сравнения содержимого двух экземпляров `IList <T>` того же типа.

По умолчанию элементы сравниваются по их порядку в списке и сами элементы, передавая `false` параметру `isOrdered` будут сравнивать только сами элементы, независимо от их

порядка.

Чтобы этот метод работал, общий тип (`T`) должен переопределять методы `Equals` и `GetHashCode` .

Использование:

```
List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2); //this gives false
list1.Compare(list2, false); //this gives true. they are equal when the order is disregarded
```

Метод:

```
public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                (l1 == null && l2 != null) ||
                (l1 != null && l2 == null) ||
                (!l1.Equals(l2)))
            {
                return false;
            }
        }
        return true;
    }
    else
    {
        List<T> list2Copy = new List<T>(list2);
        //Can be done with Dictionary without O(n^2)
        for (int i = 0; i < list1.Count; i++)
        {
            if (!list2Copy.Remove(list1[i]))
                return false;
        }
        return true;
    }
}
```

Методы расширения с перечислением

Методы расширения полезны для добавления функциональности в перечисления.

Одним из распространенных способов использования является метод преобразования.

```

public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }

    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}

```

Теперь вы можете быстро преобразовать значение enum в другой тип. В этом случае bool.

```

bool yesNoBool = YesNo.Yes.ToBoolean(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();    // yesNoEnum == YesNo.No

```

Альтернативно, методы расширения могут использоваться для добавления свойств, подобных методам.

```

public enum Element
{
    Hydrogen,
    Helium,
    Lithium,
    Beryllium,
    Boron,
    Carbon,
    Nitrogen,
    Oxygen
    //Etc
}

public static class ElementExtensions
{
    public static double AtomicMass(this Element element)
    {
        switch(element)
        {
            case Element.Hydrogen: return 1.00794;
            case Element.Helium:   return 4.002602;
            case Element.Lithium:  return 6.941;
            case Element.Beryllium: return 9.012182;
            case Element.Boron:    return 10.811;
            case Element.Carbon:   return 12.0107;
            case Element.Nitrogen: return 14.0067;
            case Element.Oxygen:   return 15.9994;
            //Etc
        }
        return double.NaN;
    }
}

```

```
var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();
```

Расширения и интерфейсы вместе позволяют DRY-код и смешанную функциональность

Методы расширения позволяют упростить определения вашего интерфейса, включая только основные необходимые функции в самом интерфейсе и позволяющие определять методы удобства и перегрузки как методы расширения. Интерфейсы с меньшим количеством методов легче реализовать в новых классах. Сохранение перегрузок в виде расширений, а не включение их в интерфейс напрямую избавляет вас от копирования кода шаблона в каждую реализацию, помогая вам сохранить код DRY. На самом деле это похоже на шаблон mixin, который C# не поддерживает.

Расширение `System.Linq.Enumerable` для `IEnumerable<T>` - отличный пример этого. `IEnumerable<T>` требует, чтобы реализующий класс реализовал два метода: общий и не общий `GetEnumerator()`. Но `System.Linq.Enumerable` предоставляет множество полезных утилит в качестве расширений, обеспечивающих краткое и четкое потребление `IEnumerable<T>`.

Ниже приведен очень простой интерфейс с удобными перегрузками, предоставляемыми в качестве расширений.

```
public interface ITimeFormatter
{
    string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
        => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// Implementations only need to provide one method. Very easy to
// write additional implementations.
public class SecondsTimeFormatter : ITimeFormatter
{
    public string Format(TimeSpan span)
    {
        return $"{(int)span.TotalSeconds}s";
    }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is roughly {formatter.Format(4500)}");
    }
}
```

```

        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}

```

Методы расширения для обработки особых случаев

Методы расширения могут использоваться для «скрытия» обработки неэлегантных бизнес-правил, которые в противном случае потребовали бы загромождения вызывающей функции с помощью операторов if / then. Это аналогично и аналогично обработке нулей с помощью методов расширения. Например,

```

public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty
        but are known never to be lies. If the cake isn't a lie, don't do anything and return it.
        return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() :
        cake;
    }
}

```

```

Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake); //Eat the cake, confident that it is not a lie.

```

Использование методов расширения со статическими методами и обратными вызовами

Рассмотрите возможность использования методов расширения как функций, которые оберывают другой код, вот отличный пример, который использует как метод статического метода, так и метод расширения, чтобы обернуть конструкцию Try Catch. Сделайте свой код Bullet Proof ...

```

using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {

```

```

try
{
    code();
}
catch (Exception iox)
{
    //extension method used here
    iox.Log("BP2200-ERR-Unexpected Error");
    //callback, exception already handled and logged
    error(iox);
}
}
/// <summary>
/// Example of a logging method helper, this is the extension method
/// </summary>
/// <param name="error">The Exception to log</param>
/// <param name="messageID">A unique error ID header</param>
public static void Log(this Exception error, string messageID)
{
    Trace.WriteLine(messageID);
    Trace.WriteLine(error.Message);
    Trace.WriteLine(error.StackTrace);
    Trace.WriteLine("");
}
}
/// <summary>
/// Shows how to use both the wrapper and extension methods.
/// </summary>
public class UseBulletProofing
{
    public UseBulletProofing()
    {
        var ok = false;
        var result = DoSomething();
        if (!result.Contains("ERR"))
        {
            ok = true;
            DoSomethingElse();
        }
    }
}

/// <summary>
/// How to use Bullet Proofing in your code.
/// </summary>
/// <returns>A string</returns>
public string DoSomething()
{
    string result = string.Empty;
    //Note that the Bullet.Proof method forces this construct.
    Bullet.Proof(() =>
    {
        //this is the code callback
        result = "DST5900-INF-No Exceptions in this code";
    }, error =>
    {
        //error is the already logged and handled exception
        //determine the base result
        result = "DTS6200-ERR-An exception happened look at console log";
        if (error.Message.Contains("SomeMarker"))
        {
            //filter the result for Something within the exception message
        }
    });
}

```

```
        result = "DST6500-ERR-Some marker was found in the exception";
    }
})
return result;
}

/// <summary>
/// Next step in workflow
/// </summary>
public void DoSomethingElse()
{
    //Only called if no exception was thrown before
}
}
```

Методы расширения на интерфейсах

Одной из полезных функций методов расширения является то, что вы можете создавать общие методы для интерфейса. Обычно интерфейс не может иметь общие реализации, но с помощью методов расширения, которые они могут.

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}
```

В этом примере метод `FeetDriven` может использоваться на любом `IVehicle`. Эта логика в этом методе применима ко всем `IVehicle`, так что это можно сделать так, чтобы не было `FeetDriven` в определении `IVehicle` которое будет реализовано одинаково для всех детей.

Использование методов расширения для создания красивых классов сопоставления

Мы можем создать лучшие классы карт с методами расширения. Предположим, что у меня есть некоторые классы DTO, такие как

```
public class UserDTO
{
    public AddressDTO Address { get; set; }
}

public class AddressDTO
{
    public string Name { get; set; }
}
```

```
}
```

И мне нужно сопоставить соответствующие классы модели представления

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

public class AddressViewModel
{
    public string Name { get; set; }
}
```

то я могу создать свой класс mapper, как показано ниже

```
public static class ViewModelMapper
{
    public static UserViewModel ToViewModel(this UserDTO user)
    {
        return user == null ?
            null :
            new UserViewModel()
            {
                Address = user.Address.ToViewModel()
                // Job = user.Job.ToViewModel(),
                // Contact = user.Contact.ToViewModel() .. and so on
            };
    }

    public static AddressViewModel ToViewModel(this AddressDTO userAddr)
    {
        return userAddr == null ?
            null :
            new AddressViewModel()
            {
                Name = userAddr.Name
            };
    }
}
```

Затем, наконец, я могу вызвать свой картограф, как показано ниже.

```
UserDTO userDTOObj = new UserDTO() {
    Address = new AddressDTO() {
        Name = "Address of the user"
    }
};

UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

Красота здесь - все методы сопоставления имеют общее имя (ToViewModel), и мы можем использовать его несколькими способами

Использование методов расширения для создания новых типов

коллекций (например, `DictList`)

Вы можете создавать методы расширения для улучшения удобства использования для вложенных коллекций, таких как `Dictionary` со значением `List<T>`.

Рассмотрим следующие методы расширения:

```
public static class DictListExtensions
{
    public static void Add<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
        where TCollection : ICollection<TValue>, new()
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            list = new TCollection();
            dict.Add(key, list);
        }

        list.Add(value);
    }

    public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection>
dict, TKey key, TValue value)
        where TCollection : ICollection<TValue>
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            return false;
        }

        var ret = list.Remove(value);
        if (list.Count == 0)
        {
            dict.Remove(key);
        }
        return ret;
    }
}
```

Вы можете использовать методы расширения следующим образом:

```
var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15
```

```
dictList.Remove("example", 15);  
Console.WriteLine(dictList.ContainsKey("example")); // False
```

[Посмотреть демо](#)

Прочтайте Методы расширения онлайн: <https://riptutorial.com/ru/csharp/topic/20/методы-расширения>

глава 103: Многопоточность

замечания

Поток является частью программы, которая может выполняться независимо от других частей. Он может выполнять задачи одновременно с другими потоками. **Многопоточность** - это функция, которая позволяет программам выполнять параллельную обработку, чтобы одновременно выполнять несколько операций.

Например, вы можете использовать потоки для обновления таймера или счетчика в фоновом режиме, одновременно выполняя другие задачи на переднем плане.

Многопоточные приложения более восприимчивы к пользовательскому вводу, а также легко масштабируются, поскольку разработчик может добавлять потоки по мере увеличения рабочей нагрузки.

По умолчанию программа C # имеет один поток - основной поток программы. Тем не менее, вторичные потоки могут быть созданы и использованы для выполнения кода параллельно с основным потоком. Такие потоки называются рабочими потоками.

Для управления работой потока CLR делегирует функцию в операционную систему, известную как Thread Scheduler. Планировщик потоков гарантирует, что всем потокам будет назначено надлежащее время выполнения. Он также проверяет, что потоки, которые заблокированы или заблокированы, не потребляют большую часть времени процессора.

Пространство имен .NET Framework `System.Threading` упрощает использование потоков. `System.Threading` позволяет многопоточность, предоставляя несколько классов и интерфейсов. Помимо предоставления типов и классов для определенного потока, он также определяет типы для хранения коллекции потоков, класса таймера и т. д. Он также обеспечивает поддержку, позволяя синхронизировать доступ к общим данным.

`Thread` является основным классом в пространстве имен `System.Threading`. Другие классы включают `AutoResetEvent`, `Interlocked`, `Monitor`, `Mutex` и `ThreadPool`.

Некоторые из делегатов, присутствующих в пространстве имен `System.Threading` включают `ThreadStart`, `TimerCallback` и `WaitCallback`.

Перечисления в пространстве имен `System.Threading` включают `ThreadPriority`, `ThreadState` и `EventResetMode`.

В .NET Framework 4 и более поздних версиях многопоточное программирование упрощается и упрощается с помощью классов `System.Threading.Tasks.Parallel` и `System.Threading.Tasks.Task`, Parallel LINQ (PLINQ), новых классов параллельной коллекции в

Examples

Простая полная версия Threading Demo

```
class Program
{
    static void Main(string[] args)
    {
        // Create 2 thread objects.  We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Простая полная демонстрация потоков с использованием задач

```
class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });
    }

    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

```

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Task: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}

```

Явный целевой паралич

```

private static void explicitTaskParallism()
{
    Thread.CurrentThread.Name = "Main";

    // Create a task and supply a user delegate by using a lambda expression.
    Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
    Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

    // Start the task.
    taskA.Start();
    taskB.Start();

    // Output a message from the calling thread.
    Console.WriteLine("Hello from thread '{0}'.",
                      Thread.CurrentThread.Name);
    taskA.Wait();
    taskB.Wait();
    Console.Read();
}

```

Неявный параллелизм задач

```

private static void Main(string[] args)
{
    var a = new A();
    var b = new B();
    //implicit task parallelism
    Parallel.Invoke(
        () => a.DoSomeWork(),
        () => b.DoSomeOtherWork()
    );
}

```

Создание и запуск второй темы

Если вы выполняете несколько длинных вычислений, вы можете запускать их

одновременно в разных потоках на вашем компьютере. Для этого мы создаем новый **поток** и указываем на другой метод.

```
using System.Threading;

class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}
```

Запуск потока с параметрами

используя `System.Threading`:

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

Создание одной нити на процессор

`Environment.ProcessorCount` Возвращает количество **логических** процессоров на текущем компьютере.

Затем CLR будет планировать каждый поток на логическом процессоре, это теоретически может означать каждый поток на другом логическом процессоре, все потоки на одном логическом процессоре или какую-то другую комбинацию.

```
using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }
    }

    static void Secondary(object threadNumber) {
```

```
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}
```

Избегание одновременного чтения и записи данных

Иногда вы хотите, чтобы ваши потоки одновременно обменивались данными. Когда это происходит, важно знать код и блокировать любые части, которые могут пойти не так. Ниже приведен простой пример подсчета двух потоков.

Вот какой-то опасный (неправильный) код:

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of count to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

Вы заметите, вместо подсчета 1,2,3,4,5 ... мы считаем 1,1,2,2,3 ...

Чтобы устранить эту проблему, нам нужно **заблокировать** значение count, чтобы несколько разных потоков не могли читать и писать на него одновременно. С добавлением блокировки и ключа мы можем предотвратить потоки доступа к данным одновременно.

```
using System.Threading;

class MainClass
{
```

```

static int count { get; set; }
static readonly object key = new object();

static void Main()
{
    for (int i = 1; i <= 2; i++)
    {
        var thread = new Thread(ThreadMethod);
        thread.Start(i);
        Thread.Sleep(500);
    }
}

static void ThreadMethod(object threadNumber)
{
    while (true)
    {
        lock (key)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of count to:" + count);
        }
        Thread.Sleep(1000);
    }
}
}

```

Параллельный цикл foreach

Если у вас есть цикл foreach, который вы хотите ускорить, и вы не возражаете против того, в каком порядке находится выход, вы можете преобразовать его в параллельный цикл foreach, выполнив следующие действия:

```

using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }
}

```

```

private static int longCalculation(int number) {
    Thread.Sleep(1000); // Sleep to simulate a long calculation
    return number * number;
}
}

```

Тупики (два потока, ожидающие друг друга)

Тупик - это то, что происходит, когда два или более потока ожидают, что каждый может завершить или выпустить ресурс таким образом, что они будут ждать навсегда.

Типичный сценарий двух потоков, ожидающих завершения каждого из них, - это когда поток графического интерфейса Windows Forms ожидает рабочий поток, а рабочий поток пытается вызвать объект, управляемый потоком графического интерфейса пользователя. Обратите внимание, что с помощью этого кода, нажатие кнопки1 приведет к зависанию программы.

```

private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}

```

`workerthread.Join()` - это вызов, который блокирует вызывающий поток до тех пор, пока `workthread` не завершится. `textBox1.Invoke(invoker_delegate)` - это вызов, который блокирует вызывающий поток до тех пор, пока поток GUI не обработает `invoker_delegate`, но этот вызов вызывает взаимоблокировки, если поток GUI уже ожидает завершения вызова.

Чтобы обойти это, можно использовать неблокирующий способ вызова текстового поля:

```

private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}

```

Однако это вызовет проблемы, если вам нужно запустить код, зависящий от первого обновляемого текстового поля. В этом случае запустите это как часть вызова, но имейте в виду, что это запустит его в потоке графического интерфейса.

```
private void dowork()
```

```

{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    }));
    // Do work that is not dependent on textBox1 being updated first
}

```

В качестве альтернативы, запустите весь новый поток, и пусть это сделает ожидание в потоке графического интерфейса, так что workthread может завершиться.

```

private void dowork()
{
    // Do work
    Thread workerthread2 = new Thread(() =>
    {
        textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    });
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}

```

Чтобы свести к минимуму риск захода в тупик взаимного ожидания, всегда избегайте циркулярных ссылок между нитями, когда это возможно. Иерархия потоков, в которых потоки нижнего уровня оставляют сообщения только для высокопоставленных потоков и никогда не ждут от них, не будут сталкиваться с такой проблемой. Однако он все равно будет уязвим для блокировок, основанных на блокировке ресурсов.

Тупики (удерживайте ресурс и подождите)

Тупик - это то, что происходит, когда два или более потока ожидают, что каждый может завершить или выпустить ресурс таким образом, что они будут ждать навсегда.

Если thread1 удерживает блокировку на ресурсе A и ожидает, пока ресурс B будет выпущен, а thread2 будет содержать ресурс B и ожидает освобождения ресурса A, они блокируются.

Нажатие кнопки1 для следующего примера кода приведет к тому, что ваше приложение войдет в вышеупомянутое тупиковое состояние и повесит

```

private void button_Click(object sender, EventArgs e)
{
    DeadlockWorkers workers = new DeadlockWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class DeadlockWorkers

```

```

{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            Thread.Sleep(100);
            lock (resourceB)
            {
                output += "T1#";
            }
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}

```

Чтобы избежать блокировки таким образом, можно использовать `Monitor.TryEnter` (`lock_object, timeout_in_milliseconds`), чтобы проверить, не заблокирован ли замок на объекте. Если `Monitor.TryEnter` не удалось получить блокировку `lock_object` до `timeout_in_milliseconds`, он возвращает `false`, давая потоку возможность освободить другие удерживаемые ресурсы и уступить, тем самым предоставив другим потокам возможность завершить, как в этой слегка модифицированной версии выше :

```
private void button_Click(object sender, EventArgs e)
```

```

{
    MonitorWorkers workers = new MonitorWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        bool mustDoWork = true;
        Thread.Sleep(100);
        while (mustDoWork)
        {
            lock (resourceA)
            {
                Thread.Sleep(100);
                if (Monitor.TryEnter(resourceB, 0))
                {
                    output += "T1#";
                    mustDoWork = false;
                    Monitor.Exit(resourceB);
                }
            }
            if (mustDoWork) Thread.Yield();
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}

```

```
    }  
}
```

Обратите внимание, что это обходное решение полагается на то, что `thread2` упрям в отношении его блокировок и `thread1`, которые готовы дать, так что `thread2` всегда имеет приоритет. Также обратите внимание, что `thread1` должен повторить работу, которую он выполнил после блокировки ресурса A, когда он дает. Поэтому будьте осторожны при реализации этого подхода с более чем одним выходным потоком, так как тогда вы рискуете попасть в так называемый `livelock` - состояние, которое произойдет, если два потока продолжат делать первый бит их работы, а затем взаимно , начиная многократно.

Прочтайте Многопоточность онлайн: [https://riptutorial.com/ru/csharp/topic/51/
многопоточность](https://riptutorial.com/ru/csharp/topic/51/mногопоточность)

глава 104: Модификаторы доступа

замечания

Если модификатор доступа опущен,

- классы по умолчанию являются `internal`
- методы по default `private`
- геттеры и сеттеры наследуют модификатор свойства, по умолчанию это `private`

Модификаторы доступа на сеттерах или получателях свойств могут ограничивать доступ, а не расширять его: `public string someProperty {get; private set;}`

Examples

общественности

Ключевое слово `public` делает класс (включая вложенные классы), свойство, метод или поле доступным для каждого потребителя:

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

частный

`private` Ключевое слово отмечает свойства, методы, поля и вложенные классы для использования внутри класса:

```
public class Foo()
{
```

```

private string someProperty { get; set; }

private class Baz
{
    public string Value { get; set; }
}

public void Do()
{
    var baz = new Baz { Value = 42 };
}
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}

```

внутренний

Внутреннее ключевое слово делает класс (включая вложенные классы), свойство, метод или поле доступным для каждого потребителя в той же сборке:

```

internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}

```

Это может быть нарушено, чтобы тестовая сборка могла получить доступ к коду посредством добавления кода в файл AssemblyInfo.cs:

```

using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]

```

защищенный

`protected` Ключевое слово отмечает поле, свойства методов и вложенные классы для использования только внутри одного класса и только с производными классами:

```
public class Foo()
{
    protected void SomeFooMethod()
    {
        //do something
    }

    protected class Thing
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar() : Foo
{
    private void someBarMethod()
    {
        SomeFooMethod(); // inside derived class
        var thing = new Thing(); // can use nested class
    }
}

public class Baz()
{
    private void someBazMethod()
    {
        var foo = new Foo();
        foo.SomeFooMethod(); //not accessible due to protected modifier
    }
}
```

защищенный внутренний

`protected internal` Ключевое слово отмечает поле, методы, свойства и вложенные классы для использования внутри одного и того же сборного или производного классов в другой сборке:

Ассамблея 1

```
public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}
```

```

}

public class Bar
{
    void MyMethod1()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}

```

Ассамблея 2

```

public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

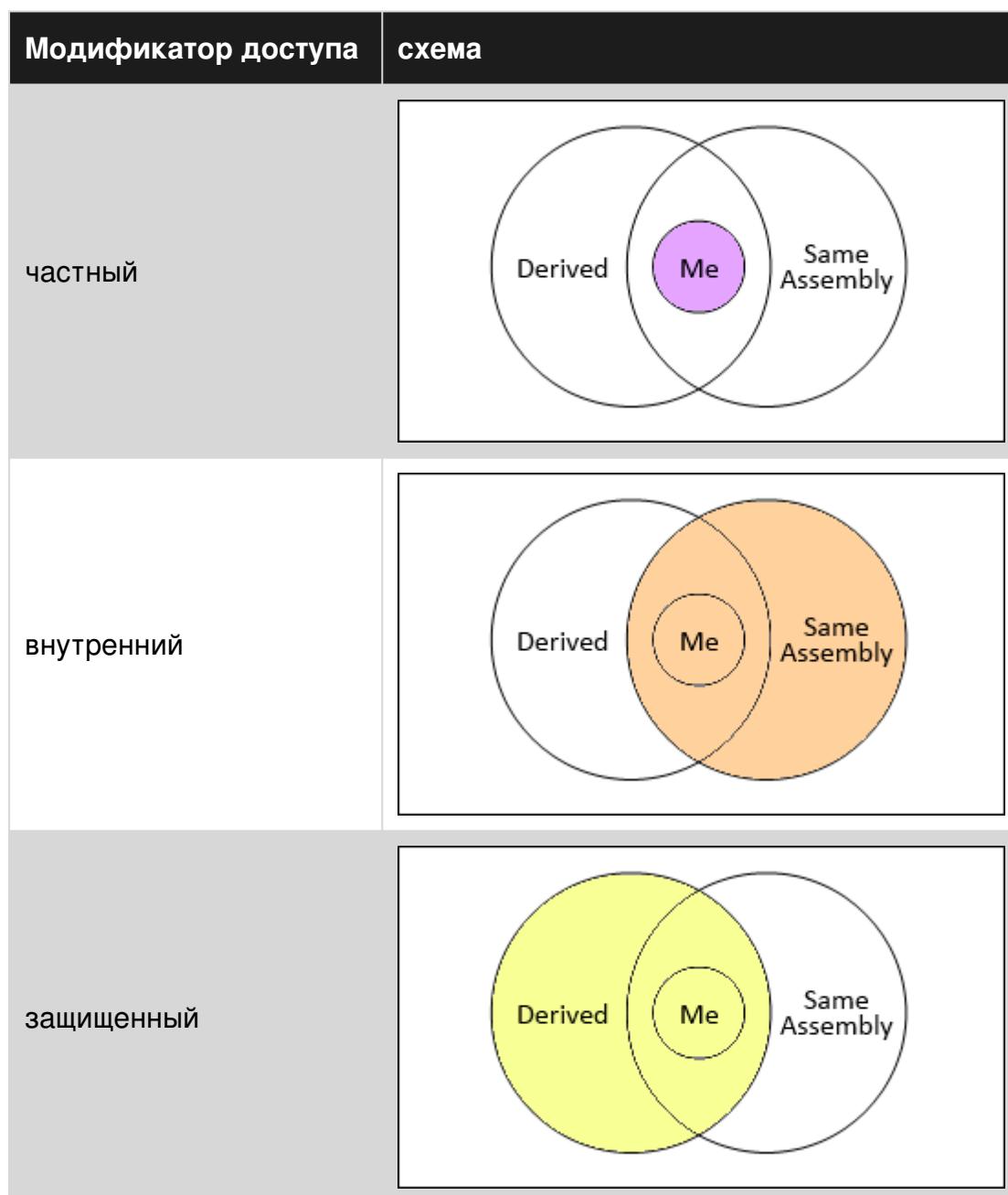
        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
    }
}

```

```
// Compile Error
var myProtectedInternalNestedInstance =
    new Foo.MyProtectedInternalNestedClass();
}
```

Диаграммы доступа Модификаторы

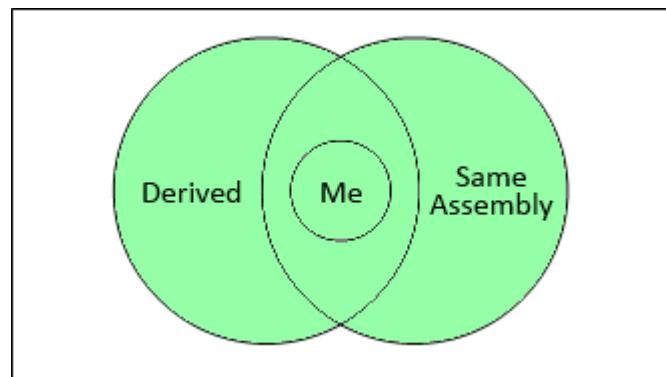
Вот все модификаторы доступа в диаграммах venn, от более ограничивающих доступ к более доступным:



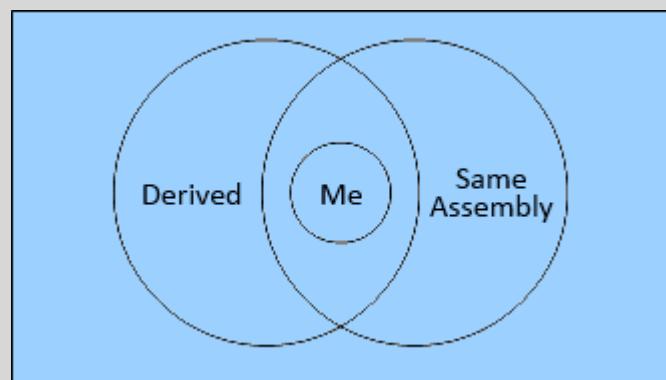
Модификатор доступа

схема

защищенный внутренний



общественности



Ниже вы можете найти дополнительную информацию.

Прочтайте Модификаторы доступа онлайн: <https://riptutorial.com/ru/csharp/topic/960/модификаторы-доступа>

глава 105: наследование

Синтаксис

- класс DerivedClass: BaseClass
- класс DerivedClass: BaseClass, IExampleInterface
- класс DerivedClass: BaseClass, IExampleInterface, IAnotherInterface

замечания

Классы могут наследовать непосредственно только из одного класса, но (вместо этого или в одно и то же время) могут реализовывать один или несколько интерфейсов.

Структуры могут реализовывать интерфейсы, но не могут явно наследовать от любого типа. Они неявно наследуют от `System.ValueType`, который, в свою очередь, наследуется непосредственно из `System.Object`.

Статические классы **не могут** реализовывать интерфейсы.

Examples

Наследование от базового класса

Чтобы избежать дублирования кода, определите общие методы и атрибуты в общем классе в качестве базы:

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

Теперь, когда у вас есть класс, который представляет `Animal` вообще, вы можете определить класс, который описывает особенности конкретных животных:

```

public class Cat : Animal
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}

```

Класс Cat получает доступ не только к методам, описанным в его определении, но также ко всем методам, определенным в общем базовом классе `Animal`. Любое животное (независимо от того, было ли это кошкой) могло съесть, посмотреть или бросить. Однако Животное не могло бы царапать, если только это не было кошкой. Затем вы можете определить другие классы, описывающие других животных. (Например, Гофер с методом уничтожения цветников и ленивца без каких-либо дополнительных методов).

Наследование от класса и реализация интерфейса

```

public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}

```

Наследование от класса и реализация нескольких интерфейсов

```

public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{

```

```

        bool HasHair { get; set; }
    }

    public interface INoiseMaker
    {
        string MakeNoise();
    }

    //Note that in C#, the base class name must come before the interface names
    public class Cat : LivingBeing, IAnimal, INoiseMaker
    {
        public Cat()
        {
            Name = "Cat";
            HasHair = true;
        }

        public bool HasHair { get; set; }

        public string Name { get; set; }

        public string MakeNoise()
        {
            return "Nyan";
        }
    }
}

```

Тестирование и навигационное наследование

```

interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name); //BaseClass
Console.WriteLine(baseType.BaseType.Name); //Object
Console.WriteLine(typeof(object).BaseType); //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance)); //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance)); //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));

```

```
//BaseInterface,DerivedInterface

Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False
```

Расширение абстрактного базового класса

В отличие от интерфейсов, которые могут быть описаны как контракты для реализации, абстрактные классы действуют как контракты для расширения.

Абстрактный класс не может быть инстанцирован, он должен быть расширен, и полученный класс (или производный класс) может быть затем создан.

Абстрактные классы используются для предоставления общих реализаций

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
}

public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}
```

В приведенном выше примере показано, как любой класс, расширяющий Автомобиль, автоматически получит метод HonkHorn с реализацией. Это означает, что любому разработчику, создающему новый автомобиль, не нужно будет беспокоиться о том, как он будет показывать его.

Конструкторы в подклассе

Когда вы создаете подкласс базового класса, вы можете построить базовый класс, используя `: base` после параметров конструктора подкласса.

```
class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}
```

```
class Trumpet : Instrument
{
    bool oiled;

    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}
```

Наследование. Последовательность вызовов конструкторов

Рассмотрим, что у нас есть класс `Animal` которого есть дочерний класс `Dog`

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

По умолчанию каждый класс неявно наследует класс `Object`.

Это то же самое, что и приведенный выше код.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

При создании экземпляра класса `Dog` **будет вызываться конструктор базовых классов по умолчанию (без параметров), если нет явного вызова другого конструктора в родительском классе**. В нашем случае сначала будет называться конструктор `Object's`, а затем `Animal's` и в конце конструктора `Dog's`.

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

Выход будет

В конструкторе Animal
В конструкторе Dog

[Посмотреть демо](#)

Вызовите конструктор родителя явно.

В приведенных выше примерах наш конструктор класса `Dog` вызывает конструктор по **умолчанию** класса `Animal`. Если вы хотите, вы можете указать, какой конструктор должен вызываться: можно вызвать любой конструктор, определенный в родительском классе.

Рассмотрим эти два класса.

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
        Console.WriteLine("Animal's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

Что здесь происходит?

У нас есть 2 конструктора в каждом классе.

Что означает `base` ?

`base` - ссылка на родительский класс. В нашем случае, когда мы создаем экземпляр класса `Dog` **подобный этому**

```
Dog dog = new Dog();
```

Среда выполнения сначала вызывает `Dog()` , который является конструктором без параметров. Но его тело не работает сразу. После круглых скобок конструктора мы имеем такой вызов: `base()` , что означает, что когда мы вызываем конструктор `Dog` по умолчанию, он, в свою очередь, вызывает конструктор по **умолчанию** родителя. После запуска конструктора родителя он вернется и, наконец, запустит тело конструктора `Dog()` .

Таким образом, вывод будет следующим:

```
Конструктор по умолчанию для Animal  
Собственный конструктор по умолчанию
```

[Посмотреть демо](#)

Теперь, если мы будем называть конструктор `Dog's` параметром?

```
Dog dog = new Dog("Rex");
```

Вы знаете, что члены родительского класса, которые не являются частными, наследуются дочерним классом, а это означает, что у `Dog` также будет поле `name` .

В этом случае мы передали аргумент нашему конструктору. Он, в свою очередь, передает аргумент конструктору родительского класса **с параметром** , который инициализирует поле `name` .

Выход будет

```
Animal's constructor with 1 parameter  
Rex  
Dog's constructor with 1 parameter  
Rex
```

Резюме:

Каждое создание объекта начинается с базового класса. В наследовании классы, находящиеся в иерархии, связаны цепями. Поскольку все классы производятся от `Object` , первый конструктор, который будет вызываться при создании любого объекта, является конструктором класса `Object` ; Затем вызывается следующий конструктор в цепочке и только после того, как все они называются, объект создается

ключевое слово base

1. Ключевое слово `base` используется для доступа к членам базового класса из производного класса:
2. Вызвать метод в базовом классе, который был переопределен другим методом.
Укажите, какой конструктор базового класса следует вызывать при создании экземпляров производного класса.

Методы наследования

Существует несколько способов унаследовать методы

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }

    // virtual methods CAN be overridden in derived classes
    public virtual void ChangeGear() {
        // Implementation of gears being changed
    }

    // abstract methods MUST be overridden in derived classes
    public abstract void Accelerate();
}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}
```

Наследование Анти-шаблоны

Неправильное наследование

Допустим, есть 2 класса класса `Foo` и `Bar`. `Foo` имеет две функции: `Do1` и `Do2`. `Bar` должен использовать `Do1` из `Foo`, но ему не нужна `Do2` или функция, эквивалентная `Do2` но делает что-то совершенно другое.

Плохой путь : сделайте `Do2()` на виртуальном `Foo` затем переопределите его в `Bar` или просто `throw Exception` в `Bar` для `Do2()`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
    }
}
```

```
        //or simply throws new Exception
    }
}
```

Хороший способ

Выньте `Do1()` из `Foo` и поместите его в новый класс `Baz` затем наследуйте как `Foo` и `Bar` от `Baz` и реализуйте `Do2()` отдельно

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
    }
}
```

Теперь почему первый пример плох, а второй хорош: когда разработчик `nr2` должен внести изменения в `Foo`, возможно, он сломает реализацию `Bar` потому что `Bar` теперь неотделим от `Foo`. Когда это делается по последнему примеру, `Foo` и `Bar` commonality были перемещены в `Baz` и они не влияют друг на друга (как и не должны).

Базовый класс с рекурсивной спецификацией типа

Однократное определение базового класса с рекурсивным спецификатором типа. Каждый узел имеет один родительский элемент и несколько дочерних элементов.

```
/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
```

```

{
    this.Parent=parent;
    this.Children=new List<T>();
    if(parent!=null)
    {
        parent.Children.Add(this as T);
    }
}
public T Parent { get; private set; }
public List<T> Children { get; private set; }
public bool IsRoot { get { return Parent==null; } }
public bool IsLeaf { get { return Children.Count==0; } }
/// <summary>
/// Returns the number of hops to the root object
/// </summary>
public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}

```

Вышеупомянутое может быть повторно использовано каждый раз, когда необходимо определить иерархию дерева объектов. Объект узла в дереве должен наследовать от базового класса с помощью

```

public class MyNode : Tree<MyNode>
{
    // stuff
}

```

каждый класс узла знает, где он находится в иерархии, каков родительский объект, а также какие объекты-дети. В нескольких встроенных типах используется древовидная структура, например `Control` или `XmlElement` и вышеупомянутое `Tree<T>` может использоваться как базовый класс *любого* типа в вашем коде.

Например, чтобы создать иерархию частей, где общий вес вычисляется по весу *всех* детей, выполните следующие действия:

```

public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); } }
}

```

для использования в качестве

```

// [Q:2.5] -- [P:4.2] -- [R:0.4]
// \

```

```

//      - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;

```

Другим примером может служить определение относительных координат. В этом случае истинное положение кадра координат зависит от положения *всех* исходных координат.

```

public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null,
    PointF.Empty) { };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
            var parent_pos = Parent.GlobalPosition;
            return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
        }
    }
    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }
    public RelativeCoordinate Add(PointF local_position)
    {
        return new RelativeCoordinate(this, local_position);
    }
    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}

```

для использования в качестве

```

// Define the following coordinate system hierarchy
// 
// o--> [A1] ---> [B1] -----> [C1]
//           |
//           +--> [B2] --+--> [C2]

```

```
// |  
// +--> [C3]  
  
var A1 = RelativeCoordinate.Start;  
var B1 = A1.Add(100, 20);  
var B2 = A1.Add(160, 10);  
  
var C1 = B1.Add(120, -40);  
var C2 = B2.Add(80, -20);  
var C3 = B2.Add(60, -30);  
  
double dist1 = C1.TotalDistance;
```

Прочтайте наследование онлайн: <https://riptutorial.com/ru/csharp/topic/29/наследование>

глава 106: Начало работы: Json с C

Вступление

В следующем разделе будет представлен способ работы с Json с использованием языка C # и концепций сериализации и десериализации.

Examples

Простой пример Json

```
{  
    "id": 89,  
    "name": "Aldous Huxley",  
    "type": "Author",  
    "books": [  
        {  
            "name": "Brave New World",  
            "date": 1932  
        },  
        {  
            "name": "Eyeless in Gaza",  
            "date": 1936  
        },  
        {  
            "name": "The Genius and the Goddess",  
            "date": 1955  
        }  
    ]  
}
```

Если вы новичок в Json, вот [пример учебника](#).

Первые вещи Сначала: Библиотека для работы с Json

Для работы с Json с использованием C # необходимо использовать библиотеку Newtonsoft (.net). Эта библиотека предоставляет методы, которые позволяют программировать сериализацию и десериализацию объектов и многое другое. [Есть учебник](#), если вы хотите узнать подробности о его методах и обычаях.

Если вы используете Visual Studio, откройте «Диспетчер пакетов / Управление пакетами» / «Управление пакетом» в Solution / и введите «Newtonsoft» в панель поиска и установите пакет. Если у вас нет NuGet, это [подробное руководство](#) может вам помочь.

Реализация C

Прежде чем читать какой-то код, важно отказаться от основных концепций, которые помогут программировать приложения с помощью json.

Сериализация : процесс преобразования объекта в поток байтов, который может быть отправлен через приложения. Следующий код может быть сериализован и преобразован в предыдущий json.

Deserialization : процесс преобразования json / потока байтов в объект. Это точно противоположный процесс сериализации. Предыдущий json можно десериализовать в объект C #, как показано в примерах ниже.

Чтобы это исправить, важно превратить структуру json в классы, чтобы использовать уже описанные процессы. Если вы используете Visual Studio, вы можете автоматически превратить json в класс, просто выбрав «*Edit / Paste Special / Paste JSON as Classes*» и вставку json-структуры.

```
using Newtonsoft.Json;

class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;      // "id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type = type;
        this.books = books;
    }
}

class Book
{
    [JsonProperty("name")]
    public string name;
    [JsonProperty("date")]
    public DateTime date;
}
```

Сериализация

```
static void Main(string[] args)
{
    Book[] books = new Book[3];
    Author author = new Author(89, "Aldous Huxley", "Author", books);
    string objectDeserialized = JsonConvert.SerializeObject(author);
    //Converting author into json
}
```

Метод «.SerializeObject» получает в качестве параметра *объект типа* , поэтому вы можете

поместить в него что-либо.

Десериализация

Вы можете получить json из любого места, файла или даже сервера, чтобы он не был включен в следующий код.

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

Метод «`DeserializeObject`» десериализует «`jsonExample`» в объект «`Автор`». Вот почему важно установить json-переменные в определении классов, поэтому метод обращается к нему, чтобы заполнить его.

Функция сериализации и де-сериализации Common Utilities

Этот пример используется для общей функции для сериализации и десериализации всех типов объектов.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
    public static class IGUtilities
    {
        public static string Serialization(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            return data;
        }

        public static T Deserialization(this string jsonData)
        {
            T copy = JsonConvert.DeserializeObject<T>(jsonData);
            return copy;
        }

        public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject<T>(data);
            return copy;
        }
    }
}
```

Прочтайте Начало работы: Json с C # онлайн: <https://riptutorial.com/ru/csharp/topic/9910/начало-работы-json-c-c-sharp>

глава 107: Небезопасный код в .NET.

замечания

- Чтобы использовать ключевое слово `unsafe` в проекте .Net, вы должны установить флагок «Разрешить небезопасный код» в Project Properties => Build
- Использование небезопасного кода может повысить производительность, однако это за счет безопасности кода (отсюда и термин `unsafe`).

Например, когда вы используете цикл `for` для такого массива:

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

.NET Framework гарантирует, что вы не превысите границы массива, бросая `IndexOutOfRangeException` если индекс превышает границы.

Однако, если вы используете небезопасный код, вы можете превысить границы массива следующим образом:

```
unsafe
{
    fixed (int* ptr = array)
    {
        for (int i = 0; i <= array.Length; i++)
        {
            *(ptr+i) = 0;
        }
    }
}
```

Examples

Небезопасный индекс массива

```
void Main()
{
    unsafe
    {
        int[] a = {1, 2, 3};
        fixed(int* b = a)
        {
            Console.WriteLine(b[4]);
        }
    }
}
```

Запуск этого кода создает массив длиной 3, но затем пытается получить 5-й элемент (индекс 4). На моей машине это напечатано 1910457872, но поведение не определено.

Без `unsafe` блока вы не можете использовать указатели и, следовательно, не можете получать доступ к значениям за конец массива, не вызывая исключения.

Использование небезопасных массивов

При доступе к массивам с помощью указателей проверка границ не выполняется, поэтому исключение `IndexOutOfRangeException` не будет выбрано. Это делает код быстрее.

Присвоение значений массиву с указателем:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    * (ptr+i) = i; //assigning the value with the pointer
                }
            }
        }
    }
}
```

В то время как безопасный и нормальный аналог будет:

```
class Program
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

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

Небезопасная часть, как правило, будет быстрее, и разница в производительности может варьироваться в зависимости от сложности элементов в массиве, а также от логики, применяемой к каждому из них. Несмотря на то, что это может быть быстрее, его следует использовать с осторожностью, поскольку его сложнее поддерживать и легче сломать.

Использование небезопасных строк

```
var s = "Hello";           // The string referenced by variable 's' is normally immutable, but
                           // since it is memory, we could change it if we can access it in an
                           // unsafe way.

unsafe                      // allows writing to memory; methods on System.String don't allow this
{
    fixed (char* c = s) // get pointer to string originally stored in read only memory
        for (int i = 0; i < s.Length; i++)
            c[i] = 'a';      // change data in memory allocated for original string "Hello"
}
Console.WriteLine(s); // The variable 's' still refers to the same System.String
                     // value in memory, but the contents at that location were
                     // changed by the unsafe write above.
                     // Displays: "aaaaa"
```

Прочтайте Небезопасный код в .NET. онлайн: <https://riptutorial.com/ru/csharp/topic/81/небезопасный-код-в--net>

глава 108: Неверные типы

Синтаксис

- `Nullable<int> i = 10;`
- `Int? j = 11;`
- `Int? k = null;`
- `DateTime? DateOfBirth = DateTime.Now;`
- `десятичный? Сумма = 1,0 м;`
- `BOOL? IsAvailable = true;`
- `обугливается? Letter = 'a';`
- `(тип)? имяПеременной`

замечания

Nullable типы могут представлять все значения базового типа, а также `null`.

Синтаксис `t?` является сокращением для `Nullable<T>`

Нулевыми значениями являются объекты `System.ValueType`, поэтому они могут быть в коробке и распакованы. Кроме того, `null` значение объекта с `null` значением не совпадает с `null` значением ссылочного объекта, это всего лишь флаг.

При боксировании с нулевым объектом нулевое значение преобразуется в `null` ссылку, а значение, отличное от `null`, преобразуется в базовый тип с нулевым значением.

```
DateTime? dt = null;
var o = (object)dt;
var result = (o == null); // is true

DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var dt2 = (DateTime)dt; // correct cause o contains DateTime value
```

Второе правило приводит к правильному, но парадоксальному коду:

```
DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var type = o.GetType(); // is DateTime, not Nullable<DateTime>
```

В краткой форме:

```
DateTime? dt = new DateTime(2015, 12, 11);
var type = dt.GetType(); // is DateTime, not Nullable<DateTime>
```

Examples

Инициализация нулевой

Для `null` значений:

```
Nullable<int> i = null;
```

Или же:

```
int? i = null;
```

Или же:

```
var i = (int?)null;
```

Для непустых значений:

```
Nullable<int> i = 0;
```

Или же:

```
int? i = 0;
```

Проверьте, имеет ли значение `Nullable` значение

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Это то же самое, что:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Получить значение типа NULL

Учитывая следующее значение nullable `int`

```
int? i = 10;
```

В случае, если значение по умолчанию необходимо, вы можете назначить его с помощью [оператора нулевой коалесценции](#), метода `GetValueOrDefault` или проверить, является ли `HasValue` `int HasValue` перед назначением.

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

Следующее использование всегда *небезопасно*. Если `i` равно нулю во время выполнения, будет `System.InvalidOperationException`. Во время разработки, если значение не задано, вы получите сообщение `Use of unassigned local variable 'i'` ошибке `Use of unassigned local variable 'i'`.

```
int j = i.Value;
```

Получение значения по умолчанию из значения NULL

Метод `.GetValueOrDefault()` возвращает значение, даже если свойство `.HasValue` является ложным (в отличие от свойства `Value`, которое генерирует исключение).

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

Выход:

```
0
1
```

Проверьте, является ли типовой параметр типа NULL

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) ) != null;
}
```

Значение по умолчанию для типов с нулевым значением равно null

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Выход:

ноль

Эффективное использование базового Nullable аргумент

Любой тип с нулевым значением является **общим** типом. И любой тип с нулевым значением - тип **значения**.

Существуют некоторые трюки, которые позволяют **эффективно использовать** результат метода `Nullable.GetUnderlyingType` при создании кода, связанного с целью **отражения** / генерации кода:

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
```

```
        return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

Использование:

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsDBNull())
    Console.WriteLine("Type is nullable.");
Type underlyingType;
if(type.IsDBNull(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

Выход:

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PS. NullableTypesCache определяется следующим образом:

```
static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type, Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}
```

Прочтайте Неверные типы онлайн: <https://riptutorial.com/ru/csharp/topic/1240/неверные-типы>

глава 109: неизменность

Examples

Класс System.String

В С # (и .NET) строка представлена классом System.String. Ключевое слово `string` - это псевдоним для этого класса.

Класс System.String неизменен, т. Е. После создания его состояние не может быть изменено.

Таким образом, все операции, выполняемые над строкой типа Substring, Remove, Replace, concatenation с помощью оператора + т. Д., Создадут новую строку и вернут ее.

См. Следующую программу демонстрации -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

Это будет печатать `string` и `mystring` соответственно.

Строки и неизменность

Неизменяемые типы - это типы, которые при изменении создают новую версию объекта в памяти, вместо того, чтобы изменять существующий объект в памяти. Простейшим примером этого является встроенный тип `string`.

Принимая следующий код, который добавляет «мир» к слову «Привет»,

```
string myString = "hello";
myString += " world";
```

В этом случае происходит то, что новый объект создается при добавлении к `string` во второй строке. Если вы сделаете это как часть большого цикла, есть вероятность, что это вызовет проблемы с производительностью в вашем приложении.

Изменчивым эквивалентом для `string` является `StringBuilder`

Принимая следующий код

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

Когда вы запустите это, вы сами модифицируете объект `StringBuilder` в памяти.

Прочтайте неизменность онлайн: <https://riptutorial.com/ru/csharp/topic/1863/неизменность>

глава 110: Нуль-условные операторы

Синтаксис

- ? X .Y; // null, если X имеет значение null else XY
- ? X .Y .Z; // null, если X является нулевым, или Y имеет значение null else XYZ
- ? X [индекс]; // null, если X имеет значение null else X [index]
- ? X .ValueMethod (); // null, если X является null else результатом X.ValueMethod ();
- ? X .VoidMethod (); // ничего не делать, если X является null else, вызывать X.VoidMethod ();

замечания

Обратите внимание, что при использовании оператора нулевой коалесценции на типе значений `T` вы получите `Nullable<T>` назад.

Examples

Null-Conditional Operator

`?. operator` - синтаксический сахар, чтобы избежать подробных нулевых проверок. Он также известен как [безопасный навигатор](#).

Класс, используемый в следующем примере:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

Если объект является потенциально нулевым (например, функция, возвращающая ссылочный тип), сначала объект должен быть проверен на значение `null`, чтобы предотвратить возможное исключение `NullReferenceException`. Без оператора с нулевым условием это выглядело бы так:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

В этом же примере используется оператор с нулевым условием:

```
Person person = GetPerson();

var age = person?.Age;      // 'age' will be of type 'int?', even if 'person' is not null
```

Цепочка оператора

Оператор с нулевым условием может быть объединен с элементами и под-членами объекта.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

Объединение с оператором Null-Coalescing

Оператор с нулевым условием может быть объединен с оператором с [нулевым коалесцированием](#) для предоставления значения по умолчанию:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

Условный индекс

Аналогично « ?. » оператор, оператор индекса с нулевым условием проверяет нулевые значения при индексировании в коллекцию, которая может быть нулевой.

```
string item = collection?[index];
```

синтаксический сахар для

```
string item = null;
if(collection != null)
{
    item = collection[index];
}
```

Избегание NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

Этот эффект можно связать вместе:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

Оператор Null-условный может использоваться с методом расширения

Метод расширения может работать с нулевыми ссылками, но вы можете использовать `?.` для того, чтобы все равно проверить нуль.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Обычно метод запускается для `null` ссылок и возвращает -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Использование `?.` метод не будет запущен для `null` ссылок, а `ТИП - int? :`

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

Такое поведение действительно ожидается от того, как `?.` оператор работает: он избегает вызова экземпляров экземпляра для нулевых экземпляров, чтобы избежать `NullReferenceExceptions`. Однако эта же логика применяется к методу расширения, несмотря на разницу в том, как объявлен метод.

Для получения дополнительной информации о том, почему метод расширения вызывается в первом примере, см. [Методы расширения - нулевая проверка](#) документации.

Прочтайте Нуль-условные операторы онлайн: <https://riptutorial.com/ru/csharp/topic/41/нуль-условные-операторы>

глава 111: Обеспечение безопасности переменной потока

Examples

Управление доступом к переменной в цикле Parallel.For

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500. sum is: {0}", sum );
    }
}
```

Недостаточно просто делать `sum = sum + i` без блокировки, потому что операция чтения-изменения-записи не является атомарной. Нить будет перезаписывать любые внешние изменения в `sum` которая возникает после того, как она прочитает текущее значение `sum`, но до того, как она сохранит измененное значение `sum + i` обратно в `sum`.

Прочтайте Обеспечение безопасности переменной потока онлайн:

<https://riptutorial.com/ru/csharp/topic/4140/обеспечение-безопасности-переменной-потока>

глава 112: Обзор коллекций с

Examples

HashSet

Это коллекция уникальных предметов, с $O(1)$ поиском.

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

Для сравнения, выполнение `Contains` в списке дает более низкую производительность:

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

`HashSet.Contains` использует хеш-таблицу, так что поиск выполняется очень быстро, независимо от количества элементов в коллекции.

SortedSet

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

T [] (массив T)

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
}
```

```

// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";

// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// two

```

Список

`List<T>` - это список заданного типа. Элементы могут быть добавлены, вставлены, удалены и адресованы индексом.

```

using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]); // 5

```

`List<T>` можно рассматривать как массив, размер которого можно изменить. Перечисление коллекции в порядке выполняется быстро, так же как и доступ к отдельным элементам через их индекс. Чтобы получить доступ к элементам, основанным на каком-то аспекте их значения или другом ключе, `Dictionary<T>` обеспечит более быстрый поиск.

ТОЛКОВЫЙ СЛОВАРЬ

Словарь `< TKey, TValue >` - это карта. Для данного ключа в словаре может быть одно значение.

```

using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, { "Mary", 35 }, { "Jack", 40 }
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException

```

```

int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40;      // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}

```

Дублирующий ключ при использовании инициализации коллекции

```

var people = new Dictionary<string, int>
{
    {"John", 30}, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists

```

стек

```

// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
Console.WriteLine(stack.Pop()); // prints 3

```

LinkedList

```
// initialize a LinkedList of integers
LinkedList<int> list = new LinkedList<int>();

// add some numbers to our list.
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

Обратите внимание, что `LinkedList<T>` представляет собой *дважды связанный* список. Таким образом, это просто набор узлов, и каждый узел содержит элемент типа `T`. Каждый узел связан с предыдущим узлом и следующим узлом.

Очередь

```
// Initialize a new queue of integers
var queue = new Queue<int>();

// Add some data
queue.Enqueue(6);
queue.Enqueue(4);
queue.Enqueue(9);

// Elements in a queue are stored in "first in, first out" order.
// The queue from first to last is: 6, 4, 9

// View the next element in the queue, without removing it.
Console.WriteLine(queue.Peek()); // prints 6

// Removes the first element in the queue, and returns it.
Console.WriteLine(queue.Dequeue()); // prints 6
Console.WriteLine(queue.Dequeue()); // prints 4
Console.WriteLine(queue.Dequeue()); // prints 9
```

Головки резьбы вверх! Используйте `ConcurrentQueue` в многопоточных средах.

Прочтайте Обзор коллекций с # онлайн: <https://riptutorial.com/ru/csharp/topic/2344/обзор-коллекций-c-sharp>

глава 113: Обработка `FormatException` при преобразовании строки в другие типы

Examples

Преобразование строки в целое число

Существуют различные методы для явного преобразования `string` в `integer`, например:

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

Но все эти методы будут `FormatException`, если входная строка содержит нечисловые символы. Для этого нам нужно написать дополнительную обработку исключений (`try..catch`), чтобы обрабатывать их в таких случаях.

Объяснение примерами:

Итак, пусть наш вклад будет:

```
string inputString = "10.2";
```

Пример 1: `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

Примечание. То же самое касается других упомянутых методов, а именно:

`Convert.ToInt16();` и `Convert.ToInt64();`

Пример 2: `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct
format.
```

Как мы обходим это?

Как было сказано ранее, для обработки исключений нам обычно требуется `try..catch` как показано ниже:

```

try
{
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
{
    //Display some message, that the conversion has failed.
}

```

Но использование `try..catch` **всё** `try..catch` мире не будет хорошей практикой, и могут быть некоторые сценарии, в которых мы хотели бы дать `0` если вход неверен (*если мы следуем описанному выше методу, нам нужно назначить `0 convertedInt` из блок `catch`*). Для обработки таких сценариев мы можем использовать специальный метод под названием `.TryParse()`.

Метод `.TryParse()` имеющий внутреннюю обработку `Exception`, которая даст вам выход параметру `out` и возвращает логическое значение, указывающее статус преобразования (`true` если преобразование было успешным, `false` если оно не удалось). На основе возвращаемого значения мы можем определить статус конверсии. Посмотрим один пример:

Использование 1: сохранить возвращаемое значение в булевой переменной

```

int convertedInt; // Be the required integer
bool isSuccessConversion = int.TryParse(inputString, out convertedInt);

```

Мы можем проверить переменную `isSuccessConversion` **после выполнения**, чтобы проверить статус преобразования. Если оно **ложно**, значение `convertedInt` будет `0` (*нет необходимости проверять возвращаемое значение, если вы хотите `0` для отказа преобразования*).

Использование 2: Проверьте возвращаемое значение, `if`

```

if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
    // Display an error message
}

```

Использование 3: Не проверяя возвращаемое значение, вы можете использовать следующее, если вы не заботитесь о возвращаемом значении (*преобразованном или нет, 0 будет в порядке*)

```

int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted

```

Прочтите Обработка `FormatException` при преобразовании строки в другие типы онлайн:

<https://riptutorial.com/ru/csharp/topic/2886/обработка-formatexception-при-преобразовании-строки-в-другие-типы>

глава 114: Обработка исключений

Examples

Основная обработка исключений

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

Обратите внимание, что обработка всех исключений с помощью одного и того же кода часто не является наилучшим подходом.

Это обычно используется, когда какие-либо внутренние процедуры обработки исключений терпят неудачу, в крайнем случае.

Обработка определенных типов исключений

```
try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}
```

Будьте осторожны, чтобы исключения оценивались по порядку и применялось наследование. Поэтому вам нужно начать с самых конкретных и закончить с их предком. В любой заданной точке будет выполнен только один блок catch.

Использование объекта исключения

Вы можете создавать исключения и создавать исключения из своего собственного кода. Создание экземпляра исключения выполняется так же, как и любой другой объект С#.

```
Exception ex = new Exception();  
  
// constructor with an overload that takes a message string  
Exception ex = new Exception("Error message");
```

Затем вы можете использовать ключевое слово `throw` для повышения исключения:

```
try  
{  
    throw new Exception("Error");  
}  
catch (Exception ex)  
{  
    Console.WriteLine(ex.Message); // Logs 'Error' to the output window  
}
```

Примечание. Если вы выбрали новое исключение внутри блока `catch`, убедитесь, что исходное исключение передано как «внутреннее исключение», например

```
void DoSomething()  
{  
    int b=1; int c=5;  
    try  
    {  
        var a = 1;  
        b = a - 1;  
        c = a / b;  
        a = a / c;  
    }  
    catch (DivideByZeroException dEx) when (b==0)  
    {  
        // we're throwing the same kind of exception  
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);  
    }  
    catch (DivideByZeroException dEx) when (c==0)  
    {  
        // we're throwing the same kind of exception  
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);  
    }  
}  
  
void Main()  
{  
    try  
    {  
        DoSomething();  
    }  
    catch (Exception ex)  
    {  
        // Logs full error information (incl. inner exception)  
        Console.WriteLine(ex.ToString());  
    }  
}
```

В этом случае предполагается, что исключение не может быть обработано, но в сообщение добавляется некоторая полезная информация (и исходное исключение по-прежнему можно получить через `ex.InnerException` **внешним блоком исключения**).

Он покажет что-то вроде:

```
System.DivideByZeroException: не может делить на b, потому что он равен нулю -  
> System.DivideByZeroException: Попытка деления на ноль.  
  в UserQuery.g__DoSomething0_0 () в C: [...] \ LINQPadQuery.cs: строка 36  
  --- Конец внутренней проверки стека исключений ---  
  в UserQuery.g__DoSomething0_0 () в C: [...] \ LINQPadQuery.cs: строка 42  
  в UserQuery.Main () в C: [...] \ LINQPadQuery.cs: строка 55
```

Если вы попробуете этот пример в LinqPad, вы заметите, что номера строк не очень значимы (они не всегда помогают вам). Но передача полезного текста ошибки, как было предложено выше, часто значительно сокращает время отслеживания местоположения ошибки, которая в этом примере явно соответствует линии

```
c = a / b;
```

в функции `DoSomething()`.

[Попробуйте в .NET Fiddle](#)

Наконец, блок

```
try  
{  
    /* code that could throw an exception */  
}  
catch (Exception)  
{  
    /* handle the exception */  
}  
finally  
{  
    /* Code that will be executed, regardless if an exception was thrown / caught or not */  
}
```

Блок `try / catch / finally` может быть очень удобен при чтении из файлов.

Например:

```
FileStream f = null;  
  
try  
{  
    f = File.OpenRead("file.txt");  
    /* process the file here */  
}  
finally  
{
```

```
    f?.Close(); // f may be null, so use the null conditional operator.  
}
```

За блоком `try` должен следовать либо `catch` либо `finally`. Однако, поскольку нет блокировки `catch`, выполнение приведет к завершению. Перед завершением будут выполнены инструкции внутри блока `finally`.

В чтении файла мы могли использовать `using` блок как `FileStream` (что возвращает `OpenRead`) реализует `IDisposable`.

Даже если в блоке `try` есть оператор `return` блок `finally`, как правило, выполняется; есть несколько случаев, когда это не будет:

- Когда происходит [StackOverflow](#).
- `Environment.FailFast`
- Процесс приложения убит, как правило, внешним источником.

Реализация `IErrorHandler` для служб WCF

Реализация `IErrorHandler` для служб WCF - отличный способ централизовать обработку ошибок и протоколирование. Представленная здесь реализация должна поймать любое необработанное исключение, которое вызывается в результате вызова одной из ваших служб WCF. В этом примере также показано, как вернуть пользовательский объект и как вернуть JSON, а не XML по умолчанию.

Внедрение `IErrorHandler`:

```
using System.ServiceModel.Channels;  
using System.ServiceModel.Dispatcher;  
using System.Runtime.Serialization.Json;  
using System.ServiceModel;  
using System.ServiceModel.Web;  
  
namespace BehaviorsAndInspectors  
{  
    public class ErrorHandler : IErrorHandler  
    {  
  
        public bool HandleError(Exception ex)  
        {  
            // Log exceptions here  
  
            return true;  
        } // end  
  
        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)  
        {  
            // Get the outgoing response portion of the current context  
            var response = WebOperationContext.Current.OutgoingResponse;  
  
            // Set the default http status code
```

```

        response.StatusCode = HttpStatusCode.InternalServerError;

        // Add ContentType header that specifies we are using JSON
        response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

        // Create the fault message that is returned (note the ref parameter) with
        BaseDataResponseContract
        fault = Message.CreateMessage(
            version,
            string.Empty,
            new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
            new DataContractJsonSerializer(typeof(BaseDataResponseContract)), new
        List<Type> { typeof(BaseDataResponseContract) }));

        if (ex.GetType() == typeof(VariousExceptionTypes))
        {
            // You might want to catch different types of exceptions here and process
            them differently
        }

        // Tell WCF to use JSON encoding rather than default XML
        var webBodyFormatMessageProperty = new
        WebBodyFormatMessageProperty(WebContentFormat.Json);
        fault.Properties.Add(WebBodyFormatMessageProperty.Name,
        webBodyFormatMessageProperty);

    } // end

} // end class

} // end namespace

```

В этом примере мы присоединяем обработчик к поведению службы. Вы также можете присоединить это к IEndpointBehavior, IContractBehavior или IOperationBehavior аналогичным образом.

Приложите к поведению обслуживания:

```

using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }
    }
}

```

```

    private IErrorHandler GetInstance()
    {
        return new ErrorHandler();
    }

    void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
BindingParameterCollection bindingParameters) { } // end

    void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
    {
        var errorHandlerInstance = GetInstance();

        foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
        {
            dispatcher.ErrorHandlers.Add(errorHandlerInstance);
        }
    }

    void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end

    } // end class

} // end namespace

```

Конфигурации в Web.config:

```

...
<system.serviceModel>

    <services>
        <service name="WebServices.MyService">
            <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
        </service>
    </services>

    <extensions>
        <behaviorExtensions>
            <!-- This extension if for the WCF Error Handling-->
            <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
        </behaviorExtensions>
    </extensions>

    <behaviors>
        <serviceBehaviors>
            <behavior>
                <serviceMetadata httpGetEnabled="true"/>
                <serviceDebug includeExceptionDetailInFaults="true"/>
                <ErrorHandlerBehavior />
            </behavior>
        </serviceBehaviors>
    </behaviors>

    ...
</system.serviceModel>

```

...

Вот несколько ссылок, которые могут быть полезны по этой теме:

[https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx)

[http://www.brainhud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they-realizatsii,i](http://www.brainhud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they-realizatsii-i)

Другие примеры:

[IErrorHandler возвращает неправильное тело сообщения, когда код статуса HTTP равен 401. Неавторизованный](#)

[IErrorHandler, похоже, не обрабатывает мои ошибки в WCF .. любые идеи?](#)

[Как заставить пользовательский обработчик ошибок WCF возвращать ответ JSON с кодом не-OK http?](#)

[Как установить заголовок Content-Type для запроса HttpClient?](#)

Создание пользовательских исключений

Вам разрешено выполнять пользовательские исключения, которые могут быть выбрасываться точно так же, как и любое другое исключение. Это имеет смысл, если вы хотите, чтобы ваши исключения отличались от других ошибок во время выполнения.

В этом примере мы создадим настраиваемое исключение для четкой обработки проблем, которые может иметь приложение при анализе сложного ввода.

Создание пользовательского класса исключений

Чтобы создать настраиваемое исключение, создайте подкласс класса `Exception`:

```
public class ParserException : Exception
{
    public ParserException() :
        base("The parsing went wrong and we have no additional information.") { }
}
```

Пользовательское исключение становится очень полезным, когда вы хотите предоставить дополнительную информацию олову:

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
        base($"Parser error in {fileName}:{lineNumber}")
    {
        FileName = fileName;
        LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

Теперь, когда вы `catch(ParserException x)` вас будет дополнительная семантика для точной настройки обработки исключений.

Пользовательские классы могут реализовать следующие функции для поддержки дополнительных сценариев.

ПОВТОРНО МЕТАНИЯ

Во время процесса синтаксического анализа исходное исключение по-прежнему представляет интерес. В этом примере это `FormatException` потому что код пытается разобрать кусок строки, который, как ожидается, будет числом. В этом случае пользовательское исключение должно поддерживать включение « `InnerException` »:

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) { }
```

СЕРИАЛИЗАЦИЯ

В некоторых случаях ваши исключения могут пересекать границы `AppDomain`. Это так, если ваш парсер работает в своем `AppDomain` для поддержки горячей перезагрузки новых конфигураций парсера. В `Visual Studio` вы можете использовать шаблон `Exception` для генерации кода, подобного этому.

```
[Serializable]
public class ParserException : Exception
{
    // Constructor without arguments allows throwing your exception without
    // providing any information, including error message. Should be included
    // if your exception is meaningful without any additional details. Should
    // set message by calling base constructor (default message is not helpful).
    public ParserException()
        : base("Parser failure.")
    {}

    // Constructor with message argument allows overriding default error message.
    // Should be included if users can provide more helpful messages than
```

```
// generic automatically generated messages.
public ParserException(string message)
    : base(message)
{ }

// Constructor for serialization support. If your exception contains custom
// properties, read their values here.
protected ParserException(SerializationInfo info, StreamingContext context)
    : base(info, context)
{ }
}
```

Использование исключения ParserException

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

Вы также можете использовать пользовательские исключения для исключений catching и wrapping. Таким образом, многие различные ошибки могут быть преобразованы в один тип ошибки, который более полезен для приложения:

```
try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
        $"Failed to read {token} as number.",
        FileName,
        LineNumber,
        ex);
}
```

При обработке исключений, создавая собственные пользовательские исключения, вы обычно должны включать ссылку на исходное исключение в свойстве `InnerException`, как показано выше.

Проблемы безопасности

Если разоблачение причины исключения может поставить под угрозу безопасность, позволяя пользователям видеть внутреннюю работу вашего приложения, может быть плохая идея обернуть внутреннее исключение. Это может применяться, если вы создаете библиотеку классов, которая будет использоваться другими.

Вот как вы могли бы создать настраиваемое исключение без обертывания внутреннего исключения:

```
try
{
    // ...
}
catch (SomeStandardException ex)
{
    // ...
    throw new MyCustomException(someMessage);
}
```

Заключение

При создании настраиваемого исключения (с оберткой или с помощью развернутого нового исключения) вы должны создать исключение, имеющее смысл для вызывающего.

Например, пользователь библиотеки классов может не знать о том, как эта библиотека выполняет свою внутреннюю работу. Исключения, вызванные зависимостями библиотеки классов, не имеют смысла. Скорее, пользователь хочет исключение, которое имеет отношение к тому, как библиотека классов использует эти зависимости ошибочным способом.

```
try
{
    // ...
}
catch (IOException ex)
{
    // ...
    throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

Исключение Анти-шаблоны

Глотание исключений

Всегда следует перебрасывать исключение следующим образом:

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

Повторное бросание исключения, как показано ниже, приведет к запутыванию исходного исключения и потеряет исходную трассировку стека. Никогда не следует этого делать! Трассировка стека до улова и ретрона будет потеряна.

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw ex;
}
```

Обработка исключений в бейсболе

Не следует использовать исключения в качестве [замены для обычных конструкций управления потоком](#), таких как операторы if-then и while. Этот анти-шаблон иногда называют [обработкой исключений для бейсбола](#).

Вот пример анти-шаблона:

```
try
{
    while (AccountManager.HasMoreAccounts ())
    {
        account = AccountManager.GetNextAccount ();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

Вот лучший способ сделать это:

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.WriteLine("Here are your account details: " + found.Details.ToString());
```

улов (исключение)

Почти нет (некоторые говорят, что нет!) Причин, чтобы поймать общий тип исключения в вашем коде. Вы должны поймать только те типы исключений, которые вы ожидаете, потому что в противном случае вы скрываете ошибки в своем коде.

```
try
{
    var f = File.Open(myfile);
    // do something
}
catch (Exception x)
{
    // Assume file not found
    Console.WriteLine("Could not open file");
    // but maybe the error was a NullReferenceException because of a bug in the file handling
    code?
}
```

Лучше сделать:

```
try
{
    var f = File.Open(myfile);
    // do something which should normally not throw exceptions
}
catch (IOException)
{
    Console.WriteLine("File not found");
}
// Unfortunately, this one does not derive from the above, so declare separately
catch (UnauthorizedAccessException)
{
    Console.WriteLine("Insufficient rights");
}
```

Если произойдет какое-либо другое исключение, мы автоматически разрешим приложение сбой, поэтому он непосредственно работает в отладчике, и мы можем исправить эту проблему. Мы не должны отправлять программу, где любые другие исключения, кроме этих, происходят в любом случае, так что это не проблема с крахом.

Ниже приведен неверный пример, так как он использует исключения для работы с ошибкой программирования. Это не то, для чего они предназначены.

```
public void DoSomething(String s)
{
    if (s == null)
        throw new ArgumentNullException(nameof(s));
    // Implementation goes here
}

try
{
    DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}
```

Совокупные исключения / множественные исключения из одного метода

Кто говорит, что вы не можете бросить несколько исключений в один метод. Если вы не привыкли играть с `AggregateExceptions`, у вас может возникнуть соблазн создать свою собственную структуру данных, чтобы представить, что многие вещи идут не так. Есть, конечно, другая структура данных, которая не является исключением, была бы более идеальной, такой как результаты проверки. Даже если вы играете с `AggregateExceptions`, вы можете быть на стороне приема и всегда обращаться с ними, не понимая, что они могут вам пригодиться.

Весьма правдоподобно, что метод выполняется, и даже если это будет провал в целом, вы захотите выделить несколько вещей, которые поступили не так в тех исключениях, которые были выбраны. В качестве примера это поведение можно увидеть с помощью методов параллельных вычислений, которые были разбиты на несколько потоков, и любое число из них могло генерировать исключения, и об этом необходимо сообщать. Вот глупый пример того, как вы могли бы воспользоваться этим:

```
public void Run()
{
    try
    {
        this.SillyMethod(1, 2);
    }
    catch (AggregateException ex)
    {
        Console.WriteLine(ex.Message);
        foreach (Exception innerException in ex.InnerExceptions)
        {
            Console.WriteLine(innerException.Message);
        }
    }
}
```

```

private void SillyMethod(int input1, int input2)
{
    var exceptions = new List<Exception>();

    if (input1 == 1)
    {
        exceptions.Add(new ArgumentException("I do not like ones"));
    }
    if (input2 == 2)
    {
        exceptions.Add(new ArgumentException("I do not like twos"));
    }
    if (exceptions.Any())
    {
        throw new AggregateException("Funny stuff happened during execution",
exceptions);
    }
}

```

Вложение исключений и попытка блокировки.

Один из них может вставить одно исключение / `try catch` block внутри другого.

Таким образом, можно управлять небольшими блоками кода, которые способны работать без нарушения всего вашего механизма.

```

try
{
//some code here
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
    //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}

```

Примечание. Избегайте [проглатывания исключений](#) при бросании в родительский блок `catch`

Лучшие практики

Cheatsheet

ДЕЛАТЬ	НЕ
Управляющий поток с контрольными инструкциями	Управляющий поток с исключениями
Следить за проигнорированным (поглощенным) исключением путем ведения журнала	Игнорировать исключение
Повторить исключение, используя <code>throw</code>	Исключение повторного броска - <code>throw new ArgumentNullException() ИЛИ throw ex</code>
Выбросить предопределенные системные исключения	Выбросить пользовательские исключения, аналогичные предопределенным системным исключениям
Выбросить пользовательское / предопределенное исключение, если оно имеет решающее значение для логики приложения	Выбросить пользовательские / предопределенные исключения, чтобы указать предупреждение в потоке
Ловить исключения, которые вы хотите обработать	Поймать каждое исключение

НЕ управляйте бизнес-логикой с исключениями.

Контроль потока НЕ должен выполняться исключениями. Вместо этого используйте условные утверждения. Если элемент управления можно сделать с инструкцией `if-else`, не используйте исключения, поскольку он снижает читаемость и производительность.

Рассмотрите следующий фрагмент г-ном Bad-Practices:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

Когда выполнение достигает `Console.WriteLine(myObject.ToString());`, приложение выкинет исключение `NullReferenceException`. Г-н Bad Practices понял, что `myObject` имеет значение `null` и отредактировал его фрагмент, чтобы поймать и обработать `NullReferenceException`:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    try
    {
        Console.WriteLine(myObject.ToString());
    }
    catch(NullReferenceException ex)
    {
        // Hmm, if I create a new instance of object and assign it to myObject:
        myObject = new object();
        // Nice, now I can continue to work with myObject
        DoSomethingElseWithMyObject();
    }
}
```

Поскольку предыдущий фрагмент кода охватывает только логику исключения, что мне делать, если `myObject` на данный момент не равен нулю? Где я должен освещать эту часть логики? Сразу после `Console.WriteLine(myObject.ToString());`? Как насчет после `try...catch` block?

Как насчет г-на лучших практик? Как он справится с этим?

```
// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}
```

Г-н Лучшие практики достигли той же логики с меньшим количеством кода и понятной и понятной логикой.

НЕ перебрасывать исключения

Переброски исключений стоят дорого. Это негативно сказывается на производительности. Для кода, который обычно терпит неудачу, вы можете использовать шаблоны проектирования для сведения к минимуму проблем с производительностью. В этом разделе описываются два шаблона проектирования, которые полезны, когда исключения могут значительно влиять на производительность.

НЕ поглощайте исключения без регистрации

```
try
{
    //Some code that might throw an exception
```

```
    }
    catch (Exception ex)
    {
        //empty catch block, bad practice
    }
}
```

Никогда не проглатывайте исключения. Игнорирование исключений сэкономит этот момент, но позже создаст хаос для ремонтопригодности. При регистрации исключений вы всегда должны регистрировать экземпляр исключения, чтобы отслеживать полную трассировку стека, а не только сообщение об исключении.

```
try
{
    //Some code that might throw an exception
}
catch (NullException ex)
{
    LogManager.Log(ex.ToString());
}
```

Не перехватывайте исключения, которые вы не можете обработать

Многие ресурсы, такие как [этот](#), настоятельно рекомендуют вам рассмотреть, почему вы ловите исключение в том месте, где вы его ловите. Вы должны только поймать исключение, если вы можете обработать его в этом месте. Если вы можете что-то сделать, чтобы помочь устранить проблему, например, попробовать альтернативный алгоритм, подключиться к резервной базе данных, попробовать другое имя файла, подождать 30 секунд и повторить попытку или уведомить администратора, вы можете поймать ошибку и сделать это. Если вы ничего не можете сделать правдоподобно и разумно, просто «отпустите», и пусть исключение будет обрабатываться на более высоком уровне. Если исключение является достаточно катастрофическим, и нет разумного выбора, кроме как для всей программы, чтобы сбой из-за серьезности проблемы, то пусть это сбой.

```
try
{
    //Try to save the data to the main database.
}
catch (SqlException ex)
{
    //Try to save the data to the alternative database.
}
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the
exception bubble up to a level where it can be handled.
```

Необработанное и исключение потоков

AppDomain.UnhandledException Это событие предоставляет уведомление о

неперехваченных исключениях. Оно позволяет приложению регистрировать информацию об исключении до того, как обработчик по умолчанию системы сообщит об исключении для пользователя и прекратит применение. Если имеется достаточная информация о состоянии приложения, другие могут быть предприняты действия, такие как сохранение программных данных для последующего восстановления. Предупреждение рекомендуется, так как данные программы могут быть повреждены, если исключения не обрабатываются.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
private static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += new
    UnhandledExceptionEventHandler(UnhandledException);
}
```

Application.ThreadException Это событие позволяет вашему приложению Windows Forms обрабатывать иначе необработанные исключения, возникающие в потоках Windows Forms. Прикрепите обработчики событий к событию ThreadException для устранения этих исключений, которые оставят ваше приложение в неизвестном состоянии. По возможности исключения должны обрабатываться блоком обработки структурированных исключений.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
private static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += new
    UnhandledExceptionEventHandler(UnhandledException);
    Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
}
```

И, наконец, обработка исключений

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    Exception ex = (Exception)e.ExceptionObject;
    // your code
}

static void ThreadException(object sender, ThreadExceptionEventArgs e)
{
    Exception ex = e.Exception;
    // your code
}
```

Выброс исключения

Ваш код может и часто должен вызывать исключение, если что-то необычное произошло.

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

Прочтайте Обработка исключений онлайн: <https://riptutorial.com/ru/csharp/topic/40/обработка-исключений>

глава 115: Обработчик проверки подлинности C

Examples

Обработчик аутентификации

```
public class AuthenticationHandler : DelegatingHandler
{
    /// <summary>
    /// Holds request's header name which will contains token.
    /// </summary>
    private const string securityToken = "__RequestAuthToken";

    /// <summary>
    /// Default overridden method which performs authentication.
    /// </summary>
    /// <param name="request">Http request message.</param>
    /// <param name="cancellationToken">Cancellation token.</param>
    /// <returns>Returns http response message of type <see cref="HttpResponseMessage"/>
    class asynchronously.</returns>
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
    CancellationToken cancellationToken)
    {
        if (request.Headers.Contains(securityToken))
        {
            bool authorized = Authorize(request);
            if (!authorized)
            {
                return ApiHttpUtility.FromResult(request, false,
                    HttpStatusCode.Unauthorized, MessageTypes.Error, Resource.UnAuthenticatedUser);
            }
        }
        else
        {
            return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
                MessageTypes.Error, Resource.UnAuthenticatedUser);
        }

        return base.SendAsync(request, cancellationToken);
    }

    /// <summary>
    /// Authorize user by validating token.
    /// </summary>
    /// <param name="requestMessage">Authorization context.</param>
    /// <returns>Returns a value indicating whether current request is authenticated or
    not.</returns>
    private bool Authorize(HttpRequestMessage requestMessage)
    {
        try
        {
            HttpRequest request = HttpContext.Current.Request;
            string token = request.Headers[securityToken];
```

```
        return SecurityUtility.IsTokenValid(token, request.UserAgent,
HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
    }
    catch (Exception)
    {
        return false;
    }
}
```

Прочтайте Обработчик проверки подлинности C # онлайн:

<https://riptutorial.com/ru/csharp/topic/5430/обработчик-проверки-подлинности-c-sharp>

глава 116: Общие операции с строками

Examples

Разделение строки по определенному символу

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');
//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');
//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

Получение подстрок заданной строки

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

Substring возвращает строку вверх из заданного индекса или между двумя индексами (оба включительно).

Определите, начинается ли строка с заданной последовательностью

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

Поиск строки внутри строки

Используя `System.String.Contains` вы можете узнать, существует ли определенная строка внутри строки. Метод возвращает логическое значение, `true`, если строка существует `false` иначе.

```
string s = "Hello World";
bool stringExists = s.Contains("ello"); //stringExists =true as the string contains the
substring
```

Обрезка нежелательных символов Отключение и / или конец строк.

`String.Trim()`

```
string x = "    Hello World!    ";
string y = x.Trim(); // "Hello World!"
```

```
string q = "{(Hi!*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

String.TrimStart() И String.TrimEnd()

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' ); // "{(Hi"
```

Форматирование строки

Используйте метод `String.Format()` для замены одного или нескольких элементов в строке строковым представлением указанного объекта:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

Объединение массива строк в новый

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz" };
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

Заполнение строки до фиксированной длины

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);           // paddedLeft = " Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+');    // paddedRight = "Foo++"
string noPadded = s.PadLeft(2);            // noPadded = "Foo" (original string is never
                                         // shortened)
```

Построить строку из массива

Метод `String.Join` поможет нам построить строку из массива / списка символов или строки. Этот метод принимает два параметра. Первый - разделитель или разделитель, который поможет вам разделить каждый элемент массива. И вторым параметром является сам массив.

Строка из `char array`:

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

Выход: `a,b,c` если мы изменим `delimiter` как `""` тогда выход станет `abc`.

Строка из List of char :

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

Выход : a|b|c

Строка из List of Strings :

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a", "boy" };
string inputString = String.Join(delimiter, stringList);
```

Результат : Ram is a boy

Строка из array of strings :

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a", "boy" };
string inputString = String.Join(delimiter, stringArray);
```

Выход : Ram_is_a_boy

Форматирование с использованием ToString

Обычно мы используем метод `String.Format` для целей форматирования, `.ToString` обычно используется для преобразования других типов в строку. Мы можем указать формат вместе с методом `ToString` во время преобразования, поэтому мы можем избежать дополнительного форматирования. Позвольте мне объяснить, как он работает с разными типами;

Целое число в отформатированную строку:

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be "Input value
is 10"
```

дважды в отформатированную строку:

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00"); // output 10
string customFormat = doubleValue.ToString("Input value is 0.0"); // Input value is 10.5
```

Форматирование DateTime с использованием ToString

```
DateTime currentDate = DateTime.Now; // {7/21/2016 7:23:15 PM}
```

```
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

Получение символов с правой стороны строки

Visual Basic имеет функции `Left`, `Right` и `Mid`, которые возвращают символы из левой, правой и средней строки. Эти методы не существуют в C #, но могут быть реализованы с помощью `Substring()`. Они могут быть реализованы как методы расширения, такие как:

```
public static class StringExtensions
{
    /// <summary>
    /// VB Left function
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>Left-most numchars characters</returns>
    public static string Left( this string stringparam, int numchars )
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( 0, numchars );
    }

    /// <summary>
    /// VB Right function
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>Right-most numchars characters</returns>
    public static string Right( this string stringparam, int numchars )
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( stringparam.Length - numchars );
    }

    /// <summary>
    /// VB Mid function - to end of string
    /// </summary>
```

```

/// <param name="stringparam"></param>
/// <param name="startIndex">VB-Style startIndex, 1st char startIndex = 1</param>
/// <returns>Balance of string beginning at startIndex character</returns>
public static string Mid( this string stringparam, int startIndex )
{
    // Handle possible Null or numeric stringparam being passed
    stringparam += string.Empty;

    // Handle possible negative startIndex being passed
    startIndex = Math.Abs( startIndex );

    // Validate numchars parameter
    if (startIndex > stringparam.Length)
        startIndex = stringparam.Length;

    // C# strings are zero-based, convert passed startIndex
    return stringparam.Substring( startIndex - 1 );
}

/// <summary>
/// VB Mid function - for number of characters
/// </summary>
/// <param name="stringparam"></param>
/// <param name="startIndex">VB-Style startIndex, 1st char startIndex = 1</param>
/// <param name="numchars">number of characters to return</param>
/// <returns>Balance of string beginning at startIndex character</returns>
public static string Mid( this string stringparam, int startIndex, int numchars )
{
    // Handle possible Null or numeric stringparam being passed
    stringparam += string.Empty;

    // Handle possible negative startIndex being passed
    startIndex = Math.Abs( startIndex );

    // Handle possible negative numchars being passed
    numchars = Math.Abs( numchars );

    // Validate numchars parameter
    if (startIndex > stringparam.Length)
        startIndex = stringparam.Length;

    // C# strings are zero-based, convert passed startIndex
    return stringparam.Substring( startIndex - 1, numchars );
}

}
}

```

Этот метод расширения можно использовать следующим образом:

```

string myLongString = "Hello World!";
string myShortString = myLongString.Right(6);    // "World!"
string myLeftString = myLongString.Left(5);        // "Hello"
string myMidString1 = myLongString.Left(4);        // "lo World"
string myMidString2 = myLongString.Left(2,3);       // "ell"

```

Проверка пустой строки с помощью `String.IsNullOrEmpty ()` и `String.IsNullOrWhiteSpace ()`

```

string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";

bool result;

result = String.IsNullOrEmpty(nullString);           // true
result = String.IsNullOrEmpty(emptyString);         // true
result = String.IsNullOrEmpty(whitespaceString);    // false
result = String.IsNullOrEmpty(tabString);           // false
result = String.IsNullOrEmpty(newlineString);        // false
result = String.IsNullOrEmpty(nonEmptyString);      // false

result = String.IsNullOrWhiteSpace(nullString);      // true
result = String.IsNullOrWhiteSpace(emptyString);    // true
result = String.IsNullOrWhiteSpace(tabString);       // true
result = String.IsNullOrWhiteSpace(newlineString);   // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);  // false

```

Получение символа по определенному индексу и перечисление строки

Вы можете использовать метод `Substring` для получения любого количества символов из строки в любом месте. Однако, если вам нужен только один символ, вы можете использовать строковый индексатор для получения одного символа в любом заданном индексе, как это делается с массивом:

```

string s = "hello";
char c = s[1]; //Returns 'e'

```

Обратите внимание, что тип возврата - `char`, в отличие от метода `Substring` который возвращает тип `string`.

Вы также можете использовать индексатор для итерации по символам строки:

```

string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
***** This will print each character on a new line:
h
e
l
l
o
*****

```

Преобразование десятичного числа в двоичный, восьмеричный и шестнадцатеричный формат

1. Для преобразования десятичного числа в двоичный формат используйте **base 2**

```
Int32 Number = 15;
Console.WriteLine(Convert.ToString(Number, 2)); //OUTPUT : 1111
```

2. Для преобразования десятичного числа в восьмеричный формат используйте **базу 8**

```
int Number = 15;
Console.WriteLine(Convert.ToString(Number, 8)); //OUTPUT : 17
```

3. Для преобразования десятичного числа в шестнадцатеричный формат используйте **базу 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16)); //OUTPUT : f
```

Разделение строки другой строкой

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Результат:

```
["this", "is", "a", "complete", "sentence"]
```

Правильное изменение строки

В большинстве случаев, когда людям приходится менять строку, они делают это более или менее:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

Однако эти люди не понимают, что это на самом деле неправильно.

И я не имею ввиду из-за отсутствия проверки NULL.

Это на самом деле неправильно, потому что Glyph / GraphemeCluster может состоять из нескольких кодовых точек (например, символов).

Чтобы понять, почему это так, мы сначала должны знать о том, что на самом деле означает термин «характер».

[Ссылка:](#)

Характер - это перегруженный термин, чем может означать многое.

Кодовая точка - это атомная единица информации. Текст - это последовательность кодовых точек. Каждая кодовая точка - это номер, который

задается стандартом Unicode.

Графема представляет собой последовательность из одной или нескольких кодовых точек, которые отображаются как единое графическое устройство, которое читатель распознает как один элемент системы письма. Например, и а, и ё являются графемами, но они могут состоять из нескольких кодовых точек (например, ё может быть двумя кодовыми точками, один для базового символа а, за которым следует один для diareis, но есть и альтернативный, устаревший, один код точка, представляющая эту графему). Некоторые кодовые точки никогда не являются частью какой-либо графемы (например, нулевой ширины без участия или переопределения направления).

Глиф представляет собой изображение, обычно хранящееся в шрифте (который представляет собой набор глифов), используемый для представления графемов или их частей. Шрифты могут составлять несколько глифов в одно представление, например, если вышеуказанная ё является одной кодовой точкой, шрифт может выбрать, чтобы отобразить это как два отдельных, пространственно перекрытых глифа. Для OTF таблицы GSUB и GPOS шрифта содержат информацию о замещении и позиционировании, чтобы сделать эту работу. Шрифт может содержать несколько альтернативных глифов для одной и той же графемы.

Таким образом, в C # символ на самом деле является CodePoint.

Это означает, что если вы просто отмените действительную строку, например `Les Misérables`, которая может выглядеть так

```
string s = "Les Mis\u00e9rables";
```

как последовательность символов, вы получите:

selbařesíM seL

Как вы можете видеть, акцент делается на символе R вместо символа e.

Хотя `string.reverse.reverse` даст исходную строку, если вы оба раза измените массив символов, этот вид разворота определенно **НЕ** является обратным исходной строке.

Вам нужно только отменить каждый GraphemeCluster.

Итак, если все сделано правильно, вы можете изменить строку следующим образом:

```
private static System.Collections.Generic.List<string> GraphemeClusters(string s)
{
    System.Collections.Generic.List<string> ls = new
System.Collections.Generic.List<string>();

    System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
```

```

        while (enumerator.MoveNext())
    {
        ls.Add((string)enumerator.Current);
    }

    return ls;
}

// this
private static string ReverseGraphemeClusters(string s)
{
    if(string.IsNullOrEmpty(s) || s.Length == 1)
        return s;

    System.Collections.Generic.List<string> ls = GraphemeClusters(s);
    ls.Reverse();

    return string.Join("", ls.ToArray());
}

public static void TestMe()
{
    string s = "Les Mise\u0301rables";
    // s = "no\u00ebl";
    string r = ReverseGraphemeClusters(s);

    // This would be wrong:
    // char[] a = s.ToCharArray();
    // System.Array.Reverse(a);
    // string r = new string(a);

    System.Console.WriteLine(r);
}

```

И - о, радость - вы поймете, что если вы сделаете это правильно, это также будет работать для азиатских / южноазиатских / восточно-азиатских языков (и французского / шведского / норвежского и т. Д.) ...

Замена строки в строке

Используя метод `System.String.Replace`, вы можете заменить часть строки другой строкой.

```

string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"

```

Все вхождения строки поиска заменяются.

Этот метод также можно использовать для удаления части строки, используя поле `String.Empty`:

```

string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"

```

Изменение случая символов внутри строки

Класс `System.String` поддерживает ряд методов для преобразования между строчными и строчными символами в строке.

- `System.String.ToLowerInvariant` используется для возврата объекта `String`, преобразованного в нижний регистр.
- `System.String.ToUpperInvariant` используется для возврата объекта `String`, преобразованного в верхний регистр.

Примечание . Причина использования *инвариантных* версий этих методов заключается в предотвращении создания неожиданных букв, специфичных для конкретной культуры. Это объясняется [здесь подробно](#) .

Пример:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Обратите внимание, что вы можете указать конкретную **Культуру** при преобразовании в строчные и прописные буквы, используя [методы `String.ToLower \(CultureInfo\)`](#) и [`String.ToUpper \(CultureInfo\)`](#) соответственно.

Объединение массива строк в одну строку

Метод `System.String.Join` позволяет объединить все элементы в массив строк, используя указанный разделитель между каждым элементом:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(", ", words); // singleString = "One,Two,Three,Four"
```

Конкатенация строк

Конкатенация строк может быть выполнена с помощью метода `System.String.Concat` или (намного проще) с помощью оператора `+`:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

В C # 6 это можно сделать следующим образом:

```
string concat = $"{first},{second}";
```

Прочтайте Общие операции с строками онлайн: <https://riptutorial.com/ru/csharp/topic/73/общие-операции-с-строками>

глава 117: Объектно-ориентированное программирование на C

Вступление

В этом разделе мы расскажем, как мы можем писать программы на основе подхода ООП. Но мы не пытаемся преподавать парадигму объектно-ориентированного программирования. Мы рассмотрим следующие темы: Классы, Свойства, Наследование, Полиморфизм, Интерфейсы и т. Д.

Examples

Классы:

Скелет объявляющего класса:

<>: Требуется

[]:Необязательный

```
[private/public/protected/internal] class <Desired Class Name> [:Inherited  
class] [,][[Interface Name 1],[Interface Name 2],...]  
{  
    //Your code  
}
```

Не беспокойтесь, если вы не можете понять весь синтаксис, мы познакомимся со всей частью этого. Для первого примера рассмотрим следующий класс:

```
class MyClass  
{  
    int i = 100;  
    public void getMyValue()  
    {  
        Console.WriteLine(this.i); //Will print number 100 in output  
    }  
}
```

в этом классе мы создаем переменную `i` с типом `int` и с помощью индивидуальных модификаторов доступа и `getMyValue()` с модификаторами общедоступного доступа.

Прочтайте Объектно-ориентированное программирование на C # онлайн:

<https://riptutorial.com/ru/csharp/topic/9856/объектно-ориентированное-программирование-на-c-sharp>

глава 118: Оператор Null-Coalescing

Синтаксис

- var result = possibleNullObject ?? значение по умолчанию;

параметры

параметр	подробности
possibleNullObject	Значение для проверки нулевого значения. Если значение не равно null, это значение возвращается. Должен быть тип с нулевым значением.
defaultValue	Значение, возвращаемое, если possibleNullObject имеет значение NULL. Должен быть того же типа, что и possibleNullObject .

замечания

Нулевой оператор коалесцирования сам по себе является двумя последовательными символами вопросительного знака: ??

Это сокращение условного выражения:

```
possibleNullObject != null ? possibleNullObject : defaultValue
```

Левый операнд (тестируемый объект) должен быть типом типа NULL или ссылочным типом, или произойдет ошибка компиляции.

The ?? оператор работает как для ссылочных типов, так и для типов значений.

Examples

Основное использование

Используя `null-coalescing operator (??)` вы можете указать значение по умолчанию для типа NULL, если левый операнд имеет значение `null` .

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

Живая демонстрация на .NET скрипке

Это логически эквивалентно:

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

или используя [оператор тернарного оператора \(? :\)](#):

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

Null fall-through and chaining

Левый operand должен быть нулевым, в то время как правый operand может быть или не быть. Результат будет набран соответствующим образом.

Ненулевые

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

Выход:

Тип: System.Int32

значение: 3

[Посмотреть демо](#)

Nullable

```
int? a = null;
int? b = null;
var output = a ?? b;
```

output будет иметь тип int? и равна b или null.

Многократное объединение

Коалесцирование также может выполняться в цепях:

```
int? a = null;
int? b = null;
int c = 3;
var output = a ?? b ?? c;

var type = output.GetType();
Console.WriteLine($"Type :{type}");
Console.WriteLine($"value :{output}");
```

Выход:

Тип: System.Int32

значение: 3

[Посмотреть демо](#)

Null Conditional Chaining

Оператор нулевого коалесцирования может использоваться в tandemе с [оператором распространения пустоты](#) для обеспечения более безопасного доступа к свойствам объектов.

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

Выход:

Тип: System.String

Значение: Значение по умолчанию

[Посмотреть демо](#)

Оператор коллаборации с вызовом метода

Оператор null coalescing позволяет легко убедиться, что метод, который может вернуть значение `null`, вернется к значению по умолчанию.

Без оператора нулевой коалесценции:

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

С нулевым коалесцирующим оператором:

```
string name = GetName() ?? "Unknown!";
```

Использовать существующие или создавать новые

Обычный сценарий использования, с которым эта функция действительно помогает, - это когда вы ищете объект в коллекции и вам нужно создать новый, если он еще не существует.

```
IEnumerable<MyClass> myList = GetMyList();
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

Инициализация ленивых свойств с помощью оператора нулевой коалесценции

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

Первый раз, когда недвижимость `.FooBars` осуществляется доступ к `_fooBars` переменному будет оценивать, как `null`, таким образом, падает через к правопреемникам оператора присваивания и оценивает к полученному значению.

Безопасность резьбы

Это **не потокобезопасный** способ реализации ленивых свойств. Для поточнобезопасной лени используйте класс `Lazy<T>` встроенный в .NET Framework.

С # 6 Синтаксический сахар с использованием тел экспрессии

Обратите внимание, что с С # 6 этот синтаксис можно упростить с помощью тела выражения для свойства:

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Последующий доступ к свойству даст значение, хранящееся в переменной `_fooBars`.

Пример в шаблоне MVVM

Это часто используется при реализации команд в шаблоне MVVM. Вместо того, чтобы

инициализировать команды с помощью конструкции модели, команды инициализируются лениво, используя этот шаблон следующим образом:

```
private ICommand _actionCommand = null;  
public ICommand ActionCommand =>  
    _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Прочтайте Оператор Null-Coalescing онлайн: <https://riptutorial.com/ru/csharp/topic/37/operator-null-coalescing>

глава 119: Оператор равенства

Examples

Виды равенства в C# и оператор равенства

В C# существует два разных типа равенства: ссылочное равенство и равенство значений. Ценностное равенство - это общепринятое значение равенства: это означает, что два объекта содержат одни и те же значения. Например, два целых числа со значением 2 имеют значение равенства. Ссылочное равенство означает, что не существует двух объектов для сравнения. Вместо этого существуют две ссылки на объекты, которые относятся к одному и тому же объекту.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b); //returns true
```

Для предопределенных типов значений оператор равенства (==) возвращает true, если значения его operandов равны, в противном случае - false. Для ссылочных типов, отличных от строки, == возвращает true, если два операнда относятся к одному и тому же объекту.

Для типа строки == сравнивает значения строк.

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

Прочтите Оператор равенства онлайн: <https://riptutorial.com/ru/csharp/topic/1491/оператор-равенства>

глава 120: операторы

Вступление

В С # **оператор** - это программный элемент, который применяется к одному или нескольким операндам в выражении или инструкции. Операторы, которые принимают один операнд, такой как оператор приращения (++) или новый, называются унарными операторами. Операторы, которые принимают два операнда, такие как арифметические операторы (+, -, *, /), называются двоичными операторами. Один оператор, условный оператор (? :), принимает три операнда и является единственным тернарным оператором в С #.

Синтаксис

- публичный статический оператор OperandType оператораSymbol (операнд операнда1)
- публичный статический оператор OperandType оператораSymbol (OperandType operand1, OperandType2 operand2)

параметры

параметр	подробности
operatorSymbol	Перегруженный оператор, например +, -, /, *
OperandType	Тип, который будет возвращен перегруженным оператором.
operand1	Первый операнд, который будет использоваться при выполнении операции.
operand2	Второй операнд, который будет использоваться при выполнении операции, при выполнении двоичных операций.
заявления	Необязательный код, необходимый для выполнения операции перед возвратом результата.

замечания

Все операторы определяются как `static methods` и они не являются `virtual` и они не наследуются.

Приоритет оператора

Все операторы имеют особый «приоритет» в зависимости от того, к какой группе относится оператор (операторы той же группы имеют одинаковый приоритет). Значение некоторых операторов будет применено к другим. Ниже следует список групп (содержащих их соответствующие операторы), упорядоченные по приоритету (сначала по высоте):

- **Первичные операторы**

- `ab` - доступ к члену.
- `a?.b` - Нулевой доступ к условному члену.
- `->` - разыменование указателя в сочетании с доступом участника.
- `f(x)` - вызов функции.
- `a[x]` - индекс.
- `a?[x]` - Нулевой условный индекс.
- `x++` - Постерическое приращение.
- `x--` - Постфиксный декремент.
- `new` - Создание экземпляра.
- `default(T)` - Возвращает инициализированное значение по умолчанию типа `T`.
- `typeof` - Возвращает объект `Type` операнда.
- `checked` - Включает проверку числового переполнения.
- `unchecked` - отключает проверку числового переполнения.
- `delegate` - объявляет и возвращает экземпляр делегата.
- `sizeof` - Возвращает размер в байтах операнда типа.

- **Унарные операторы**

- `+x` - Возвращает `x`.
- `-x` - Числовое отрицание.
- `!x` - Логическое отрицание.
- `~x` - Побитовое дополнение / объявляет деструкторы.
- `++x` - Приращение префикса.
- `--x` - --x префиксов.
- `(T)x` - Тип литья.
- `await` - await выполнения `Task`.
- `&x` - возвращает адрес (указатель) `x`.
- `*x` - разыменование указателя.

- **Мультипликативные операторы**

- `x * y` - Умножение.
- `x / y` - Отдел.
- `x % y` - Модуль.

- **Аддитивные операторы**

- `x + y` - Дополнение.

- `x - y` - вычитание.

- **Побитовые операторы сдвига**

- `x << y` - сдвинуть бит влево.
- `x >> y` - сдвинуть бит вправо.

- **Операторы реляционного / типа тестирования**

- `x < y` - Меньше чем.
- `x > y` - больше.
- `x <= y` - меньше или равно.
- `x >= y` - больше или равно.
- `is` - совместимость Тип.
- `as` - Преобразование типа.

- **Операторы равенства**

- `x == y` - Равенство.
- `x != y` - Не равно.

- **Логический И Оператор**

- `x & y` - Логическое / побитовое И.

- **Логический оператор XOR**

- `x ^ y` - Логический / побитовый XOR.

- **Логический оператор ИЛИ**

- `x | y` - логическое / побитовое ИЛИ.

- **Условный И Оператор**

- `x && y` - Короткое замыкание логического И.

- **Условный ИЛИ Оператор**

- `x || y` - Короткое замыкание логического ИЛИ.

- **Оператор Null-coalescing**

- `x ?? y` - возвращает `x` если он не равен нулю; в противном случае возвращает `y`.

- **Условный оператор**

- `x ? y : z` - вычисляет / возвращает `y` если `x` ИСТИННО; в противном случае - `z`.

Связанный контент

- [Оператор Null-Coalescing](#)
- [Null-Conditional Operator](#)
- [имя оператора](#)

Examples

Перегружаемые операторы

С # позволяет пользовательским типам перегружать операторы путем определения статических функций-членов с использованием ключевого слова `operator`. Следующий пример иллюстрирует реализацию оператора `+`.

Если у нас есть класс `Complex` представляющий комплексное число:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

И мы хотим добавить опцию для использования оператора `+` для этого класса. то есть:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

Нам нужно будет перегрузить оператор `+` для класса. Это делается с использованием статической функции и ключевого слова `operator`:

```
public static Complex operator +(Complex c1, Complex c2)
{
    return new Complex
    {
        Real = c1.Real + c2.Real,
        Imaginary = c1.Imaginary + c2.Imaginary
    };
}
```

Операторы, такие как `+`, `-`, `*`, `/` могут быть перегружены. Сюда также входят операторы, которые не возвращают один и тот же тип (например, `==` и `!=`). Могут быть перегружены, несмотря на возврат булевых элементов). Здесь также применяется правило, относящееся к парам.

Операторы сравнения должны быть перегружены парами (например, если `<` перегружен, `>` также необходимо перегрузить).

Полный список перегружаемых операторов (а также неперегружаемых операторов и ограничений, помещенных на некоторые перегружаемые операторы) можно увидеть в [MSDN - Overloadable Operators \(Руководство по программированию на C #\)](#).

7,0

перегрузка `operator is` была введена с помощью механизма сопоставления шаблонов C # 7.0. Для получения дополнительной информации см [Pattern Matching](#)

Учитывая тип `Cartesian` определенный следующим образом

```
public class Cartesian
{
    public int X { get; }
    public int Y { get; }
}
```

Перегружаемый `operator is` может быть определен, например, для `Polar` координат

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

который можно использовать следующим образом

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(Пример взята из [сопоставимой документации по шаблону Roslyn](#))

Реляционные операторы

Равно

Проверяет, равны ли поставленные операнды (аргументы)

```
"a" == "b"      // Returns false.
"a" == "a"      // Returns true.
1 == 0          // Returns false.
1 == 1          // Returns true.
false == true   // Returns false.
false == false  // Returns true.
```

В отличие от Java, оператор сравнения равенства работает со строками.

Оператор сравнения равенства будет работать с операндами разных типов, если неявный листинг существует от одного к другому. Если подходящий неявный бросок не существует, вы можете вызвать явное приведение или использовать метод для преобразования в совместимый тип.

```
1 == 1.0          // Returns true because there is an implicit cast from int to double.  
new Object() == 1.0 // Will not compile.  
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

В отличие от Visual Basic.NET оператор сравнения равенства не совпадает с оператором присваивания равенства.

```
var x = new Object();  
var y = new Object();  
x == y // Returns false, the operands (objects in this case) have different references.  
x == x // Returns true, both operands have the same reference.
```

Не путать с оператором присваивания (=).

Для типов значений оператор возвращает `true` если оба операнда равны по значению. Для ссылочных типов оператор возвращает `true` если оба операнда равны в ссылке (не значение). Исключением является то, что строковые объекты будут сравниваться со значением равенства.

Неравномерно

Проверяет, *не* равны ли поставленные операнды.

```
"a" != "b"      // Returns true.  
"a" != "a"      // Returns false.  
1 != 0          // Returns true.  
1 != 1          // Returns false.  
false != true   // Returns true.  
false != false  // Returns false.  
  
var x = new Object();  
var y = new Object();  
x != y // Returns true, the operands have different references.  
x != x // Returns false, both operands have the same reference.
```

Этот оператор фактически возвращает обратный результат к результату оператора `equals` (`==`)

Лучше чем

Проверяет, превышает ли первый операнд второй операнд.

```
3 > 5      // Returns false.
```

```
1 > 0      //Returns true.  
2 > 2      //Return false.  
  
var x = 10;  
var y = 15;  
x > y      //Returns false.  
y > x      //Returns true.
```

Меньше, чем

Проверяет, является ли первый операнд меньше второго операнда.

```
2 < 4      //Returns true.  
1 < -3     //Returns false.  
2 < 2      //Return false.  
  
var x = 12;  
var y = 22;  
x < y      //Returns true.  
y < x      //Returns false.
```

Больше, чем равно

Проверяет, является ли первый операнд больше, чем второй операнд.

```
7 >= 8     //Returns false.  
0 >= 0     //Returns true.
```

Меньше, чем равно

Проверяет, является ли первый операнд меньше, чем второй операнд.

```
2 <= 4      //Returns true.  
1 <= -3     //Returns false.  
1 <= 1      //Returns true.
```

Операторы короткого замыкания

По определению короткозамкнутые булевые операторы будут оценивать только второй операнд, если первый операнд не может определить общий результат выражения.

Это означает, что, если вы используете `&&` operator как `firstCondition && secondCondition`, он будет оценивать `secondCondition` только в том случае, когда `firstCondition` является истинным, а `fromcourse` - общий результат будет истинным, только если оба `firstOperand` и `secondOperand` оценены как `true`. Это полезно во многих сценариях, например, представьте, что вы хотите проверить, в то время как ваш список содержит более трех элементов, но вы также должны проверить, был ли список инициализирован, чтобы не запускаться в `исключение NullReferenceException`. Вы можете достичь этого, как показано ниже:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mList.Count > 3 не будет проверяться до тех пор, пока *myList != null* не будет выполнен.

Логические И

`&&` является короткозамкнутым аналогом стандартного логического оператора AND (`&`).

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

Логический ИЛИ

`||` является короткозамкнутым аналогом стандартного булева оператора OR (`|`).

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

Пример использования

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
    Action1();
else
    Action2();
```

размер

Возвращает `int` содержащий размер типа `*` в байтах.

```
sizeof(bool)      // Returns 1.
sizeof(byte)      // Returns 1.
sizeof(sbyte)     // Returns 1.
sizeof(char)      // Returns 2.
sizeof(short)     // Returns 2.
sizeof(ushort)    // Returns 2.
sizeof(int)       // Returns 4.
sizeof(uint)      // Returns 4.
sizeof(float)     // Returns 4.
sizeof(long)      // Returns 8.
sizeof(ulong)     // Returns 8.
sizeof(double)    // Returns 8.
sizeof(decimal)   // Returns 16.
```

* Поддерживает только определенные примитивные типы в безопасном контексте.

В небезопасном контексте `sizeof` может использоваться для возврата размера других примитивных типов и структур.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Перегрузка операторов равенства

Недостаточно перегрузки только операторов равенства. В разных обстоятельствах можно назвать все следующие:

1. `object.Equals` И `object.GetHashCode`
2. `IEquatable<T>.Equals` (необязательно, позволяет избежать бокса)
3. `operator ==` И `operator !=` (необязательно, позволяет использовать операторы)

При переопределении `Equals`, `GetHashCode` также должен быть переоценен. При реализации `Equals` существует множество особых случаев: по сравнению с объектами другого типа, по сравнению с самим собой.

Когда НЕ переопределенный метод `Equals` и оператор `==` работают по-разному для классов и структур. Для классов сравниваются только ссылки, а для структур значения свойств сравниваются посредством отражения, что может отрицательно повлиять на производительность. `==` не может использоваться для сравнения структур, если он не переопределен.

Как правило, операция равенства должна подчиняться следующим правилам:

- Не нужно бросать исключения .
- Рефлексивность: `a` всегда равна `a` (может быть неверна для значений `NULL` в некоторых системах).
- Транзитивность: если `a` равно `b`, а `b` равно `c`, то `a` равно `c`
- Если `a` равно `b`, то `a` и `b` имеют одинаковые хэш-коды.
- Независимость дерева наследования: если `b` и `c` являются экземплярами `Class2` унаследованными от `Class1`: `Class1.Equals(A, B)` всегда должны возвращать то же значение, что и вызов `Class2.Equals(A, B)` .

```

class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}

```

Операторы класса: членский доступ

```

var now = DateTime.UtcNow;
//accesses member of a class. In this case the UtcNow property.

```

Операторы класса: Null Conditional Member Access

```

var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;

```

Операторы класса: вызов функции

```

var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.

```

Операторы класса: индексирование агрегированных объектов

```
var letters = "letters".ToCharArray();
char letter = letters[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

Операторы класса: нулевая условная индексация

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example rather than throwing an error because letters is null
//letter is assigned the value null
```

«Эксклюзив» или «Оператор»

Оператор для «исключительного» или «(для короткого XOR): ^

Этот оператор возвращает true, когда один, но только один, из предоставленных bools являются истинными.

```
true ^ false    // Returns true
false ^ true    // Returns true
false ^ false   // Returns false
true ^ true     // Returns false
```

Операторы битового сдвига

Операторы сдвига позволяют программистам настраивать целое число, сдвигая все его биты влево или вправо. Следующая диаграмма показывает влияние сдвига значения слева на одну цифру.

Сдвиг влево

```
uint value = 15;           // 00001111
uint doubled = value << 1; // Result = 00011110 = 30
uint shiftFour = value << 4; // Result = 11110000 = 240
```

Правый-Shift

```
uint value = 240;          // 11110000
uint halved = value >> 1; // Result = 01111000 = 120
uint shiftFour = value >> 4; // Result = 00001111 = 15
```

Неявные трансляции и явные операторы

C # позволяет пользовательским типам управлять назначением и кастом посредством использования `explicit` и `implicit` ключевых слов. Подпись метода принимает вид:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

Метод не может принимать больше аргументов и не может быть методом экземпляра. Тем не менее, он может получить доступ к любым частным членам типа, которые определены внутри.

Пример как `implicit` и `explicit` приведения в действие:

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
    {
        return im._pixels;
    }
}
```

Разрешить следующий синтаксис:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

Операторы литья могут работать в обоих направлениях, переходя от вашего типа и переходя к вашему типу:

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

И, наконец, `as` ключевых слов, которые могут быть вовлечены в литеином в иерархии типов, не действуют в этой ситуации. Даже после определения `explicit` или `implicit` броска

вы не можете:

```
ColorImage cm = myBinaryImage as ColorImage;
```

Он будет генерировать ошибку компиляции.

Бинарные операторы с присвоением

C # имеет несколько операторов, которые могут быть объединены с знаком = чтобы оценить результат оператора, а затем назначить результат исходной переменной.

Пример:

```
x += y
```

такой же как

```
x = x + y
```

Операторы присваивания:

- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- <<=
- >>=

? : Тернарный оператор

Возвращает одно из двух значений в зависимости от значения булевого выражения.

Синтаксис:

```
condition ? expression_if_true : expression_if_false;
```

Пример:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

Тернарный оператор является право-ассоциативным, что позволяет использовать сложные тернарные выражения. Это делается путем добавления дополнительных тернарных уравнений в истинную или ложную позицию родительского тройного уравнения. Следует

проявлять осторожность, чтобы обеспечить читаемость, но в некоторых случаях это может быть полезным сокращением.

В этом примере сложная тройная операция оценивает функцию `clamp` и возвращает текущее значение, если оно находится в пределах диапазона, `min` значение, если оно находится ниже диапазона, или `max` значение, если оно находится выше диапазона.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

Тернарные операторы также могут быть вложенными, например:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

При написании сложных тернарных утверждений обычно используется скобка или отступы для улучшения удобочитаемости.

Типы `expression_if_true` и `expression_if_false` должны быть идентичными или должны быть явным преобразованием из одного в другое.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
                             conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
                     ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
                     ternary operator will return a `double`.
```

Требования типа и преобразования применяются также к вашим собственным классам.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
```

```
'SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit
conversion from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough
to realize that both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate
to a reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

ТИП

Возвращает объект `System.Type` для типа.

```
System.Type type = typeof(Point)           //System.Drawing.Point
System.Type type = typeof(IDisposable)      //System.IDisposable
System.Type type = typeof(Colors)           //System.Drawing.Color
System.Type type = typeof(List<T>)         //System.Collections.Generic.List`1[T]
```

Чтобы получить тип времени выполнения, используйте метод `GetType` для получения `System.Type` текущего экземпляра.

Оператор `typeof` принимает имя типа в качестве параметра, который задается во время компиляции.

```
public class Animal {}
public class Dog : Animal {}

var animal = new Dog();

Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
Assert.IsTrue(animal.GetType() == typeof(Dog));     // pass, animal is typeof(Dog)
Assert.IsTrue(animal is Animal);                   // pass, animal implements Animal
```

по умолчанию

Тип значения (где T: struct)

Встроенные примитивные типы данных, такие как `char`, `int` и `float`, а также пользовательские типы, объявленные с помощью `struct` или `enum`. Их значением по умолчанию является `new T()`:

```
default(int)           // 0
default(DateTime)      // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
```

```
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)           // (MyEnum) 0
```

Тип ссылки (где T: класс)

Любой `class`, `interface`, массив или тип делегата. Их значение по умолчанию равно `null`:

```
default(object)           // null
default(string)            // null
default(MyClass)           // null
default(IDisposable)       // null
default(dynamic)           // null
```

имя оператора

Возвращает строку, которая представляет неквалифицированное имя `variable`, `type` или `member`.

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

Оператор `nameof` был введен в C # 6.0. Он оценивается во время компиляции, а возвращаемое строковое значение вставляется встроенным компилятором, поэтому его можно использовать в большинстве случаев, где может использоваться константная строка (например, метки `case` в инструкции `switch`, атрибуты и т. д.). Это может быть полезно в таких случаях, как повышение и регистрация исключений, атрибутов, ссылок на действия MVC и т. д. ...

? . (Null условный оператор)

6,0

Представлен в C # 6.0, Null Conditional Operator `?`. будет немедленно возвращать значение `null` если выражение в левой части оценивается как `null`, а не бросает `NullReferenceException`. Если его левая часть оценивается как `null` значение, она рассматривается как нормальная . оператор. Обратите внимание, что поскольку он может возвращать значение `null`, его возвращаемый тип всегда является типом с `null` значением. Это означает, что для структурного или примитивного типа он завернут в `Nullable<T>`.

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

Это удобно при запуске событий. Обычно вам придется обернуть вызов события в инструкции `if`, проверяющей значение `null` и затем поднять событие, что будет означать

возможность гонки. Используя условный оператор Null, это можно зафиксировать следующим образом:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Приращение и уменьшение префиксов и префиксов

Постерическое приращение `x++` добавит 1 к x

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Постфиксный декремент `x--` вычитает один

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` называется приращением префикса, он увеличивает значение x, а затем возвращает x, в то время как `x++` возвращает значение x, а затем увеличивает

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

в то время как

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

оба они обычно используются в цикле for

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

=> Лямбда-оператор

3.0

Оператор `=>` имеет тот же приоритет, что и оператор присваивания `=` и является правоассоциативным.

Он используется для объявления лямбда-выражений, а также широко используется с запросами LINQ :

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

При использовании в расширениях LINQ или запросах тип объектов обычно может быть пропущен, поскольку он выводится компилятором:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

Общая форма лямбда-оператора следующая:

```
(input parameters) => expression
```

Параметры выражения лямбда указаны перед оператором `=>`, а фактическое выражение / оператор / блок, который должен быть выполнен, находится справа от оператора:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
    x += " says Hello!";
    Console.WriteLine(x);
}
```

Этот оператор может использоваться для простого определения делегатов без написания явного метода:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

Вместо

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

Оператор присваивания '='

Оператор присваивания = устанавливает значение левого операнда в значение правого операнда и возвращает это значение:

```
int a = 3;      // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

?? Оператор Null-Coalescing

Нуль-Коалесцентный оператор ?? будет возвращать левую сторону, если не null. Если он равен нулю, он вернет правую часть.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

The ?? оператор может быть цепным, что позволяет удалить, if проверки.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
    RetrieveConfigFromService() ??
    new DefaultConfiguration();
```

Прочтайте операторы онлайн: <https://riptutorial.com/ru/csharp/topic/18/операторы>

глава 121: отражение

Вступление

Reflection - это механизм языка C # для доступа к свойствам динамических объектов во время выполнения. Как правило, отражение используется для получения информации о динамических типах объектов и значениях атрибутов объекта. Например, в приложении REST отражение может использоваться для итерации через сериализованный объект ответа.

Примечание. Согласно рекомендациям MS, критический код должен избегать отражения. См. <Https://msdn.microsoft.com/en-us/library/ff647790.aspx>.

замечания

Reflection позволяет коду получать информацию о сборках, модулях и типах во время выполнения (выполнение программы). Затем его можно использовать для динамического создания, изменения или доступа к типам. Типы включают свойства, методы, поля и атрибуты.

Дальнейшее чтение :

[Отражение \(C #\)](#)

[Отражение в .Net Framework](#)

Examples

Получить System.Type

Для экземпляра типа:

```
var theString = "hello";
var theType = theString.GetType();
```

Из самого типа:

```
var theType = typeof(string);
```

Получить членов типа

```
using System;
using System.Reflection;
```

```

using System.Linq;

public class Program
{
    public static void Main()
    {
        var members = typeof(object)
            .GetMembers(BindingFlags.Public |
                BindingFlags.Static |
                BindingFlags.Instance);

        foreach (var member in members)
        {
            bool inherited = member.DeclaringType.Equals( typeof(object).Name );
            Console.WriteLine($"'{member.Name}' is a {member.MemberType}, " +
                $"it has {(inherited ? ":" not)} been inherited.");
        }
    }
}

```

Выход (см. Примечание о порядке вывода ниже):

```

GetType is a Method, it has not been inherited.
GetHashCode is a Method, it has not been inherited.
ToString is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
ReferenceEquals is a Method, it has not been inherited.
.ctor is a Constructor, it has not been inherited.

```

Мы также можем использовать `GetMembers()` без передачи каких-либо `BindingFlags`. Это вернет всех публичных членов этого конкретного типа.

Следует отметить, что `GetMembers` не возвращает членов в каком-либо конкретном порядке, поэтому никогда не полагайтесь на то, что `GetMembers` вернет вас.

[Посмотреть демо](#)

Получить метод и вызвать его

Получить экземпляр метода и вызвать его

```

using System;

public class Program
{
    public static void Main()
    {
        var theString = "hello";
        var method = theString
            .GetType()
            .GetMethod("Substring",
                new[] {typeof(int), typeof(int)}); //The types of the method
        arguments
        var result = method.Invoke(theString, new object[] {0, 4});
    }
}

```

```
        Console.WriteLine(result);
    }
}
```

Выход:

ад

[Посмотреть демо](#)

Получить метод Static и вызвать его

С другой стороны, если метод статичен, для его вызова не требуется экземпляр.

```
var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] { 2}); //Pass null as the first argument (no need
for an instance)
Console.WriteLine(result); //You'll get e^2
```

Выход:

+7,38905609893065

[Посмотреть демо](#)

Получение и настройка свойств

Основное использование:

```
 PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
 // get the value myInstance.myProperty
 object value = prop.GetValue(myInstance);

 int newValue = 1;
 // set the value myInstance.myProperty to newValue
 prop.SetValue(myInstance, newValue);
```

Настройка автоматически реализованных свойств только для чтения может быть выполнена через его поле поддержки (в .NET Framework имя поля поддержки - «k__BackingField»):

```
// get backing field info
 FieldInfo fieldInfo = myInstance.GetType()
 .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

 int newValue = 1;
 // set the value of myInstance.myProperty backing field to newValue
 fieldInfo.SetValue(myInstance, newValue);
```

Пользовательские атрибуты

Поиск свойств с помощью настраиваемого атрибута - MyAttribute

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |  
    BindingFlags.Instance).Where(  
    prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

Найти все пользовательские атрибуты для данного свойства

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

Перечислить все классы с помощью настраиваемого атрибута - MyAttribute

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {  
    foreach(Type type in assembly.GetTypes()) {  
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {  
            yield return type;  
        }  
    }  
}
```

Чтение значения настраиваемого атрибута во время выполнения

```
public static class AttributeExtensions  
{  
  
    ///<summary>  
    /// Returns the value of a member attribute for any member in a class.  
    /// (a member is a Field, Property, Method, etc...)  
    ///<remarks>  
    /// If there is more than one member of the same name in the class, it will return the  
    /// first one (this applies to overloaded methods)  
    ///</remarks>  
    ///<example>  
    /// Read System.ComponentModel.Description Attribute from method 'MyMethodName' in  
    class 'MyClass':  
        /// var Attribute = typeof(MyClass).GetAttribute("MyMethodName",  
        (DescriptionAttribute d) => d.Description);  
    ///</example>  
    ///<param name="type">The class that contains the member as a type</param>  
    ///<param name="MemberName">Name of the member in the class</param>  
    ///<param name="valueSelector">Attribute type and property to get (will return first  
    instance if there are multiple attributes of the same type)</param>  
    ///<param name="inherit">true to search this member's inheritance chain to find the  
    attributes; otherwise, false. This parameter is ignored for properties and events</param>  
    ///</summary>  
    public static TValue GetAttribute<TAttribute, TValue>(this Type type, string  
    MemberName, Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute :  
    Attribute  
    {  
        var att =  
        type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),  
        inherit).FirstOrDefault() as TAttribute;  
        if (att != null)  
        {  
            return valueSelector(att);  
        }  
    }  
}
```

```
        return default(TValue);
    }
}
```

использование

```
//Read System.ComponentModel.Description Attribute from method 'MyMethodName' in class
'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

Пересечение всех свойств класса

```
Type type = obj.GetType();
//To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
 PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}
```

Определение общих аргументов экземпляров родовых типов

Если у вас есть экземпляр родового типа, но по какой-то причине не знаю определенного типа, вы можете определить общие аргументы, которые были использованы для создания этого экземпляра.

Предположим, кто-то создал экземпляр `List<T>` и передал его методу:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

где `ShowGenericArguments` имеет эту подпись:

```
public void ShowGenericArguments(object o)
```

поэтому во время компиляции вы не знаете, какие общие аргументы были использованы для создания `o`. [Отражение](#) предоставляет множество методов для проверки типичных типов. Сначала мы можем определить, является ли тип `o` общим типом:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType` возвращает `true` если тип является общим типом и `false` если нет.

Но это еще не все, что мы хотим знать. `List<>` сам также является общим типом. Но мы хотим только изучить экземпляры конкретных построенных общих типов. Построенный общий тип - это, например, `List<int>` который имеет аргумент определенного типа для всех его общих параметров .

Класс `Type` предоставляет еще два свойства: `IsConstructedGenericType` и `IsGenericTypeDefinition`, чтобы отличать эти построенные общие типы от общих описаний типов:

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
typeof(List<>).IsConstructedGenericType// false

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true
```

Чтобы перечислить общие аргументы экземпляра, мы можем использовать метод `GetGenericArguments()` который возвращает массив `Type` содержащий аргументы общего типа:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach (Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}
```

Таким образом, вызов сверху (`ShowGenericArguments(myList)`) приводит к этому выводу:

```
Int32
```

Получить общий метод и вызвать его

Предположим, у вас есть класс с общими методами. И вам нужно вызвать его функции с отражением.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Предположим, мы хотим вызвать `GenericMethod` с строкой типа.

```
Sample sample = new Sample(); //or you can get an instance via reflection
MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null); //Since there are no arguments, we are passing null
```

Для статического метода вам не нужен экземпляр. Поэтому первый аргумент также будет нулевым.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

Создайте экземпляр типа Generic и вызовите его метод

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

Создание экземпляров классов, реализующих интерфейс (например, активация плагина)

Если вы хотите, чтобы ваше приложение поддерживало подключаемую систему, например, для загрузки плагинов из сборок, расположенных в папке `plugins`:

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

Этот класс будет располагаться в отдельной dll

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

Плагин-загрузчик вашего приложения найдет файлы DLL, получит все типы в тех сборках, которые реализуют `IPlugin`, и создаст экземпляры этих файлов.

```
public IEnumerable<IPlugin> InstantiatePlugins(string directory)
```

```

{
    var pluginAssemblyNames = Directory.GetFiles(directory, "*.addin.dll").Select(name =>
    new FileInfo(name).FullName).ToArray();
    //load the assemblies into the current AppDomain, so we can instantiate the types
    later
    foreach (var fileName in pluginAssemblyNames)
        AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
    var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
    var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
    var pluginTypes = typesInAssembly.Where(type => typeof
    (IPlugin).IsAssignableFrom(type));
    return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
}

```

Создание экземпляра типа

Самый простой способ - использовать класс `Activator`.

Однако, несмотря на то, что производительность `Activator` была улучшена с .NET 3.5, использование `Activator.CreateInstance()` иногда является плохим вариантом из-за (относительно) низкой производительности: [Test 1](#) , [Test 2](#) , [Test 3](#) ...

С классом `Activator`

```

Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an
'int' compatible argument.
Console.WriteLine(result); //Output: 123

```

Вы можете передать массив объектов `Activator.CreateInstance` если у вас есть несколько параметров.

```

// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] { 1, 2, "Hello World" });

Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);

```

Для общего типа

Метод `MakeGenericType` превращает открытый общий тип (например, `List<>`) в конкретный тип (например, `List<string>`), применяя к нему аргументы типа.

```

// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };

```

```
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

Синтаксис `List<>` не допускается вне выражения `typeof`.

Без класса `Activator`

Использование `new` ключевого слова (сделайте для конструкторов без параметров)

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

Использование метода `Invoke`

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

Использование деревьев выражений

Деревья выражений представляют код в древовидной структуре данных, где каждый узел является выражением. Как объясняет [MSDN](#) :

Выражение представляет собой последовательность из одного или нескольких операндов и ноль или более операторов, которые могут быть оценены по одному значению, объекту, методу или пространству имен. Выражения могут состоять из буквального значения, вызова метода, оператора и его operandов или простого имени. Простыми именами могут быть имя переменной, член типа, параметр метода, пространство имен или тип.

```
public class GenericFactory<TKey, TType>
{
    private readonly Dictionary<TKey, Func<object[], TType>> _registeredTypes; // dictionary, that holds constructor functions.
    private object _locker = new object(); // object for locking dictionary, to guarantee thread safety

    public GenericFactory()
    {
        _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
```

```

    }

    /// <summary>
    /// Find and register suitable constructor for type
    /// </summary>
    /// <typeparam name="TType"></typeparam>
    /// <param name="key">Key for this constructor</param>
    /// <param name="parameters">Parameters</param>
    public void Register(TKey key, params Type[] parameters)
    {
        ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] { });
    } // Get the instance of ctor.
    if (ci == null)
        throw new InvalidOperationException(string.Format("Constructor for type '{0}' was not found.", typeof(TType)));
    Func<object[], TType> ctor;

    lock (_locker)
    {
        if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor already been registered
        {
            var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create parameter Expression
            var ctorParams = ci.GetParameters(); // get parameter info from constructor

            var argExpressions = new Expression[ctorParams.Length]; // array that will contains parameter expressions
            for (var i = 0; i < parameters.Length; i++)
            {

                var indexedAcccess = Expression.ArrayIndex(pExp,
Expression.Constant(i));

                if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if parameter is a value type
                {
                    var localVariable = Expression.Variable(parameters[i],
"localVariable"); // if so - we should create local variable that will store paraameter value

                    var block = Expression.Block(new[] { localVariable },
Expression.IfThenElse(Expression.Equal(indexedAcccess,
Expression.Constant(null)),
Expression.Assign(localVariable,
Expression.Default(parameters[i])),
Expression.Assign(localVariable,
Expression.Convert(indexedAcccess, parameters[i]))
),
localVariable
);

                    argExpressions[i] = block;
                }
                else
                    argExpressions[i] = Expression.Convert(indexedAcccess,
parameters[i]);
            }
        }
    }
}

```

```

        var newExpr = Expression.New(ci, argExpressions); // create expression
        that represents call to specified ctor with the specified arguments.

        _registeredTypes.Add(key, Expression.Lambda(new[] { pExp
    }).Compile() as Func<object[], TType>); // compile expression to create delegate, and add
    fucntion to dictionary
    }
}
}

/// <summary>
/// Returns instance of registered type by key.
/// </summary>
/// <typeparam name="TType"></typeparam>
/// <param name="key"></param>
/// <param name="args"></param>
/// <returns></returns>
public TType Create(TKey key, params object[] args)
{
    Func<object[], TType> foo;
    if (_registeredTypes.TryGetValue(key, out foo))
    {
        return (TType) foo(args);
    }

    throw new ArgumentException("No type registered for this key.");
}
}
}

```

Может использоваться следующим образом:

```

public class TestClass
{
    public TestClass(string parameter)
    {
        Console.Write(parameter);
    }
}

public void TestMethod()
{
    var factory = new GenericFactory<string, TestClass>();
    factory.Register("key", typeof(string));
    TestClass newInstance = factory.Create("key", "testParameter");
}

```

Использование **FormatterServices.GetUninitializedObject**

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

В случае использования **FormatterServices.GetUninitializedObject** **конструкторы и инициализаторы полей не будут вызываться**. Он предназначен для использования в сериализаторах и удаленных двигателях

Получить тип по имени с пространством имен

Для этого вам нужна ссылка на сборку, которая содержит этот тип. Если у вас есть другой тип, который, как вы знаете, находится в той же сборке, что и тот, который вы хотите, вы можете сделать это:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- где `typeName` - это имя типа, который вы ищете (включая пространство имен), а `KnownType` - тот тип, который, как вы знаете, находится в той же сборке.

Менее эффективным, но более общим является следующее:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

Обратите внимание на проверку, чтобы исключить возможности сканирования системного пространства `System` namespace для ускорения поиска. Если ваш тип может быть CLR-типов, вам придется удалить эти две строки.

Если у вас есть полное имя типа сборки, включая сборку, вы можете просто получить ее с помощью

```
Type.GetType(fullyQualifiedName);
```

Получить строго типизированного делегата для метода или свойства посредством отражения

Когда производительность вызывает беспокойство, вызов метода через отражение (`t MethodInfo.Invoke` `Методом MethodInfo.Invoke`) не идеален. Тем не менее, относительно просто получить более производительный сильно типизированный делегат, используя функцию `Delegate.CreateDelegate`. Снижение производительности за использование отражения происходит только во время процесса создания делегата. Как только делегат создан, для его активации есть мало шансов на производительность:

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int, int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

Этот метод можно распространить и на свойства. Если у нас есть класс с именем MyClass с свойством int именем MyIntProperty, то код для получения сильно типизированного getter будет (следующий пример предполагает, что «target» является допустимым экземпляром MyClass):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass,
int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

... и то же самое можно сделать для сеттера:

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass,
int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

Прочтайте отражение онлайн: <https://riptutorial.com/ru/csharp/topic/28/отражение>

глава 122: Параллельная библиотека задач

Examples

Parallel.ForEach

Пример, который использует цикл Parallel.ForEach для ping заданного массива URL-адресов веб-сайта.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

Parallel.For

Пример, который использует цикл Parallel.For для пинга заданного массива URL-адресов веб-сайта.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(urls[i]);
    });
}
```

```

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", urls[i]));
    }
});
}

```

Parallel.Invoke

Вызов методов или действий параллельно (Параллельная область)

```

static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", url));
    }
}

```

Асинхронный аннулируемый опрос Задача, ожидающая между итерациями

```

public class Foo
{
    private const int TASK_ITERATION_DELAY_MS = 1000;
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {

```

```

        Task.Factory.StartNew(this.OwnCodeCancelableTask_EveryNSeconds, this._cts.Token);
    }

    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop that runs every N seconds. Good for checking for a heartbeat or
    updates.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
    StartNew call</param>
    private async void OwnCodeCancelableTask_EveryNSeconds(object taskState)
    {
        var token = (CancellationToken)taskState;

        while (!token.IsCancellationRequested)
        {
            Console.WriteLine("Do the work that needs to happen every N seconds in this
loop");

            // Passing token here allows the Delay to be cancelled if your task gets
            cancelled.
            await Task.Delay(TASK_ITERATION_DELAY_MS, token);
        }
    }
}

```

Задача аннулирования опроса с использованием CancellationTokenSource

```

public class Foo
{
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask, this._cts.Token);
    }

    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop with no delays. Writing to a database while pulling from a buffer for
    example.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
    StartNew call</param>
    private void OwnCodeCancelableTask(object taskState)

```

```

    {
        var token = (CancellationToken) taskState; //Our cancellation token passed from
        StartNew();

        while ( !token.IsCancellationRequested )
        {
            Console.WriteLine("Do your task work in this loop");
        }
    }
}

```

Асинхронная версия PingUrl

```

static void Main(string[] args)
{
    string url = "www.stackoverflow.com";
    var pingTask = PingUrlAsync(url);
    Console.WriteLine($"Waiting for response from {url}");
    Task.WaitAll(pingTask);
    Console.WriteLine(pingTask.Result);
}

static async Task<string> PingUrlAsync(string url)
{
    string response = string.Empty;
    var ping = new System.Net.NetworkInformation.Ping();

    var result = await ping.SendPingAsync(url);

    await Task.Delay(5000); //simulate slow internet

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        response = $"{url} is online";
    }

    return response;
}

```

Прочтайте Параллельная библиотека задач онлайн:

<https://riptutorial.com/ru/csharp/topic/1010/параллельная-библиотека-задач>

глава 123: Параллельный LINQ (PLINQ)

Синтаксис

- ParallelEnumerable.Aggregate (FUNC)
- ParallelEnumerable.Aggregate (seed, func)
- ParallelEnumerable.Aggregate (seed, updateAccumulatorFunc, combAccumulatorsFunc, resultSelector)
- ParallelEnumerable.Aggregate (seedFactory, updateAccumulatorFunc, combAccumulatorsFunc, resultSelector)
- ParallelEnumerable.All (предикат)
- ParallelEnumerable.Any ()
- ParallelEnumerable.Any (предикат)
- ParallelEnumerable.AsEnumerable ()
- ParallelEnumerable.AsOrdered ()
- ParallelEnumerable.AsParallel ()
- ParallelEnumerable.AsSequential ()
- ParallelEnumerable.AsUnordered ()
- ParallelEnumerable.Average (селектор)
- ParallelEnumerable.Cast ()
- ParallelEnumerable.Concat (второй)
- ParallelEnumerable.Contains (значение)
- ParallelEnumerable.Contains (значение, сравнение)
- ParallelEnumerable.Count ()
- ParallelEnumerable.Count (предикат)
- ParallelEnumerable.DefaultIfEmpty ()
- ParallelEnumerable.DefaultIfEmpty (DefaultValue)
- ParallelEnumerable.Distinct ()
- ParallelEnumerable.Distinct (Comparer)
- ParallelEnumerable.ElementAt (индекс)
- ParallelEnumerable.ElementAtOrDefault (индекс)
- ParallelEnumerable.Empty ()
- ParallelEnumerable.Except (второй)
- ParallelEnumerable.Except (во-вторых, сравнение)
- ParallelEnumerable.First ()
- ParallelEnumerable.First (предикат)
- ParallelEnumerable.FirstOrDefault ()
- ParallelEnumerable.FirstOrDefault (предикат)
- ParallelEnumerable.ForAll (действие)
- ParallelEnumerable.GroupBy (keySelector)
- ParallelEnumerable.GroupBy (keySelector, comparer)
- ParallelEnumerable.GroupBy (keySelector, elementSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, comparer)

- ParallelEnumerable.GroupBy (keySelector, resultSelector)
- ParallelEnumerable.GroupBy (keySelector, resultSelector, comparer)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector, comparer)
- ParallelEnumerable.GroupJoin (внутренний, внешнийKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.GroupJoin (внутренний, внешнийKeySelector, innerKeySelector, resultSelector, comparer)
- ParallelEnumerable.Intersect (второй)
- ParallelEnumerable.Intersect (второй, сравнительный)
- ParallelEnumerable.Join (внутренний, внешнийKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.Join (внутренний, внешнийKeySelector, innerKeySelector, resultSelector, comparer)
- ParallelEnumerable.Last ()
- ParallelEnumerable.Last (предикат)
- ParallelEnumerable.LastOrDefault ()
- ParallelEnumerable.LastOrDefault (предикат)
- ParallelEnumerable.LongCount ()
- ParallelEnumerable.LongCount (предикат)
- ParallelEnumerable.Max ()
- ParallelEnumerable.Max (селектор)
- ParallelEnumerable.Min ()
- ParallelEnumerable.Min (селектор)
- ParallelEnumerable.OfType ()
- ParallelEnumerable.OrderBy (keySelector)
- ParallelEnumerable.OrderBy (keySelector, comparer)
- ParallelEnumerable.OrderByDescending (keySelector)
- ParallelEnumerable.OrderByDescending (keySelector, comparer)
- ParallelEnumerable.Range (начало, количество)
- ParallelEnumerable.Repeat (элемент, счетчик)
- ParallelEnumerable.Reverse ()
- ParallelEnumerable.Select (селектор)
- ParallelEnumerable.SelectMany (селектор)
- ParallelEnumerable.SelectMany (collectionSelector, resultSelector)
- ParallelEnumerable.SequenceEqual (второй)
- ParallelEnumerable.SequenceEqual (второй, сравнительный)
- ParallelEnumerable.Single ()
- ParallelEnumerable.Single (предикат)
- ParallelEnumerable.SingleOrDefault ()
- ParallelEnumerable.SingleOrDefault (предикат)
- ParallelEnumerable.Skip (количество)
- ParallelEnumerable.SkipWhile (предикат)
- ParallelEnumerable.Sum ()

- ParallelEnumerable.Sum (селектор)
- ParallelEnumerable.Take (количество)
- ParallelEnumerable.TakeWhile (предикат)
- ParallelEnumerable.ThenBy (keySelector)
- ParallelEnumerable.ThenBy (keySelector, comparer)
- ParallelEnumerable.ThenByDescending (keySelector)
- ParallelEnumerable.ThenByDescending (keySelector, comparer)
- ParallelEnumerable.ToArray ()
- ParallelEnumerable.ToDictionary (keySelector)
- ParallelEnumerable.ToDictionary (keySelector, comparer)
- ParallelEnumerable.ToDictionary (ElementSelector)
- ParallelEnumerable.ToDictionary (elementSelector, comparer)
- ParallelEnumerable.ToList ()
- ParallelEnumerable.ToLookup (keySelector)
- ParallelEnumerable.ToLookup (keySelector, comparer)
- ParallelEnumerable.ToLookup (keySelector, elementSelector)
- ParallelEnumerable.ToLookup (keySelector, elementSelector, comparer)
- ParallelEnumerable.Union (второй)
- ParallelEnumerable.Union (второй, сравнительный)
- ParallelEnumerable.Where (предикат)
- ParallelEnumerable.WithCancellation (CancellationToken)
- ParallelEnumerable.WithDegreeOfParallelism (degreeOfParallelism)
- ParallelEnumerable.WithExecutionMode (executionMode)
- ParallelEnumerable.WithMergeOptions (mergeOptions)
- ParallelEnumerable.Zip (второй, resultSelector)

Examples

Простой пример

В этом примере показано, как PLINQ может использоваться для вычисления четных чисел от 1 до 10000 с использованием нескольких потоков. Обратите внимание, что результирующий список не будет заказан!

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .Where(x => x % 2 == 0)
    .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

WithDegreeOfParallelism

Степень параллелизма - это максимальное количество одновременно выполняемых задач, которые будут использоваться для обработки запроса.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .WithDegreeOfParallelism(4)
    .Where(x => x % 2 == 0);
```

AsOrdered

В этом примере показано, как PLINQ может использоваться для вычисления четных чисел от 1 до 10000 с использованием нескольких потоков. Заказ будет сохранен в результирующем списке, однако имейте в виду, что `AsOrdered` может повредить производительность для большого количества элементов, поэтому `AsOrdered` обработка является предпочтительной, когда это возможно.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .AsOrdered()
    .Where(x => x % 2 == 0)
    .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

AsUnordered

Упорядоченные последовательности могут повредить производительность при работе с большим количеством элементов. Чтобы смягчить это, можно вызвать `AsUnordered` когда порядок последовательности больше не нужен.

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
    .OrderBy(x => x)
    .Take(5000)
    .AsUnordered()
    .Where(x => x % 2 == 0) // This line won't be affected by ordering
    .ToList();
```

Прочтайте Параллельный LINQ (PLINQ) онлайн: <https://riptutorial.com/ru/csharp/topic/3569/параллельный-linq--plinq->

глава 124: перекручивание

Examples

Стиль петли

В то время как

Самый тривиальный тип цикла. Единственный недостаток - нет никакой внутренней подсказки, чтобы знать, где вы находитесь в цикле.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

Делать

Подобно `while`, но условие оценивается в конце цикла вместо начала. Это приводит к выполнению циклов хотя бы один раз.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

За

Еще один тривиальный стиль цикла. Цикл индекса (`i`) увеличивается, и вы можете его использовать. Он обычно используется для обработки массивов.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

Для каждого

Модернизированный способ петли через `IEnumerable` объекты. Хорошо, что вам не нужно думать об индексе элемента или о количестве элементов списка.

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

Метод Foreach

В то время как другие стили используются для выбора или обновления элементов в коллекциях, этот стиль обычно используется для *вызыва метода* сразу для всех элементов в коллекции.

```
list.ForEach(item => item.DoSomething());  
  
// or  
list.ForEach(item => DoSomething(item));  
  
// or using a method group  
list.ForEach(Console.WriteLine);  
  
// using an array  
Array.ForEach(myArray, Console.WriteLine);
```

Важно отметить, что этот метод доступен только в экземплярах `List<T>` и как статический метод для `Array` - он **не** является частью `Linq`.

Linq Parallel Foreach

Точно так же, как `Linq Foreach`, за исключением того, что эта задача выполняется параллельно. Это означает, что все элементы в коллекции будут одновременно выполнять задание одновременно.

```
collection.AsParallel().ForAll(item => item.DoSomething());  
  
/// or  
collection.AsParallel().ForAll(item => DoSomething(item));
```

перерыв

Иногда условие цикла должно быть проверено в середине цикла. Первый, возможно, более изящный, чем последний:

```
for (;;) {  
    // precondition code that can change the value of should_end_loop expression  
  
    if (should_end_loop)  
        break;  
  
    // do something  
}
```

Альтернатива:

```
bool endLoop = false;  
for (; !endLoop;) {
```

```
// precondition code that can set endLoop flag

if (!endLoop)
{
    // do something
}
}
```

Примечание. В вложенных циклах и / или `switch` необходимо использовать не просто `break`.

Перекрестная петля

`foreach` будет перебирать любой объект класса, который реализует `IEnumerable` (обратите внимание, что `IEnumerable<T>` наследует от него). Такие объекты включают некоторые встроенные, но не ограничивают: `List<T>` , `T[]` (массивы любого типа), `Dictionary< TKey, TValue >` , а также интерфейсы, такие как `IQueryable` И `ICollection` И Т. Д.

Синтаксис

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

Замечания

1. Тип `ItemType` не должен соответствовать точному типу элементов, его просто необходимо назначить из типа элементов
2. Вместо `ItemType` можно также использовать `var` , который выведет тип элементов из `ItemType` , `ItemType` общий аргумент реализации `IEnumerable`
3. Оператором может быть блок, один оператор или даже пустой оператор (;)
4. Если `enumerableObject` не реализует `IEnumerable` , код не будет компилироваться
5. Во время каждой итерации текущий элемент `ItemType` в `ItemType` (даже если это не указано, а компилятор - выводится через `var`), и если элемент не может быть `InvalidCastException` будет `InvalidCastException` .

Рассмотрим этот пример:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

Эквивалентно:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
```

```
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
    {
        string name = (string)enumerator.Current;
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

Пока цикл

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Выход:

```
0
1
2
3
4
```

IEnumerators можно повторить с помощью цикла while:

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of theh largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

Пример вывода:

```
Токио / Йокогама
Нью-Йорк Метро
Сан-Паулу
Сеул / Инчхон
```

Для цикла

А For Loop отлично подходит для выполнения определенных задач. Это похоже на цикл While Loop, но приращение включено в условие.

А For Loop настроен следующим образом:

```
for (Initialization; Condition; Increment)
{
    // Code
}
```

Инициализация - создает новую локальную переменную, которая может использоваться только в цикле.

Условие. Цикл работает только тогда, когда условие истинно.

Приращение - как переменная изменяется при каждом запуске цикла.

Пример:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Выход:

```
0
1
2
3
4
```

Вы также можете оставить пробелы в For Loop, но для этого вам нужно иметь все точки с запятой.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Выход для 3:

```
3
5
7
9
11
```

Do - While Loop

Это похоже на `while` цикл, за исключением того, что он проверяет условие в конце тела цикла. Цикл Do - While выполняет цикл один раз независимо от того, является ли условие истинным или нет.

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// Sum values from the array until we get a total that's greater than 10,
// or until we run out of values.
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

Вложенные петли

```
// Print the multiplication table up to 5s
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        int product = i * j;
        Console.WriteLine("{0} times {1} is {2}", i, j, product);
    }
}
```

Продолжить

В дополнение к `break` существует также ключевое слово `continue`. Вместо полного прорыва цикла он просто пропустит текущую итерацию. Это может быть полезно, если вы не хотите, чтобы какой-либо код выполнялся, если задано определенное значение.

Вот простой пример:

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
}
```

Это приведет к:

```
9
10
```

Примечание. `Continue` часто очень полезно в циклах `while` или `do-while`. For-loops, с четко

определенными условиями выхода, могут не принести столько пользы.

Прочтайте перекручивание онлайн: <https://riptutorial.com/ru/csharp/topic/2064/перекручивание>

глава 125: перелив

Examples

Целочисленное переполнение

Максимальная емкость, которую может хранить целое число. И когда вы перейдете этот предел, он вернется к отрицательной стороне. Для `int` это 2147483647

```
int x = int.MaxValue;           //.MaxValue is 2147483647
x = unchecked(x + 1);          //make operation explicitly unchecked so that the example
also works when the check for arithmetic overflow/underflow is enabled in the project settings

Console.WriteLine(x);           //Will print -2147483648
Console.WriteLine(int.MinValue); //Same as Min value
```

Для любых целых чисел из этого диапазона используйте пространство имен `System.Numerics`, которое имеет тип данных `BigInteger`. Проверьте ссылку ниже для получения дополнительной информации [https://msdn.microsoft.com/en-us/library/system.numerics.biginteger\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx)

Переполнение во время работы

Переполнение также происходит во время операции. В следующем примере `x` - это `int`, `1` - `int` по умолчанию. Поэтому добавление является добавлением `int`. И результатом будет `int`. И это переполнится.

```
int x = int.MaxValue;           //.MaxValue is 2147483647
long y = x + 1;                //It will be overflowed
Console.WriteLine(y);           //Will print -2147483648
Console.WriteLine(int.MinValue); //Same as Min value
```

Вы можете предотвратить это, используя `1L`. Теперь `1` будет `long` и добавление будет `long` дополнением

```
int x = int.MaxValue;           //.MaxValue is 2147483647
long y = x + 1L;                //It will be OK
Console.WriteLine(y);           //Will print 2147483648
```

Вопросы для заказа

Переполнение в следующем коде

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L); //prints -1
```

Если в следующем коде нет переполнения

```
int x = int.MaxValue;  
Console.WriteLine(x + 1L + x); //prints 4294967295
```

Это связано с упорядочением операций слева направо. В первом фрагменте кода `x + x` переполняется, и после этого он становится `long`. С другой стороны, `x + 1L` становится `long` и после этого `x` добавляется к этому значению.

Прочтайте перелив онлайн: <https://riptutorial.com/ru/csharp/topic/3303/перелив>

глава 126: Платформа компилятора .NET (Roslyn)

Examples

Создание рабочей области из проекта MSBuild

Прежде чем продолжить, сначала запустите `Microsoft.CodeAnalysis.CSharp.Workspaces` nuget.

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

Чтобы загрузить существующий код в рабочую область, выполните компиляцию и отправьте сообщение об ошибках. После этого код будет находиться в памяти. Отсюда и синтаксическая, и семантическая стороны будут доступны для работы.

Дерево синтаксиса

Дерево синтаксиса - это неизменяемая структура данных, представляющая программу как дерево имен, команд и меток (как ранее было настроено в редакторе).

Например, предположим, что экземпляр `Microsoft.CodeAnalysis.Compilation` именем `compilation` был настроен. Существует несколько способов перечислить имена каждой переменной, объявленной в загруженном коде. Чтобы сделать это наивно, возьмите все фрагменты синтаксиса в каждом документе (метод `DescendantNodes`) и используйте Linq для выбора узлов, описывающих объявление переменных:

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

Каждый тип конструкции C # с соответствующим типом будет существовать в дереве синтаксиса. Чтобы быстро найти определенные типы, используйте окно Syntax Visualizer Visual Studio Visual Studio. Это будет интерпретировать текущий открытый документ как синтаксическое дерево Roslyn.

Семантическая модель

Семантическая модель предлагает более глубокий уровень интерпретации и понимания кода по сравнению с деревом синтаксиса. Если деревья синтаксиса могут указывать имена переменных, семантические модели также дают тип и все ссылки. Деревья синтаксиса замечают вызовы методов, но семантические модели дают ссылки на точное местоположение, объявленное методом (после применения разрешения перегрузки).

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379]"
            }
        }
    }
}
```

Это выводит список локальных переменных, используя дерево синтаксиса. Затем он консультируется с семантической моделью, чтобы получить полное имя типа и найти все ссылки каждой переменной.

Прочтите Платформа компилятора .NET (Roslyn) онлайн:

<https://riptutorial.com/ru/csharp/topic/4886/платформа-компилятора--net--roslyn->

глава 127: Полиморфизм

Examples

Другой пример полиморфизма

Полиморфизм является одним из столпов ООП. Поли происходит от греческого термина, что означает «множественные формы».

Ниже приведен пример, демонстрирующий полиморфизм. Класс `Vehicle` принимает несколько форм в качестве базового класса.

Производные классы `Ducati` и `Lamborghini` наследуют от `Vehicle` и переопределяют метод `Display()` базового класса для отображения собственных `NumberOfWheels`.

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is
{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is
{NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
}
```

```
    }  
}
```

Ниже приведен фрагмент кода, в котором проявляется полиморфизм. Объект создается для базового типа `Vehicle` использующее переменное `vehicle` в строке 1. Он вызывает метод базового класса `Display()` в строке 2 и отображает вывод, как показано.

```
static void Main(string[] args)  
{  
    Vehicle vehicle = new Vehicle();      //Line 1  
    vehicle.Display();                   //Line 2  
    vehicle = new Ducati();             //Line 3  
    vehicle.Display();                   //Line 4  
    vehicle = new Lamborghini();        //Line 5  
    vehicle.Display();                   //Line 6  
}
```

В строке 3 объект `vehicle` указывается на производный класс `Ducati` и вызывает его метод `Display()`, который отображает вывод, как показано. А вот полиморфное поведение, даже если объект `vehicle` имеет тип `Vehicle`, он вызывает производный метод класса `Display()` в качестве типа `Ducati` переопределяет базовый класс `Display()` метод, так как `vehicle` объект направлен в стороне `Ducati`.

Такое же объяснение применимо, когда оно вызывает метод `Display()` типа `Lamborghini`.

Результат показан ниже

```
The number of wheels for the Vehicle is 0      // Line 2  
The number of wheels for the Ducati is 2        // Line 4  
The number of wheels for the Lamborghini is 4    // Line 6
```

Типы полиморфизма

Полиморфизм означает, что операция может также применяться к значениям некоторых других типов.

Существует несколько типов полиморфизма:

- **Специальный полиморфизм:**

содержит `function overloading`. Цель состоит в том, что метод может использоваться с разными типами без необходимости генерации.

- **Параметрический полиморфизм:**

является использование общих типов. См. [Generics](#)

- **Подтипы:**

имеет целевое наследование класса для обобщения аналогичной функциональности

Специальный полиморфизм

Целью `Ad hoc polymorphism` является создание метода, который может быть вызван различными типами данных без необходимости преобразования типов в вызове функции или дженериках. Следующие методы `sumInt(par1, par2)` могут вызываться с различными типами данных и для каждой комбинации типов имеют собственную реализацию:

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

Вот пример вызова:

```
public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 )); // 3
    Console.WriteLine(sumInt("3","4")); // 7
    Console.WriteLine(sumInt("5", 6 )); // 11
    Console.WriteLine(sumInt( 7 , "8")); // 15
}
```

Подтипы

Подтипование - это использование наследования из базового класса для обобщения аналогичного поведения:

```
public interface Car{
    void refuel();
}

public class NormalCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}
```

Оба класса `NormalCar` и `ElectricCar` теперь имеют способ дозаправки, но их собственная реализация. Вот пример:

```
public static void Main()
{
    List<Car> cars = new List<Car>() {
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

Выход будет следующий:

Заправка бензином
Зарядка аккумулятора

Прочтайте Полиморфизм онлайн: <https://riptutorial.com/ru/csharp/topic/1589/полиморфизм>

глава 128: Поток

Examples

Использование потоков

Поток - это объект, который обеспечивает низкоуровневое средство для передачи данных. Они сами не действуют как контейнеры данных.

Данные, с которыми мы имеем дело, находятся в форме байтового массива (`byte []`).

Функции для чтения и записи ориентированы по `WriteByte()`, например `WriteByte()`.

Нет функций для работы с целыми числами, строками и т. д. Это делает поток очень универсальным, но менее простым в работе, если, скажем, вы просто хотите передать текст. Потоки могут быть особенно полезными, когда вы имеете дело с большим объемом данных.

Нам нужно будет использовать другой тип Stream, на котором он должен быть записан / прочитан (т. е. хранилище резервных копий). Например, если источником является файл, нам нужно использовать `FileStream`:

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

Аналогично, `MemoryStream` используется, если резервным хранилищем является память:

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes("C:\\file.txt");

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
    // do stuff here...
}
```

Аналогично, `System.Net.Sockets.NetworkStream` используется для доступа к сети.

Все потоки выводятся из общего класса `System.IO.Stream`. Данные не могут быть непосредственно прочитаны или записаны из потоков. .NET Framework предоставляет вспомогательные классы, такие как `StreamReader`, `StreamWriter`, `BinaryReader` и `BinaryWriter` которые преобразуют между родными типами и интерфейсом потока низкого уровня и

переносят данные в поток или из потока для вас.

Чтение и запись в потоки можно выполнять с помощью `StreamReader` и `StreamWriter`. Будьте осторожны, закрывая их. По умолчанию закрытие также закрывает поток, содержащийся в потоке, и делает его непригодным для дальнейшего использования. Это поведение по умолчанию может быть изменено с помощью **конструктора**, который имеет параметр `bool leaveOpen` и устанавливает его значение как `true`.

`StreamWriter`:

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.WriteLine();
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

`StreamReader`:

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close();      This will close ms and when we try to use ms later it will cause an
exception
    sw.Flush();      //You can send the remaining data to stream. Closing will do this
automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Поскольку классы `Stream`, `StreamReader`, `StreamWriter` и т. д. **имплементируют** `IDisposable` **интерфейс** `IDisposable`, мы можем вызвать метод `Dispose()` для объектов этих классов.

Прочтите Поток онлайн: <https://riptutorial.com/ru/csharp/topic/3114/поток>

глава 129: Преобразование типа

замечания

Преобразование типов преобразует один тип данных в другой тип. Он также известен как Type Casting. В C # тип литья имеет две формы:

Неявное преобразование типов. Эти преобразования выполняются C # безопасным типом. Например, это преобразования от меньших до более крупных интегральных типов и преобразования из производных классов в базовые классы.

Явное преобразование типов. Эти преобразования выполняются явно пользователями с использованием предварительно определенных функций. Для явных преобразований требуется оператор трансляции.

Examples

Пример неявного оператора MSDN

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implicit conversion called");
        return d.val;
    }
    // User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implicit conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

Выход:

Цифра для двойного преобразования импликации называется
преобразование двойного значения в цифру, вызванное
num = 7 dig2 = 12

[Живая демонстрация на .NET скрипке](#)

Явное преобразование типов

```
using System;
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // cast double to int.
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();
        }
    }
}
```

Прочтайте Преобразование типа онлайн: <https://riptutorial.com/ru/csharp/topic/3489/преобразование-типа>

глава 130: Препроцессорные директивы

Синтаксис

- `#define [symbol]` // Определяет символ компилятора.
- `#undef [symbol]` // Уndefine символ компилятора.
- `#warning [warning message]` // Создает предупреждение компилятора. Полезно с `#if`.
- `#error [сообщение об ошибке]` // Создает ошибку компилятора. Полезно с `#if`.
- `#line [номер строки] (имя файла)` // Переопределяет номер строки компилятора (и, возможно, имя исходного файла). Используется с [текстовыми шаблонами T4](#).
- `#pragma warning [disable | restore] [warning numbers]` // Отключает / восстанавливает предупреждения компилятора.
- `#pragma checksum " [filename]" " [guid]" " [checksum]"` // Проверяет содержимое исходного файла.
- `#region [имя региона]` // Определяет область складного кода.
- `#endregion` // Заканчивает блок области кода.
- `#if [condition]` // Выполняет код ниже, если условие истинно.
- `#else` // Используется после `#if`.
- `#elif [condition]` // Используется после `#if`.
- `#endif` // Завершает условный блок, начинающийся с `#if`.

замечания

Директивы препроцессора обычно используются для облегчения преобразования исходных программ и их легкой компиляции в разных средах исполнения. Директивы в исходном файле сообщают препроцессору выполнять определенные действия. Например, препроцессор может заменить токены в тексте, вставить содержимое других файлов в исходный файл или подавить компиляцию части файла, удалив разделы текста. Линии препроцессора распознаются и выполняются перед расширением макроса. Поэтому, если макрос расширяется во что-то, что выглядит как команда препроцессора, эта команда не распознается препроцессором.

Операторы препроцессора используют тот же набор символов, что и для операторов исходного файла, за исключением того, что escape-последовательности не поддерживаются. Набор символов, используемый в инструкциях препроцессора, совпадает с набором символов выполнения. Препроцессор также распознает отрицательные значения символов.

Условные выражения

Условные выражения (`#if`, `#elif` и т. д.) Поддерживают ограниченное подмножество

булевых операторов. Они есть:

- == и != . Они могут использоваться только для проверки того, является ли символ истинным (определенным) или ложным (не определено)
- && , || , !
- ()

Например:

```
#if !DEBUG && (SOME_SYMBOL || SOME_OTHER_SYMBOL) && RELEASE == true
Console.WriteLine("OK!");
#endif
```

будет компилировать код, который печатает "OK!" на консоль, если DEBUG не определен, определяется SOME_SYMBOL ИЛИ SOME_OTHER_SYMBOL , и определен RELEASE .

Примечание. Эти замены выполняются *во время компиляции* и поэтому недоступны для проверки во время выполнения. Код, исключенный с помощью #if , не является частью выхода компилятора.

См. Также: [Директивы препроцессора C #](#) в MSDN.

Examples

Условные выражения

Когда компилируется следующее, оно возвращает другое значение в зависимости от того, какие директивы определены.

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
#if A
    return "A";
#elif B
    return "B";
#else
    return "C";
#endif
}
```

Условные выражения обычно используются для регистрации дополнительной информации для отладочных сборников.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
```

```
        catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
    }
}
```

Генерирование предупреждений и ошибок компилятора

Предупреждения компилятора могут быть сгенерированы с `#warning` директивы `#warning`, а ошибки могут быть сгенерированы с `#error` директивы `#error`.

```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
#endif
```

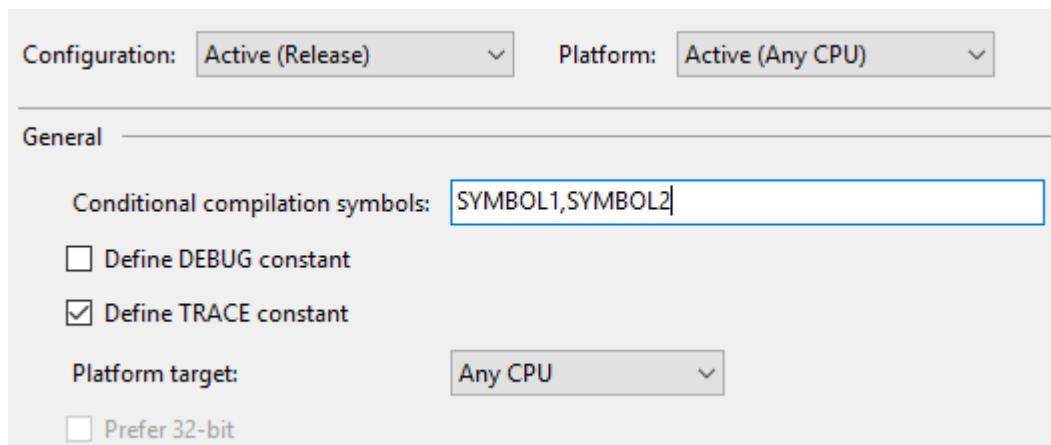
Определение и определение символов

Символом компилятора является ключевое слово, которое определено во время компиляции, которое можно проверить для условного выполнения определенных разделов кода.

Существует три способа определения символа компилятора. Их можно определить с помощью кода:

```
#define MYSYMBOL
```

Они могут быть определены в Visual Studio в разделе «Свойства проекта» > «Создать» > «Условные символы компиляции»:



(Обратите внимание, что `DEBUG` и `TRACE` имеют свои собственные флаги и не обязательно должны быть указаны явно).

Или они могут быть определены во время компиляции с использованием ключа

`/define:[name]` на компиляторе C #, `csc.exe`.

Вы также можете использовать неопределенные символы, используя директиву `#undefine`.

Наиболее распространенным примером этого является символ `DEBUG`, который определяется Visual Studio, когда приложение скомпилировано в режиме отладки (в отличие от режима `Release`).

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
        #if DEBUG
            System.Diagnostics.Trace.WriteLine("Unhandled exception!");
            System.Diagnostics.Trace.WriteLine(ex);
            throw;
        #else
            LoggingFramework.LogError(ex);
            DisplayFriendlyErrorMessage();
        #endif
    }
}
```

В приведенном выше примере, когда в бизнес-логике приложения возникает ошибка, если приложение скомпилировано в режиме отладки (и установлен символ `DEBUG`), ошибка будет записана в журнал трассировки, и исключение будет повторно -отброса для отладки. Однако, если приложение скомпилировано в режиме деблокирования (и не установлен символ `DEBUG`), среда регистрации используется для тихого регистрации ошибки, и для конечного пользователя отображается дружественное сообщение об ошибке.

Региональные блоки

Используйте `#region` и `#endregion` чтобы определить область `#endregion` кода.

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}
```

```
#endregion
```

Эти директивы полезны только тогда, когда для редактирования кода используется среда IDE, которая поддерживает разборчивые области (например, [Visual Studio](#)).

Другие инструкции компилятора

Линия

`#line` управляет номером строки и именем файла, сообщенным компилятором при выводе предупреждений и ошибок.

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

Контрольная сумма Pragma

`#pragma checksum` позволяет специфицировать определенную контрольную сумму для сгенерированной базы данных программ (PDB) для отладки.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

Использование условного атрибута

Добавление `Conditional` атрибута из `System.Diagnostics` имен `System.Diagnostics` в метод является чистым способом управления тем, какие методы вызываются в ваших сборках, а какие нет.

```
#define EXAMPLE_A

using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

    [Conditional("EXAMPLE_A")]
}
```

```
static void ExampleA() { ... }

[Conditional("EXAMPLE_B")]
static void ExampleB() { ... }

}
```

Отключение и восстановление предупреждений компилятора

Вы можете отключить предупреждения компилятора с помощью `#pragma warning disable` и `#pragma warning disable ИХ С ПОМОЩЬЮ #pragma warning restore`:

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;

#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Разрешены номера предупреждений с запятыми:

```
#pragma warning disable CS0168, CS0219
```

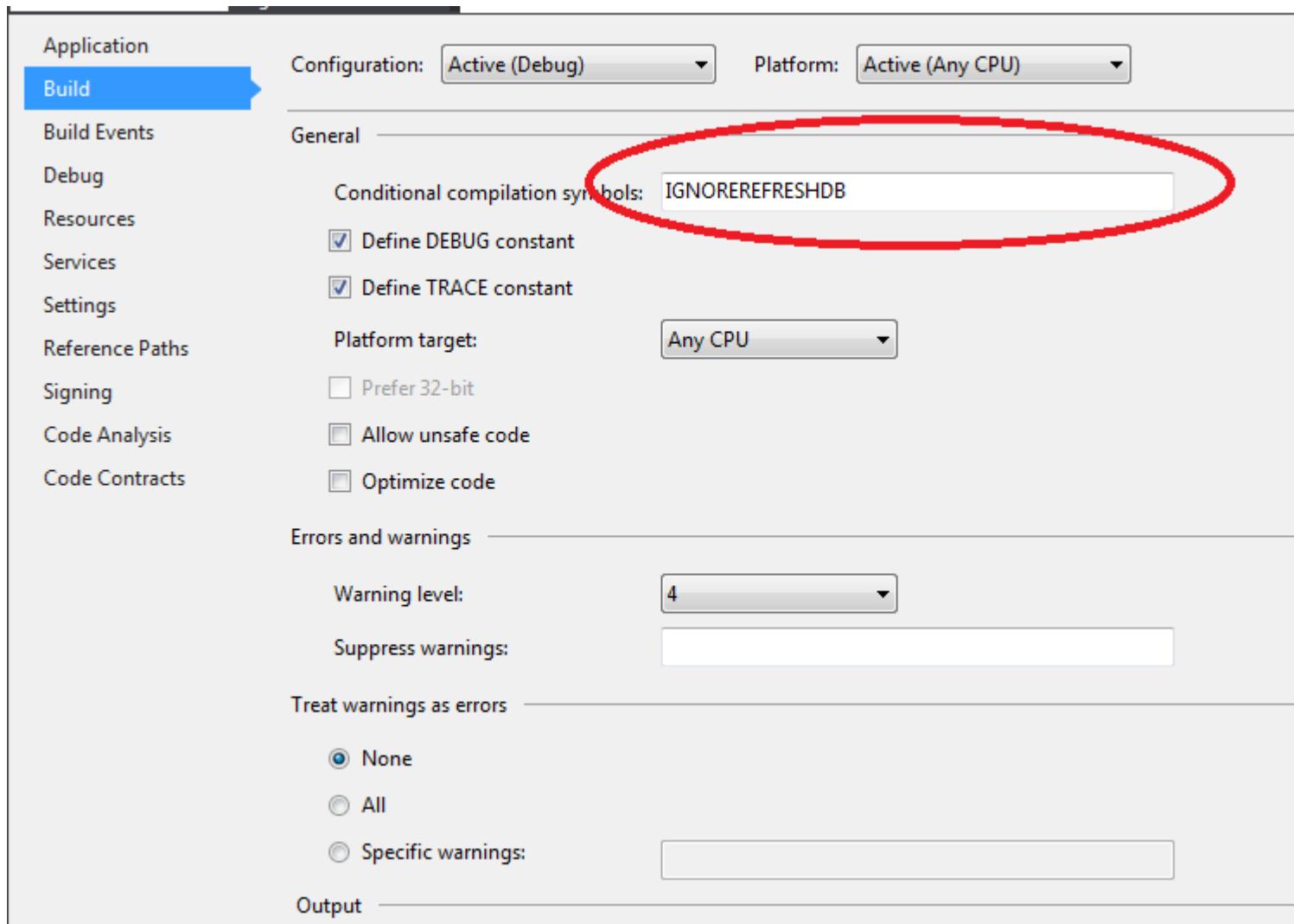
Префикс `CS` является необязательным и может быть даже смешанным (хотя это не лучшая практика):

```
#pragma warning disable 0168, 0219, CS0414
```

Пользовательские препроцессоры на уровне проекта

Удобно устанавливать пользовательскую условную предварительную обработку на уровне проекта, когда некоторые действия нужно пропустить, скажем, для тестов.

Перейдите в `Solution Explorer` -> Щелкните правой кнопкой мыши по проекту, для которого вы хотите установить переменную, -> `Properties` -> `Build` -> В поле «`Поиск`». `Conditional compilation symbols` и введите условную переменную здесь.



Пример кода, который пропустит некоторый код:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
    db.Initialize();
    #endif
}
```

Прочтите Препроцессорные директивы онлайн: <https://riptutorial.com/ru/csharp/topic/755/препроцессорные-директивы>

глава 131: Примеры AssemblyInfo.cs

замечания

Имя файла `AssemblyInfo.cs` используется в качестве исходного файла, где разработчики размещают атрибуты метаданных, которые описывают всю сборку, которую они строят.

Examples

[AssemblyTitle]

Этот атрибут используется для указания имени этой конкретной сборки.

```
[assembly: AssemblyTitle("MyProduct")]
```

[AssemblyProduct]

Этот атрибут используется для описания продукта, для которого предназначена эта конкретная сборка. Несколько сборок могут быть компонентами одного и того же продукта, и в этом случае все они могут иметь одинаковое значение для этого атрибута.

```
[assembly: AssemblyProduct("MyProduct")]
```

Глобальная и местная AssemblyInfo

Наличие глобального допускает улучшение СУХОЙ, вам нужно только поместить значения, которые отличаются от `AssemblyInfo.cs` для проектов, имеющих дисперсию. Это использование предполагает, что ваш продукт имеет более одного проекта визуальной студии.

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
//using Stackoverflow domain as a made up example

// It is common, and mostly good, to use one GlobalAssemblyInfo.cs that is added
// as a link to many projects of the same product, details below
// Change these attribute values in local assembly info to modify the information.
[assembly: AssemblyProduct("Stackoverflow Q&A")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("Copyright © Stackoverflow 2016")]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] //not going to expose ;)
```

```
// Version information for an assembly consists of the following four values:  
// roughly translated from I reckon it is for SO, note that they most likely  
// dynamically generate this file  
//     Major Version - Year 6 being 2016  
//     Minor Version - The month  
//     Day Number     - Day of month  
//     Revision       - Build number  
// You can specify all the values or you can default the Build and Revision Numbers  
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]  
[assembly: AssemblyVersion("2016.7.00.00")]  
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - по одному для каждого проекта

```
//then the following might be put into a separate Assembly file per project, e.g.  
[assembly: AssemblyTitle("Stackoveflow.Redis")]
```

Вы можете добавить GlobalAssemblyInfo.cs в локальный проект, используя [следующую процедуру](#) :

1. Выберите Добавить / Существующий элемент ... в контекстном меню проекта
2. Выберите GlobalAssemblyInfo.cs
3. Разверните кнопку «Добавить», нажав на эту маленькую стрелку вниз по правой руке
4. Выберите «Добавить как ссылку» в раскрывающемся списке кнопок

[AssemblyVersion]

Этот атрибут применяет версию к сборке.

```
[assembly: AssemblyVersion("1.0.*")]
```

Символ * используется для автоматического увеличения части версии автоматически каждый раз, когда вы компилируете (часто используется для числа «build»)

Атрибуты ассемблера чтения

Используя богатые API-интерфейсы .NET, вы можете получить доступ к метаданным сборки. Например, вы можете получить атрибут title this сборки со следующим кодом

```
using System.Linq;  
using System.Reflection;  
  
...  
  
Assembly assembly = typeof(this).Assembly;  
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();  
  
Console.WriteLine($"This assembly title is {titleAttribute?.Title}");
```

Автоматическое управление версиями

У вашего кода в исходном элементе есть номера версий по умолчанию (SVN-идентификаторы или хэши Git SHA1) или явно (теги Git). Вместо того, чтобы вручную обновлять версии в AssemblyInfo.cs, вы можете использовать процесс времени сборки для записи версии из вашей системы управления версиями в файлы AssemblyInfo.cs и, следовательно, на свои сборки.

Примерами вышеупомянутых пакетов являются пакеты [GitVersionTask](#) или [SemVer.Git.Fody](#) NuGet. Например, чтобы использовать GitVersionTask, после установки пакета в вашем проекте удалите атрибуты `Assembly*Version` из файлов AssemblyInfo.cs. Это ставит GitVersionTask на управление версиями ваших сборок.

Обратите внимание, что Semantic Versioning становится стандартом *de facto*, поэтому эти методы рекомендуют использовать теги управления версиями, которые следуют за SemVer.

Общие поля

Хорошей практикой является заполнение полей по умолчанию в AssemblyInfo. Информация может быть получена установщиками и затем появится при использовании программ и функций (Windows 10) для удаления или изменения программы.

Минимум должен быть:

- AssemblyTitle - обычно пространство имен, *то есть* MyCompany.MySolution.MyProject
- AssemblyCompany - полное юридическое лицо
- AssemblyProduct - маркетинг может иметь представление здесь
- AssemblyCopyright - держите его в курсе событий, поскольку он выглядит неряшливым в противном случае

«`AssemblyTitle`» становится «описанием файла» при просмотре вкладки «Свойства» библиотеки DLL.

[AssemblyConfiguration]

`AssemblyConfiguration`: Атрибут `AssemblyConfiguration` должен иметь конфигурацию, которая была использована для сборки сборки. Используйте условную компиляцию для правильного включения различных конфигураций сборок. Используйте блок, подобный приведенному ниже примеру. Добавьте столько разных конфигураций, сколько вы обычно используете.

```
#if (DEBUG)  
[assembly: AssemblyConfiguration("Debug")]
```

```
#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

[InternalsVisibleTo]

Если вы хотите сделать `internal` классы или функции сборки доступными из другой сборки, вы объявляете это по `InternalsVisibleTo` и имени сборки, доступ к которой разрешен.

В этом примере код в сборке `MyAssembly.UnitTests` разрешен для вызова `internal` элементов из `MyAssembly`.

```
[assembly: InternalsVisibleTo("MyAssembly.UnitTests")]
```

Это особенно полезно для модульного тестирования, чтобы предотвратить ненужные `public` объявления.

[AssemblyKeyFile]

Всякий раз, когда мы хотим, чтобы наша сборка была установлена в GAC, необходимо иметь сильное имя. Для сильной сборки именования мы должны создать открытый ключ. Чтобы сгенерировать файл `.snk`.

Чтобы создать сильный ключевой файл имени

1. Командная строка разработчика для VS2015 (с доступом администратора)
2. В командной строке введите `cd C:\Directory_Name` и нажмите клавишу ВВОД.
3. В командной строке введите `sn -k KeyFileName.snk` и нажмите клавишу ВВОД.

как только `keyFileName.snk` создается в указанном каталоге, тогда дайте ссылку в своем проекте. атрибут `AssemblyKeyFileAttribute` `snk` путь к `.snk` файлу для генерации ключа при создании нашей библиотеки классов.

Свойства -> `AssemblyInfo.cs`

```
[assembly: AssemblyKeyFile(@"c:\Directory_Name\KeyFileName.snk")]
```

Это создаст сильную сборку имен после сборки. Создав сильную сборку имен, вы можете установить ее в GAC

Счастливое кодирование :)

Прочтайте Примеры `AssemblyInfo.cs` онлайн: <https://riptutorial.com/ru/csharp/topic/4264/>

глава 132: Проверено и не отмечено

Синтаксис

- checked (a + b) // проверено выражение
- unchecked (a + b) // непроверенное выражение
- checked {c = a + b; c += 5; } // заблокирован блок
- unchecked {c = a + b; c += 5; } // unchecked block

Examples

Проверено и не отмечено

Операторы C# выполняются в любом проверенном или непроверенном контексте. В проверенном контексте арифметическое переполнение вызывает исключение. В неконтролируемом контексте арифметическое переполнение игнорируется, и результат усекается.

```
short m = 32767;
short n = 32767;
int result1 = checked((short)(m + n));    //will throw an OverflowException
int result2 = unchecked((short)(m + n)); // will return -2
```

Если ни один из них не указан, контекст по умолчанию будет опираться на другие факторы, такие как параметры компилятора.

Проверено и не отмечено как область действия

Ключевые слова также могут создавать области для (up) проверки нескольких операций.

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

Прочтайте Проверено и не отмечено онлайн: <https://riptutorial.com/ru/csharp/topic/2394/проверено-и-не-отмечено>

глава 133: Псевдонимы встроенных типов

Examples

Таблица встроенных типов

В следующей таблице показаны ключевые слова для встроенных типов C#, которые являются псевдонимами предопределенных типов в пространствах имен System.

Тип C #	Тип .NET Framework
BOOL	System.Boolean
байт	System.Byte
SByte	System.SByte
голец	System.Char
десятичный	System.Decimal
двойной	System.Double
поплавок	System.Single
ИНТ	System.Int32
UINT	System.UInt32
долго	System.Int64
ULONG	System.UInt64
объект	System.Object
короткая	System.Int16
USHORT	System.UInt16
строка	System.String

Ключевые слова C# и их псевдонимы взаимозаменямы. Например, вы можете объявить целочисленную переменную, используя одно из следующих объявлений:

```
int number = 123;  
System.Int32 number = 123;
```

Прочтите Псевдонимы встроенных типов онлайн:

<https://riptutorial.com/ru/csharp/topic/1862/псевдонимы-встроенных-типов>

глава 134: Равные и GetHashCode

замечания

Каждая реализация `Equals` должна соответствовать следующим требованиям:

- **Рефлексивный** : объект должен быть равен самому себе.
`x.Equals(x)` возвращает `true`.
- **Симметрично** : нет разницы, если я сравниваю `x` с `y` или `y` с `x` - результат тот же.
`x.Equals(y)` возвращает то же значение, что и `y.Equals(x)`.
- **Транзитив** : если один объект равен другому объекту, и он равен третьему, первый должен быть равен третьему.
`if (x.Equals(y) && y.Equals(z))` возвращает `true`, тогда `x.Equals(z)` возвращает `true`.
- **Согласовано** : если вы сравниваете объект с другим несколько раз, результат всегда один и тот же.
Последовательные вызовы `x.Equals(y)` возвращают одно и то же значение, если объекты, на которые ссылаются `x` и `y`, не изменяются.
- **Сравнение с null** : ни один объект не равен `null`.
`x.Equals(null)` возвращает `false`.

Реализации `GetHashCode` :

- **Совместимость с Equals** : если два объекта равны (это означает, что `Equals` возвращает `true`), то `GetHashCode` **должен** возвращать одинаковое значение для каждого из них.
- **Большой диапазон** : если два объекта не равны (`Equals` говорит `false`), должна быть **высокая вероятность, что** их хэш-коды различны. *Идеальное* хеширование часто невозможно, так как существует ограниченное количество значений на выбор.
- **Дешево** : во всех случаях вычислять хэш-код должно быть недорого.

См. [Руководство по перегрузке Equals \(\) и оператору ==](#)

Examples

Поведение по умолчанию по умолчанию.

`Equals` **объявляется в самом классе Object**.

```
public virtual bool Equals(Object obj);
```

По умолчанию `Equals` имеет следующее поведение:

- Если экземпляр является ссылочным типом, то `Equals` вернет `true`, только если ссылки совпадают.
- Если экземпляр является типом значения, то `Equals` вернет `true`, только если тип и значение совпадают.
- `string` - это особый случай. Он ведет себя как тип значения.

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

Запись хорошего `GetHashCode` переопределения

`GetHashCode` оказывает значительное влияние на словарь `< >` и `HashTable`.

Хорошие методы `GetHashCode`

- должны иметь равномерное распределение
 - каждое целое число должно иметь примерно равную вероятность возврата для случайного экземпляра
 - если ваш метод возвращает одно и то же целое число (например, константу «999») для каждого экземпляра, у вас будет плохая производительность
- должен быть быстрым
 - Это НЕ криптографические хеши, где медлительность - это функция
 - чем медленнее ваша хэш-функция, тем медленнее ваш словарь
- должен возвращать тот же GetHashCode в двух экземплярах, что Equals оценивает true
 - если они этого не делают (например, потому что GetHashCode возвращает случайное число), элементы не могут быть найдены в List, Dictionary ИЛИ аналогичном.

Хорошим методом реализации GetHashCode является использование одного простого числа в качестве начального значения и добавление хэш-кодов полей типа, умноженного на другие простые числа, на:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Для хэш-функции должны использоваться только те поля, которые используются в Equals методе.

Если у вас есть необходимость рассматривать один и тот же тип по-разному для словаря / хэш-таблиц, вы можете использовать IEqualityComparer.

Переопределить Equals и GetHashCode на пользовательских типах

Для класса Person like:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };
```

```
bool result = person1.Equals(person2); //false because it's reference Equals
```

Но определяя Equals и GetHashCode следующим образом:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }

    public override bool Equals(object obj)
    {
        var person = obj as Person;
        if (person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode() * Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

Также использование LINQ для разных запросов для людей будет проверять как Equals и GetHashCode :

```
var persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes" },
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes" },
    new Person{ Name = "Jon", Age = 20, Clothes = "" }
};

var distinctPersons = persons.Distinct().ToList(); //distinctPersons has Count = 2
```

Равно и GetHashCode в IEqualityComparer

Для данного типа Person :

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes" },
```

```
new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},  
new Person{ Name = "Jon", Age = 20, Clothes = ""}  
};  
  
var distinctPersons = persons.Distinct().ToList(); // distinctPersons has Count = 3
```

Но определяя Equals и GetHashCode в IEqualityComparer:

```
public class PersonComparator : IEqualityComparer<Person>  
{  
    public bool Equals(Person x, Person y)  
    {  
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when  
        comparing two persons;  
    }  
  
    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }  
}  
  
var distinctPersons = persons.Distinct(new PersonComparator()).ToList(); // distinctPersons has  
Count = 2
```

Обратите внимание, что для этого запроса два объекта считаются равными, если оба значения Equals true, а GetHashCode возвращают одинаковый хэш-код для двух человек.

Прочтайте Равные и GetHashCode онлайн: <https://riptutorial.com/ru/csharp/topic/3429/равные-и-gethashcode>

глава 135: Разрешение перегрузки

замечания

Процесс разрешения перегрузки описан в [спецификации C #](#), раздел 7.5.3. Также актуальны разделы 7.5.2 (вывод типа) и 7.6.5 (выражения вызова).

Как работает разрешение перегрузки, вероятно, будет изменено на C # 7. Замечания по дизайну показывают, что Microsoft развернет новую систему для определения того, какой метод лучше (в сложных сценариях).

Examples

Пример базовой перегрузки

Этот код содержит перегруженный метод с именем **Hello** :

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

Когда вызывается метод **Main**, он будет печатать

```
int
double
```

Во время компиляции, когда компилятор находит вызов метода `Hello(0)`, он находит все методы с именем `Hello`. В этом случае он находит два из них. Затем он пытается определить, какой из методов лучше. Алгоритм определения того, какой метод лучше, сложный, но он обычно сводится к «сделать как можно меньше имплицитных преобразований».

Таким образом, в случае `Hello(0)` преобразование не требуется для метода `Hello(int)` но

для метода `Hello(double)` требуется неявное числовое преобразование. Таким образом, первый метод выбирается компилятором.

В случае `Hello(0.0)` нет возможности конвертировать `0.0` в `int` неявно, поэтому метод `Hello(int)` даже не рассматривается для разрешения перегрузки. Остается только метод, поэтому он выбирается компилятором.

«params» не расширяется, если это необходимо.

Следующая программа:

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }
    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }
    static void Main(string[] args)
    {
        object[] objectArray = new object[5];

        Method(objectArray);
        Method(objectArray, objectArray);
        Method(objectArray, objectArray, objectArray);
    }
}
```

будет печатать:

```
5
two
3
```

Вызов `Method(objectArray)` можно интерпретировать двумя способами: одним аргументом `Object` который является массивом (поэтому программа будет выводить 1 потому что это будет число аргументов или массив аргументов, приведенный в нормальная форма, как бы метод `Method` не имеет ключевое слово `params`). В таких ситуациях нормальная, нераскрытая форма всегда имеет приоритет. Таким образом, программа выводит 5.

Во втором выражении `Method(objectArray, objectArray)`, как расширенная форма первого метода, так и традиционный второй метод. В этом случае также нерасширенные формы имеют приоритет, поэтому программа печатает two.

В третьем выражении `Method(objectArray, objectArray, objectArray)` единственным вариантом является использование расширенной формы первого метода, и поэтому программа печатает 3.

Передача null в качестве одного из аргументов

Если у вас есть

```
void F1(MyType1 x) {  
    // do something  
}  
  
void F1(MyType2 x) {  
    // do something else  
}
```

и по какой-то причине вам нужно вызвать первую перегрузку F1 но с x = null, а затем просто

```
F1(null);
```

не будет компилироваться, поскольку вызов неоднозначен. Чтобы противостоять этому, вы можете сделать

```
F1(null as MyType1);
```

Прочтайте Разрешение перегрузки онлайн: <https://riptutorial.com/ru/csharp/topic/77/разрешение-перегрузки>

глава 136: Реактивные расширения (Rx)

Examples

Наблюдение за событием TextChanged в TextBox

Наблюдаемое создается из события TextChanged TextBox. Также любой вход выбирается только в том случае, если он отличается от последнего входа, и если в течение 0,5 секунд не было введенного значения. Результат в этом примере отправляется на консоль.

```
Observable
    .FromEventPattern(textBoxInput, "TextChanged")
    .Select(s => ((TextBox) s.Sender).Text)
    .Throttle(TimeSpan.FromSeconds(0.5))
    .DistinctUntilChanged()
    .Subscribe(text => Console.WriteLine(text));
```

Потоковая передача данных из базы данных с возможностью наблюдения

Предположим, что метод, возвращающий `IEnumerable<T>`, фе

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch (Exception exception)
    {
        throw;
    }
}
```

Создает Observable и запускает метод асинхронно. `SelectMany` выравнивает коллекцию, и подписка запускается каждые 200 элементов через `Buffer`.

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
```

```
() => Console.WriteLine("Completed loading"));
```

Прочтайте **Реактивные расширения (Rx)** онлайн: <https://riptutorial.com/ru/csharp/topic/5770/реактивные-расширения--rx->

глава 137: Реализация Singleton

Examples

Статически инициализированный синглтон

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

Эта реализация является потокобезопасной, поскольку в этом случае `instance` объекта инициализируется в статическом конструкторе. CLR уже гарантирует, что все статические конструкторы выполняются поточно-безопасными.

Мутирующий `instance` не является потокобезопасной операцией, поэтому атрибут `readonly` гарантирует неизменность после инициализации.

Ленивый, потокобезопасный синглтон (с использованием Double Checked Locking)

Эта поточно-безопасная версия синглкета была необходима в ранних версиях .NET, где `static` инициализация не гарантировалась для потоковой безопасности. В более современных версиях структуры [статически инициализированный синглтон](#) обычно предпочтительнее, потому что очень легко сделать ошибки реализации в следующем шаблоне.

```
public sealed class ThreadSafeSingleton
{
    private static volatile ThreadSafeSingleton instance;
    private static object lockObject = new Object();

    private ThreadSafeSingleton()
    {
    }

    public static ThreadSafeSingleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (lockObject)
                {
                    if (instance == null)
                    {
                        instance = new ThreadSafeSingleton();
                    }
                }
            }
            return instance;
        }
    }
}
```

```
        }
    }
}

return instance;
}
}
```

Обратите внимание, что проверка `if (instance == null)` выполняется дважды: один раз до того, как блокировка будет получена, и один раз после этого. Эта реализация по-прежнему была бы потокобезопасной даже без первой проверки нуля. Однако это будет означать, что блокировка будет получена *каждый раз*, когда запрашивается экземпляр, и это может привести к потере производительности. Первая нулевая проверка добавляется, чтобы блокировка не была получена, если это не необходимо. Вторая нулевая проверка гарантирует, что только первый поток, получающий блокировку, затем создает экземпляр. Другие потоки найдут экземпляр, который будет заполнен, и пропустят вперед.

Ленивый, потокобезопасный Синглтон (с использованием Lazy)

.Net 4.0 Lazy гарантирует инициализацию объектов с потоком, поэтому этот тип можно использовать для создания Singletons.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }

}
```

Использование `Lazy<T>` гарантирует, что объект создается только тогда, когда он используется где-то в вызывающем коде.

Простое использование будет выглядеть так:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

Живая демонстрация на .NET скрипте

Lazy, потокобезопасный singleton (для .NET 3.5 или старше, альтернативная реализация)

Поскольку в .NET 3.5 и старше у вас нет класса `Lazy<T>` вы используете следующий шаблон:

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {

    }

    public static Singleton Instance
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

Это вдохновляет [блог Джона Скита](#).

Поскольку класс `Nested` является вложенным и закрытым, экземпляр экземпляра `singleton` не запускается при доступе к другим членам класса `Sigleton` (например, к `Singleton readonly`).

Устранение экземпляра Singleton, когда он больше не нужен

В большинстве примеров создается экземпляр и `LazySingleton` объект `LazySingleton` до тех пор, пока приложение-владелец не завершится, даже если этот объект больше не нужен приложению. Решением этого является реализация `IDisposable` и установка экземпляра объекта в `null` следующим образом:

```
public class LazySingleton : IDisposable
{
    private static volatile Lazy<LazySingleton> _instance;
    private static volatile int _instanceCount = 0;
    private bool _alreadyDisposed = false;

    public static LazySingleton Instance
    {
        get
        {
            if (_instance == null)
```

```

        _instance = new Lazy<LazySingleton>(() => new LazySingleton());
        _instanceCount++;
        return _instance.Value;
    }
}

private LazySingleton() { }

// Public implementation of Dispose pattern callable by consumers.
public void Dispose()
{
    if (--_instanceCount == 0) // No more references to this object.
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}

// Protected implementation of Dispose pattern.
protected virtual void Dispose(bool disposing)
{
    if (_alreadyDisposed) return;

    if (disposing)
    {
        _instance = null; // Allow GC to dispose of this instance.
        // Free any other managed objects here.
    }

    // Free any unmanaged objects here.
    _alreadyDisposed = true;
}

```

Вышеупомянутый код предоставляет экземпляр до завершения приложения, но только если потребители вызывают `Dispose()` на объекте после каждого использования. Поскольку нет никакой гарантии, что это произойдет или способ заставить его, также нет гарантии, что экземпляр когда-либо будет удален. Но если этот класс используется внутри, то легче обеспечить, чтобы метод `Dispose()` вызывался после каждого использования. Ниже приведен пример:

```

public class Program
{
    public static void Main()
    {
        using (var instance = LazySingleton.Instance)
        {
            // Do work with instance
        }
    }
}

```

Обратите внимание, что этот пример **не является потокобезопасным**.

Прочтайте Реализация Singleton онлайн: <https://riptutorial.com/ru/csharp/topic/1192/реализация-singleton>

глава 138: Рекурсия

замечания

Обратите внимание, что использование рекурсии может оказаться серьезное влияние на ваш код, так как каждый вызов рекурсивной функции будет добавлен в стек. Если слишком много вызовов, это может привести к исключению **StackOverflow**. Большинство «естественных рекурсивных функций» можно записать как конструкцию цикла `for`, `while` или `foreach`, и в то же время не выглядящие настолько **шикарными** или **умными**, будут более эффективными.

Всегда думайте дважды и тщательно используйте рекурсию - знайте, почему вы ее используете:

- рекурсия должна использоваться, когда вы знаете, что количество рекурсивных вызовов не является **чрезмерным**
 - *чрезмерные* средства, это зависит от того, сколько памяти доступно
- рекурсия используется, потому что это более ясная и чистая версия кода, более читаемая, чем итеративная или петлевая функция. Часто это происходит потому, что он дает более чистый и более компактный код (также меньше строк кода).
 - но помните, что он может быть менее эффективным! Например, в рекурсии Фибоначчи для вычисления n -го числа в последовательности время вычисления будет экспоненциально расти!

Если вы хотите больше теории, прочитайте:

- <https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html>
- https://en.wikipedia.org/wiki/Recursion#In_computer_science

Examples

Рекурсивно описывать структуру объекта

Рекурсия - это когда метод вызывает себя. Предпочтительно он будет делать это до тех пор, пока не будет выполнено конкретное условие, а затем оно нормально выйдет из метода, возвращаясь к точке, из которой был вызван метод. Если нет, исключение переполнения стека может возникнуть из-за слишком большого количества рекурсивных вызовов.

```
/// <summary>
/// Create an object structure the code can recursively describe
/// </summary>
public class Root
{
```

```

public string Name { get; set; }
public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
        DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
        Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}

```

Рекурсия на английском языке

Рекурсия может быть определена как:

Метод, который вызывает себя до тех пор, пока не будет выполнено определенное условие.

Отличным и простым примером рекурсии является метод, который получит факториал определенного числа:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

В этом методе мы видим, что метод примет аргумент, `number`.

Шаг за шагом:

Учитывая пример, выполнение `Factorial(4)`

1. `number (4) == 1 ?`
2. Нет? `return 4 * Factorial(number-1) (3)`
3. Поскольку метод вызывается еще раз, он теперь повторяет первый шаг, используя `Factorial(3)` в качестве нового аргумента.
4. Это продолжается до тех пор, пока не будет выполнено `Factorial(1)` и `number (1) == 1` вернется 1.
5. В целом, расчет «накапливает» `4 * 3 * 2 * 1` и, наконец, возвращает 24.

Ключом к пониманию рекурсии является то, что метод вызывает *новый экземпляр* самого себя. После возвращения выполнение вызывающего экземпляра продолжается.

Использование рекурсии для получения дерева каталогов

Одним из видов использования рекурсии является перемещение по иерархической структуре данных, как дерево каталогов файловой системы, не зная, сколько уровней имеет дерево или количество объектов на каждом уровне. В этом примере вы увидите, как использовать рекурсию в дереве каталогов, чтобы найти все подкаталоги указанного каталога и распечатать все дерево на консоли.

```
internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()
    {
        Console.WriteLine("Enter the path of the root directory:");
        var rootDirectorypath = Console.ReadLine();

        Console.WriteLine(
            $"Getting directory tree of '{rootDirectorypath}'");

        PrintDirectoryTree(rootDirectorypath);
        Console.WriteLine("Press 'Enter' to quit...");
        Console.ReadLine();
    }

    internal static void PrintDirectoryTree(string rootDirectoryPath)
    {
        try
        {
```

```

        if (!Directory.Exists(rootDirectoryPath))
        {
            throw new DirectoryNotFoundException(
                $"Directory '{rootDirectoryPath}' not found.");
        }

        var rootDirectory = new DirectoryInfo(rootDirectoryPath);
        PrintDirectoryTree(rootDirectory, RootLevel);
    }
    catch (DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
    }
}

private static void PrintDirectoryTree(
    DirectoryInfo directory, int currentLevel)
{
    var indentation = string.Empty;
    for (var i = RootLevel; i < currentLevel; i++)
    {
        indentation += Tab;
    }

    Console.WriteLine($"{indentation}-{directory.Name}");
    var nextLevel = currentLevel + 1;
    try
    {
        foreach (var subDirectory in directory.GetDirectories())
        {
            PrintDirectoryTree(subDirectory, nextLevel);
        }
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine($"{indentation}-{e.Message}");
    }
}
}

```

Этот код несколько сложнее, чем минимальный минимум для выполнения этой задачи, поскольку он включает проверку исключений для обработки любых проблем с получением каталогов. Ниже вы найдете разбивку кода на более мелкие сегменты с объяснениями каждого из них.

Main :

Основной метод принимает вход пользователя как строку, которая должна использоваться как путь к корневому каталогу. Затем он вызывает метод `PrintDirectoryTree` с этой строкой в качестве параметра.

`PrintDirectoryTree(string)` :

Это первый из двух методов, которые обрабатывают фактическую печать дерева каталогов. Этот метод принимает строку, представляющую путь к корневому каталогу в качестве параметра. Он проверяет, является ли путь фактическим каталогом, а если нет,

генерирует исключение `DirectoryNotFoundException` которое затем обрабатывается в блоке `catch`. Если путь является реальным каталогом, `DirectoryInfo` объект `rootDirectory` создается с пути, а второй `PrintDirectoryTree` метод вызывается с `rootDirectory` объектом и `RootLevel`, которая является целой константой со значением, равным нулю.

`PrintDirectoryTree(DirectoryInfo, int) :`

Этот второй метод обрабатывает основной результат работы. В качестве параметров требуется `DirectoryInfo` и целое число. `DirectoryInfo` - это текущий каталог, а целое число - это глубина каталога относительно корня. Для удобства чтения выходной сигнал имеет отступ для каждого уровня в глубину текущего каталога, так что вывод выглядит следующим образом:

```
-Root
  -Child 1
  -Child 2
    -Grandchild 2.1
  -Child 3
```

Как только текущий каталог печатается, его вспомогательные каталоги извлекаются, и затем этот метод вызывается для каждого из них с уровнем глубины более одного значения, чем текущий. Эта часть - это рекурсия: метод, вызывающий себя. Программа будет работать таким образом, пока она не посетит все каталоги в дереве. Когда он достигнет каталога без подкаталогов, метод автоматически вернется.

Этот метод также захватывает `UnauthorizedAccessException`, которое вызывается, если какая-либо из подкаталогов текущего каталога защищена системой. Сообщение об ошибке печатается на текущем уровне отступа для согласованности.

В приведенном ниже методе приведен более общий подход к этой проблеме:

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Это не включает в себя конкретную проверку ошибок или форматирование вывода первого подхода, но это фактически делает то же самое. Поскольку он использует только строки,

а не `DirectoryInfo`, он не может предоставить доступ к другим свойствам каталога, например разрешениям.

Последовательность Фибоначчи

Вы можете рассчитать число в последовательности Фибоначчи, используя рекурсию.

Следуя математической теории $F(n) = F(n-2) + F(n-1)$, для любого $i > 0$,

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return fib(i - 2) + fib(i - 1);
}

fib(10); // Returns 55
```

Факториальный расчет

Факториал числа (обозначаемого символом!), Как, например, $9!$, Является умножением этого числа на факториал одного ниже. Так, например, $9! = 9 \times 8! = 9 \times 8 \times 7! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$.

Таким образом, в коде, который становится, используя рекурсию:

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

Расчет PowerOf

Вычисление мощности заданного числа может быть также рекурсивным. Учитывая базовое число n и показатель e , мы должны обязательно разделить проблему на куски, уменьшив

показатель e .

Теоретический пример:

- $2^2 = 2 \times 2$
- $2^3 = 2 \times 2 \times 2$ или, $2^3 = 2^2 \times 2$

Там лежит секрет нашего рекурсивного алгоритма (см. Код ниже). Речь идет о том, чтобы решить проблему и разделить ее на меньшие и более простые, чтобы решить куски.

- **Заметки**

- когда базовое число равно 0, мы должны знать, что он возвращает 0 как $0^3 = 0 \times 0 \times 0$
- когда показатель степени равен 0, мы должны знать, что всегда нужно возвращать 1, поскольку это математическое правило.

Пример кода:

```
public int CalcPowerOf(int b, int e) {  
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0  
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1  
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem,  
    aka:  $2^3 = 2 \times 2^2$  etc..  
}
```

Тесты в xUnit для проверки логики:

Хотя это необязательно, всегда полезно писать тесты для проверки вашей логики. Я включаю те, которые здесь написаны в рамках **xUnit**.

```
[Theory]  
[MemberData(nameof(PowerOfTestData))]  
public void PowerOfTest(int @base, int exponent, int expected) {  
    Assert.Equal(expected, CalcPowerOf(@base, exponent));  
}  
  
public static IEnumerable<object[]> PowerOfTestData() {  
    yield return new object[] { 0, 0, 0 };  
    yield return new object[] { 0, 1, 0 };  
    yield return new object[] { 2, 0, 1 };  
    yield return new object[] { 2, 1, 2 };  
    yield return new object[] { 2, 2, 4 };  
    yield return new object[] { 5, 2, 25 };  
    yield return new object[] { 5, 3, 125 };  
    yield return new object[] { 5, 4, 625 };  
}
```

Прочтите Рекурсия онлайн: <https://riptutorial.com/ru/csharp/topic/2470/рекурсия>

глава 139: Сборщик мусора в .Net

Examples

Крупное уплотнение кучи объекта

По умолчанию куча больших объектов не уплотняется в отличие от классической кучи объектов, которая [может привести к фрагментации памяти](#) и, кроме того, может привести к `OutOfMemoryException`

Начиная с .NET 4.5.1 есть [возможность](#) явно сжать кучу больших объектов (вместе с сборкой мусора):

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;  
GC.Collect();
```

Точно так же, как любой запрос на сбор мусора (он называется запросом, потому что CLR не вынужден его выполнять), используйте его с осторожностью и по умолчанию избегайте его, если сможете, поскольку он может [GC статистику](#) `GC`, уменьшая ее производительность.

Слабые ссылки

В .NET GC выделяет объекты, когда нет ссылок на них. Поэтому, хотя объект все еще может быть достигнут из кода (существует сильная ссылка на него), GC не будет выделять этот объект. Это может стать проблемой, если есть много крупных объектов.

Слабая ссылка - это ссылка, которая позволяет GC собирать объект, сохраняя при этом доступ к объекту. Слабая ссылка действительна только в течение неопределенного промежутка времени, пока объект не будет собран, если нет сильных ссылок. Когда вы используете слабую ссылку, приложение все же может получить сильную ссылку на объект, что предотвращает его сбор. Таким образом, слабые ссылки могут быть полезны для хранения больших объектов, которые дорого инициализируются, но должны быть доступны для сбора мусора, если они не активно используются.

Простое использование:

```
WeakReference reference = new WeakReference(new object(), false);  
  
GC.Collect();  
  
object target = reference.Target;  
if (target != null)  
    DoSomething(target);
```

Таким образом, слабые ссылки могут использоваться для поддержания, например, кеша объектов. Однако важно помнить, что всегда существует риск того, что сборщик мусора доберется до объекта, пока не будет восстановлена сильная ссылка.

Слабые ссылки также удобны для предотвращения утечек памяти. Типичный вариант использования событий.

Предположим, что у нас есть обработчик события на источнике:

```
Source.Event += new EventHandler(Handler)
```

Этот код регистрирует обработчик событий и создает сильную ссылку от источника события на объект прослушивания. Если исходный объект имеет более длительный срок службы, чем слушатель, и слушателю больше не нужно это событие, когда нет других ссылок на него, использование обычных событий .NET вызывает утечку памяти: исходный объект содержит объекты-слушатели в памяти, которые должен быть собран мусором.

В этом случае может быть хорошей идеей использовать [шаблон слабых событий](#).

Что-то вроде:

```
public static class WeakEventManager
{
    public static void SetHandler<S, TArgs>(
        Action<EventHandler<TArgs>> add,
        Action<EventHandler<TArgs>> remove,
        S subscriber,
        Action<S, TArgs> action)
    where TArgs : EventArgs
    where S : class
    {
        var subscrWeakRef = new WeakReference(subscriber);
        EventHandler<TArgs> handler = null;

        handler = (s, e) =>
        {
            var subscrStrongRef = subscrWeakRef.Target as S;
            if (subscrStrongRef != null)
            {
                action(subscrStrongRef, e);
            }
            else
            {
                remove(handler);
                handler = null;
            }
        };
        add(handler);
    }
}
```

и используется так:

```
EventSource s = new EventSource();
Subscriber subscriber = new Subscriber();
WeakEventManager<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });
```

В этом случае, конечно, мы имеем некоторые ограничения - событие должно быть

```
public event EventHandler<SomeEventArgs> Event;
```

Как [MSDN](#) предлагает:

- Используйте длинные слабые ссылки только тогда, когда это необходимо, поскольку состояние объекта непредсказуемо после завершения.
- Избегайте использования слабых ссылок на небольшие объекты, потому что сам указатель может быть как большим, так и большим.
- Избегайте использования слабых ссылок в качестве автоматического решения проблем управления памятью. Вместо этого создайте эффективную политику кэширования для обработки объектов вашего приложения.

Прочтайте Сборщик мусора в .Net онлайн: <https://riptutorial.com/ru/csharp/topic/1287/сборщик-мусора-в--net>

глава 140: свойства

замечания

Свойства объединяют хранение данных классов по классам с возможностью доступа к методам. Иногда бывает сложно решить, использовать ли свойство, свойство, ссылающееся на поле, или метод, ссылающийся на поле. Как правило большого пальца:

- Свойства должны использоваться без внутреннего поля, если они только получают и / или устанавливают значения; без какой-либо другой логики. В таких случаях добавление внутреннего поля было бы добавлением кода без каких-либо преимуществ.
- Свойства должны использоваться с внутренними полями, когда вам нужно манипулировать или проверять данные. Примером может быть удаление начальных и конечных пробелов из строк или обеспечение того, что дата не была в прошлом.

Что касается методов vs Properties, где вы можете как получить (`get`), так и обновить (`set`) значение, лучшим вариантом является свойство. Кроме того, .Net предоставляет множество функций, которые используют структуру класса; например, добавление сетки в форму, .Net по умолчанию перечислит все свойства класса в этой форме; поэтому наилучшим образом использовать такие соглашения планируют использовать свойства, когда это поведение будет обычно желательным, и методы, в которых вы предпочитаете, чтобы типы не добавлялись автоматически.

Examples

Различные свойства в контексте

```
public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set; }

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned
    //through the public property.
    public DateTime DOB
    {
        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
```

```

        get
    {
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return DateTime.UtcNow.Year - this.dob.Year + offset;
    }
}

//this is not a property but a method; though it could be rewritten as a property if
desired.
private bool HasHadBirthdayThisYear()
{
    bool hasHadBirthdayThisYear = true;
    DateTime today = DateTime.UtcNow;
    if (today.Month > this.dob.Month)
    {
        hasHadBirthdayThisYear = true;
    }
    else
    {
        if (today.Month == this.dob.Month)
        {
            hasHadBirthdayThisYear = today.Day > this.dob.Day;
        }
        else
        {
            hasHadBirthdayThisYear = false;
        }
    }
    return hasHadBirthdayThisYear;
}
}

```

Публичный Get

Getters используются для отображения значений из классов.

```

string name;
public string Name
{
    get { return this.name; }
}

```

Общественный набор

Setters используются для назначения значений свойствам.

```

string name;
public string Name
{
    set { this.name = value; }
}

```

Доступ к свойствам

```

class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //example of accessing properties (Id, Name & DOB)
        Console.WriteLine("Id is: \t{0}\nName is:\t{1}.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = " Hans Trimmer ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only
accessible within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as
there's validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is: \t{0}\nName is:\t{1}.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }
}

public class Person
{
    private static int nextId = 0;
    private string name;
    private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
    public Person(string name, DateTime dob)
    {
        this.Id = ++Person.nextId;
        this.Name = name;
        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDOBException(value);
            this.dob = value;
        }
    }
}

```

```

public int GetAgeInYears()
{
    DateTime today = DateTime.UtcNow;
    int offset = HasHadBirthdayThisYear() ? 0 : -1;
    return today.Year - this.dob.Year + offset;
}
private bool HasHadBirthdayThisYear()
{
    bool hasHadBirthdayThisYear = true;
    DateTime today = DateTime.UtcNow;
    if (today.Month > this.dob.Month)
    {
        hasHadBirthdayThisYear = true;
    }
    else
    {
        if (today.Month == this.dob.Month)
        {
            hasHadBirthdayThisYear = today.Day > this.dob.Day;
        }
        else
        {
            hasHadBirthdayThisYear = false;
        }
    }
    return hasHadBirthdayThisYear;
}
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage,value)){}
}
public class InvalidDobException : ApplicationException
{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB. The date
must not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}

```

Значения по умолчанию для свойств

Установка значения по умолчанию может быть выполнена с помощью инициализаторов (С # 6)

```

public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}

```

Если он доступен только для чтения, вы можете вернуть такие значения:

```

public class Name

```

```
{  
    public string First => "James";  
    public string Last => "Smith";  
}
```

Автореализованные свойства

Авто-реализованные свойства были введены в C # 3.

Авто-реализованное свойство объявляется с пустым getter и setter (accessors):

```
public bool IsValid { get; set; }
```

Когда в вашем коде записывается автоматически реализованное свойство, компилятор создает личное анонимное поле, доступ к которому можно получить только через аксессуры свойства.

Вышеприведенный автозаполняемый оператор свойства эквивалентен написанию этого длинного кода:

```
private bool _isValid;  
public bool IsValid  
{  
    get { return _isValid; }  
    set { _isValid = value; }  
}
```

Автообновленные свойства не могут иметь никакой логики в их аксессуарах, например:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

Однако свойство автообновления может иметь разные модификаторы доступа для своих аксессуаров:

```
public bool IsValid { get; private set; }
```

C # 6 позволяет автоматически реализованным свойствам вообще не устанавливать сеттер (делая его неизменным, поскольку его значение может быть установлено только внутри конструктора или жестко закодировано):

```
public bool IsValid { get; }  
public bool IsValid { get; } = true;
```

Для получения более подробной информации об инициализации авто Реализуемого свойства, читать [Инициализаторы Auto-свойство](#) документацию.

Свойства только для чтения

декларация

Обычное недоразумение, особенно новички, имеет свойство только для чтения, которое имеет ключевое слово `readonly`. Это неверно, и на самом деле *следующая ошибка времени компиляции*:

```
public readonly string SomeProp { get; set; }
```

Свойство доступно только для чтения, когда у него есть только получатель.

```
public string SomeProp { get; }
```

Использование свойств только для чтения для создания неизменяемых классов

```
public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}
```

Прочтайте свойства онлайн: <https://riptutorial.com/ru/csharp/topic/49/свойства>

глава 141: Секундомеры

Синтаксис

- `stopWatch.Start ()` - запускает секундомер.
- `stopWatch.Stop ()` - останавливает секундомер.
- `stopWatch.Elapsed` - Получает общее прошедшее время, измеренное текущим интервалом.

замечания

Секундомеры часто используются в программах бенчмаркинга для временного кода и видят, как оптимальные различные сегменты кода выполняются.

Examples

Создание экземпляра секундомера

Экземпляр секундомера может измерять прошедшее время в течение нескольких интервалов с общим временем, прошедшим сложение всех отдельных интервалов. Это дает надежный метод измерения прошедшего времени между двумя или более событиями.

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
    d += 1;
}

stopWatch.Stop();
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

Stopwatch находится в `System.Diagnostics` поэтому вам нужно добавить `using System.Diagnostics;` к вашему файлу.

IsHighResolution

- Свойство `IsHighResolution` указывает, основан ли таймер на счетчике производительности с высоким разрешением или на основе класса `DateTime`.
- Это поле доступно только для чтения.

```
// Display the timer frequency and resolution.
if (Stopwatch.IsHighResolution)
```

```

{
    Console.WriteLine("Operations timed using the system's high-resolution performance
counter.");
}
else
{
    Console.WriteLine("Operations timed using the DateTime class.");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine(" Timer frequency in ticks per second = {0}",
    frequency);
long nanosecPerTick = (1000L*1000L*1000L) / frequency;
Console.WriteLine(" Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}

```

<https://dotnetfiddle.net/ckrWUo>

Таймер, используемый классом секундомера, зависит от системного оборудования и операционной системы. `IsHighResolution` истинна, если таймер секундомера основан на счетчике производительности с высоким разрешением. В противном случае `IsHighResolution` является ложным, что указывает на то, что таймер секундомера основан на системном таймере.

Клещи в секундомере зависят от машины / ОС, поэтому вам не следует рассчитывать на то, что количество секунд секундомера на секундах будет одинаковым между двумя системами и, возможно, даже в одной и той же системе после перезагрузки. Таким образом, вы никогда не можете рассчитывать на тики секундомера на тот же интервал, что и метки `DateTime` / `TimeSpan`.

Чтобы получить независимое от системы время, убедитесь, что вы используете свойства «Секундомер истек» или «Истекшие миллисекунды», которые уже учитывают значение секундомера. Частота (тики в секунду).

Секундомер всегда должен использоваться в течение `Datetime` для процессов синхронизации, поскольку он более легкий и использует `Dateime`, если он не может использовать счетчик производительности с высоким разрешением.

[Источник](#)

Прочтайте Секундомеры онлайн: <https://riptutorial.com/ru/csharp/topic/3676/секундомеры>

глава 142: сетей

Синтаксис

- TcpClient (string host, int port);

замечания

Вы можете получить NetworkStream из TcpClient с `client.GetStream()` и передать его в `StreamReader/StreamWriter` чтобы получить доступ к их методам чтения и записи `async`.

Examples

Основной клиент TCP-связи

В этом примере кода создается клиент TCP, он отправляет «Hello World» через соединение сокета, а затем записывает ответ сервера на консоль перед закрытием соединения.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// The client and stream will close as control exits the using block (Equivalent but safer
than calling Close());
```

Загрузите файл с веб-сервера

Загрузка файла из Интернета - очень обычная задача, требуемая почти для каждого приложения, которое вы можете построить.

Для этого вы можете использовать класс « [System.Net.WebClient](#) ».

Простейшее использование этого, используя шаблон «using», показано ниже:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

В этом примере он использует «использование», чтобы убедиться, что ваш веб-клиент правильно очищен по окончании и просто передает именованный ресурс из URL-адреса в первом параметре, в именованный файл на локальном жестком диске во втором параметре.

Первый параметр имеет тип « [System.Uri](#) », второй параметр имеет тип « [System.String](#) »,

Вы также можете использовать эту функцию как асинхронную форму, так что она отключается и выполняет загрузку в фоновом режиме, в то время как ваше приложение получает что-то другое, используя вызов таким образом, имеет большое значение в современных приложениях, поскольку это помогает чтобы ваш пользовательский интерфейс реагировал.

Когда вы используете методы `Async`, вы можете подключить обработчики событий, которые позволяют отслеживать прогресс, чтобы вы могли, например, обновить индикатор выполнения, что-то вроде следующего:

```
var webClient = new WebClient();
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

Один важный момент, который следует помнить, если вы используете версии `Async`, и это «Будьте очень осторожны в использовании их в синтаксисе« `using` ».

Причина этого довольно проста. После вызова метода файла загрузки он немедленно вернется. Если у вас есть это в используемом блоке, вы вернетесь, затем выйдите из этого блока и немедленно удалите объект класса, и, таким образом, отмените загрузку.

Если вы используете способ «использования» для передачи `Async`, обязательно оставайтесь внутри закрывающего блока до завершения передачи.

Async TCP Client

Использование `async/await` в приложениях C # упрощает многопоточность. Так вы можете использовать `async/await` `wait` в сочетании с `TcpClient`.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;
```

```

// Create TCP client and connect
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
    // Asynchronously attempt to connect to server
    await client.ConnectAsync(host, port);

    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}
// The client and stream will close as control exits
// the using block (Equivalent but safer than calling Close());

```

Основной клиент UDP

В этом примере кода создается клиент UDP, а затем отправляется «Hello World» по сети назначаемому получателю. Слушатель не должен быть активным, так как UDP является без установления соединения и будет транслировать сообщение независимо. После отправки сообщения выполняется работа с клиентами.

```

byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
    using (var client = new UdpClient())
    {
        IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
        client.Connect(ep);
        client.Send(data, data.Length);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}

```

Ниже приведен пример прослушивателя UDP для дополнения вышеупомянутого клиента. Он будет постоянно сидеть и слушать трафик на данном порту и просто записывать эти данные на консоль. В этом примере содержится флаг управления «`done`», который не

задан внутри, и полагается на что-то, чтобы установить это, чтобы разрешить конец слушателя и выйти.

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}",
listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World"
sent from above client
    }
}
```

Прочтайте сетей онлайн: <https://riptutorial.com/ru/csharp/topic/1352/сетей>

глава 143: События

Вступление

Событие - это уведомление о том, что что-то произошло (например, щелчок мыши) или, в некоторых случаях, произойдет (например, изменение цены).

Классы могут определять события, и их экземпляры (объекты) могут создавать эти события. Например, кнопка может содержать событие Click, которое поднимается, когда пользователь щелкнул его.

Обработчики событий - это методы, которые вызываются при возникновении соответствующего события. Форма может содержать обработчик события Clicked для каждой кнопки, которая содержит, например.

параметры

параметр	подробности
EventArgsT	Тип, который выводится из EventArgs и содержит параметры события.
Название события	Название события.
HandlerName	Имя обработчика события.
SenderObject	Объект, вызывающий событие.
EventArgs	Экземпляр типа EventArgsT, который содержит параметры события.

замечания

При поднятии события:

- Всегда проверяйте, является ли делегат `null`. Нулевой делегат означает, что у события нет подписчиков. Поднятие события без подписчиков приведет к `NullReferenceException`.

6,0

- Скопируйте делегат (например, `EventName`) в локальную переменную (например,

`eventName`), прежде чем проверять значение `null` / повышение события. Это позволяет избежать условий гонки в многопоточных средах:

Неправильно :

```
if(Changed != null)      // Changed has 1 subscriber at this point
    // In another thread, that one subscriber decided to unsubscribe
    Changed(this, args); // `Changed` is now null, `NullReferenceException` is thrown.
```

Справа :

```
// Cache the "Changed" event as a local. If it is not null, then use
// the LOCAL variable (handler) to raise the event, NOT the event itself.
var handler = Changed;
if(handler != null)
    handler(this, args);
```

6,0

- Используйте оператор с нулевым условием `(?.)` Для повышения метода вместо нулевой проверки делегата для подписчиков в выражении `if` :
`EventName?.Invoke(SenderObject, new EventArgs());`
- При использовании `Action <>` для объявления типов делегатов подпись анонимного метода / события должна быть такой же, как объявленный анонимный тип делегата в объявлении события.

Examples

Объявление и повышение активности

Объявление события

Вы можете объявить событие в любом `class` ИЛИ `struct` используя следующий синтаксис:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

Существует расширенный синтаксис для объявления событий, в котором вы держите частный экземпляр события и определяете открытый экземпляр, используя `add` и `set`

accessors. Синтаксис очень похож на свойства C#. Во всех случаях синтаксис, продемонстрированный выше, должен быть предпочтительным, поскольку компилятор испускает код, чтобы гарантировать, что несколько потоков могут безопасно добавлять и удалять обработчики событий в событие на вашем классе.

Поднятие мероприятия

6,0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

6,0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;

    // If eventName == null, then it means there are no event-subscribers,
    // and therefore, we cannot raise the event.
    if(eventName != null)
        eventName(this, EventArgs.Empty);
}
```

Обратите внимание, что события могут быть подняты только типом объявления. Клиенты могут подписаться или отказаться от подписки.

Для версий C# до 6.0, где `EventName?.Invoke` не поддерживается, рекомендуется назначить событие временной переменной перед вызовом, как показано в примере, что обеспечивает безопасность потоков в случаях, когда несколько потоков выполняют одинаковые код. В противном случае может возникнуть `NullReferenceException` в некоторых случаях, когда несколько потоков используют один и тот же экземпляр объекта. В C# 6.0 компилятор испускает код, подобный тому, который показан в примере кода для C# 6.

Стандартное объявление о мероприятии

Объявление события:

```
public event EventHandler<EventArgsT> EventName;
```

Объявление обработчика события:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

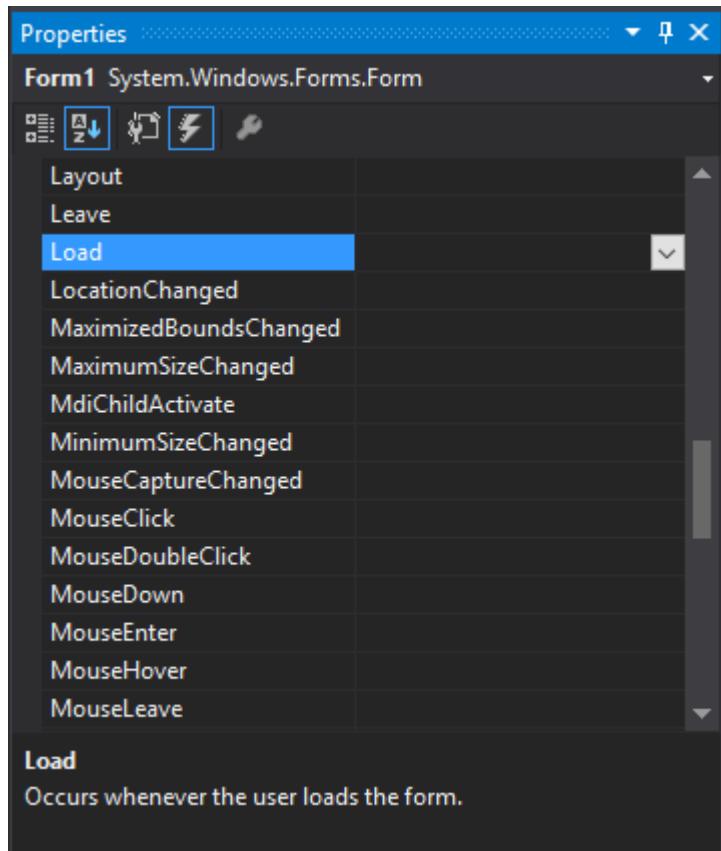
Подписка на событие:

Динамически:

```
EventName += HandlerName;
```

Через конструктора:

1. Нажмите кнопку «События» в окне свойств элемента управления («Молниеносный болт»)
2. Дважды щелкните имя события:



3. Visual Studio создаст код события:

```
private void Form1_Load(object sender, EventArgs e)
{
}
```

Вызов метода:

```
EventName (SenderObject, EventArgs);
```

Объявление анонимного обработчика событий

Объявление события:

```
public event EventHandler<EventArgsType> EventName;
```

Объявление обработчика события с помощью **lambda operator =>** и подписки на событие:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Объявление обработчика события с использованием синтаксиса анонимного **делегата**:

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Декларация и подписка обработчика событий, который не использует параметр события, и поэтому может использовать указанный выше синтаксис, не указывая параметры:

```
EventName += delegate { /* Handler Logic */ }
```

Вызов события:

```
EventName?.Invoke(SenderObject, EventArguments);
```

Нестандартная заявка на мероприятие

События могут иметь любой тип делегата, а не только `EventHandler` и `EventHandler<T>`. Например:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

Это используется аналогично стандартным событиям `EventHandler`:

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

Можно объявить несколько событий одного и того же типа в одном выражении, аналогично полям и локальным переменным (хотя это часто может быть плохой идеей):

```
public event EventHandler Event1, Event2, Event3;
```

Это объявляет три отдельных события (`Event1` , `Event2` И `Event3`) для всех типов `EventHandler`

Примечание. Хотя некоторые компиляторы могут принять этот синтаксис как в интерфейсах, так и в классах, спецификация C # (v5.0 §13.2.3) предоставляет грамматику для интерфейсов, которые этого не позволяют, поэтому использование этого в интерфейсах может быть ненадежным для разных компиляторов.

Создание настраиваемых `EventArgs`, содержащих дополнительные данные

Для пользовательских событий обычно требуются настраиваемые аргументы событий, содержащие информацию о событии. Например `MouseEventArgs` , который используется на события мыши , как `MouseDown` ИЛИ `MouseUp` событий, содержит информацию о `Location` ИЛИ `Buttons` , которые используются для генерации события.

При создании новых событий для создания настраиваемого события `arg`:

- Создайте класс, полученный из `EventArgs` и определите свойства необходимых данных.
- В качестве условного обозначения имя класса должно заканчиваться `EventArgs` .

пример

В приведенном ниже примере мы создаем событие `PriceChangingEventArgs` для свойства `Price` для класса. Класс данных событий содержит `CurrentPrice` И `NewPrice` . Событие возникает, когда вы назначаете новое значение для свойства `Price` И позволяет потребителю знать, что значение меняется и позволяет узнать о текущей цене и новой цене:

`PriceChangingEventArgs`

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

Товар

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
```

```

        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

protected void OnPriceChanging(PriceChangingEventArgs e)
{
    var handler = PriceChanging;
    if (handler != null)
        handler(this, e);
}
}

```

Вы можете улучшить пример, разрешив потребителю изменить новое значение, а затем значение будет использовано для свойства. Для этого достаточно применить эти изменения в классах.

Измените определение `NewPrice` :

```
public int NewPrice { get; set; }
```

Измените определение `Price` на использование `e.NewPrice` качестве значения свойства после вызова `OnPriceChanging` :

```

int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}

```

Создание отменяемого события

Отменное событие может быть поднято классом, когда он собирается выполнить действие, которое может быть отменено, например событие `FormClosing` `Form` .

Чтобы создать такое событие:

- Создайте новое событие `arg` from `CancelEventArgs` и добавьте дополнительные свойства для данных события.
- Создайте событие, используя `EventHandler<T>` и используйте созданный вами класс `arg` нового события `cancel`.

пример

В приведенном ниже примере мы создаем событие `PriceChangingEventArgs` для свойства `Price` для класса. Класс данных события содержит `Value` которое позволяет потребителю узнать о новом. Событие возникает, когда вы назначаете новое значение для свойства `Price` и позволяет потребителю знать, что значение меняется, и позволить им отменить событие. Если потребитель отменяет событие, будет использоваться предыдущее значение для `Price`:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

Товар

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

Свойства события

Если класс вызывает большое количество событий, стоимость хранения одного поля на одного делегата может быть неприемлемым. .NET Framework предоставляет [свойства](#)

событий для этих случаев. Таким образом, вы можете использовать другую структуру данных, такую как `EventHandlerList` для хранения делегатов событий:

```
public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}
```

Этот подход широко используется в графических интерфейсах, таких как WinForms, где элементы управления могут иметь десятки и даже сотни событий.

Обратите внимание, что `EventHandlerList` не является потокобезопасным, поэтому, если вы ожидаете, что ваш класс будет использоваться из нескольких потоков, вам нужно будет добавить операторы блокировки или другой механизм синхронизации (или использовать хранилище, обеспечивающее безопасность потоков).

Прочтайте События онлайн: <https://riptutorial.com/ru/csharp/topic/64/события>

глава 144: Соглашения об именах

Вступление

В этом разделе описываются основные соглашения об именах, используемые при записи на языке C#. Как и все соглашения, они не применяются компилятором, но обеспечивают читаемость между разработчиками.

Подробные рекомендации по разработке .NET Framework см. в документе docs.microsoft.com/dotnet/standard/design-guidelines.

замечания

Выбор легко читаемых имен идентификаторов

Например, свойство с именем `HorizontalAlignment` более читается на английском языке, чем `AlignmentHorizontal`.

Благодарите читателя за краткость

Имя свойства `CanScrollHorizontally` лучше, чем `ScrollableX` (`CanScrollHorizontally` ссылка на ось X).

Избегайте использования символов подчеркивания, дефис или любых других не-буквенно-цифровых символов.

Не используйте венгерскую нотацию

Венгерская нотация - это практика включения префикса в идентификаторы для кодирования некоторых метаданных о параметре, таких как тип данных идентификатора, например `string strName`.

Также избегайте использования идентификаторов, которые конфликтуют с ключевыми словами, уже используемыми в C#.

Аббревиатуры и Акронимы

В общем, вы не должны использовать сокращения или аббревиатуры; они делают ваши имена менее читабельными. Точно так же трудно понять, когда можно с уверенностью предположить, что акроним широко признан.

Examples

Соглашения о капитализации

Следующие термины описывают различные способы идентификации идентификаторов.

Корпус Pascal

Первая буква в идентификаторе и первая буква каждого последующего конкатенированного слова капитализируются. Вы можете использовать случай Pascal для идентификаторов трех или более символов. Например: `BackColor`

Корпус верблюда

Первая буква идентификатора является строчной буквой и первая буква каждого последующего конкатенированного слова капитализируется. Например: `backColor`

Верхний регистр

Все буквы в идентификаторе капитализируются. Например: `IO`

правила

Если идентификатор состоит из нескольких слов, не используйте разделители, такие как символы подчеркивания («_») или дефисы («-») между словами. Вместо этого используйте обсадную колонну, чтобы указать начало каждого слова.

В следующей таблице приведены правила капитализации для идентификаторов и приведены примеры для разных типов идентификаторов:

Идентификатор	случай	пример
Локальное значение	верблюд	<code>carName</code>
Учебный класс	паскаль	<code>AppDomain</code>
Тип перечисления	паскаль	<code>Равен</code>
Значения перечисления	паскаль	<code>Фатальная ошибка</code>
Событие	паскаль	<code>ValueChanged</code>
Класс исключений	паскаль	<code>WebException</code>

Идентификатор	случай	пример
Статическое поле только для чтения	паскаль	RedValue
Интерфейс	паскаль	IDisposable
метод	паскаль	Нанизывать
Пространство имен	паскаль	System.Drawing
параметр	верблюд	TYPENAME
Имущество	паскаль	BackColor

Более подробную информацию можно найти в [MSDN](#) .

Интерфейсы

Интерфейсы должны быть названы именами или существительными фразами или прилагательными, описывающими поведение. Например, `IComponent` использует описательное существительное, `ICustomAttributeProvider` использует именную фразу, а `IPersistable` использует прилагательное.

Названия интерфейсов должны иметь префикс с буквой `I`, чтобы указать, что тип является интерфейсом, и использовать паскаль.

Ниже приведены правильно названные интерфейсы:

```
public interface IServiceProvider
public interface IFormattable
```

Частные поля

Существуют два общих соглашения для частных полей: `camelCase` и `_camelCaseWithLeadingUnderscore` .

Чехол для верблюда

```
public class Rational
{
    private readonly int numerator;
    private readonly int denominator;

    public Rational(int numerator, int denominator)
    {
        // "this" keyword is required to refer to the class-scope field
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

```
    }
}
```

Костюм верблюда с подчеркиванием

```
public class Rational
{
    private readonly int _numerator;
    private readonly int _denominator;

    public Rational(int numerator, int denominator)
    {
        // Names are unique, so "this" keyword is not required
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

Пространства имен

Общий формат пространств имен:

```
<Company>.(<Product>|<Technology>) [<Feature>] [<Subnamespace>].
```

Примеры включают:

```
Fabrikam.Math
Litware.Security
```

Префикс имен пространства имен с названием компании не позволяет именам из разных компаний иметь одно и то же имя.

Перечисления

Используйте уникальное имя для большинства Enums

```
public enum Volume
{
    Low,
    Medium,
    High
}
```

Используйте множественное имя для типов Enum, которые являются битовыми полями

```
[Flags]
```

```
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

Примечание. Всегда добавляйте [FlagsAttribute](#) к типу `Enum` типа бит.

Не добавляйте «перечисление» в качестве суффикса

```
public enum VolumeEnum // Incorrect
```

Не следует использовать имя перечисления в каждой записи

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

Исключения

Добавить «исключение» в качестве суффикса

Пользовательские имена исключений должны быть помечены как «Исключение».

Ниже приведены корректно названные исключения:

```
public class MyCustomException : Exception
public class FooException : Exception
```

Прочтайте Соглашения об именах онлайн: <https://riptutorial.com/ru/csharp/topic/2330/соглашения-об-именах>

глава 145: Создание консольного приложения с использованием редактора Plain-Text и компилятора C # (csc.exe)

Examples

Создание приложения консоли с использованием редактора Plain-Text и компилятора C

Чтобы использовать текстовый редактор для создания приложения консоли, написанного на C #, вам понадобится компилятор C #. Компилятор C # (csc.exe) можно найти в следующем месте: %WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe

NB В зависимости от версии .NET Framework, установленной в вашей системе, вам может потребоваться изменить путь выше, соответственно.

Сохранение кода

Цель данной темы не научить вас , как написать консольное приложение, но и научить вас , как составить один [для получения одного исполняемого файла], ни с чем другим , чем C # компилятор и любой редактор Plain-Text (например, Блокнот).

1. Откройте диалоговое окно «Запуск» с помощью сочетания клавиш Windows Key + R
2. Ведите notepad , затем нажмите Enter.
3. Вставьте примерный код ниже, в Блокнот
4. Сохраните файл как ConsoleApp.cs , перейдя в **Файл → Сохранить как ...** , а затем введите ConsoleApp.cs в текстовое поле «Имя файла», а затем выберите « All Files В качестве типа файла.
5. Нажмите Save

Компиляция исходного кода

1. Откройте диалог «Запуск», используя Windows Key + R
2. Ведите:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe  
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
```

```
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Теперь вернитесь туда, где вы первоначально сохранили файл `ConsoleApp.cs`. Теперь вы должны увидеть исполняемый файл (`ConsoleApp.exe`). Дважды щелкните `ConsoleApp.exe` чтобы открыть его.

Это оно! Консольное приложение было скомпилировано. Создан исполняемый файл, и теперь у вас есть рабочее консольное приложение.

```
using System;

namespace ConsoleApp
{
    class Program
    {
        private static string input = String.Empty;

        static void Main(string[] args)
        {
            goto DisplayGreeting;

            DisplayGreeting:
            {
                Console.WriteLine("Hello! What is your name?");

                input = Console.ReadLine();

                if (input.Length >= 1)
                {
                    Console.WriteLine(
                        "Hello, " +
                        input +
                        ", enter 'Exit' at any time to exit this app.");

                    goto AwaitFurtherInstruction;
                }
                else
                {
                    goto DisplayGreeting;
                }
            }

            AwaitFurtherInstruction:
            {
                input = Console.ReadLine();

                if(input.ToLower() == "exit")
                {
                    input = String.Empty;

                    Environment.Exit(0);
                }
                else
                {
                    goto AwaitFurtherInstruction;
                }
            }
        }
    }
}
```

```
    }  
}
```

Прочтайте Создание консольного приложения с использованием редактора Plain-Text и компилятора C # (csc.exe) онлайн: <https://riptutorial.com/ru/csharp/topic/6676/создание-консольного-приложения-с-использованием-редактора-plain-text-и-компилятора-c-sharp--csc-exe->

глава 146: Создание собственного MessageBox в приложении Windows Form

Вступление

Сначала нам нужно знать, что такое MessageBox ...

Элемент управления MessageBox отображает сообщение с указанным текстом и может быть настроено путем указания пользовательских изображений, заголовков и кнопок (эти наборы кнопок позволяют пользователю выбирать более простой ответ «да / нет»).

Создавая собственный MessageBox, мы можем повторно использовать этот MessageBox Control в любых новых приложениях, просто используя сгенерированную dll или скопировав файл, содержащий класс.

Синтаксис

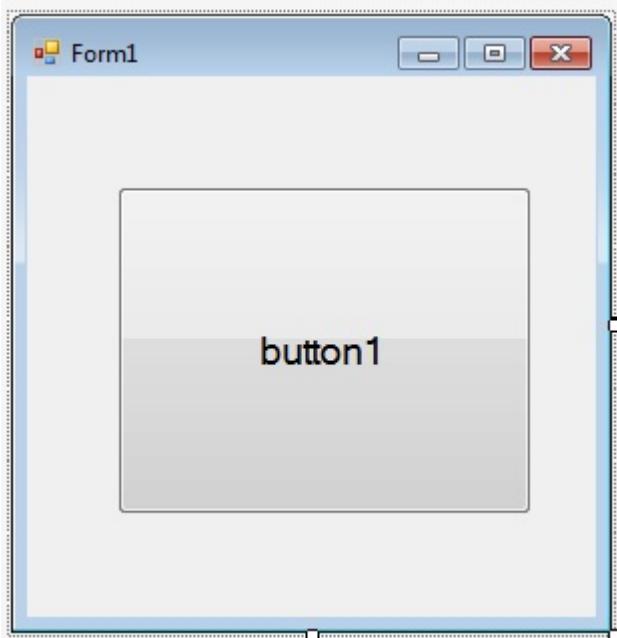
- 'статический результат DialogResult = DialogResult.No; // DialogResult возвращается по диалогам после увольнения. '

Examples

Создание собственного элемента управления MessageBox.

Чтобы создать собственный элемент управления MessageBox, просто следуйте приведенному ниже руководству ...

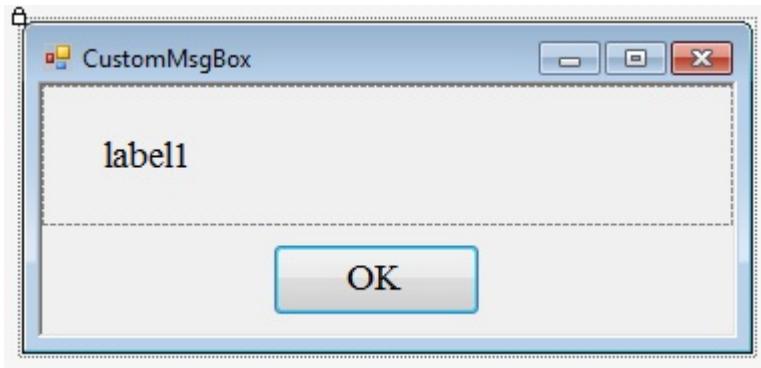
1. Откройте экземпляр Visual Studio (VS 2008/2010/2012/2015/2017)
2. Перейдите на панель инструментов вверху и нажмите «Файл» -> «Новый проект» -> «Приложение Windows Forms» -> Дайте проекту имя и нажмите «OK».
3. После загрузки перетащите элемент управления Button из панели инструментов (находится слева) в форму (как показано ниже).



4. Дважды щелкните по кнопке, и интегрированная среда разработки автоматически создаст для вас обработчик событий кликов.
5. Измените код формы, чтобы он выглядел следующим образом (вы можете щелкнуть правой кнопкой мыши форму и нажать «Изменить код»):

```
namespace MsgBoxExample {  
    public partial class MsgBoxExampleForm : Form {  
        //Constructor, called when the class is initialised.  
        public MsgBoxExampleForm() {  
            InitializeComponent();  
        }  
  
        //Called whenever the button is clicked.  
        private void btnShowMessageBox_Click(object sender, EventArgs e) {  
            CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");  
        }  
    }  
}
```

6. Обозреватель решений -> Щелкните правой кнопкой мыши по вашему проекту -> Добавить -> Форма Windows и укажите имя как «CustomMsgBox.cs»
7. Перетащите кнопку и управление меткой из панели инструментов в форму (после этого она будет выглядеть примерно так):



8. Теперь выпишите код ниже во вновь созданную форму:

```
private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    msgBox.Close();
}
```

9. Теперь запустите программу, просто нажав клавишу F5. Поздравляем, вы сделали многоразовый контроль.

Как использовать собственный созданный элемент управления MessageBox в другом приложении Windows Form.

Чтобы найти существующие файлы .cs, щелкните правой кнопкой мыши проект в вашем экземпляре Visual Studio и выберите «Открыть папку» в Проводнике.

1. Visual Studio -> Ваш текущий проект (Windows Form) -> Обозреватель решений -> Название проекта -> Щелкните правой кнопкой мыши -> Добавить -> Существующий элемент -> Затем найдите существующий файл .cs.
2. Теперь есть еще одна вещь, чтобы использовать этот элемент управления. Добавьте в свой код инструкцию using, чтобы ваша сборка узнала о ее зависимостях.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
.
```

```
•  
•  
using CustomMsgBox; //Here's the using statement for our dependency.
```

3. Чтобы отобразить сообщение, просто используйте следующие ...

CustomMsgBox.Show («Ваше сообщение для окна сообщений ...», «MSG», «OK»);

Прочтите Создание собственного MessageBox в приложении Windows Form онлайн:
<https://riptutorial.com/ru/csharp/topic/9788/создание-собственного-messagebox-в-приложении-windows-form>

глава 147: Создание шаблонов проектирования

замечания

Шаги создания нацелены на то, чтобы отделить систему от того, как ее объекты созданы, составлены и представлены. Они повышают гибкость системы с точки зрения того, что, кто, как и когда создает объект. Творческие шаблоны инкапсулируют знания о том, какие классы использует система, но они скрывают детали того, как создаются и объединяются экземпляры этих классов. Программисты поняли, что системы компоновки с наследованием делают эти системы слишком жесткими. Созданные шаблоны предназначены для разрыва этой тесной связи.

Examples

Шаблон Singleton

Шаблон Singleton предназначен для ограничения создания класса ровно на один экземпляр.

Этот шаблон используется в сценарии, где имеет смысл иметь только одно из следующего:

- один класс, который организует взаимодействия других объектов, например. Класс менеджера
- или один класс, который представляет собой уникальный, единственный ресурс, например. Компонент регистрации

Одним из наиболее распространенных способов реализации шаблона Singleton является статический **заводской метод**, такой как `CreateInstance()` или `GetInstance()` (или статическое свойство в C #, `Instance`), который затем предназначен для возврата одного и того же экземпляра.

Первый вызов метода или свойства создает и возвращает экземпляр Singleton. После этого метод всегда возвращает один и тот же экземпляр. Таким образом, существует только один экземпляр объекта `singleton`.

Предотвращение создания экземпляров с помощью `new` может быть выполнено путем создания `private` конструктора классов `private`.

Вот типичный пример кода для реализации шаблона Singleton в C #:

```
class Singleton
```

```

{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}

```

Чтобы проиллюстрировать этот шаблон далее, приведенный ниже код проверяет, возвращается ли идентичный экземпляр Singleton, когда свойство экземпляра вызывается более одного раза.

```

class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both Singleton objects above should now reference the same Singleton instance.
        if (Object.ReferenceEquals(s1, s2))
        {
            Console.WriteLine("Singleton is working");
        }
        else
        {
            // Otherwise, the Singleton Instance property is returning something
            // other than the unique, single instance when called.
            Console.WriteLine("Singleton is broken");
        }
    }
}

```

Примечание: эта реализация не является потокобезопасной.

Чтобы увидеть больше примеров, в том числе о том, как сделать этот поточно-безопасный,

посетите: [Реализация Singleton](#)

Синглтоны концептуально похожи на глобальную ценность и вызывают подобные конструктивные недостатки и проблемы. Из-за этого шаблон Singleton широко рассматривается как анти-шаблон.

Посещение [«Что так плохо в синглтонах?»](#) для получения дополнительной информации о проблемах, возникающих при их использовании.

В C# у вас есть возможность создать класс `static`, который делает все члены статическими, а класс не может быть создан. Учитывая это, обычно можно увидеть статические классы, используемые вместо шаблона Singleton.

Для основных различий между ними, посетите [C# Singleton Pattern Versus Static Class](#).

Заводская модель

Фабричный метод - один из шаблонов дизайна. Он используется для решения проблемы создания объектов без указания точного типа результата. В этом документе вы научитесь правильно использовать Factory Method DP.

Позвольте мне объяснить вам это на простом примере. Представьте, что вы работаете на заводе, производящем три типа устройств - Амперметр, вольтметр и измеритель сопротивления. Вы пишете программу для центрального компьютера, которая будет создавать выбранное устройство, но вы не знаете окончательного решения своего начальника о том, что производить.

Давайте создадим `IDevice` интерфейса с некоторыми общими функциями, которые есть у всех устройств:

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

Теперь мы можем создавать классы, которые представляют наши устройства. Эти классы должны реализовывать интерфейс `IDevice`:

```
public class AmMeter : IDevice
{
    private Random r = null;
    public AmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-25, 60); }
    public void TurnOff() { Console.WriteLine("AmMeter flashes lights saying good bye!"); }
}
```

```

    public void TurnOn() { Console.WriteLine("AmMeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(0, 1000000); }
    public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class VoltMeter : IDevice
{
    private Random r = null;
    public VoltMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-230, 230); }
    public void TurnOff() { Console.WriteLine("VoltMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("VoltMeter turns on..."); }
}

```

Теперь мы должны определить фабричный метод. Давайте создадим класс `DeviceFactory` со статическим методом внутри:

```

public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new AmMeter();
            case Device.VOLT: return new VoltMeter();
            case Device.OHM: return new OhmMeter();
            default: return new AmMeter();
        }
    }
}

```

Большой! Давайте проверим наш код:

```

public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
    }
}

```

```
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();

device = DeviceFactory.CreateDevice(Device.VOLT);
device.TurnOn();
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();

device = DeviceFactory.CreateDevice(Device.OHM);
device.TurnOn();
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();
}

}
```

Это пример вывода, который вы можете увидеть после запуска этого кода:

AmMeter включается ...

36

6

33

43

24

AmMeter мигает огнями, прощаясь!

Вольтметр включается ...

102

-61

85

138

36

Вольтметр мигает огнями, прощаясь!

OhmMeter включается ...

723828

368536

685412

800266

578595

OhmMeter мигает огнями, прощаясь!

Шаблон Builder

Отделите построение сложного объекта от его представления, чтобы тот же процесс построения мог создавать различные представления и обеспечивать высокий уровень контроля над сборкой объектов.

В этом примере демонстрируется шаблон Builder, в котором различные машины собираются поэтапно. Магазин использует VehicleBuilders для создания различных транспортных средств в серии последовательных шагов.

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

namespace GangOfFour.Builder
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Builder Design Pattern.
    /// </summary>
    public class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        public static void Main()
        {
            VehicleBuilder builder;

            // Create shop with vehicle builders
            Shop shop = new Shop();

            // Construct and display vehicles
            builder = new ScooterBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            builder = new CarBuilder();
            shop.Construct(builder);
        }
    }
}
```

```

builder.Vehicle.Show();

builder = new MotorCycleBuilder();
shop.Construct(builder);
builder.Vehicle.Show();

// Wait for user
Console.ReadKey();
}

}

/// <summary>
/// The 'Director' class
/// </summary>
class Shop
{
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
        vehicleBuilder.BuildFrame();
        vehicleBuilder.BuildEngine();
        vehicleBuilder.BuildWheels();
        vehicleBuilder.BuildDoors();
    }
}

/// <summary>
/// The 'Builder' abstract class
/// </summary>
abstract class VehicleBuilder
{
    protected Vehicle vehicle;

    // Gets vehicle instance
    public Vehicle Vehicle
    {
        get { return vehicle; }
    }

    // Abstract build methods
    public abstract void BuildFrame();
    public abstract void BuildEngine();
    public abstract void BuildWheels();
    public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
    public MotorCycleBuilder()
    {
        vehicle = new Vehicle("MotorCycle");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "MotorCycle Frame";
    }
}

```

```

public override void BuildEngine()
{
    vehicle["engine"] = "500 cc";
}

public override void BuildWheels()
{
    vehicle["wheels"] = "2";
}

public override void BuildDoors()
{
    vehicle["doors"] = "0";
}

/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class CarBuilder : VehicleBuilder
{
    public CarBuilder()
    {
        vehicle = new Vehicle("Car");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "Car Frame";
    }

    public override void BuildEngine()
    {
        vehicle["engine"] = "2500 cc";
    }

    public override void BuildWheels()
    {
        vehicle["wheels"] = "4";
    }

    public override void BuildDoors()
    {
        vehicle["doors"] = "4";
    }
}

/// <summary>
/// The 'ConcreteBuilder3' class
/// </summary>
class ScooterBuilder : VehicleBuilder
{
    public ScooterBuilder()
    {
        vehicle = new Vehicle("Scooter");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "Scooter Frame";
    }
}

```

```

}

public override void BuildEngine()
{
    vehicle["engine"] = "50 cc";
}

public override void BuildWheels()
{
    vehicle["wheels"] = "2";
}

public override void BuildDoors()
{
    vehicle["doors"] = "0";
}

/// <summary>
/// The 'Product' class
/// </summary>
class Vehicle
{
    private string _vehicleType;
    private Dictionary<string, string> _parts =
        new Dictionary<string, string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
        this._vehicleType = vehicleType;
    }

    // Indexer
    public string this[string key]
    {
        get { return _parts[key]; }
        set { _parts[key] = value; }
    }

    public void Show()
    {
        Console.WriteLine("\n-----");
        Console.WriteLine("Vehicle Type: {0}", _vehicleType);
        Console.WriteLine(" Frame : {0}", _parts["frame"]);
        Console.WriteLine(" Engine : {0}", _parts["engine"]);
        Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
        Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
}
}

```

Выход

Тип транспортного средства: Скутер
 Рамка: Скутер
 Двигатель: нет
 #Wheels: 2
 #Doors: 0

Тип автомобиля: Автомобиль

Рамка: Автомобильная рамка

Двигатель: 2500 куб. См.

#Wheels: 4

#Doors: 4

Тип транспортного средства: Мотоцикл

Рамка: рамка мотоцикла

Двигатель: 500 куб. См

#Wheels: 2

#Doors: 0

Шаблон прототипа

Укажите тип объектов, созданных с использованием прототипа, и создайте новые объекты, скопировав этот прототип.

В этом примере демонстрируется шаблон Prototype, в котором новые объекты Color создаются путем копирования уже существующих, определяемых пользователем цветов одного и того же типа.

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

namespace GangOfFour.Prototype
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Prototype Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            ColorManager colormanager = new ColorManager();

            // Initialize with standard colors
            colormanager["red"] = new Color(255, 0, 0);
            colormanager["green"] = new Color(0, 255, 0);
            colormanager["blue"] = new Color(0, 0, 255);

            // User adds personalized colors
            colormanager["angry"] = new Color(255, 54, 0);
            colormanager["peace"] = new Color(128, 211, 128);
            colormanager["flame"] = new Color(211, 34, 20);

            // User clones selected colors
            Color color1 = colormanager["red"].Clone() as Color;
            Color color2 = colormanager["peace"].Clone() as Color;
            Color color3 = colormanager["flame"].Clone() as Color;
        }
    }
}
```

```

        // Wait for user
        Console.ReadKey();
    }
}

/// <summary>
/// The 'Prototype' abstract class
/// </summary>
abstract class ColorPrototype
{
    public abstract ColorPrototype Clone();
}

/// <summary>
/// The 'ConcretePrototype' class
/// </summary>
class Color : ColorPrototype
{
    private int _red;
    private int _green;
    private int _blue;

    // Constructor
    public Color(int red, int green, int blue)
    {
        this._red = red;
        this._green = green;
        this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
        Console.WriteLine(
            "Cloning color RGB: {0,3},{1,3},{2,3}",
            _red, _green, _blue);

        return this.MemberwiseClone() as ColorPrototype;
    }
}

/// <summary>
/// Prototype manager
/// </summary>
class ColorManager
{
    private Dictionary<string, ColorPrototype> _colors =
        new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
        get { return _colors[key]; }
        set { _colors.Add(key, value); }
    }
}

```

Выход:

Цвет клонирования RGB: 255, 0, 0

Цвет клонирования RGB: 128 211 128

Цвет клонирования RGB: 211, 34, 20

Абстрактный шаблон завода

Предоставить интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

В этом примере демонстрируется создание разных животных миров для компьютерной игры с использованием разных фабрик. Хотя животные, созданные фабриками Континента, различны, взаимодействие между животными остается неизменным.

```
using System;

namespace GangOfFour.AbstractFactory
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Abstract Factory Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        public static void Main()
        {
            // Create and run the African animal world
            ContinentFactory africa = new AfricaFactory();
            AnimalWorld world = new AnimalWorld(africa);
            world.RunFoodChain();

            // Create and run the American animal world
            ContinentFactory america = new AmericaFactory();
            world = new AnimalWorld(america);
            world.RunFoodChain();

            // Wait for user input
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'AbstractFactory' abstract class
    /// </summary>
    abstract class ContinentFactory
    {
        public abstract Herbivore CreateHerbivore();
        public abstract Carnivore CreateCarnivore();
    }

    /// <summary>
```

```

/// The 'ConcreteFactory1' class
/// </summary>
class AfricaFactory : ContinentFactory
{
    public override Herbivore CreateHerbivore()
    {
        return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
        return new Lion();
    }
}

/// <summary>
/// The 'ConcreteFactory2' class
/// </summary>
class AmericaFactory : ContinentFactory
{
    public override Herbivore CreateHerbivore()
    {
        return new Bison();
    }
    public override Carnivore CreateCarnivore()
    {
        return new Wolf();
    }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
    public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
    public override void Eat(Herbivore h)
    {
        // Eat Wildebeest
        Console.WriteLine(this.GetType().Name +
            " eats " + h.GetType().Name);
    }
}

```

```

        }
    }

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
    public override void Eat(Herbivore h)
    {
        // Eat Bison
        Console.WriteLine(this.GetType().Name +
            " eats " + h.GetType().Name);
    }
}

/// <summary>
/// The 'Client' class
/// </summary>
class AnimalWorld
{
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
        _carnivore = factory.CreateCarnivore();
        _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
        _carnivore.Eat(_herbivore);
    }
}
}

```

Выход:

Лев ест Wildebeest

Волк ест Bison

Прочтайте Создание шаблонов проектирования онлайн:

<https://riptutorial.com/ru/csharp/topic/6654/создание-шаблонов-проектирования>

глава 148: Статические классы

Examples

Статическое ключевое слово

Статическое ключевое слово означает 2 вещи:

1. Это значение не изменяется от объекта к объекту, а скорее изменяется в классе в целом
2. Статические свойства и методы не требуют экземпляра.

```
public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //Create an instance
        var fool = new Foo();
        Console.WriteLine(fool.NonStaticCounter); //this will print "1"

        //Notice this next call doesn't access the instance but calls by the class name.
        Console.WriteLine(Foo.Counter); //this will also print "1"

        //Create a second instance
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //this will print "1"

        Console.WriteLine(Foo.Counter); //this will now print "2"
        //The static property incremented on both instances and can persist for the whole
        class
    }
}
```

Статические классы

Ключевое слово «static» при обращении к классу имеет три эффекта:

1. Вы **не можете** создать экземпляр статического класса (это даже удаляет конструктор

по умолчанию)

2. Все свойства и методы в классе также **должны** быть статическими.

3. static класс является sealed классом, то есть он не может быть унаследован.

```
public static class Foo
{
    //Notice there is no constructor as this cannot be an instance
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //this will print 1

        //var fool = new Foo();
        //this line would break the code as the Foo class does not have a constructor
    }
}
```

Статический класс жизни

static класс лениво инициализируется при доступе к члену и живет в течение всего срока действия домена приложения.

```
void Main()
{
    Console.WriteLine("Static classes are lazily initialized");
    Console.WriteLine("The static constructor is only invoked when the class is first
accessed");
    Foo.SayHi();

    Console.WriteLine("Reflecting on a type won't trigger its static .ctor");
    var barType = typeof(Bar);

    Console.WriteLine("However, you can manually trigger it with
System.Runtime.CompilerServices.RuntimeHelpers");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// Define other methods and classes here
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
    {
        Console.WriteLine("Foo: Hi");
    }
}
```

```
        }
    }
public static class Bar
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}
```

Прочтайте Статические классы онлайн: <https://riptutorial.com/ru/csharp/topic/1653/статические-классы>

глава 149: Строковые escape-последовательности

Синтаксис

- \ ' - одинарная кавычка (0x0027)
- \ " - двойная кавычка (0x0022)
- \\ - обратная косая черта (0x005C)
- \ 0 - null (0x0000)
- \ a - alert (0x0007)
- \ b - backspace (0x0008)
- \ f - подача формы (0x000C)
- \ n - новая строка (0x000A)
- \ r - возврат каретки (0x000D)
- \ t - горизонтальная вкладка (0x0009)
- \ v - вертикальная вкладка (0x000B)
- \ u0000 - \ uFFFF - символ Юникода
- \ x0 - \ xFFFF - символ Юникода (код с переменной длиной)
- \ U00000000 - \ U0010FFFF - символ Юникода (для генерации суррогатов)

замечания

Последовательности escape-последовательности строк преобразуются в соответствующий символ во **время компиляции**. Обычные строки, которые содержат обратную косую черту, **не** преобразуются.

Например, строки `notEscaped` и `notEscaped2` **ниже** не преобразуются в символ новой строки, а остаются как два разных символа (' \' и ' n ').

```
string escaped = "\n";
string notEscaped = "\\\" + "n";
string notEscaped2 = "\\n";

Console.WriteLine(escaped.Length); // 1
Console.WriteLine(notEscaped.Length); // 2
Console.WriteLine(notEscaped2.Length); // 2
```

Examples

Управляющие последовательности символов Unicode

```
string sqrt = "\u221A"; // √
```

```
string emoji = "\U0001F601"; // ☺
string text = "\u0022Hello World\u0022"; // "Hello World"
string variableWidth = "\x22Hello World\x22"; // "Hello World"
```

Выделение специальных символов в символьных литералах

Апострофы

```
char apostrophe = '\'';
```

бэкслэш

```
char oneBackslash = '\\';
```

Выделение специальных символов в строковых литералах

бэкслэш

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\\myfile.txt";
```

Во втором примере используется [строковый литерал](#), который не обрабатывает обратную косую черту как escape-символ.

Цитаты

```
string text = "\"Hello World!\\"", said the quick brown fox.";
string verbatimText = @"""Hello World!""", said the quick brown fox.;"
```

Обе переменные будут содержать один и тот же текст.

«Привет, мир!», - сказала быстрая коричневая лиса.

Newlines

Вербальные строковые литералы могут содержать символы новой строки:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Обе переменные будут содержать один и тот же текст.

Неопознанные escape-последовательности создают ошибки времени компиляции

Следующие примеры не будут компилироваться:

```
string s = "\c";
char c = '\c';
```

Вместо этого они будут генерировать ошибку `Unrecognized escape sequence` **во время компиляции**.

Использование escape-последовательностей в идентификаторах

Последовательности Escape не ограничиваются `string` и `char` символами.

Предположим, вам необходимо переопределить сторонний метод:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

и предположим, что символ `€` недоступен в кодировке символов, используемой для исходных файлов C#. Вам повезло, разрешено использовать `\u#####` типа `\u###` или `\u#####` в **идентификаторах** кода. Так что законно писать:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

и компилятор C# будет знать, что `€` и `\u0152` являются `\u0152` и тем же символом.

(Тем не менее, может быть хорошей идеей переключиться на UTF-8 или аналогичную кодировку, которая может обрабатывать все символы.)

Прочтайте Строковые escape-последовательности онлайн:

<https://riptutorial.com/ru/csharp/topic/39/строковые-escape-последовательности>

глава 150: Структурные шаблоны проектирования

Вступление

Структурные шаблоны проектирования - это модели, которые описывают, как объекты и классы могут объединяться и формировать большую структуру, а также упрощают проектирование, определяя простой способ реализации отношений между сущностями. Описаны семь структурных структур. Они следующие: адаптер, мост, композитный, декоратор, фасад, муки и прокси

Examples

Шаблон проектирования адаптера

«**Адаптер**», как следует из названия, - это объект, который позволяет двум взаимно несовместимым интерфейсам взаимодействовать друг с другом.

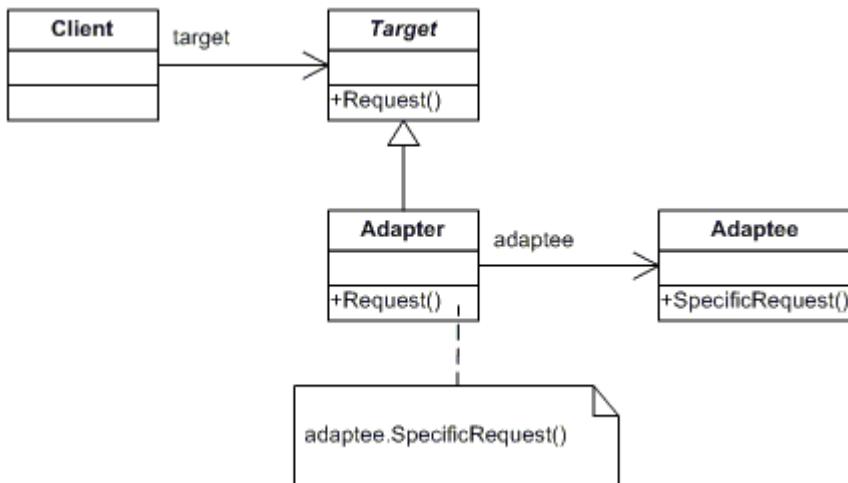
Например: если вы покупаете Iphone 8 (или любой другой продукт Apple), вам нужно много адаптеров. Поскольку интерфейс по умолчанию не поддерживает аудио jас или USB. С помощью этих адаптеров вы можете использовать наушники с проводами или использовать обычный кабель Ethernet. Таким образом, «*два взаимно несовместимых интерфейса взаимодействуют друг с другом*» .

Таким образом, в технических терминах это означает: Преобразование интерфейса класса в другой интерфейс, который ожидают клиенты. Адаптер позволяет классам работать вместе, что в противном случае не могло бы быть связано с несовместимыми интерфейсами. Классы и объекты, участвующие в этом шаблоне:

Схема адаптера выдает 4 элемента

1. **ITarget:** это интерфейс, который клиент использует для достижения функциональности.
2. **Adaptee:** Это функциональность, которую клиент хочет, но ее интерфейс несовместим с клиентом.
3. **Клиент.** Это класс, который хочет достичь некоторой функциональности, используя код адаптируемого.
4. **Адаптер:** это класс, который будет реализовывать ITarget и будет вызывать код Adaptee, который клиент хочет вызвать.

UML



Пример первого кода (теоретический пример) .

```
public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
```

Второй пример кода (Real world implementation)

```

/// <summary>
/// Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not
/// compatible with the client.
/// </summary>
public class CompanyEmployees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];
        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };

        return employees;
    }
}

/// <summary>
/// Client: This is the class which wants to achieve some functionality by using the adaptee's
/// code (list of employees).
/// </summary>
public class ThirdPartyBillingSystem
{
    /*
     * This class is from a third party and you don't have any control over it.
     * But it requires a Employee list to do its work
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // call the client list in the interface
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("##### Employee List #####");
        foreach (var item in employee)
        {
            Console.Write(item);
        }
    }
}

/// <summary>
/// Adapter: This is the class which would implement ITarget and would call the Adaptee code

```

```

which the client wants to call.
/// </summary>
public class EmployeeAdapter : CompanyEmployees, ITarget
{
    public List<string> GetEmployeeList()
    {
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }

        return employeeList;
    }
}

///
/// Demo
///
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}

```

Когда использовать

- Разрешить системе использовать классы другой системы, которые несовместимы с ней.
- Разрешить связь между новой и уже существующей системой, которые независимы друг от друга
- Ado.Net SqlDataAdapter, OracleAdapter, MySqlAdapter - лучший пример шаблона адаптера.

Прочтайте Структурные шаблоны проектирования онлайн:

<https://riptutorial.com/ru/csharp/topic/9764/структурные-шаблоны-проектирования>

глава 151: Структуры

замечания

В отличие от классов, `struct` является типом значения и создается *по умолчанию* в локальном стеке, а не в управляемой куче. Это означает, что после того, как конкретный стек выходит за пределы области, `struct` де-распределяется. Содержащиеся ссылочные типы де-выделенной `struct` также перемещаются, как только GC определяет, что они больше не ссылаются на `struct`.

`struct` `S` не может наследовать и не может быть основанием для наследования, они неявно запечатаны и также не могут включать `protected` члены. Однако `struct` может реализовать интерфейс, как это делают классы.

Examples

Объявление структуры

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- Поля `struct` экземпляра могут быть заданы с помощью параметризованного конструктора или индивидуально после построения `struct`.
- Конкретные члены могут быть инициализированы только конструктором.
- `struct` определяет закрытый тип, который неявно наследуется от `System.ValueType`.
- Структуры не могут наследовать ни от какого другого типа, но могут реализовывать интерфейсы.
- Структуры копируются при присваивании, то есть все данные копируются в новый

экземпляр, а изменения в один из них не отражаются другим.

- Стока не может быть `null`, хотя она может использоваться как тип с `null` значением:

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Структуры могут быть созданы с использованием или без использования `new` оператора.

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

Тем не менее, `new` оператор должен использоваться для использования инициализатора:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

Структура может объявлять все, что может объявить класс, за некоторыми исключениями:

- Структура не может объявить конструктор без параметров. Поля `struct` экземпляра могут быть заданы с помощью параметризованного конструктора или индивидуально после построения `struct`. Конкретные члены могут быть инициализированы только конструктором.
- Структура не может объявлять элементы защищенными, поскольку она неявно закрыта.
- Поля `Struct` могут быть инициализированы только в том случае, если они являются константными или статическими.

Использование структуры

С конструктором:

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;
```

```

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// Output X=1, Y=2, Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// Output X=0, Y=2, Z=3

Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);

```

Без конструктора:

```

Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// Output X=1, Y=2, Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;

```

Если мы используем конструкцию с ее конструктором, у нас не будет проблем с неназначенным полем (каждое неназначенное поле имеет нулевое значение).

В отличие от классов, структура не должна быть построена, то есть нет необходимости использовать новое ключевое слово, если только вам не нужно называть один из конструкторов. Структуре не требуется новое ключевое слово, потому что это тип значения и, следовательно, не может быть нулевым.

Интерфейс реализации проекта

```

public interface IShape
{
    decimal Area();
}

public struct Rectangle : IShape
{

```

```
public decimal Length { get; set; }
public decimal Width { get; set; }

public decimal Area()
{
    return Length * Width;
}
```

Структуры копируются при назначении

Since structs - типы значений, все данные копируются при назначении, а любая модификация новой копии не изменяет данные для оригинальной копии. В приведенном ниже фрагменте кода показано, что `p1` копируется на `p2` а изменения, сделанные на `p1`, не влияют на экземпляр `p2`.

```
var p1 = new Point {
    x = 1,
    y = 2
};

Console.WriteLine($"{p1.x} {p1.y}"); // 1 2

var p2 = p1;
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2

p1.x = 3;
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

Прочтайте Структуры онлайн: <https://riptutorial.com/ru/csharp/topic/778/структуры>

глава 152: Сценарий C

Examples

Простая оценка кода

Вы можете оценить любой действительный код C #:

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) -
DateTime.Now");
```

Если тип не указан, результатом будет `object` :

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

Прочтайте Сценарий C # онлайн: <https://riptutorial.com/ru/csharp/topic/3780/сценарий-c-sharp>

глава 153: Таймеры

Синтаксис

- `myTimer.Interval` - устанавливает, как часто вызывается событие «Tick» (в миллисекундах)
- `myTimer.Enabled` - логическое значение, которое устанавливает таймер для включения / отключения
- `myTimer.Start()` - запуск таймера.
- `myTimer.Stop()` - останавливает таймер.

замечания

Если вы используете Visual Studio, таймеры могут быть добавлены в виде элемента управления непосредственно в вашу форму из панели инструментов.

Examples

Многопоточные таймеры

`System.Threading.Timer` - Простой многопоточный таймер. Содержит два метода и один конструктор.

Пример: таймер вызывает метод `DataWrite`, который пишет «многопоточность, выполненный ...», по прошествии пяти секунд, а затем каждую секунду после этого, пока пользователь не нажмет `Enter`:

```
using System;
using System.Threading;
class Program
{
    static void Main()
    {
        // First interval = 5000ms; subsequent intervals = 1000ms
        Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
        Console.ReadLine();
        timer.Dispose(); // This both stops the timer and cleans up.
    }

    static void DataWrite (object data)
    {
        // This runs on a pooled thread
        Console.WriteLine (data); // Writes "multithread executed..."
    }
}
```

Примечание. Будет опубликован отдельный раздел для утилизации многопоточных таймеров.

change Этот метод можно вызвать, если вы хотите изменить интервал таймера.

timeout.Infinite - если вы хотите запустить только один раз. Задайте это в последнем аргументе конструктора.

System.Timers - еще один класс таймера, предоставляемый .NET Framework. Он обертыает System.Threading.Timer .

Особенности:

- IComponent - IComponent его размещение в лотке компонента Designer в Visual Studio
- Свойство Interval вместо метода Change
- Elapsed event вместо delegate обратного вызова
- Enabled для запуска и остановки таймера (default value = false)
- Start & Stop если вы запутались в свойстве Enabled (выше точки)
- AutoReset - для указания повторяющегося события (default value = true)
- Свойство SynchronizingObject с методами Invoke И BeginInvoke для безопасных методов вызова элементов WPF и элементов управления Windows Forms

Пример, представляющий все перечисленные выше функции:

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
    static void Main()
    {
        Timer timer = new Timer(); // Doesn't require any args
        timer.Interval = 500;
        timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
        timer.Start(); // Start the timer
        Console.ReadLine();
        timer.Stop(); // Stop the timer
        Console.ReadLine();
        timer.Start(); // Restart the timer
        Console.ReadLine();
        timer.Dispose(); // Permanently stop the timer
    }

    static void timer_Elapsed(object sender, EventArgs e)
    {
        Console.WriteLine ("Tick");
    }
}
```

Multithreaded timers - Используйте пул потоков, чтобы несколько потоков могли обслуживать множество таймеров. Это означает, что метод обратного вызова или Elapsed событие может запускаться по другому потоку каждый раз, когда он вызывается.

`Elapsed` - это событие всегда срабатывает вовремя, независимо от того, является ли предыдущим `Elapsed` закончило событие выполнения. Из-за этого обратные вызовы или обработчики событий должны быть потокобезопасными. Точность многопоточных таймеров зависит от ОС и обычно составляет 10-20 мс.

`interop` - когда вам нужна более высокая точность, используйте это и вызовите мультимедийный таймер Windows. Это имеет точность до 1 мс и определяется в `winmm.dll`.

`timeBeginPeriod` - сначала `timeBeginPeriod` это, чтобы сообщить ОС, что вам нужна высокая точность синхронизации

`timeSetEvent` - вызывать это через `timeBeginPeriod` для запуска мультимедийного таймера.

`timeKillEvent` - вызывать это, когда вы закончите, это останавливает таймер

`timeEndPeriod` - вызов этого, чтобы сообщить ОС, что вам больше не нужна высокая точность синхронизации.

Вы можете найти полные примеры в Интернете, которые используют мультимедийный таймер, `dllimport` ПОИСК КЛЮЧЕВЫХ СЛОВ `dllimport winmm.dll timesetevent`.

Создание экземпляра таймера

Таймеры используются для выполнения задач через определенные промежутки времени (до X каждые Y секунд). Ниже приведен пример создания нового экземпляра таймера.

ПРИМЕЧАНИЕ . Это относится к таймерам, использующим WinForms. Если вы используете WPF, вы можете посмотреть в `DispatcherTimer`

```
using System.Windows.Forms; //Timers use the Windows.Forms namespace

public partial class Form1 : Form
{
    Timer myTimer = new Timer(); //create an instance of Timer named myTimer

    public Form1()
    {
        InitializeComponent();
    }
}
```

Назначение обработчика события «Tick» для таймера

Все действия, выполняемые таймером, обрабатываются в событии «Tick».

```
public partial class Form1 : Form
```

```

{
    Timer myTimer = new Timer();

    public Form1()
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}

```

Пример: использование таймера для простого обратного отсчета.

```

public partial class Form1 : Form
{
    Timer myTimer = new Timer();
    int timeLeft = 10;

    public Form1()
    {
        InitializeComponent();

        //set properties for the Timer
        myTimer.Interval = 1000;
        myTimer.Enabled = true;

        //Set the event handler for the timer, named "myTimer_Tick"
        myTimer.Tick += myTimer_Tick;

        //Start the timer as soon as the form is loaded
        myTimer.Start();

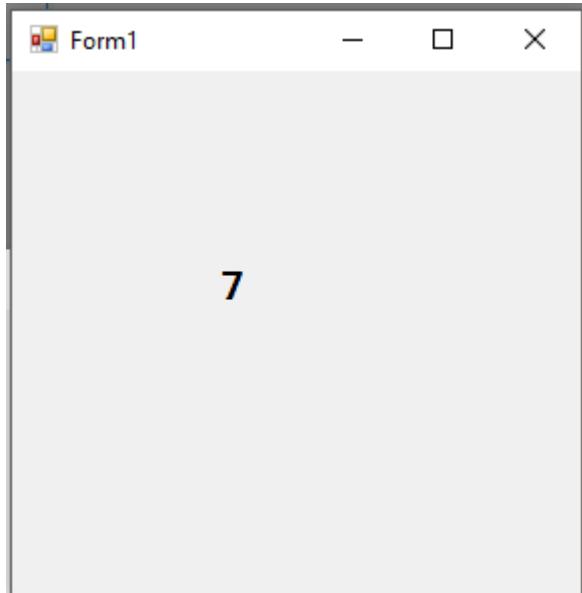
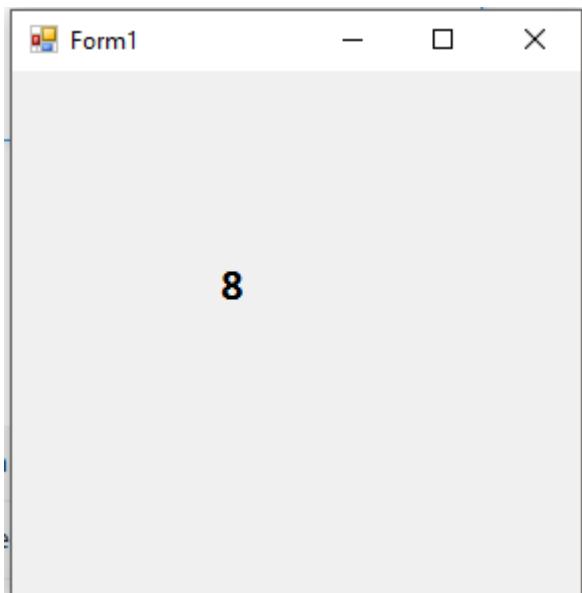
        //Show the time set in the "timeLeft" variable
        lblCountDown.Text = timeLeft.ToString();
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        //perform these actions at the interval set in the properties.
        lblCountDown.Text = timeLeft.ToString();
        timeLeft -= 1;

        if (timeLeft < 0)
        {
            myTimer.Stop();
        }
    }
}

```

Результаты в ...



И так далее...

Прочтайте Таймеры онлайн: <https://riptutorial.com/ru/csharp/topic/3829/таймеры>

глава 154: Тип значения vs Тип ссылки

Синтаксис

- Передача по ссылке: `public void Double (ref int number.ToDouble) {}`

замечания

Вступление

Типы значений

Типы значений являются более простыми из двух. Типы значений часто используются для представления самих данных. Целое число, булево или точка в трехмерном пространстве - все примеры хороших типов значений.

Типы значений (structs) объявляются с помощью ключевого слова `struct`. См. Раздел синтаксиса для примера того, как объявить новую структуру.

Вообще говоря, у нас есть 2 ключевых слова, которые используются для объявления типов значений:

- Структуры
- Перечисления

Типы ссылок

Ссылочные типы немного сложнее. Ссылочные типы являются традиционными объектами в смысле объектно-ориентированного программирования. Таким образом, они поддерживают наследование (и преимущества там), а также поддерживают финализаторы.

В C #, как правило, мы имеем такие ссылочные типы:

- Классы
- Делегаты
- Интерфейсы

Новые ссылочные типы (классы) объявляются с использованием ключевого слова `class`. Например, см. Раздел синтаксиса о том, как объявить новый тип ссылки.

Основные отличия

Основные различия между ссылочными типами и типами значений приведены ниже.

Типы значений существуют в стеке, ссылочные типы существуют в куче

Это часто упоминается различие между ними, но на самом деле то, что сводится к тому, что, когда вы используете тип значения в C #, например `int`, программа будет использовать эту переменную для прямого обращения к этому значению. Если вы скажете `int mine = 0`, то переменная шахта относится непосредственно к 0, что является эффективным. Однако ссылочные типы фактически удерживают (как следует из названия) ссылку на базовый объект, это похоже на указатели на других языках, таких как C ++.

Вы можете не заметить эффекты этого немедленно, но эффекты там, мощные и тонкие. См. Пример изменения типов ссылок в другом месте для примера.

Это различие является основной причиной следующих различий и заслуживает внимания.

Типы значений не изменяются при изменении их в методе, типы ссылок делают

Когда тип значения передается в метод в качестве параметра, если метод каким-либо образом изменяет значение, значение не изменяется. Напротив, передача ссылочного типа в тот же самый метод и его изменение изменят базовый объект, так что другие вещи, которые используют тот же самый объект, будут иметь вновь измененный объект, а не его исходное значение.

См. Пример типов значений по сравнению с типами ссылок в методах для получения дополнительной информации.

Что делать, если я хочу их изменить?

Просто передайте их в свой метод, используя ключевое слово «`ref`», и вы передаете этот объект по ссылке. Смысл, это тот же объект в памяти. Таким образом, ваши изменения будут соблюдены. См. Пример при отправке по ссылке для примера.

Типы значений не могут быть нулевыми, ссылочные типы могут

В значительной степени, как говорится, вы можете назначить `null` ссылочному типу, то есть назначенная вам переменная не может иметь никакого действительного объекта, назначенного ей. Однако в случае типов значений это невозможно. Тем не менее, вы можете использовать `Nullable`, чтобы ваш тип значения имел значение `NULL`, если это требование, но если это то, что вы рассматриваете, подумайте, действительно ли класс может быть лучшим подходом здесь, если он является вашим собственным типом.

Examples

Изменение значений в другом месте

```
public static void Main(string[] args)
{
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
    objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}
```

Вы заметите, что, несмотря на то, что список списков печати был сделан до исправления имен учеников после опечаток, метод PrintPrintingList все еще печатает исправленные имена:

```
Scott Duke
Vincent Kong
Craig Brett
```

Это связано с тем, что в обоих списках содержится список ссылок на те же ученики. SO изменение базового объекта-ученика распространяется на использование в любом списке.

Вот как выглядит класс ученика.

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Передача по ссылке

Если вы хотите, чтобы примеры методов типов и ссылочных типов в правиле корректно работали, используйте ключевое слово `ref` в вашей сигнатуре метода для параметра, который вы хотите передать по ссылке, а также при вызове метода.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

Выполнение этих изменений сделает обновление номера ожидаемым, что означает, что вывод консоли для числа будет 8.

Передача по ссылке с использованием ключевого слова `ref`.

Из [документации](#) :

В C # аргументы могут передаваться параметрам либо по значению, либо по ссылке. Передача по ссылке позволяет членам функции, методам, свойствам, индексаторам, операторам и конструкторам изменять значение параметров и сохранять это изменение в вызывающей среде. Чтобы передать параметр по ссылке, используйте ключевое слово `ref` ИЛИ `out` .

Разница между `ref` И `out` заключается в том `out` что передаваемый параметр должен быть назначен перед функцией `ends.in` контрастные параметры, переданные с `ref` могут быть изменены или оставлены без изменений.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
    }
}
```

```

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}

```

Выход :

```

Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7

```

присваивание

```

var a = new List<int>();
var b = a;
a.Add(5);
Console.WriteLine(a.Count); // prints 1
Console.WriteLine(b.Count); // prints 1 as well

```

Причение переменной `List<int>` не создает копию `List<int>`. Вместо этого он копирует ссылку на `List<int>`. Мы называем типы, которые ведут себя так, как *ссылочные типы*.

Разница с параметрами метода `ref` и `out`

Существует два возможных способа передать тип значения по ссылке: `ref` и `out`. Разница заключается в том, что при прохождении его `ref` значение должно быть инициализировано, но не при прохождении его `out`. Использование `out` гарантирует, что переменная имеет

значение после вызова метода:

```
public void ByRef(ref int value)
{
    Console.WriteLine(nameof(ByRef) + value);
    value += 4;
    Console.WriteLine(nameof(ByRef) + value);
}

public void ByOut(out int value)
{
    value += 4 // CS0269: Use of unassigned out parameter 'value'
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter
    'value'

    value = 4;
    Console.WriteLine(nameof(ByOut) + value);
}

public void TestOut()
{
    int outValue1;
    ByOut(out outValue1); // prints 4

    int outValue2 = 10;    // does not make any sense for out
    ByOut(out outValue2); // prints 4
}

public void TestRef()
{
    int refValue1;
    ByRef(ref refValue1); // S0165 Use of unassigned local variable 'refValue'

    int refValue2 = 0;
    ByRef(ref refValue2); // prints 0 and 4

    int refValue3 = 10;
    ByRef(ref refValue3); // prints 10 and 14
}
```

Улов в том, что при использовании `out` параметра `must` быть инициализирован перед выходом из метода, поэтому следующий способ можно с `ref`, но не с `out`:

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current
method
```

Это связано с тем, что если `condition` не выполняется, `value` становится неназначенным.

параметры `ref` vs `out`

Код

```
class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not initialized only declared in the
        method declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Выход

```
Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30
```

```
Inside Main - Before CalleeOut: a = 30
Inside CalleeOut a : 25
Inside Main - After CalleeOut: a = 25
```

Прочтайте Тип значения vs Тип ссылки онлайн: <https://riptutorial.com/ru/csharp/topic/3014/тип-значения-vs-тип-ссылки>

глава 155: указатели

замечания

Указатели и `unsafe`

Из-за своей природы указатели производят непроверяемый код. Таким образом, использование любого типа указателя требует `unsafe` контекста.

Тип `System.IntPtr` - безопасная оболочка вокруг `void*`. Он предназначен как более удобная альтернатива `void*` когда небезопасный контекст не требуется для выполнения задачи.

Неопределенное поведение

Как и в С и С++, неправильное использование указателей может вызывать неопределенное поведение, при этом возможны побочные эффекты, являющиеся повреждением памяти и выполнением непреднамеренного кода. Из-за непроверяемого характера большинства операций указателя правильное использование указателей полностью зависит от программиста.

Типы, которые поддерживают указатели

В отличие от С и С++, не все типы С# имеют соответствующие типы указателей. Тип `t` может иметь соответствующий тип указателя, если применимы оба следующих критерия:

- `t` - тип структуры или тип указателя.
- `t` содержит только элементы, которые рекурсивно удовлетворяют обоим этим критериям.

Examples

Указатели для доступа к массиву

В этом примере показано, как указатели могут использоваться для доступа С к массивам С#.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
```

```

    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
        }
    }
}

```

`unsafe` **ключевое слово** требуется, поскольку доступ к указателю не будет выдавать никаких проверок границ, которые обычно испускаются при регулярном доступе к массивам C #.

`fixed` **ключевое слово** сообщает компилятору C #, чтобы он выдавал инструкции для привязки объекта безопасным способом. Для обеспечения того, чтобы сборщик мусора не перемещал массив в памяти, требуется принудительное закрепление, поскольку это приведет к недействительности указателей, указывающих внутри массива.

Арифметика указателя

Сложение и вычитание в указателях работают иначе, чем целые. Когда указатель увеличивается или уменьшается, адрес, на который он указывает, увеличивается или уменьшается по размеру типа референта.

Например, тип `int` (псевдоним для `System.Int32`) имеет размер 4. Если `int` может быть сохранен в адресе 0, последующий `int` может быть сохранен в адресе 4 и так далее. В коде:

```

var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8

```

Аналогично, тип `long` (псевдоним для `System.Int64`) имеет размер 8. Если `long` может быть сохранен в адресе 0, последующий `long` может быть сохранен в адресе 8 и так далее. В коде:

```

var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16

```

Тип `void` является специальным, а указатели `void` также являются специальными, и они используются в качестве уловки, когда тип неизвестен или не имеет значения. Из-за их размерно-агностического характера указатели `void` не могут увеличиваться или уменьшаться:

```
var ptr = (void*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

Звездочка является частью типа

В С и С ++ звездочка в объявлении переменной указателя является *частью объявляемого выражения*. В С # звездочка в объявлении является *частью типа*.

В С, С ++ и С # следующий фрагмент объявляет указатель `int`:

```
int* a;
```

В С и С ++ следующий фрагмент объявляет `int` указатель и переменную `int`. В С # он объявляет два указателя `int`:

```
int* a, b;
```

В С и С ++ следующий фрагмент объявляет два указателя `int`. В С # это недопустимо:

```
int *a, *b;
```

недействительным *

С # наследует от С и С ++ использование `void*` как агностик типа-агностик и размер-агностик.

```
void* ptr;
```

Любой тип указателя можно присвоить `void*` используя неявное преобразование:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

Обратное требует явного преобразования:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

Доступ пользователей с помощью ->

С # наследует от С и С ++ использование символа `->` как средство доступа к членам экземпляра с помощью типизированного указателя.

Рассмотрим следующую структуру:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

Это пример использования `->` для доступа к его членам:

```
Vector2 v;
v.X = 5;
v.Y = 10;

Vector2* ptr = &v;
int x = ptr->X;
int y = ptr->Y;
string s = ptr->ToString();

Console.WriteLine(x); // prints 5
Console.WriteLine(y); // prints 10
Console.WriteLine(s); // prints Vector2
```

Общие указатели

Критерии, которые должен удовлетворять тип для поддержки указателей (см. *Примечания*), не могут быть выражены в терминах общих ограничений. Поэтому любая попытка объявить указатель на тип, предоставленный с помощью параметра типового типа, не будет выполнена.

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

Прочтайте указатели онлайн: <https://riptutorial.com/ru/csharp/topic/5524/указатели>

глава 156: Указатели и небезопасный код

Examples

Введение в небезопасный код

С # позволяет использовать переменные указателя в функции блока кода, когда он отмечен `unsafe` модификатором. Небезопасный код или неуправляемый код - это блок кода, который использует переменную указателя.

Указатель - это переменная, значение которой является адресом другой переменной, то есть прямым адресом ячейки памяти. подобно любой переменной или константе, вы должны объявить указатель, прежде чем сможете использовать его для хранения любого адреса переменной.

Общая форма объявления указателя:

```
type *var-name;
```

Ниже приведены допустимые объявления указателей:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch;    /* pointer to a character */
```

Следующий пример иллюстрирует использование указателей в C # с использованием небезопасного модификатора:

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ", var);
            Console.WriteLine("Address is: {0}", (int)p);
            Console.ReadKey();
        }
    }
}
```

Когда вышеуказанный код был скомпилирован и выполнен, он производит следующий результат:

```
Data is: 20
Address is: 99215364
```

Вместо того, чтобы объявить целый метод небезопасным, вы также можете объявить часть кода небезопасной:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

Получение значения данных с помощью указателя

Вы можете получить данные, хранящиеся в расположенной по ссылке переменной указателя, используя метод `ToString()`. Следующий пример демонстрирует это:

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        public static void Main()
        {
            unsafe
            {
                int var = 20;
                int* p = &var;
                Console.WriteLine("Data is: {0} " , var);
                Console.WriteLine("Data is: {0} " , p->ToString());
                Console.WriteLine("Address is: {0} " , (int)p);
            }

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

Когда вышеуказанный код был скомпилирован и выполнен, он производит следующий результат:

```
Data is: 20
Data is: 20
Address is: 77128984
```

Передача указателей в качестве параметров методов

Вы можете передать переменную указателя методу в качестве параметра. Следующий пример иллюстрирует это:

```
using System;
```

```

namespace UnsafeCodeApplication
{
    class TestPointer
    {
        public unsafe void swap(int* p, int *q)
        {
            int temp = *p;
            *p = *q;
            *q = temp;
        }

        public unsafe static void Main()
        {
            TestPointer p = new TestPointer();
            int var1 = 10;
            int var2 = 20;
            int* x = &var1;
            int* y = &var2;

            Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
            p.swap(x, y);

            Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
            Console.ReadKey();
        }
    }
}

```

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

```

Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10

```

Доступ к элементам массива с помощью указателя

В С # имя массива и указатель на тип данных, такой же, как данные массива, не являются одним и тем же типом переменной. Например, `int *p` и `int[] p` не являются одинаковыми. Вы можете увеличивать переменную указателя `p` поскольку она не фиксирована в памяти, но адрес массива фиксирован в памяти, и вы не можете увеличить его.

Поэтому, если вам нужно получить доступ к данным массива с помощью переменной указателя, как это обычно делается на С или С ++, вам нужно исправить указатель, используя ключевое слово `fixed`.

Следующий пример демонстрирует это:

```

using System;
namespace UnsafeCodeApplication
{
    class TestPointer
    {
        public unsafe static void Main()
        {

```

```
int[] list = {10, 100, 200};  
fixed(int *ptr = list)  
  
/* let us have array address in pointer */  
for (int i = 0; i < 3; i++)  
{  
    Console.WriteLine("Address of list[{0}]={1}", i, (int)(ptr + i));  
    Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));  
}  
  
Console.ReadKey();  
}  
}  
}
```

Когда вышеуказанный код был скомпилирован и выполнен, он производит следующий результат:

```
Address of list[0] = 31627168  
Value of list[0] = 10  
Address of list[1] = 31627172  
Value of list[1] = 100  
Address of list[2] = 31627176  
Value of list[2] = 200
```

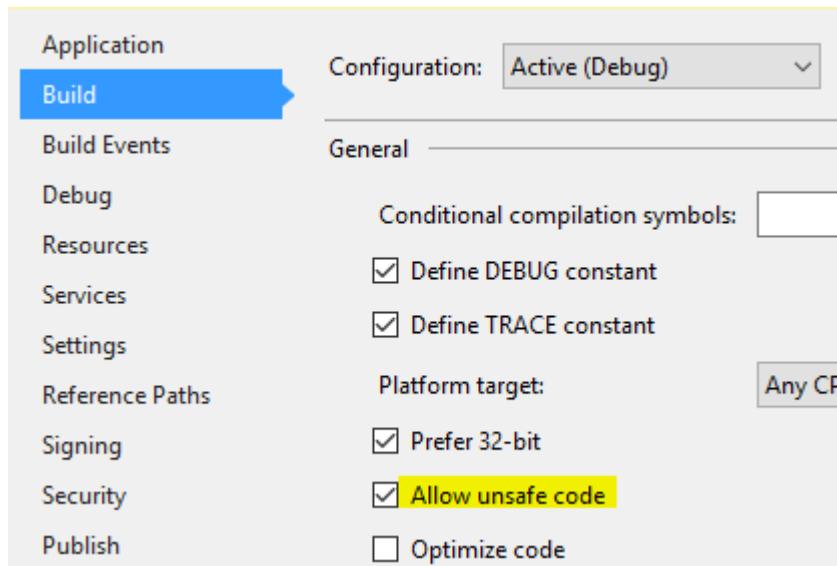
Компиляция небезопасного кода

Для компиляции небезопасного кода вы должны указать ключ командной строки `/unsafe` с компилятором командной строки.

Например, чтобы скомпилировать программу с именем `prog1.cs`, содержащую небезопасный код, из командной строки дать команду:

```
csc /unsafe prog1.cs
```

Если вы используете Visual Studio IDE, вам необходимо включить использование небезопасного кода в свойствах проекта.



Сделать это:

- Откройте свойства проекта, дважды щелкнув узел свойств в обозревателе решений.
- Перейдите на вкладку «Сборка».
- Выберите параметр «Разрешить небезопасный код»

Прочтайте Указатели и небезопасный код онлайн:

<https://riptutorial.com/ru/csharp/topic/5514/указатели-и-небезопасный-код>

глава 157: Условные заявления

Examples

If-Else Statement

Для программирования в целом часто требуется `decision` или `branch` в коде для учета того, как код работает под разными входами или условиями. В языке программирования C# (и большинстве языков программирования для этого) самым простым и иногда наиболее полезным способом создания ветки в вашей программе является оператор `If-Else`.

Предположим, что у нас есть метод (aka a function), который принимает параметр `int`, который будет составлять оценку до 100, и метод будет распечатывать сообщение о том, проходим ли мы или проваливаем.

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

При просмотре этого метода вы можете заметить эту строку кода (`score >= 50`) внутри оператора `If`. Это можно рассматривать как `boolean` условие, где, если условие оценивается равным `true`, тогда выполняется код, находящийся между `if { }`.

Например, если этот метод был вызван следующим образом: `PrintPassOrFail(60);`, результатом этого метода будет сообщение `Console Print say Pass!` поскольку значение параметра 60 больше или равно 50.

Однако, если метод был вызван как: `PrintPassOrFail(30);`, выход метода будет распечатываться, говоря `Fail!`. Это связано с тем, что значение 30 не больше или равно 50, таким образом, код между `else { }` выполняется вместо оператора `If`.

В этом примере мы сказали, что оценка должна увеличиться до 100, что не учитывается вообще. Чтобы учесть, что счет не прошел мимо 100 или, возможно, опустился ниже 0, см. Пример `If-Else If-Else`.

If-Else If-Else Statement

Следуя примеру `If-Else Statement`, пришло время ввести инструкцию `Else If`. Операция

`Else If` следует непосредственно после оператора `If` в структуре **If-Else If-Else** , но по сути имеет аналогичный синтаксис, как оператор `If` . Он используется для добавления большего количества ветвей в код, чем простой оператор **If-Else** .

В примере из **If-Else Statement** в примере указано, что оценка достигает 100; однако никогда не было никаких проверок против этого. Чтобы исправить это, измените метод из **инструкции If-Else**, чтобы выгляделеть так:

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

Все эти утверждения будут выполняться по порядку сверху до конца до тех пор, пока не будет выполнено условие. В этом новом обновлении метода мы добавили две новые ветви, которые теперь можно разместить для оценки, выходящего за *пределы* .

Например, если мы теперь вызвали метод в нашем коде как `PrintPassOrFail(110);` , результатом будет сообщение «Консольная печать». **Ошибка: оценка больше 100!**; и если мы назвали метод в нашем коде, например `PrintPassOrFail(-20);` , на выходе будет сказано **Ошибка: оценка меньше 0!** ,

Вывод операторов

Оператор `switch` позволяет проверять переменную для равенства по отношению к списку значений. Каждое значение называется случаем, а переменная, включенная, проверяется для каждого случая коммутатора.

Оператор `switch` часто более кратким и понятным, чем `if...else if... else... statement` при тестировании нескольких возможных значений для одной переменной.

Синтаксис следующий:

```
switch(expression) {
    case constant-expression:
        statement(s);
```

```

        break;
    case constant-expression:
        statement(s);
        break;

    // you can have any number of case statements
    default : // Optional
        statement(s);
        break;
}

```

есть отдельные вещи, которые необходимо учитывать при использовании оператора switch

- Выражение, используемое в операторе switch, должно иметь интегральный или нумерованный тип или быть типом класса, в котором класс имеет одну функцию преобразования для интегрального или перечисляемого типа.
- Вы можете иметь любое количество операторов case в коммутаторе. За каждым случаем следует сравнимое значение и двоеточие. Значения для сравнения должны быть уникальными в каждом операторе switch.
- Оператор switch может иметь необязательный случай по умолчанию. Случай по умолчанию может использоваться для выполнения задачи, когда ни один из случаев не является истинным.
- Каждый случай должен заканчиваться оператором `break` если он не является пустой инструкцией. В этом случае исполнение будет продолжаться в случае ниже. Оператор `break` также может быть опущен, если используется `goto case return , throw` или `goto case .`

Пример может быть задан с оценками

```

char grade = 'B';

switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent!");
        break;
    case 'B':
    case 'C':
        Console.WriteLine("Well done");
        break;
    case 'D':
        Console.WriteLine("You passed");
        break;
    case 'F':
        Console.WriteLine("Better try again");
        break;
    default:
        Console.WriteLine("Invalid grade");
        break;
}

```

Если условия оператора являются стандартными булевыми

выражениями и значениями

Следующее утверждение

```
if (conditionA && conditionB && conditionC) //...
```

точно эквивалентен

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

другими словами, условия внутри оператора «if» просто образуют обычное булево выражение.

Общей ошибкой при написании условных утверждений является явное сравнение с `true` и `false`:

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

Это можно переписать как

```
if (conditionA && !conditionB && conditionC)
```

Прочтайте Условные заявления онлайн: <https://riptutorial.com/ru/csharp/topic/3144/условные-заявления>

глава 158: Файловый и потоковый ввод-вывод

Вступление

Управляет файлами.

Синтаксис

- `new System.IO.StreamWriter(string path)`
- `new System.IO.StreamWriter(string path, bool append)`
- `System.IO.StreamWriter.WriteLine(string text)`
- `System.IO.StreamWriter.WriteAsync(string text)`
- `System.IO.Stream.Close()`
- `System.IO.File.ReadAllText(string path)`
- `System.IO.File.ReadAllLines(string path)`
- `System.IO.File.ReadLines(string path)`
- `System.IO.File.WriteAllText(string path, string text)`
- `System.IO.File.WriteAllLines(string path, IEnumerable<string> contents)`
- `System.IO.File.Copy(string source, string dest)`
- `System.IO.File.Create(string path)`
- `System.IO.File.Delete(string path)`
- `System.IO.File.Move(string source, string dest)`
- `System.IO.Directory.GetFiles(string path)`

параметры

параметр	подробности
дорожка	Местоположение файла.
присоединять	Если файл существует, <code>true</code> добавит данные в конец файла (добавляет), <code>false</code> будет перезаписывать файл.
текст	Текст, который нужно записать или сохранить.
содержание	Набор строк для записи.
источник	Расположение файла, который вы хотите использовать.
Dest	Место, куда вы хотите файл.

замечания

- Обязательно закрывайте объекты `Stream`. Это можно сделать с `using` блока, как показано выше, или путем ручного вызова `myStream.Close()` .
- Убедитесь, что текущий пользователь имеет необходимые разрешения на пути, который вы пытаетесь создать файл.
- Строки `Verbatim` должны использоваться при объявлении строки пути, которая включает обратную косую черту, например: `@"C:\MyFolder\MyFile.txt"`

Examples

Чтение из файла с использованием класса `System.IO.File`

Вы можете использовать функцию `System.IO.File.ReadAllText` для чтения всего содержимого файла в строку.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

Вы также можете прочитать файл как массив строк, используя функцию `System.IO.File.ReadAllLines` :

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

Запись строк в файл с использованием класса `System.IO.StreamWriter`

Класс `System.IO.StreamWriter` :

Реализует `TextWriter` для записи символов в поток в конкретной кодировке.

Используя метод `WriteLine` , вы можете писать контент по очереди в файл.

Обратите внимание на использование ключевого слова `using` которое гарантирует, что объект `StreamWriter` будет удален, как только он выйдет из области видимости, и, следовательно, файл будет закрыт.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

Обратите внимание, что `StreamWriter` может получить второй параметр `bool` в своем конструкторе, позволяя `Append` в файл вместо перезаписи файла:

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
```

```
appendExistingFile ))  
{  
    sw.WriteLine("This line will be appended to the existing file");  
}
```

Запись в файл с использованием класса System.IO.File

Вы можете использовать функцию [System.IO.File.WriteAllText](#) для записи строки в файл.

```
string text = "String that will be stored in the file";  
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

Вы также можете использовать функцию [System.IO.File.WriteAllLines](#), которая получает `IEnumerable<String>` как второй параметр (в отличие от одной строки в предыдущем примере). Это позволяет писать контент из массива строк.

```
string[] lines = { "My first string", "My second string", "and even a third string" };  
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

Леной чтение файла по строкам через IEnumerable

При работе с большими файлами вы можете использовать метод `System.IO.File.ReadLines` для чтения всех строк из файла в `IEnumerable<string>`. Это похоже на `System.IO.File.ReadAllLines`, за исключением того, что он не загружает весь файл в память сразу, что делает его более эффективным при работе с большими файлами.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

Второй параметр `File.ReadLines` является необязательным. Вы можете использовать его, когда требуется указать кодировку.

Важно отметить, что вызов `ToArray`, `ToList` или другой подобной функции заставит все строки загружаться сразу, что означает, что преимущество использования `ReadLines` аннулируется. Лучше всего перечислить `IEnumerable` используя цикл `foreach` или LINQ, если использовать этот метод.

Создать файл

Статический класс файла

Используя метод `CreateFile` статического класса `File` мы можем создавать файлы. Метод создает файл по заданному пути, в то же время он открывает файл и дает нам `FileStream` файла. Убедитесь, что вы закрыли файл после того, как закончите с ним.

EX1:

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

ex2:

```
using(var fileStream1 = File.Create("samplePath"))
{
    /// you can write to the fileStream1
}
```

EX3:

```
File.Create("samplePath").Close();
```

Класс FileStream

Есть много перегрузок этого конструктора классов, который на самом деле хорошо документирован [здесь](#). Ниже приведен пример, который охватывает большинство используемых функций этого класса.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

Вы можете проверить перечисления для [FileMode](#) , [FileAccess](#) и [FileShare](#) из этих ссылок. В основном они означают следующее:

FileMode: ответы «Должен ли файл быть создан? Открыт? *Create*, если не существует, тогда откроите?» любопытные вопросы.

FileAccess: Ответы «Должен ли я читать файл, писать в файл или и то, и другое?» любопытные вопросы.

FileShare: Ответы «Должны ли другие пользователи читать, писать и т. д. в файл, пока я использую его одновременно?» любопытные вопросы.

Копировать файл

Статический класс файла

Статический класс `File` может быть легко использован для этой цели.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Примечание. По этому методу файл копируется, что означает, что он будет считан из источника, а затем записан в путь назначения. Это процесс потребления ресурсов, для определения размера файла потребуется относительное время и может привести к

зависанию вашей программы, если вы не используете потоки.

Переместить файл

Статический класс файла

Статический класс файлов можно легко использовать для этой цели.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Замечание1: Изменяет только индекс файла (если файл перемещается в том же томе). Эта операция не занимает достаточного времени для размера файла.

Замечание2: Невозможно переопределить существующий файл по пути назначения.

Удалить файл

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

Хотя `Delete` не генерирует исключение, если файл не существует, он будет генерировать исключение, например, если указанный путь недействителен или у вызывающего нет необходимых разрешений. Вы всегда должны оберывать вызовы `Delete` внутри блока `try-catch` и обрабатывать все ожидаемые исключения. В случае возможных условий гонки, оберните логику внутри оператора блокировки .

Файлы и каталоги

Получить все файлы в каталоге

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Возвращает массив `FileInfo`, представляющий все файлы в указанном каталоге.

Получить файлы с определенным расширением

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Возвращает массив `FileInfo`, представляющий все файлы в указанном каталоге с указанным расширением.

Async записывает текст в файл с помощью `StreamWriter`

```
// filename is a string with the full path
// true is to append
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))
```

```
{  
    // Can write either a string or char array  
    await file.WriteAllTextAsync(text);  
}
```

Прочтайте **Файловый и потоковый ввод-вывод онлайн**:

<https://riptutorial.com/ru/csharp/topic/4266/файловый-и-потоковый-ввод-вывод>

глава 159: Фильтры действий

Examples

Пользовательские фильтры действий

Мы пишем фильтры пользовательских действий по различным причинам. У нас может быть настраиваемый фильтр действий для ведения журнала или для сохранения данных в базе данных до выполнения любого действия. У нас также может быть один для получения данных из базы данных и установки его в качестве глобальных значений приложения.

Чтобы создать настраиваемый фильтр действий, нам необходимо выполнить следующие задачи:

1. Создать класс
2. Наследовать его из класса `ActionFilterAttribute`

Переопределите хотя бы один из следующих способов:

OnActionExecuting - этот метод вызывается перед выполнением действия контроллера.

OnActionExecuted - этот метод вызывается после выполнения действия контроллера.

OnResultExecuting - этот метод вызывается до выполнения результата действия контроллера.

OnResultExecuted - этот метод вызывается после выполнения результата действия контроллера.

Фильтр может быть создан, как показано в листинге ниже:

```
using System;
using System.Diagnostics;
using System.Web.Mvc;

namespace WebApplication1
{
    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //You may fetch data from database here
            filterContext.Controller.ViewBag.GreetMesssage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }
    }
}
```

```
    }

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        var controllerName = filterContext.RouteData.Values["controller"];
        var actionName = filterContext.RouteData.Values["action"];
        var message = String.Format("{0} controller:{1} action:{2}",
"onactionexecuting", controllerName, actionName);
        Debug.WriteLine(message, "Action Filter Log");
        base.OnActionExecuting(filterContext);
    }
}
```

Прочтайте **Фильтры действий** онлайн: <https://riptutorial.com/ru/csharp/topic/1505/фильтры-действий>

глава 160: Функциональное программирование

Examples

Func и Action

Func предоставляет держатель для параметризованных анонимных функций. Ведущими типами являются входы, а последний тип всегда является возвращаемым значением.

```
// square a number.
Func<double, double> square = (x) => { return x * x; };

// get the square root.
// note how the signature matches the built in method.
Func<double, double> squareroot = Math.Sqrt;

// provide your workings.
Func<double, double, string> workings = (x, y) =>
    string.Format("The square of {0} is {1}.", x, square(y))
```

Объекты **Action** подобны void-методам, поэтому они имеют только тип ввода. В стек оценки нет результата.

```
// right-angled triangle.
class Triangle
{
    public double a;
    public double b;
    public double h;
}

// Pythagorean theorem.
Action<Triangle> pythagoras = (x) =>
    x.h = squareroot(square(x.a) + square(x.b));

Triangle t = new Triangle { a = 3, b = 4 };
pythagoras(t);
Console.WriteLine(t.h); // 5.
```

неизменность

Непрерывность распространена в функциональном программировании и редка в объектно-ориентированном программировании.

Создайте, например, тип адреса с изменчивым состоянием:

```
public class Address ()
```

```
{  
    public string Line1 { get; set; }  
    public string Line2 { get; set; }  
    public string City { get; set; }  
}
```

Любой фрагмент кода может изменить любое свойство в вышеуказанном объекте.

Теперь создайте неизменяемый тип адреса:

```
public class Address ()  
{  
    public readonly string Line1;  
    public readonly string Line2;  
    public readonly string City;  
  
    public Address(string line1, string line2, string city)  
    {  
        Line1 = line1;  
        Line2 = line2;  
        City = city;  
    }  
}
```

Имейте в виду, что наличие коллекций только для чтения не учитывает неизменность.

Например,

```
public class Classroom  
{  
    public readonly List<Student> Students;  
  
    public Classroom(List<Student> students)  
    {  
        Students = students;  
    }  
}
```

не является неизменным, так как пользователь объекта может изменять коллекцию (добавлять или удалять из нее элементы). Чтобы сделать его неизменным, нужно либо использовать интерфейс, такой как `IEnumerable`, который не предоставляет методы для добавления, или сделать его `ReadOnlyCollection`.

```
public class Classroom  
{  
    public readonly ReadOnlyCollection<Student> Students;  
  
    public Classroom(ReadOnlyCollection<Student> students)  
    {  
        Students = students;  
    }  
}  
  
List<Students> list = new List<Student>();  
// add students  
Classroom c = new Classroom(list.AsReadOnly());
```

С неизменяемым объектом мы имеем следующие преимущества:

- Он будет находиться в известном состоянии (другой код не может его изменить).
- Это поточно-безопасный.
- Конструктор предлагает одно место для проверки.
- Зная, что объект не может быть изменен, код легче понять.

Избегайте ссылок на Null

Разработчики C# получают множество нулевых ссылочных исключений. Разработчики F# не потому, что у них есть тип Option. Тип Option <> (некоторые предпочитают, возможно, <> как имя), предоставляет тип возврата Some и a None. Он делает явным, что метод может возвращать нулевую запись.

Например, вы не можете прочитать следующее и знать, если вам придется иметь дело с нулевым значением.

```
var user = _repository.GetUser(id);
```

Если вы знаете о возможном нуле, вы можете ввести код шаблона, чтобы справиться с ним.

```
var username = user != null ? user.Name : string.Empty;
```

Что делать, если вместо этого есть опция <>?

```
Option<User> maybeUser = _repository.GetUser(id);
```

Теперь код делает явным, что у нас может быть запись «Нет», и требуется шаблонный код для проверки Some или None:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

Следующий метод показывает, как вернуть параметр <>

```
public Option<User> GetUser(int id)
{
    var users = new List<User>
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Ниже приведена минимальная реализация Option <T>.

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }

    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

Чтобы продемонстрировать вышеописанное, [избегайте Null.csx](#) с C # REPL.

Как было сказано, это минимальная реализация. Поиск [пакетов «Возможно» NuGet](#) приведет к появлению ряда хороших библиотек.

Функции более высокого порядка

Функция более высокого порядка - это функция, которая принимает другую функцию в качестве аргумента или возвращает функцию (или и то, и другое).

Это обычно делается с lambdas, например, при передаче предиката в предложение LINQ Where:

```
var results = data.Where(p => p.Items == 0);
```

Предложение Where () может принимать множество разных предикатов, что дает ему большую гибкость.

Передача метода в другой метод также наблюдается при реализации шаблона проектирования стратегии. Например, различные методы сортировки могут быть выбраны

и переданы методу Сортировки на объект в зависимости от требований во время выполнения.

Неизменяемые коллекции

Пакет `System.Collections.Immutable` NuGet предоставляет неизменные классы коллекции.

Создание и добавление элементов

```
var stack = ImmutableList.Create<int>();  
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1  
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1
```

Создание с использованием строителя

У некоторых неизменяемых коллекций есть внутренний класс `Builder` который можно использовать для дешевого построения больших неизменяемых экземпляров:

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder  
builder.Add(1);  
builder.Add(2);  
var list = builder.ToImmutable();
```

Создание из существующего `IEnumerable`

```
var numbers = Enumerable.Range(1, 5);  
var list = ImmutableList.CreateRange<int>(numbers);
```

Список всех неизменяемых типов коллекций:

- `System.Collections.Immutable.ImmutableArray<T>`
- `System.Collections.Immutable.ImmutableDictionary<TKey, TValue>`
- `System.Collections.Immutable.ImmutableHashSet<T>`
- `System.Collections.Immutable.ImmutableList<T>`
- `System.Collections.Immutable.ImmutableQueue<T>`
- `System.Collections.Immutable.ImmutableSortedDictionary<TKey, TValue>`
- `System.Collections.Immutable.ImmutableSortedSet<T>`
- `System.Collections.Immutable.ImmutableStack<T>`

Прочтайте Функциональное программирование онлайн:

<https://riptutorial.com/ru/csharp/topic/2564/функциональное-программирование>

глава 161: Функция с несколькими возвращаемыми значениями

замечания

В С# нет неотъемлемого ответа на этот вопрос - так называемый - нужен. Тем не менее есть обходные пути для удовлетворения этой потребности.

Причина, по которой я квалифицирую необходимость как «так называемый», заключается в том, что нам нужны только методы с 2 или более чем двумя значениями, которые возвращаются, когда мы нарушаем принципы хорошего программирования. Особенно [принцип единой ответственности](#).

Следовательно, было бы лучше получить предупреждение, когда нам нужны функции, возвращающие 2 или более значения, и улучшаем наш дизайн.

Examples

«анонимный объект» + «динамическое ключевое слово»

Вы можете вернуть анонимный объект из своей функции

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

И присвойте результат динамическому объекту и прочитайте значения в нем.

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

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

Решение для кортежа

Вы можете вернуть экземпляр класса `Tuple` из вашей функции с двумя параметрами шаблона в виде `Tuple<string, MyClass>`:

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
```

```
}
```

И прочтайте значения, как показано ниже:

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

Параметры обратной связи и выхода

Ключевое слово `ref` используется для передачи [аргумента в качестве ссылки](#). `out` будет делать то же самое, что и `ref` но для вызова функции не требуется назначенное значение вызывающим абонентом.

Ref Parameter : -Если вы хотите передать переменную как параметр `ref`, вам необходимо инициализировать ее, прежде чем передавать ее как параметр `ref` в метод.

Out Parameter: - Если вы хотите передать переменную как параметр `out`, вам не нужно ее инициализировать, прежде чем передавать ее как параметр `out` в метод.

```
static void Main(string[] args)
{
    int a = 2;
    int b = 3;
    int add = 0;
    int mult= 0;
    AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
    Console.WriteLine(add); //5
    Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int
b, out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

Прочтайте [Функция с несколькими возвращаемыми значениями онлайн](#):

<https://riptutorial.com/ru/csharp/topic/3908/функция-с-несколькими-возвращаемыми-значениями>

глава 162: Хэш-функции

замечания

MD5 и SHA1 небезопасны и их следует избегать. Примеры существуют для образовательных целей и из-за того, что устаревшее программное обеспечение все еще может использовать эти алгоритмы.

Examples

MD5

Хэш-функции отображают двоичные строки произвольной длины в маленькие двоичные строки фиксированной длины.

Алгоритм [MD5](#) является широко используемой хэш-функцией, производящей 128-битное хэш-значение (16 байтов, 32 шестнадцатеричных символа).

Метод [ComputeHash](#) класса [System.Security.Cryptography.MD5](#) возвращает хэш как массив из 16 байтов.

Пример:

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

```
}
```

Результат: хеш MD5 Hello World! это: ED076287532E86365E841E92BFC50D8C

Проблемы с безопасностью:

Как и большинство хеш-функций, MD5 не является ни шифрованием, ни кодированием. Он может быть отменен атакой грубой силы и страдает от обширных уязвимостей против атак на столкновение и прообраз.

SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA1 sha1Hash = SHA1.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Выход:

SHA1 хеш Hello Word! это: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

SHA256

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
```

```

        using (SHA256 sha256Hash = SHA256.Create())
        {
            //From String to byte array
            byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
            byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
            string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

            Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
        }
    }
}
}

```

Выход:

Хэш SHA256 Hello World! есть: 7

F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

SHA384

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}

```

Выход:

Хэш SHA384 Hello World! это:

BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD

SHA512

```

using System;
using System.Security.Cryptography;
using System.Text;

```

```

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}

```

Результат: SHA512 хеш Hello World! is:

861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891C

PBKDF2 для шифрования паролей

PBKDF2 («Функция деривации ключа на основе пароля 2») является одной из рекомендуемых хеш-функций для хэширования паролей. Это часть [rfc-2898](#) .

.NET `Rfc2898DeriveBytes` основан на `HMACSHA1`.

```

using System.Security.Cryptography;

...

public const int SALT_SIZE = 24; // size in bytes
public const int HASH_SIZE = 24; // size in bytes
public const int ITERATIONS = 100000; // number of pbkdf2 iterations

public static byte[] CreateHash(string input)
{
    // Generate a salt
    RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
    byte[] salt = new byte[SALT_SIZE];
    provider.GetBytes(salt);

    // Generate the hash
    Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
    return pbkdf2.GetBytes(HASH_SIZE);
}

```

PBKDF2 требует [соли](#) и количества итераций.

Повторы:

Большое количество итераций замедлит алгоритм, что затрудняет процесс взлома

паролей. Поэтому рекомендуется большое количество итераций. PBKDF2 - порядок величин медленнее, чем MD5, например.

Поваренная соль:

Соль предотвратит поиск значений хэша в [радужных таблицах](#). Он должен храниться рядом с хэшем пароля. Рекомендуется одна соль на пароль (не одна глобальная соль).

Завершите решение для Хейшиングа с помощью Pbkdf2

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
    /// Compatibility: .NET 3.0 and later.
    /// </summary>
    /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
    /// hashing.</remarks>
    public static class PasswordHashProvider
    {
        /// <summary>
        /// The salt byte size, 64 length ensures safety but could be increased / decreased
        /// </summary>
        private const int SaltByteSize = 64;
        /// <summary>
        /// The hash byte size,
        /// </summary>
        private const int HashByteSize = 64;
        /// <summary>
        /// High iteration count is less likely to be cracked
        /// </summary>
        private const int Pbkdf2Iterations = 10000;

        /// <summary>
        /// Creates a salted PBKDF2 hash of the password.
        /// </summary>
        /// <remarks>
        /// The salt and the hash have to be persisted side by side for the password. They could
        /// be persisted as bytes or as a string using the convenience methods in the next class to
        /// convert from byte[] to string and later back again when executing password validation.
        /// </remarks>
        /// <param name="password">The password to hash.</param>
        /// <returns>The hash of the password.</returns>
        public static PasswordHashContainer CreateHash(string password)
        {
            // Generate a random salt
            using (var csprng = new RNGCryptoServiceProvider())
            {
                // create a unique salt for every password hash to prevent rainbow and dictionary
                // based attacks
                var salt = new byte[SaltByteSize];
                csprng.GetBytes(salt);

                // Hash the password and encode the parameters
            }
        }
    }
}
```

```

        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
    }
}

/// <summary>
/// Recreates a password hash based on the incoming password string and the stored salt
/// </summary>
/// <param name="password">The password to check.</param>
/// <param name="salt">The salt existing.</param>
/// <returns>the generated hash based on the password and salt</returns>
public static byte[] CreateHash(string password, byte[] salt)
{
    // Extract the parameters from the hash
    return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
}

/// <summary>
/// Validates a password given a hash of the correct one.
/// </summary>
/// <param name="password">The password to check.</param>
/// <param name="salt">The existing stored salt.</param>
/// <param name="correctHash">The hash of the existing password.</param>
/// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
{
    // Extract the parameters from the hash
    byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    return CompareHashes(correctHash, testHash);
}

/// <summary>
/// Compares two byte arrays (hashes)
/// </summary>
/// <param name="array1">The array1.</param>
/// <param name="array2">The array2.</param>
/// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
public static bool CompareHashes(byte[] array1, byte[] array2)
{
    if (array1.Length != array2.Length) return false;
    return !array1.Where((t, i) => t != array2[i]).Any();
}

/// <summary>
/// Computes the PBKDF2-SHA1 hash of a password.
/// </summary>
/// <param name="password">The password to hash.</param>
/// <param name="salt">The salt.</param>
/// <param name="iterations">The PBKDF2 iteration count.</param>
/// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
/// <returns>A hash of the password.</returns>
private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int
outputBytes)
{
    using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
    {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
    }
}
}

```

```

/// <summary>
/// Container for password hash and salt and iterations.
/// </summary>
public sealed class PasswordHashContainer
{
    /// <summary>
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }

    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
        this.HashedPassword = hashedPassword;
        this.Salt = salt;
    }
}

/// <summary>
/// Convenience methods for converting between hex strings and byte array.
/// </summary>
public static class ByteConverter
{
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
        var bytes = new byte[hexedString.Length / 2];
        for (var i = 0; i < bytes.Length; i++)
        {
            var strPos = i * 2;
            var chars = hexedString.Substring(strPos, 2);
            bytes[i] = Convert.ToByte(chars, 16);
        }
        return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
        return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
*/

```

```
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions are met:
*
* 1. Redistributions of source code must retain the above copyright notice,
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/

```

Пожалуйста, ознакомьтесь с этим превосходным ресурсом [Crackstation - Соленый пароль Хеширование - Делайте это правильно](#) для получения дополнительной информации. Часть этого решения (функция хеширования) была основана на коде с этого сайта.

Прочтайте Хэш-функции онлайн: <https://riptutorial.com/ru/csharp/topic/2774/хэш-функции>

глава 163: Частичный класс и методы

Вступление

Частичные классы предоставляют нам возможность разбивать классы на несколько частей и в несколько исходных файлов. Во время компиляции все части объединяются в один класс. Все части должны содержать ключевое слово `partial`, должны иметь одинаковую доступность. Все части должны присутствовать в одной и той же сборке, чтобы она включалась во время компиляции.

Синтаксис

- открытый частичный класс `MyPartialClass {}`

замечания

- Частичные классы должны быть определены внутри одной и той же сборки и пространства имен в качестве класса, который они распространяют.
- Все части класса должны использовать `partial` ключевое слово.
- Все части класса должны иметь такую же доступность; `public / protected / private` т. д.
- Если какая-либо часть использует ключевое слово `abstract`, тогда комбинированный тип считается абстрактным.
- Если какая-либо часть использует `sealed` ключевое слово, то комбинированный тип считается запечатанным.
- Если какая-либо часть использует базовый тип, тогда комбинированный тип наследуется от этого типа.
- Комбинированный тип наследует все интерфейсы, определенные для всех частичных классов.

Examples

Частичные классы

Частичные классы предоставляют возможность разбивать объявление класса (обычно в отдельные файлы). Общая проблема, которая может быть решена с помощью частичных классов, позволяет пользователям изменять автоматически сгенерированный код, не

опасаясь, что их изменения будут перезаписаны, если код будет регенерирован. Также несколько разработчиков могут работать с одним классом или методами.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("Method call from the first declaration.");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("Method call from the second declaration.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // outputs "Method call from the first declaration."
            partial.AnotherExampleMethod(); // outputs "Method call from the second
declaration."
        }
    }
}
```

Частичные методы

Частичный метод состоит из определения в одном объявлении частичного класса (как общий сценарий - в автогенерированном) и реализации в другом объявлении частичного класса.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // Auto-generated
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // Human-written
    {
        public void PartialMethod()
        {
            Console.WriteLine("Partial method called.");
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        PartialClass partial = new PartialClass();
        partial.PartialMethod(); // outputs "Partial method called."
    }
}
```

Частичные классы, наследуемые от базового класса

При наследовании от любого базового класса только один частичный класс должен иметь указанный базовый класс.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

Вы можете указать один и тот же базовый класс в нескольких неполных классах. Он будет помечен как избыточный с помощью некоторых инструментов IDE, но он правильно компилируется.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass : BaseClass {} // base class here is redundant
```

Вы не можете указать разные базовые классы в нескольких частичных классах, это приведет к ошибке компилятора.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // compiler error

// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // compiler error
```

Прочтайте Частичный класс и методы онлайн: <https://riptutorial.com/ru/csharp/topic/3674/частичный-класс-и-методы>

глава 164: Читать и понимать Stacktraces

Вступление

Трассировка стека - отличная помощь при отладке программы. Вы получите трассировку стека, когда ваша программа выбрасывает исключение, а иногда, когда программа прерывается ненормально.

Examples

Трассировка стека для простого исключения NullReferenceException в Windows Forms

Давайте создадим небольшой фрагмент кода, который выдает исключение:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

Если мы выполним это, мы получим следующую трассировку Исключения и стека:

```
System.NullReferenceException: "Object reference not set to an instance of an object."
  at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
  at System.Windows.Forms.Control.OnClick(EventArgs e)
  at System.Windows.Forms.Button.OnClick(EventArgs e)
  at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

Трассировка стека продолжается так, но эта часть будет достаточной для наших целей.

В верхней части трассировки стека мы видим строку:

в WindowsFormsApplication1.Form1.button1_Click (отправитель объекта, EventArgs e) в F: \ WindowsFormsApplication1 \ WindowsFormsApplication1 \ Form1.cs: строка 29

Это самая важная часть. Он сообщает нам *точную* строку, в которой произошло исключение: строка 29 в Form1.cs.

Итак, вот где вы начинаете свой поиск.

Вторая строка

в System.Windows.Forms.Control.OnClick (EventArgs e)

Это метод, который называется `button1_Click`. Итак, теперь мы знаем, что `button1_Click`, где произошла ошибка, `button1_Click` из `System.Windows.Forms.Control.OnClick`.

Мы можем продолжать это; третья строка

в `System.Windows.Forms.Button.OnClick (EventArgs e)`

Это, в свою очередь, код, который называется `System.Windows.Forms.Control.OnClick`.

Трассировка стека представляет собой список функций, которые вызывались до тех пор, пока ваш код не столкнулся с `Exception`. И, следуя этому, вы можете выяснить, к какому пути выполнения ваш код следовал, пока он не столкнулся с трудностями!

Обратите внимание, что трассировка стека включает вызовы из системы .Net; вам обычно не нужно следовать всем кодам `Microsoft System.Windows.Forms` чтобы узнать, что пошло не так, только код, принадлежащий вашему собственному приложению.

Итак, почему это называется «трассировкой стека»?

Поскольку каждый раз, когда программа вызывает метод, он отслеживает, где он был. Он имеет структуру данных, называемую «стек», где он выгружает свое последнее местоположение.

Если выполнение этого метода выполняется, он смотрит на стек, чтобы увидеть, где он был, прежде чем он назвал метод, и продолжит оттуда.

Таким образом, стек позволяет компьютеру узнать, где он остановился, прежде чем вызывать новый метод.

Но он также служит для отладки. Как детектив, отслеживающий шаги, предпринятые преступником при совершении преступления, программист может использовать стек для отслеживания шагов, предпринятых программой, прежде чем он разбился.

Прочтите Читать и понимать Stacktraces онлайн: <https://riptutorial.com/ru/csharp/topic/8923/читать-и-понимать-stacktraces>

глава 165: Чтение и запись .zip-файлов

Синтаксис

1. открытый статический ZipArchive OpenRead (строка archiveFileName)

параметры

параметр	подробности
archiveFileName	Путь к открытому архиву, заданный как относительный или абсолютный путь. Относительный путь интерпретируется как относительно текущего рабочего каталога.

Examples

Запись в zip-файл

Чтобы написать новый .zip-файл:

```
System.IO.Compression
System.IO.Compression.FileSystem

using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
    {
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
        {
            writer.WriteLine("Information about this package.");
            writer.WriteLine("=====");
        }
    }
}
```

Запись ZIP-файлов в память

В следующем примере будут возвращены `byte[]` данные заархивированного файла, содержащего предоставленные ему файлы, без необходимости доступа к файловой системе.

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)
{
    using (MemoryStream ms = new MemoryStream())
    {
```

```

        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))
        {
            foreach (var file in files)
            {
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file
                with this name
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))
                {
                    writer.Write(file.Value); //write the binary data
                }
            }
            //ZipArchive must be disposed before the MemoryStream has data
            return ms.ToArray();
        }
    }
}

```

Получить файлы из Zip-файла

В этом примере приводится список файлов из предоставленных двоичных данных zip-архива:

```

public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}

```

В следующем примере показано, как открыть zip-архив и извлечь все .txt-файлы в папку

```

using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

```

```
using (ZipArchive archive = ZipFile.OpenRead(zipPath))
{
    foreach (ZipArchiveEntry entry in archive.Entries)
    {
        if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
        {
            entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
        }
    }
}
```

Прочтайте Чтение и запись .zip-файлов онлайн: <https://riptutorial.com/ru/csharp/topic/6709/чтение-и-запись--zip-файлов>

кредиты

S. No	Главы	Contributors
1	Начало работы с C # Language	4444, A. Raza, A_Arnold, aalaap, Aaron Hudon, abishekshivan, Ade Stringer, Aleksandur Murfitt, Almir Vuk, Alok Singh, Andrii Abramov, AndroidMechanic, Aravind Suresh, Artemix, Ben Aaronson, Bernard Vander Beken, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, bpoiss, Br0k3nL1m1ts, Callum Watkins, Carlos Muñoz, Chad Levy, Chris Nantau, Christopher Ronning, Community, Confiqure, crunchy, David G., David Pine, DavidG, DAXaholic, Delphi.Boy, Durgpal Singh, DWright, Ehsan Sajjad, Elie Saad, Emre Bolat, enrico.bacis, fabriciorissetto, FadedAce, Florian Greinacher, Florian Koch, Frankenstein Joe, Gennady Trubach, GingerHead, Gordon Bell, gracacs, G-Wiz, H. Pauwelyn, Happypig375, Henrik H, HodofHod, Hywel Rees, iliketocode, Iordanis, Jamie Rees, Jawa, jnovo, John Slegers, Kayathiri, ken2k, Kevin Montrose, Kritner, Krzyserious, leumas1960, M Monis

		Ahmed Khan, Mahmoud Elgindy, Malick, Marcus Höglund, Mateen Ulhaq, Matt, Matt, Matt, Matt, Michael B, Michael Brandon Morris, Miljen Mikic, Millan Sanchez, Nate Barbettini, Nick, Nick Cox, Nipun Tripathi, NotMyself, Ojen, PashaPash, pijemcolu, Prateek, Raj Rao, Rajput, Rakitić, Rion Williams, RokumDev, RomCoo, Ryan Hilbert, sebingel, SeeuD1, solidcell, Steven Ackley, sumit sharma, Tofix, Tom Bowers, Travis J, Tushar patel, User 00000, user3185569, Ven, Victor Tomaili, viggity, void, Wen Qin, Ziad Akiki, Zze
2	BackgroundWorker	Bovaz, Draken, ephtee, Jacobr365, Will
3	BigInteger	4444, Ed Marty, James Hughes, Rob, The_Outsider
4	BindingList	Bovaz, Stephen Leppik, yumaikas
5	CLSCCompliantAttribute	mybirthname, Rob
6	Enum	Aaron Hudon, Abdul Rehman Sayed, Adrian Iftode, aholmes, alex, Blachshma, Chris Oldwood, Diligent Key Presser, dlatikay, Dmitry Bychenko, dove, Ghost4Man, H. Pauwelyn, ja72, Jon Schneider, Kit, konkked, Kyle Trauberman, Martin

		Zikmund, Matthew Whited, Maxime, mbrdev , Michael Mairegger, MuiBienCarlota, NikolayKondratyev, Osama AbuSitta, PSGuy , recursive, Richa Garg, Richard, Rob, sdgfsdh, Sergii Lischuk, Squirrel, Stefano d'Antonio, Tanner Swett, TarkaDaal , Theodoros Chatzigiannakis, vesi, Wasabi Fan, Yanai
7	FileSystemWatcher	Sondre
8	Generic Lambda Query Builder	4444, PedroSouki
9	Guid	Bearington, Botond Balázs, elibyy, Jonas S, Osama AbuSitta, Sherantha, TarkaDaal, The_Outsider, Tim Ebenezer, void
10	ICloneable	ja72, Rob
11	IComparable	alex
12	IEnumerable	4444, Avia, Benjamin Hodgson, Luke Ryan, Olivier De Meulder, The_Outsider
13	ILGenerator	Aleks Andreev, thehenny
14	Interoperability	Balen Danny, Benjamin Hodgson, Bovaz, Craig Brett, Dean Van Greunen, Gajendra, Jan Bońkowski, Kimmax, Marc Wittmann, Martin, Pavel Durov, René Vogt, RomCoo, Squidward
15	Linq to Objects	brijber, Christian Gollhardt, FortyTwo,

		Kevin Green, Raphael Pantaleão, Simon Halsey, Tanveer Badar
16	LINQ to XML	Denis Elkhov, Stephen Leppik, Uali
17	Microsoft.Exchange.WebServices	Bassie
18	NullReferenceException	4444, Agramer, Ashutosh, krimog, Kyle Traberman, Mathias Müller, Philip C, RamenChef, S.L. Barth, Shelby115, Squidward, vicky, Zikato
19	О (n) Алгоритм кругового вращения массива	AFT
20	ObservableCollection	demonplus, GeralexGR, Jonathan Anctil, MuiBienCarlota
21	String.Format	Aaron Hudon, Akshay Anand, Alexander Mandt, Andrius, Aseem Gautam, Benjol, BrunoLM, Dmitry Egorov, Don Vince, Dweeberly, ebattulga, ejhn5, gdoron, H. Pauwelyn, Hossein Narimani Rad, Jasmin Solanki, Marek Musielak, Mark Shevchenko, Matas Vaitkevicius, Mendhak, MGB, nikchi, Philip C, Rahul Nikate, Raidri, RamenChef, Richard, Richard, Rion Williams, ryanyuyu, teo van kot, Vincent, void, Wyck
22	StringBuilder	ATechieThought, brijber, Jeremy Kato, Jon Schneider, Robert Columbia, The_Outsider

23	System.DirectoryServices.Protocols.LdapConnection	Andrew Stollak
24	System.Management.Automation	Mikko Viitala
25	Verbatim Strings	Alan McBee, Amitay Stern, Andrew Diamond, Aphelion, Arjan Einbu, avb, Bryan Crosby, Charlie H, David G., devuxer, DLeh, Ehsan Sajjad, Freelex, goric, Jared Hooper, Jeremy Kato, Jonas S, Kevin Montrose, Kilazur, Mateen Ulhaq, Ricardo Amores, Rion Williams, Sam Johnson, Sophie Jackson-Lee, Squirrel, th1rdey3
26	Windows Communication Foundation	NtFreX
27	XDocument и пространство имен System.Xml.Linq	Crowcoder, Jon Schneider
28	XmlDocument и пространство имен System.Xml	Alexander Petrov, Rokey Ge, Rubens Farias, Timon Post, Willy David Jr
29	Анализ регулярных выражений	C4u
30	Аннотации данных	Maxime, Mikko Viitala, The_Outsider, Will Ray
31	Анонимные типы	Fernando Matsumoto, goric, Stephen Leppik
32	Асинхронные / ожидающие, фоновые работы, задачи и примеры потоков	Dieter Meemken, Kyrylo M, nik, Pavel Mayorov, sebingel, Underscore, Xander Luciano, Yehor Hromadskyi
33	Асинхронный разъем	Timon Post
34	Асинхронный-Await	Aaron Hudon, AGB, aholmes, Ant P, Benjol, BrunoLM, Conrad.Dean,

		Craig Brett, Donald Webb, EJoshuaS, EvilTak, gdyrrahitis, George Duckett, Grimm The Opiner, Guanxi, guntbert, H. Pauwelyn, jdpilgrim, ken2k, Kevin Montrose, marshal craft, Michael Richardson, Moerwald, Nate Barbettini, nickguletskii, Pavel Mayorov, Pavel Voronin, pinkfloydx33, Rob, Serg Rogovtsev, Stefano d'Antonio, Stephen Leppik, SynerCoder, trashr0x, Tseng, user2321864, Vincent
35	Атрибуты	Alexander Mandt, Andrew Diamond, Doruk, LosManos, Lukas Kolletzki, NikolayKondratyev, Pavel Sapehin, SysVoid, TKharaishvili
36	Включая ресурсы шрифтов	Bales, Facebamm
37	Внедрение зависимости	Buh Buh, iaminvincible, Kyle Trauberman, Wiktor Dębski
38	Внедрение шаблона дизайна мухи	Jan Bońkowski
39	Внедрение шаблона проектирования декоратора	Jan Bońkowski
40	Возможности C # 3.0	0xFF, bob0the0mighty, FrenkyB, H. Pauwelyn, ken2k, Maniero, Rob
41	Возможности C # 4.0	Benjamin Hodgson, Botond Balázs, H. Pauwelyn, Proxima, Sibeesh Venu, Squidward, Theodoros Chatzigiannakis

42	Возможности C # 5.0	Abob, alex.b, H. Pauwelyn
43	Возможности C # 6.0	A_Arnold, Aaron Anodide, Aaron Hudon, Adil Mammadov, Adriano Repetti, AER, AGB, Akshay Anand, Alan McBee, Alex Logan, Amitay Stern, anaximander, andre_ss6 , Andrea, AndroidMechanic, Ares, Arthur Rizzo, Ashwin Ramaswami, avishayp, Balagurunathan Marimuthu, Bardia, Ben Aaronson, Blubberguy22 , Bobson, bpoiss, Bradley Uffner, Bret Copeland, C4u, Callum Watkins, Chad Levy, Charlie H, ChrFin, Community, Conrad.Dean, Cyprien Autexier, Dan, Daniel Minnaar, Daniel Stradowski, DarkV1, dasblinkenlight, David, David G., David Pine, Deepak gupta, DLeh, dotctor, Durgpal Singh, Ehsan Sajjad, el2iot2, Emre Bolat, enrico.bacis, Erik Schierboom, fabriciorissetto, faso, Franck Dernoncourt, FrankerZ, Gabor Kecskemeti, Gary, Gates Wong, Geoff, GingerHead, Gordon Bell, Guillaume Pascal, H. Pauwelyn, hankide, Henrik H, iliketocode, Iordanis , Irfan, Ivan Yurchenko, J. Steen, Jacob Linney, Jamie

Rees, Jason Sturges, Jeppe Stig Nielsen, Jim, JNYRanger, Joe, Joel Etherton, John Slegers, Johnbot, Jojodmo, Jonas S, Juan, Kapep, ken2k, Kit, Konamiman, Krikor Ailanjian, Lafexlos, LaoR , Lasse Vågsæther Karlsen, M.kazem Akhgary, Mafii, Magisch, Makyen, MANISH KUMAR CHAUDHARY, Marc, MarcinJuraszek, Mark Shevchenko, Matas Vaitkevicius, Mateen Ulhaq, Matt, Matt, Matt, Matt Thomas, Maximillian Laumeister, mbrdev, Mellow, Michael Mairegger, Michael Richardson, Michał Perłakowski, mike z, Minhas Kamal, Mitch Talmadge, Mohammad Mirmostafa, Mr.Mindor, mshsayem, MuiBienCarlota, Nate Barbettini, Nicholas Sizer , nik, nollidge, Nuri Tasdemir, Oliver Mellet, Orlando William, Osama AbuSitta, Panda, Parth Patel, Patrick, Pavel Voronin, PSN, qJake, QoP, Racil Hilan, Radouane ROUFID, Rahul Nikate, Raidri, Rajeev, Rakitić, ravindra, rdans, Reeven, Richa Garg, Richard, Rion Williams, Rob, Robban, Robert Columbia, Ryan Hilbert, ryanyuyu, Sam, Sam Axe, Samuel, Sender, Shekhar, Shoe, Slayther, solidcell,

Squidward, Squirrel,
stackptr, stark, Stilgar,
Sunny R Gupta, Suren
Sravyan, Sworgkh,
syb0rg, takrl, Tamir
Vered, Theodoros
Chatzigiannakis, Timothy
Shields, Tom Droste,
Travis J, Trent,
Trikaldarshi, Troyen,
Tushar Patel, tzachs, Uri
Agassi, Uriil, uTeisT,
vcsjones, Ven, viggity,
Vishal Madhvani, Vlad,
Wai Ha Lee, Xiaoy312,
Yury Kerbitskov, Zano,
Ze Rubeus, Zimm1

Adil Mammadov, afuna,
Amitay Stern, Amr
Badawy, Andreas Pähler
, Andrew Diamond, Avi
Turner, Benjamin
Hodgson, Blorgbeard,
bluray, Botond Balázs,
Bovaz, Cerbrus,
Clueless, Conrad.Dean,
Dale Chen, David Pine,
Degusto, Didgeridoo,
Diligent Key Presser,
ECC-Dan, Emre Bolat,
fallaciousreasoning,
ferday, Florian
Greinacher, ganchito55,
Ginkgo, H. Pauwelyn,
Henrik H, Icy Defiance,
Igor Ševo, iliketocode,
Jatin Sanghvi, Jean-
Bernard Pellerin, Jesse
Williams, Jon Schoning,
Kimmax, Kobi, Kris
Vandermotten, Kritner,
leppie, Llwyd, Maakep,
maf-soft, Marc Gravell,
MarcinJuraszek, Mariano
Desanze, Matt Rowland,
Matt Thomas, MemphiZ,

		mnoronha, MotKohn, Name, Nate Barbettini, Nico, Niek, nietras, NikolayKondratyev, Nuri Tasdemir, PashaPash, Pavel Mayorov, PeteGO, petrzjunior, Philippe, Pratik, Priyank Gadhia, Pyritie, qJake, Raidri, Rakitić, RamenChef, Ray Vega, RBT, René Vogt, Rob, samuelesque, Squidward, Stavm, Stefano, Stefano d'Antonio, Stilgar, Tim Pohlmann, Uriil, user1304444, user2321864, user3185569, uTeisT, Uwe Keim, Vlad, Vlad, Wai Ha Lee, Wasabi Fan, WerWet, wezten, Wojciech Czerniak, Zze
45	Встроенные типы	Alexander Mandt, David, F_V, Haseeb Asif, matteeyah, Patrick Hofman, Wai Ha Lee
46	Выполнение HTTP-запросов	Gordon Bell, Jon Schneider, Mark Shevchenko
47	Генерация кода T4	lloyd, Pavel Mayorov
48	Генерация случайных чисел в C #	A. Can Aydemir, Adi Lester, Alexander Mandt, DLeh, J3soon, Rob
49	Двоичная сериализация	David, Maxim, RamenChef, Stephen Leppik
50	Делегаты	Aaron Hudon, Adam, Ben Aaronson, Benjamin Hodgson, Bradley Uffner, CalmBit, Cihan Yakar, CodeWarrior, EyasSH,

		Huseyin Durmus, Jasmin Solanki, Jeppe Stig Nielsen, Jon G, Jonas S, Matt, NikolayKondratyev, niksofteng, Rajput, Richa Garg, Sam Farajpour Ghamari, Shog9, Stu, Thulani Chivandikwa, trashr0x
51	Делегаты Func	Theodoros Chatzigiannakis, Valentin
52	Деревья выражений	Benjamin Hodgson, dasblinkenlight, Dileep, George Duckett, just.another.programmer, Matas Vaitkevicius, matteeyah, meJustAndrew, Nathan Tuggy, NikolayKondratyev, Rob, Ruben Steins, Stephen Leppik, Рахул Маквана
53	Дженерики	AGB, andre_ss6, Ben Aaronson, Benjamin Hodgson, Benjol, Bobson, Carsten, darth_phoenixx, dymanoid, Eamon Charles, Ehsan Sajjad, Gajendra, GregC, H. Pauwelyn, ja72, Jim, Kroltan, Matas Vaitkevicius, mehmetgil, meJustAndrew, Mord Zuber, Mujassir Nasir, Oly, Pavel Voronin, Richa Garg, Sam, Sebi, Sjoerd222888, Theodoros Chatzigiannakis, user3185569, VictorB, void, Wallace Zhang
54	диагностика	Jasmin Solanki, Luke

		Ryan, TylerH
55	Динамический тип	Daryl, David, H. Pauwelyn, Kilazur, Mark Shevchenko, Nate Barbettini, Rob
56	Доступ к базам данных	ATechieThought, ravindra, Rion Williams, The_Outsider, user2321864
57	Доступ к общей общей папке с именем пользователя и паролем	Mohsin khan
58	Заблокировать	Aaron Hudon, Alexey Groshev, Andrei Rînea, Benjamin Hodgson, Botond Balázs, Christopher Currens, Cihan Yakar, David Ben Knoble, Denis Elkhov, Diligent Key Presser, George Duckett, George Polevoy, Jargon, Jasmin Solanki, Jivan, Mark Shevchenko, Matas Vaitkevicius, Mikko Viitala, Nuri Tasdemir, Oluwafemi, Pavel Mayorov, Richard, Rob, Scott Hannen, Squidward, Vahid Farahmandian
59	Запросы LINQ	Adam Clifford, Ade Stringer, Adi Lester, Adil Mammadov, Akshay Anand, Aleksey L., Alexey Koptyaev, AMW, anaximander, Andrew Piliser, Ankit Vijay, Aphelion, bbonch, Benjamin Hodgson, bmadtiger, BOBS, BrunoLM, BUDI, bumbeishvili, callisto, cbale, Chad McGrath,

Chris, Chris H., coyote,
Daniel Argüelles, Daniel
Corzo, darcyq, David,
David G., David Pine,
DavidG, die maus,
Diligent Key Presser,
Dmitry Bychenko, Dmitry
Egorov, dotctor, Ehsan
Sajjad, Erick, Erik
Schierboom, EvenPrime,
fabriciorissetto, faso,
Finickyflame, Florin M,
forsvarir, fubo,
gbellmann, Gene, Gert
Arnold, Gilad Green, H.
Pauwelyn, Hari Prasad,
hellyale, HimBromBeere,
hWright, iliketocode,
Ioannis Karadimas,
Jagadisha B S, James
Ellis-Jones, jao,
jiaweizhang, Jodrell, Jon
Bates, Jon G, Jon
Schneider, Jonas S,
karaken12, KevinM,
Koopakiller, leppie, LINQ
, Lohitha Palagiri,
Itiveron, Mafii, Martin
Zikmund, Matas
Vaitkevicius, Mateen
Ulhaq, Matt, Maxime,
mburleigh, Meloviz,
Mikko Viitala,
Mohammad Dehghan,
mok, Nate Barbettini,
Neel, Neha Jain, Néstor
Sánchez A., Nico, Noctis
, Pavel Mayorov, Pavel
Yermalovich, Paweł
Hemperek, Pedro, Phuc
Nguyen, pinkfloydx33,
przno, qJake, Racil Hilan
, rdans, Rémi, Rion
Williams, rjdevereux,
RobPethi, Ryan Abbott,
S. Rangeley, S.Akbari,
S.L. Barth, Salvador

		Rubio Martinez, Sanjay Radadiya, Satish Yadav, sebingel, Sergio Domínguez, SilentCoder, Sivanantham Padikkasu, slawekwin, Sondre, Squidward, Stephen Leppik, Steve Trout, Tamir Vered, techspider, teo van kot, th1rdey3, Theodoros Chatzigiannakis, Tim Iles, Tim S. Van Haren, Tobbe, Tom, Travis J, tungns304, Tushar patel, user1304444, user3185569, Valentin, varocarbas, VictorB, Vitaliy Fedorchenco, vivek nuna, void, wali, wertzui, WMios, Xiaoy312, Yaakov Ellis, Zev Spitz
60	Идентификация ASP.NET	HappyCoding, Skulldomani
61	Именованные аргументы	Cihan Yakar, Danny Chen, mehrandvd, Pan, Pavel Mayorov, Stephen Leppik
62	Именованные и необязательные аргументы	RamenChef, Sibeesh Venu, Testing123, The_Outsider, Tim Yusupov
63	Импорт контактов Google	4444, Supraj v
64	имя оператора	Chad, Danny Chen, heltonbiker, Kane, MotKohn, Philip C, pinkfloydx33, Racil Hilan, Rob, Robert Columbia, Sender, Sondre, Stephen Leppik, Wasabi Fan

65	Индексатор	A_Arnold , Ehsan Sajjad , jHilscher
66	Инициализаторы коллекции	Aphelion , ASh , Bart Jolling , Chronocide , CodeCaster , CyberFox , DLeh , Jacob Linney , Jeromy Irvine , Jonas S , Matas Vaitkevicius , Rob , robert demartino , rudygt , Squidward , Tamir Vered , TarkaDaal , Thulani Chivandikwa , WMios
67	Инициализаторы объектов	Andrei , Kroltan , LeopardSkinPillBoxHat , Marco , Nick DeVore , Stephen Leppik
68	Инициализация свойств	Blorgbeard , hatchet , jaycer , Michael Sorens , Parth Patel , Stephen Leppik
69	Интерполяция строк	Arjan Einbu , ATechieThought , avs099 , bluray , Brendan L , Dave Zych , DLeh , Ehsan Sajjad , fabriciorissetto , Guilherme de Jesus Santos , H. Pauwelyn , Jon Skeet , Nate Barbettini , RamenChef , Rion Williams , Squidward , Stephen Leppik , Tushar patel , Wasabi Fan
70	Интерфейс IDisposable	Aaron Hudon , Adam , BatteryBackupUnit , binki , Bogdan Gavril , Bryan Crosby , ChrisWue , Dmitry Bychenko , Ehsan Sajjad , H. Pauwelyn , Jarrod Dixon , Josh Peterson , Matas Vaitkevicius , Maxime , Nicholas Sizer

		OliPro007, Pavel Mayorov, pinkfloydx33, pyrocumulus, RamenChef, Rob, Thennarasan, Will Ray
71	Интерфейс INotifyPropertyChanged	mbrdev, Stephen Leppik, Vlad
72	Интерфейс IQueryable	lucavgobbi, Michiel van Oosterhout, RamenChef, Rob
73	Интерфейсы	Avia, Botond Balázs, CyberFox, harriyott, hellyale, Jeremy Kato, MarcE, MSE, PMF, Preston, Sigh, Sometowngeek, Stagg, Steven, user2441511
74	Использование json.net	Aleks Andreev, Snipzwolf
75	Использование SQLite в C #	Carmine, NikolayKondratyev, th1rdey3, Tim Yusupov
76	Использование директивы	Fernando Matsumoto, Jesse Williams, JohnLBevan, Kit, Michael Freidgeim, Nuri Tasdemir, RamenChef, Tot Zam
77	Использование заявления	Adam Houldsworth, Ahmar, Akshay Anand, Alex Wiese, andre_ss6, Aphelion, Benjol, Boris Callens, Bradley Grainger, Bradley Uffner, bubbleking, Chris Marisic, ChrisWue, Cristian T, cubrr, Dan Ling, Danny Chen, dav_i, David Stockinger, dazerdude, Denis Elkhov, Dmitry Bychenko, Erik

		Schierboom, Florian Greinacher, gdonor, H. Pauwelyn, Herbstein, Jon Schneider, Jon Skeet, Jonesopolis, JT., Ken Keenan, Kev, Kobi, Kyle Trauberman, Lasse Vågsæther Karlsen, LegionMammal978, Lorentz Vedeler, Martin, Martin Zikmund, Maxime , Nuri Tasdemir, Peter K, Philip C, pid, René Vogt, Rion Williams, Ryan Abbott, Scott Koland, Sean, Sparrow, stylef, Sunny R Gupta, Sworgkh, Thaoden, The_Cthulhu_Kid, Tom Droste, Tot Zam, Zaheer UI Hassan
78	итераторы	Botond Balázs, Lijo, Nate Barbettini, Tagc
79	Как использовать C # Structs для создания типа Union (аналогично C Unions)	DLeh, Milton Hernandez, Squidward, usr
80	Кастинг	Benjamin Hodgson, MSE , RamenChef, StriplingWarrior
81	Ключевое слово доходности	Aaron Hudon, Andrew Diamond, Ben Aaronson , ChrisPatrick, Damon Smithies, David G., David Pine, Dmitry Bychenko, dotctor, Ehsan Sajjad, erfanzri, Gajendra, George Duckett, H. Pauwelyn, HimBromBeere, Jeremy Kato, João Lourenço, Joe Amenta, Julien Roncaglia, just.ru, Karthik AMR, Mark Shevchenko, Michael Richardson,

MuiBienCarlota, Myster,
Nate Barbettini, Noctis,
Nuri Tasdemir, Olivier
De Meulder, OP313,
ravindra, Ricardo
Amores, Rion Williams,
rocky, Sompom, Tot
Zam, un-lucky, Vlad,
void, Wasabi Fan,
Xiaoy312, ZenLulz

4444, A_Arnold, Aaron
Hudon, Ade Stringer, Adi
Lester, Aditya Korti,
Adriano Repetti, AJ.,
Akshay Anand, Alex
Filatov, Alexander Pacha
, Amir Pourmand, Andrei
Rînea, Andrew Diamond,
Angela, Anna, Avia, Bart
, Ben, Ben Fogel,
Benjamin Hodgson,
Bjørn-Roger Kringsjå,
Botz3000, Brandon,
brijber, BrunoLM,
BunkerMentality,
BurnsBA, bwegs, Callum
Watkins, Chris, Chris
Akridge, Chris H., Chris
Skardon, ChrisPatrick,
Chuu, Cihan Yakar, cl3m
, Craig Brett, Daniel,
Daniel J.G., Danny Chen
, Darren Davies, Daryl,
dasblinkenlight, David,
David G., David L, David
Pine, DAxaholic,
deadManN,
DeanoMachino,
digitworld, Dmitry
Bychenko, dotctor,
DPenner1, Drew
Kennedy, DrewJordan,
Ehsan Sajjad, EJoshuaS
, Elad Lachmi, Eric
Lippert, EvenPrime, F_V,
Felix, fernacolo,

82 Ключевые слова

Fernando Matsumoto, forsvarir, Francis Lord, Gavin Greenwalt, gdonor, George Duckett, Gilad Naaman, goric, greatwolf, H. Pauwelyn, Happypig375, Icemanind, Jack, Jacob Linney, Jake, James Hughes, Jcoffman, Jeppe Stig Nielsen, jHilscher, João Lourenço, John Slegers, JohnD, Jon Schneider, Jon Skeet, JoshuaBehrens, Kilazur, Kimmax, Kirk Woll, Kit, Kjartan, kjhf, Konamiman, Kyle Trauberman, kyurthich, levininja, lokusking, Mafii, Mamta D, Mango Wong, MarcE, MarcinJuraszek, Marco Scabbiolo, Martin, Martin Klinke, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matěj Pokorný, Mat's Mug, Matthew Whited, Max, Maximilian Ast, Medeni Baykal, Michael Mairegger, Michael Richardson, Michel Keijzers, Mihail Shishkov, mike z, Mr.Mindor, Myster, Nicholas Sizer, Nicholaus Lawson, Nick Cox, Nico, nik, niksofteng, NotEnoughData, numaroth, Nuri Tasdemir, pascalhein, Pavel Mayorov, Pavel Pája Halbich, Pavel Yermalovich, Paweł Mach, Kraskoŭski, Paweł Mach, petelids, Peter Gordon,

		Peter L., PMF, Rakitić, RamenChef, ranieuwe, Razan, RBT, Renan Gemignani, Ringil, Rion Williams, Rob, Robert Columbia, robinmckenzie, RobSiklos, Romain Vincent, RomCoo, ryanyuyu, Sain Pradeep, Sam, Sándor Mátyás Márton, Sanjay Radadiya, Scott, sebingel, Skipper, Sobieck, sohnryang, somebody, Sondre, Squidward, Stephen Leppik, Sujay Sarma, Suyash Kumar Singh, svick, TarkaDaal, th1rdey3, Thaoden, Theodoros Chatzigiannakis, Thorsten Dittmar, Tim Ebenezer, titol, tonirush, topolm, Tot Zam, user3185569, Valentin, vcsjones, void, Wasabi Fan, Wavum, Woodchipper, Xandrmore, Zaheer UI Hassan, Zalomon, Zohar Peled
83	Кодовые контракты	MegaTron
84	Кодовые контракты и утверждения	Roy Dictus
85	Комментарии и регионы	Bad, Botond Balázs, Jonathan Zúñiga, MrDKOz, Ranjit Singh, Squidward
86	Комментарии к документации XML	Alexander Mandt, James, jHilscher, Jon Schneider, Nathan Tuggy, teo van kot, tsjnsn

87	Компиляция времени выполнения	Artificial Stupidity, Stephen Leppik, Tommy
88	Конкатенация строк	Abdul Rehman Sayed, Callum Watkins, ChaoticTwist, Doruk, Dweeberly, Jon Schneider, Oluwafemi, Rob, RubberDuck, Testing123, The_Outsider
89	Конструкторы и финализаторы	Adam Sills, Adi Lester, Adriano Repetti, Andrei Rînea, Andrew Diamond, Arjan Einbu, Avia, BackDoorNoBaby, BanksySan, Ben Fogel, Benjamin Hodgson, Benjol, Bogdan Gavril, Bovaz, Carlos Muñoz, Dan Hulme, Daryl, DLeh, Dmitry Bychenko, drusellers, Ehsan Sajjad, Fernando Matsumoto, guntbert, hatchet, Ian, Jeremy Kato, Jon Skeet, Julien Roncaglia, kamilk, Konamiman, Itiveron, Michael Richardson, Neel, Oly, Pavel Mayorov, Pavel Sapehin, Pavel Voronin, Peter Hommel, pinkfloydx33, Robert Columbia, RomCoo, Roy Dictus, Sam, Saravanan Sachi, Seph, Sklivvz, The_Cthulhu_Kid, Tim Medora, usr, Verena Haunschmid, void, Wouter, ZenLulz
90	Конструкции потока данных параллельной библиотеки задач (TPL)	Droritos, Stephen Leppik
91	Контекст синхронизации в Async-Await	codeape, Mark Shevchenko,

		RamenChef
92	Кортеж	Bovaz, Chawin, EFrank, H. Pauwelyn, Mark Benovsky, Muhammad Albarmawi, Nathan Tuggy, Nikita, Nuri Tasdemir, petrzjunior, PMF, RaYell, slawekwin, Squidward, tire0011
93	Криптография (System.Security.Cryptography)	glaubergrt, MikeS159, Oggas, Pete
94	Кэширование	Aliaksei Futryn, th1rdey3
95	литералы	jaycer, NotEnoughData, Racil Hilan
96	Лямбда-выражения	Andrei Rînea, Benjamin Hodgson, Benjol, David L, David Pine, Federico Allocati, Feelbad Soussi Wolfgun DZ, Fernando Matsumoto, H. Pauwelyn, haim770, Matas Vaitkevicius, Matt Sherman, Michael Mairegger, Michael Richardson, NotEnoughData, Oly, RubberDuck, S.L. Barth, Sunny R Gupta, Tagc, Thriggle
97	Манипуляция строк	Blachshma, Jon Schneider, sferencik, The_Outsider
98	Массивы	A_Arnold, Aaron Hudon, Alexey Groshev, Anas Tasadduq, Andrii Abramov, Baddie, Benjamin Hodgson, bluray, coyote, D.J., das_keyboard, Fernando Matsumoto, granmirupa, Jaydip Jadhav, Jeppe

		Stig Nielsen, Jon Schneider, Ogoun, RamenChef, Robert Columbia, Shyju, The_Outsider, Thomas Weller, tonirush, Tormod Haugene, Wasabi Fan, Wen Qin, Xiobiq, Yotam Salmon
99	методы	Botz3000, F_V, fubo, H. Pauwelyn, Icy Defiance, Jasmin Solanki, Jeremy Kato, Jon Schneider, ken2k, Marco, meJustAndrew, MSL, S.Dav, Sjoerd222888, TarkaDaal, un-lucky
100	Методы DateTime	AbdulRahman Ansari, C4u, Christian Gollhardt, Felipe Oriani, Guilherme de Jesus Santos, James Hughes, Matas Vaitkevicius, midnightsyntax, Mostafiz, Oluwafemi, Pavel Yermalovich, Sondre, theinarasu, Thulani Chivandikwa
101	Методы расширения	Aaron Hudon, AbdulRahman Ansari, Adi Lester, Adil Mammadov, AGB, AldoRomo88, anaximander, Aphelion, Ashwin Ramaswami, ATechieThought, Ben Aaronson, Benjol, binki, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, Brett Veenstra, brijber, Callum Watkins, Chad McGrath, Charlie H, Chris Akridge, Chronocide, CorrectorBot, cubrr,

Dan-Cook, Daniel
Stradowski, David G.,
David Pine, Deepak
gupta, diiN_____,
DLeh, Dmitry Bychenko,
DoNot, DWright, Đan,
Ehsan Sajjad, ekolis,
el2iot2, Elton,
enrico.bacis, Erik
Schierboom, ethorn10,
extremeboredom, Ezra,
fahadash, Federico
Allocati, Fernando
Matsumoto, FrankerZ,
gdziadkiewicz, Gilad
Naaman, GregC,
Gudradain, H. Pauwelyn,
HimBromBeere, Hsu Wei
Cheng, Icy Defiance,
Jamie Rees, Jeppe Stig
Nielsen, John Peters,
John Slegers, Jon
Erickson, Jonas S,
Jonesopolis, Kev, Kevin
Avignon, Kevin DiTraglia
, Kobi, Konamiman,
krillgar, Kurtis Beavers,
Kyle Trauberman,
Lafexlos, LMK, Iothlarias,
Lukáš Lánský, Magisch,
Marc, MarcE, Marek
Musielak, Martin
Zikmund, Matas
Vaitkevicius, Matt, Matt
Dillard, Maximilian Ast,
mbrdev,
MDTech.us_MAN,
meJustAndrew, Michael
Benford, Michael
Freidgeim, Michael
Richardson, Michał
Perłakowski, Nate
Barbettini, Nick Larsen,
Nico, Nisarg Shah, Nuri
Tasdemir, Parth Patel,
pinkfloydx33, PMF,
Prashanth Benny, QoP,

		Raidri, Reddy, Reeven, Ricardo Amores, Richard, Rion Williams, Rob, Robert Columbia, Ryan Hilbert, ryanyuyu, S. Tarık Çetin, Sam Axe, Shoe, Sibeesh Venu, solidcell, Sondre, Squidward, Steven, style, SysVoid, Tanner Swett, Timothy Rascher, TKharaishvili, T-moty, Tobbe, Tushar patel, unarist, user3185569, user40521, Ven, Victor Tomaili, viggity
102	Многопоточность	Aaron Hudon, Alexander Petrov, Austin T French, captainjamie, Eldar Dordzhiev, H. Pauwelyn, ionmike, Jacob Linney, JohnLBevan, leondepedlaw, Mamta D, Matthijs Wessels, Mellow, RamenChef, Zoba
103	Модификаторы доступа	Botond Balázs, H. Pauwelyn, hatcyl, John, Justin Rohr, Kobi, Robert Woods, Thaoden, ZenLulz
104	наследование	Almir Vuk, andre_ss6, Andrew Diamond, Barathon, Ben Aaronson, Ben Fogel, Benjol, David L, deloreyk, Ehsan Sajjad, harriyott, ja72, Jon Ericson, Karthik, Konamiman, MarcE, Matas Vaitkevicius, Pete Uh, Rion Williams, Robert Columbia, Steven, Suren Srapyan, VirusParadox, Yehuda Shapira

105	Начало работы: Json с C #	Neo Vijay, Rob, VitorCioletti
106	Небезопасный код в .NET.	Andrew Piliser, cbale, codekaizen, Danny Varod, Isac, Jaroslav Kadlec, MSE, Nisarg Shah, Rahul Nikate, Stephen Leppik, Uwe Keim, ZenLulz
107	Неверные типы	Benjamin Hodgson, Braydie, DmitryG, Gordon Bell, Jasmin Solanki, Jon Schneider, Konstantin Vdovkin, Maximilian Ast, Mikko Viitala, Nicholas Sizer, Patrick Hofman, Pavel Mayorov, pinkfloydx33, Vitaliy Fedorchenco
108	неизменность	Boggin, Jon Schneider, Oluwafemi, Tim Ebenezer
109	Нуль-условные операторы	Alpha, dazerdude, DLeh, Draken, George Duckett, Jon Schneider, Kobi, Max, Nathan, Nicholas Sizer, Rob, Stephen Leppik, tehDorf, Timothy Shields, topolm, Wasabi Fan
110	Обеспечение безопасности переменной потока	Wyck
111	Обзор коллекций с #	Aaron Hudon, Andrew Diamond, Denuath, Jeremy Kato, Jon Schneider, Jorge, Juha Palomäki, Leon Husmann, Michael Mairegger, Michael Richardson, Nikita, rene, Rob, Sebi, TarkaDaal, wertzui, Will Ray

112	Обработка FormatException при преобразовании строки в другие типы	Rakitić, un-lucky
113	Обработка исключений	0x49D1, Abdul Rehman Sayed, Adam Lear, Adil Mammadov, Andrew Diamond, Aseem Gautam, Athafoud, Botond Balázs, Collin Stevens, Danny Chen, Dmitry Bychenko, dove, Eldar Dordzhiev, fabriciorissetto, faso, flq, George Duckett, Gilad Naaman, Gudradain, Jack, James Hughes, Jamie Rees, John Meyer, Jonesopolis, MaddinTribleD, Marimba, Matas Vaitkevicius, Matt, matteeyah, Menthak, Michael Bisbjerg, Nate Barbettini, Nathaniel Ford, nik0lias, niksofteng, Oly, Pavel Pája Halbich, Pavel Voronin, PMF, Racil Hilan, raidensan, Rasa, Robert Columbia, RomCoo, Sam Hanley, Scott Koland, Squidward, Steve Dunn, Thulani Chivandikwa, vesi
114	Обработчик проверки подлинности C #	Abbas Galiyakotwala
115	Общие операции с строками	Austin T French, Blachshma, bluish, CharithJ, Chief Wiggum, cyberj0g, Daryl, deloreyk, jaycer, Jaydip Jadhav, Jon G, Jon Schneider, juergen d, Konamiman, Maniero, Paul Weiland, Racil Hilan, RoelF, Stefan Steiger, Steven, The_Outsider, tiedied61,

		un-lucky, WizardOfMenlo
116	Объектно-ориентированное программирование на C #	Yashar Aliabasi
117	Оператор Null-Coalescing	aashishkoirala, Ankit Rana, Aristos, Bradley Uffner, David Arno, David G., David Pine, demonplus, Denis Elkhov, Diligent Key Presser, Eamon Charles, Ehsan Sajjad, eouw0o83hf, Fernando Matsumoto, H. Pauwelyn, Jodrell, Jon Schneider, Jonesopolis, Martin Zikmund, Mike C, Nate Barbettini, Nic Foster, petelids, Prateek, Rahul Nikate, Rion Williams, Rob, smead, tonirush, Wasabi Fan, Will Ray
118	Оператор равенства	Vadim Martynov
119	операторы	Adam Houldsworth, Adi Lester, Adil Mammadov, Akshay Anand, Alan McBee, Avi Turner, Ben Fogel, Blorgbeard, Blubberguy22, Chris Jester-Young, David Basarab, DLeh, Dmitry Bychenko, dotctor, Ehsan Sajjad, fabriciorissetto, Fernando Matsumoto, H. Pauwelyn, Henrik H, Jake Farley, Jasmin Solanki, Jephron, Jeppe Stig Nielsen, Jesse Williams, Joe, JohnLBevan, Jon Schneider, Jonas S, Kevin Montrose, Kimmax, lokusking, Matas

		Vaitkevicius, meJustAndrew, Mikko Viitala, mmushtaq, Mohamed Belal, Nate Barbettini, Nico, Oly, pascalhein, Pavel Voronin, petelids, Philip C, Racil Hilan, RhysO, Robert Columbia, Rodolfo Fadino Junior, Sachin Joseph, Sam, slawekwin, slinzerthegod , Squidward, Testing123, TyCobb, Wasabi Fan, Xiaoy312, Zaheer UI Hassan
120	отражение	Alexander Mandt, Aman Sharma, artemisart, Aseem Gautam, Axarydax, Benjamin Hodgson, Botond Balázs , Carson McManus, Cigano Morrison Mendez , Cihan Yakar, da_sann, DVJex, Ehsan Sajjad, H. Pauwelyn, Haim Bendanan, HimBromBeere, James Ellis-Jones, James Hughes, Jamie Rees, Jan Peldřimovský, Johny Skovdal, JSF, Kobi, Konamiman, Kristijan, Lovy, Matas Vaitkevicius , Mourndark, Nuri Tasdemir, pinkfloydx33, Rekshino, René Vogt, Sachin Chavan, Shuffler, Sjoerd222888, Sklivvz, Tamir Vered, Thriggle, Travis J, uygar.raf, Vadim Ovchinnikov, wablab, Wai Ha Lee
121	Параллельная библиотека задач	Benjamin Hodgson, Brandon, Collin Stevens,

		i3arnon , Mokhtar Ashour , Murtuza Vohra
122	Параллельный LINQ (PLINQ)	Adi Lester
123	перекручивание	Alisson , Andrei Rînea , B Hawkins , Benjamin Hodgson , Botond Balázs , connor , Dialecticus , DJCubed , Freelex , Jon Schneider , Oluwafemi , Racil Hilan , Squidward , Testing123 , Tolga Evcimen
124	перелив	Akshay Anand , Nuri Tasdemir , tonirush
125	Платформа компилятора .NET (Roslyn)	4444 , Lukáš Lánský
126	Полиморфизм	Ade Stringer , ganchito55 , H. Pauwelyn , Karthik , Maximilian Ast , void
127	Поток	Danny Bogers , jlawcordova , Jon Schneider , Nuri Tasdemir , Pushpendra
128	Преобразование типа	Community , connor , Ehsan Sajjad , Lijo
129	Препроцессорные директивы	Andrei , Gilad Naaman , Matas Vaitkevicius , qJake , RamenChef , theB , volvis
130	Примеры AssemblyInfo.cs	Adi Lester , Ameya Deshpande , AndreyAkinshin , Boggin , Dodzi Dzakuma , dove , Joel Martinez , pinkfloydx33 , Ralf Bönning , Theodoros Chatzigiannakis , Wasabi Fan
131	Проверено и не отмечено	Botond Balázs , Rahul Nikate , Sam Johnson ,

		ZenLulz
132	Псевдонимы встроенных типов	Racil Hilan, Rahul Nikate, Stephen Leppik
133	Равные и GetHashCode	Alexey, BanksySan, hatcyl, ja72, Jeppe Stig Nielsen, meJustAndrew, Rob, scher, Timiry, viggity
134	Разрешение перегрузки	Dunno, Petr Hudeček, Stephen Leppik, TorbenJ
135	Реактивные расширения (Rx)	stefankmitph
136	Реализация Singleton	Aaron Hudon, Adam, Adi Lester, Andrei Rînea, cbale, Disk Crash, Ehsan Sajjad, Krzysztof Branicki, lothlarias, Mark Shevchenko, Pavel Mayorov, Sklivvz, snickro, Squidward, Squirrel, Stephen Leppik, Victor Tomaili, Xandrmore
137	Рекурсия	Alexey Groshev, Botond Balázs, connor, ephtee, Florian Koch, Kroltan, Michael Brandon Morris, Mulder, Pan, qJake, Robert Columbia, Roy Dictus, SlaterCodes, Yves Schelpe
138	Сборщик мусора в .Net	Andrei Rînea, da_sann, Eamon Charles, J3soon, Luke Ryan, Squidward, Suren Srapyan
139	свойства	Botond Balázs, Callum Watkins, Jeremy Kato, John, JohnLBevan, niksofteng, Stephen Leppik, Zohar Peled

140	Секундомеры	Adam, demonplus, dotctor, Gavin Greenwalt, Jeppe Stig Nielsen, Sondre
141	сетей	Adi Lester, Nicholaus Lawson, Salih Karagoz, shawty, Squirrel, Xander Luciano
142	События	Aaron Hudon, Adi Lester, Benjol, CheGuevarasBeret, dcastro, matteeyah, meJustAndrew, mhoward, nik, niksofteng, NotEnoughData, OliPro007, paulius_I, PSGuy, Reza Aghaei, Roy Dictus, Squidward, Steven, vbnet3d
143	Соглашения об именах	Ben Aaronson, Callum Watkins, PMF, ZenLulz
144	Создание консольного приложения с использованием редактора Plain-Text и компилятора C # (csc.exe)	delete me
145	Создание собственного MessageBox в приложении Windows Form	Mansel Davies, Vaibhav_Welcomes_You
146	Создание шаблонов проектирования	DWright, Jan Bońkowski, Mark Shevchenko, Parth Patel, PedroSouki, Pierre Theate, Sondre, Tushar patel
147	Статические классы	MCronin, The_Outsider, Xiaoy312
148	Строковые escape-последовательности	Benjol, Botond Balázs, cubrr, Ed Gibbs, Jeppe Stig Nielsen, LegionMammal978, Michael Richardson, Peter Gordon, Petr Hudeček, Squidward, tonirush

149	Структурные шаблоны проектирования	Timon Post
150	Структуры	abto, Alexey Groshev, Benjamin Hodgson, Botz3000, David, Elad Lachmi, ganchito55, Jon Schneider, NikolayKondratyev
151	Сценарий C #	mehrandvd, Squidward, Stephen Leppik
152	Таймеры	Adam, Akshay Anand, Benjamin Kozuch, ephtee, RamenChef, Thennarasan
153	Тип значения vs Тип ссылки	Abdul Rehman Sayed, Adam, Amir Pourmand, Blubberguy22, Chronocide, Craig Brett, docesam, GWigWam, matiaslauriti, meJustAndrew, Michael Mairegger, Michele Ceo, Moe Farag, Nate Barbettini, RamenChef, Rob, scher, Snympi, Tagc, Theodoros Chatzigiannakis
154	указатели	Jeppe Stig Nielsen, Theodoros Chatzigiannakis
155	Указатели и небезопасный код	Aaron Hudon, Botond Balázs, undefined
156	Условные заявления	Alexander Mandt, Ameya Deshpande, EJoshuaS, H. Pauwelyn, Hayden, Kroltan, RamenChef, Sklivvz
157	Файловый и потоковый ввод-вывод	BanksySan, Blachshma, dbmuller, DJCubed, Feelbad Soussi Wolfgang DZ, intox, Mikko Viitala,

		Sender, Squidward, Tolga Evcimen, Wasabi Fan
158	Фильтры действий	Lokesh_Ram
159	Функциональное программирование	Andrei Epure, Boggin, Botond Balázs, richard
160	Функция с несколькими возвращаемыми значениями	Adam, Alexey Mitev, Durgpal Singh, Tolga Evcimen
161	Хэш-функции	Adi Lester, Callum Watkins, EvenPrime, ganchito55, Igor, jHilscher, RamenChef, ZenLulz
162	Частичный класс и методы	Ben Jenkinson, Jonas S, Rahul Nikate, Stephen Leppik, Taras, The_Outsider
163	Читать и понимать Stacktraces	S.L. Barth
164	Чтение и запись .zip-файлов	4444, DLeh, Naveen Gogineni, Nisarg Shah