

LinksPlatform's Platform.Data Class Library

./Exceptions/ArgumentLinkDoesNotExistsException.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Exceptions
6  {
7      public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
8      {
9          public ArgumentLinkDoesNotExistsException(TLinkAddress link, string paramName) :
10             ↪ base(FormatMessage(link, paramName), paramName) { }
11          public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link))
12             ↪ { }
13          private static string FormatMessage(TLinkAddress link, string paramName) => $"Связь
14             ↪ [{link}] переданная в аргумент [{paramName}] не существует.";
15          private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
16             ↪ качестве аргумента не существует.";
17      }
18 }

```

./Exceptions/ArgumentLinkHasDependenciesException.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Exceptions
6  {
7      public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
8      {
9          public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) :
10             ↪ base(FormatMessage(link, paramName), paramName) { }
11          public ArgumentLinkHasDependenciesException(TLinkAddress link) :
12             ↪ base(FormatMessage(link)) { }
13          private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи
14             ↪ [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые
15             ↪ препятствуют изменению её внутренней структуры.";
16          private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной
17             ↪ в качестве аргумента присутствуют зависимости, которые препятствуют изменению её
18             ↪ внутренней структуры.";
19      }
20 }

```

./Exceptions/LinksLimitReachedException.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Exceptions
6  {
7      public class LinksLimitReachedException<TLinkAddress> : Exception
8      {
9          public static readonly string DefaultMessage = "Достигнут лимит количества связей в
10             ↪ хранилище.";
11          public LinksLimitReachedException(string message) : base(message) { }
12          public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }
13          public LinksLimitReachedException() : base(DefaultMessage) { }
14          private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества
15             ↪ связей в хранилище ({limit}).";
16      }
17 }

```

./Exceptions/LinkWithSameValueAlreadyExistsException.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Exceptions
6  {
7      public class LinkWithSameValueAlreadyExistsException : Exception
8      {
9          public static readonly string DefaultMessage = "Связь с таким же значением уже
10             ↪ существует.";
11          public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
12          public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
13      }
14 }

```

./ILinks.cs

```
1 using System;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data
7 {
8     /// <summary>
9     /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
10    /// </summary>
11    /// <remarks>
12    /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит
13    /// → подходит как для дуплетов, так и для триплетов и т.п.
14    /// Возможно этот интерфейс подходит даже для Sequences.
15    /// </remarks>
16    public interface ILinks<TLinkAddress, TConstants>
17    where TConstants : LinksConstants<TLinkAddress>
18    {
19        #region Constants
20
21        /// <summary>
22        /// Возвращает набор констант, который необходим для эффективной коммуникации с методами
23        /// → этого интерфейса.
24        /// Эти константы не меняются с момента создания точки доступа к хранилищу.
25        /// </summary>
26        TConstants Constants { get; }
27
28        #endregion
29
30        #region Read
31
32        /// <summary>
33        /// Подсчитывает и возвращает общее число связей находящихся в хранилище,
34        /// → соответствующих указанным ограничениям.
35        /// </summary>
36        /// <param name="restriction">Ограничения на содержимое связей.</param>
37        /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным
38        /// → ограничениям.</returns>
39        TLinkAddress Count(IList<TLinkAddress> restriction);
40
41        /// <summary>
42        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
43        /// → (handler) для каждой подходящей связи.
44        /// </summary>
45        /// <param name="handler">Обработчик каждой подходящей связи.</param>
46        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
47        /// → может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
48        /// → Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
49        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
50        /// → случае.</returns>
51        TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress>
52        → restrictions);
53
54        #endregion
55
56        #region Write
57
58        /// <summary>
59        /// Создает связь.
60        /// </summary>
61        /// <returns>Индекс созданной связи.</returns>
62        TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно
63        → принимать restrictions, возможно и возвращать связь нужно целиком.
64
65        /// <summary>
66        /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
67        /// на связь с указанным новым содержимым.
68        /// </summary>
69        /// <param name="restrictions">
70        /// Ограничения на содержимое связей.
71        /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
72        /// → и далее за ним будет следовать содержимое связи.
73        /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
74        /// → ссылку на пустоту,
75        /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
76        /// → другой связи.
77        /// </param>
78        /// <param name="substitution"></param>
```

```

66     /// <returns>Индекс обновлённой связи.</returns>
67     TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
    → // TODO: Возможно и возвращать связь нужно целиком.
68
69     /// <summary>Удаляет связь с указанным индексом.</summary>
70     void Delete(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно принимать
    → restrictions, а так же возвращать удалённую связь, если удаление было реально
    → выполнено, и Null, если нет.
71
72     #endregion
73 }
74 }

```

./ILinksExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Setters;
5  using Platform.Data.Exceptions;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data
10 {
11     public static class ILinksExtensions
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    → TConstants> links, params TLinkAddress[] restrictions)
15             where TConstants : LinksConstants<TLinkAddress>
16             => links.Count(restrictions);
17
18         /// <summary>
19         /// Возвращает значение, определяющее существует ли связь с указанным индексом в
    → хранилище связей.
20         /// </summary>
21         /// <param name="links">Хранилище связей.</param>
22         /// <param name="link">Индекс проверяемой на существование связи.</param>
23         /// <returns>Значение, определяющее существует ли связь.</returns>
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    → TConstants> links, TLinkAddress link)
26             where TConstants : LinksConstants<TLinkAddress>
27         {
28             var constants = links.Constants;
29             return constants.IsExternalReference(link) || (constants.IsInnerReference(link) &&
    → Comparer<TLinkAddress>.Default.Compare(links.Count(new
    → LinkAddress<TLinkAddress>(link)), default) > 0);
30         }
31
32         /// <param name="links">Хранилище связей.</param>
33         /// <param name="link">Индекс проверяемой на существование связи.</param>
34         /// <remarks>
35         /// TODO: May be move to EnsureExtensions or make it both there and here
36         /// </remarks>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    → TConstants> links, TLinkAddress link)
39             where TConstants : LinksConstants<TLinkAddress>
40         {
41             if (!links.Exists(link))
42             {
43                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
44             }
45         }
46
47         /// <param name="links">Хранилище связей.</param>
48         /// <param name="link">Индекс проверяемой на существование связи.</param>
49         /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    → TConstants> links, TLinkAddress link, string argumentName)
52             where TConstants : LinksConstants<TLinkAddress>
53         {
54             if (!links.Exists(link))
55             {
56                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
57             }
58         }
59     }
60 }

```

```

59
60 /// <summary>
61 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
62   ↳ (handler) для каждой подходящей связи.
63 /// </summary>
64 /// <param name="links">Хранилище связей.</param>
65 /// <param name="handler">Обработчик каждой подходящей связи.</param>
66 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
67   ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
68   ↳ Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
69 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
70   ↳ случае.</returns>
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
73   ↳ TConstants> links, Func<IList<TLinkAddress>, TLinkAddress> handler, params
74   ↳ TLinkAddress[] restrictions)
75     where TConstants : LinksConstants<TLinkAddress>
76     => links.Each(handler, restrictions);
77
78 /// <summary>
79 /// Возвращает части-значения для связи с указанным индексом.
80 /// </summary>
81 /// <param name="links">Хранилище связей.</param>
82 /// <param name="link">Индекс связи.</param>
83 /// <returns>Уникальную связь.</returns>
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
86   ↳ ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
87     where TConstants : LinksConstants<TLinkAddress>
88 {
89     var constants = links.Constants;
90     if (constants.IsExternalReference(link))
91     {
92         return new Point<TLinkAddress>(link, constants.TargetPart + 1);
93     }
94     var linkPartsSetter = new Setter<IList<TLinkAddress>,
95       ↳ TLinkAddress>(constants.Continue, constants.Break);
96     links.Each(linkPartsSetter.SetAndReturnTrue, link);
97     return linkPartsSetter.Result;
98 }
99
100 #region Points
101
102 /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
103   ↳ точкой полностью (связью замкнутой на себе дважды).</summary>
104 /// <param name="links">Хранилище связей.</param>
105 /// <param name="link">Индекс проверяемой связи.</param>
106 /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
107 /// <remarks>
108 /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
109   ↳ связь.
110 /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
111   ↳ точка и пара существовать одновременно?
112 /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
113   ↳ сортировать по индексу в массиве связей?
114 /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
115 /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
116   ↳ самой себя любого размера?
117 /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
118   ↳ одной ссылки на себя (частичной точки).
119 /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
120   ↳ самостоятельный цикл через себя? Что если предоставить все варианты использования
121   ↳ связей?
122 /// Что если разрешить и нули, а так же частичные варианты?
123 ///
124 /// Что если точка, это только в том случае когда link.Source == link &&
125   ↳ link.Target == link, т.е. дважды ссылка на себя.
126 /// А пара это тогда, когда link.Source == link.Target && link.Source != link,
127   ↳ т.е. ссылка не на себя а во вне.
128 ///
129 /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
130   ↳ промежуточную связь,
131   ↳ например "DoubletOf" обозначить что является точно парой, а что точно точкой.
132 /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
133 /// </remarks>
134 public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
135   ↳ TConstants> links, TLinkAddress link)
136     where TConstants : LinksConstants<TLinkAddress>

```



```

27         return Index;
28     }
29     else
30     {
31         throw new IndexOutOfRangeException();
32     }
33 }
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 set => throw new NotSupportedException();
36 }
37
38 public int Count => 1;
39
40 public bool IsReadOnly => true;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public LinkAddress(TLinkAddress index) => Index = index;
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public void Add(TLinkAddress item) => throw new NotSupportedException();
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public void Clear() => throw new NotSupportedException();
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
53     ↪ ? true : false;
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public IEnumerator<TLinkAddress> GetEnumerator()
60 {
61     yield return Index;
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
66     ↪ 0 : -1;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public bool Remove(TLinkAddress item) => throw new NotSupportedException();
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public void RemoveAt(int index) => throw new NotSupportedException();
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 IEnumerator IEnumerable.GetEnumerator()
79 {
80     yield return Index;
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public virtual bool Equals(LinkAddress<TLinkAddress> other) =>
85     ↪ _equalityComparer.Equals(Index, other.Index);
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
89     ↪ new LinkAddress<TLinkAddress>(linkAddress);
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
93     ↪ ? Equals(linkAddress) : false;
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 public override int GetHashCode() => Index.GetHashCode();
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 public override string ToString() => Index.ToString();

```

```

100     public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
101         ↪ right) => left.Equals(right);
102
103     [MethodImpl(MethodImplOptions.AggressiveInlining)]
104     public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
105         ↪ right) => !(left == right);
106 }

```

./LinksConstants.cs

```

1  using Platform.Numbers;
2  using Platform.Ranges;
3  using Platform.Reflection;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data
8  {
9      public class LinksConstants<TLinkAddress>
10     {
11         public static readonly int DefaultTargetPart = 2;
12
13         #region Link parts
14
15         /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
16         ↪ самой связи.</summary>
17         public int IndexPart { get; }
18
19         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
20         ↪ часть-значение).</summary>
21         public int SourcePart { get; }
22
23         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
24         ↪ (последняя часть-значение).</summary>
25         public int TargetPart { get; }
26
27         #endregion
28
29         #region Flow control
30
31         /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
32         /// <remarks>Используется в функции обработчике, который передаётся в функцию
33         ↪ Each.</remarks>
34         public TLinkAddress Continue { get; }
35
36         /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
37         public TLinkAddress Skip { get; }
38
39         /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
40         /// <remarks>Используется в функции обработчике, который передаётся в функцию
41         ↪ Each.</remarks>
42         public TLinkAddress Break { get; }
43
44         #endregion
45
46         #region Special symbols
47
48         /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
49         public TLinkAddress Null { get; }
50
51         /// <summary>Возвращает значение, обозначающее любую связь.</summary>
52         /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
53         ↪ создавать все варианты последовательностей в функции Create.</remarks>
54         public TLinkAddress Any { get; }
55
56         /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
57         public TLinkAddress Itself { get; }
58
59         #endregion
60
61         #region References
62
63         /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
64         ↪ ссылок).</summary>
65         public Range<TLinkAddress> PossibleInnerReferencesRange { get; }
66
67         /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
68         ↪ ссылок).</summary>
69         public Range<TLinkAddress>? PossibleExternalReferencesRange { get; }
70
71     }
72 }

```

```

63     #endregion
64
65     public LinksConstants(int targetPart, Range<TLinkAddress> possibleInnerReferencesRange,
66     ↪ Range<TLinkAddress>? possibleExternalReferencesRange)
67     {
68         IndexPart = 0;
69         SourcePart = 1;
70         TargetPart = targetPart;
71         Null = Integer<TLinkAddress>.Zero;
72         Break = Integer<TLinkAddress>.Zero;
73         var currentInnerReferenceIndex = possibleInnerReferencesRange.Maximum;
74         Continue = currentInnerReferenceIndex;
75         Decrement(ref currentInnerReferenceIndex);
76         Skip = currentInnerReferenceIndex;
77         Decrement(ref currentInnerReferenceIndex);
78         Any = currentInnerReferenceIndex;
79         Decrement(ref currentInnerReferenceIndex);
80         Itself = currentInnerReferenceIndex;
81         Decrement(ref currentInnerReferenceIndex);
82         PossibleInnerReferencesRange = (possibleInnerReferencesRange.Minimum,
83     ↪ currentInnerReferenceIndex);
84         PossibleExternalReferencesRange = possibleExternalReferencesRange;
85     }
86
87     private static void Decrement(ref TLinkAddress currentInnerReferenceIndex) =>
88     ↪ currentInnerReferenceIndex = Arithmetic.Decrement(currentInnerReferenceIndex);
89
90     public LinksConstants(Range<TLinkAddress> possibleInnerReferencesRange,
91     ↪ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
92     ↪ possibleInnerReferencesRange, possibleExternalReferencesRange) { }
93
94     public LinksConstants(int targetPart, Range<TLinkAddress> possibleInnerReferencesRange)
95     ↪ : this(targetPart, possibleInnerReferencesRange, null) { }
96
97     public LinksConstants(Range<TLinkAddress> possibleInnerReferencesRange) :
98     ↪ this(DefaultTargetPart, possibleInnerReferencesRange, null) { }
99
100    public LinksConstants() : this(DefaultTargetPart, (Integer<TLinkAddress>.One,
101    ↪ NumericType<TLinkAddress>.MaxValue), null) { }
102
103    }
104 }

```

./LinksConstantsExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data
6  {
7      public static class LinksConstantsExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static bool IsReference<TLinkAddress>(this LinksConstants<TLinkAddress>
11         ↪ linksConstants, TLinkAddress address) => linksConstants.IsInnerReference(address) ||
12         ↪ linksConstants.IsExternalReference(address);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static bool IsInnerReference<TLinkAddress>(this LinksConstants<TLinkAddress>
16         ↪ linksConstants, TLinkAddress address) =>
17         ↪ linksConstants.PossibleInnerReferencesRange.ContainsValue(address);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
21         ↪ linksConstants, TLinkAddress address) =>
22         ↪ linksConstants.PossibleExternalReferencesRange?.ContainsValue(address) ?? false;
23     }
24 }

```

./Point.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Exceptions;
6  using Platform.Ranges;
7  using Platform.Collections;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10

```



```

11 namespace Platform.Data
12 {
13     public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
14     {
15         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
16             ↪ EqualityComparer<TLinkAddress>.Default;
17
18         public TLinkAddress Index
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public int Size
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public TLinkAddress this[int index]
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get
34             {
35                 if (index < Size)
36                 {
37                     return Index;
38                 }
39                 else
40                 {
41                     throw new IndexOutOfRangeException();
42                 }
43             }
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set => throw new NotSupportedException();
46         }
47
48         public int Count => int.MaxValue;
49
50         public bool IsReadOnly => true;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public Point(TLinkAddress index, int size)
54         {
55             Index = index;
56             Size = size;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public void Add(TLinkAddress item) => throw new NotSupportedException();
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public void Clear() => throw new NotSupportedException();
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
67             ↪ ? true : false;
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public IEnumerator<TLinkAddress> GetEnumerator()
74         {
75             for (int i = 0; i < Size; i++)
76             {
77                 yield return Index;
78             }
79         }
80
81         [MethodImpl(MethodImplOptions.AggressiveInlining)]
82         public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
83             ↪ 0 : -1;
84
85         [MethodImpl(MethodImplOptions.AggressiveInlining)]
86         public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
87
88         [MethodImpl(MethodImplOptions.AggressiveInlining)]
89         public bool Remove(TLinkAddress item) => throw new NotSupportedException();
90

```

```

88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public void RemoveAt(int index) => throw new NotSupportedException();
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 IEnumerator IEnumerable.GetEnumerator()
93 {
94     for (int i = 0; i < Size; i++)
95     {
96         yield return Index;
97     }
98 }
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public virtual bool Equals(LinkAddress<TLinkAddress> other) =>
102     ↪ _equalityComparer.Equals(Index, other.Index);
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
106     ↪ linkAddress.Index;
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
110     ↪ Equals(linkAddress) : false;
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public override int GetHashCode() => Index.GetHashCode();
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public override string ToString() => Index.ToString();
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
120     ↪ left.Equals(right);
121
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
124     ↪ !(left == right);
125
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 public static bool IsFullPoint(params TLinkAddress[] link) =>
128     ↪ IsFullPoint((IList<TLinkAddress>)link);
129
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 public static bool IsFullPoint(IList<TLinkAddress> link)
132 {
133     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
134     Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
135         ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
136     return IsFullPointUnchecked(link);
137 }
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
141 {
142     var result = true;
143     for (var i = 1; result && i < link.Count; i++)
144     {
145         result = _equalityComparer.Equals(link[0], link[i]);
146     }
147     return result;
148 }
149
150 [MethodImpl(MethodImplOptions.AggressiveInlining)]
151 public static bool IsPartialPoint(params TLinkAddress[] link) =>
152     ↪ IsPartialPoint((IList<TLinkAddress>)link);
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 public static bool IsPartialPoint(IList<TLinkAddress> link)
156 {
157     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
158     Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
159         ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
160     return IsPartialPointUnchecked(link);
161 }
162
163 [MethodImpl(MethodImplOptions.AggressiveInlining)]
164 public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
165 {

```

```

157         var result = false;
158         for (var i = 1; !result && i < link.Count; i++)
159         {
160             result = _equalityComparer.Equals(link[0], link[i]);
161         }
162         return result;
163     }
164 }
165 }

```

./Sequences/ISequenceAppender.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Sequences
4  {
5      public interface ISequenceAppender<TLinkAddress>
6      {
7          TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant);
8      }
9  }

```

./Sequences/ISequenceWalker.cs

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Sequences
6  {
7      public interface ISequenceWalker<TLinkAddress>
8      {
9          IEnumerable<IList<TLinkAddress>> Walk(TLinkAddress sequence);
10     }
11 }

```

./Sequences/SequenceWalker.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Sequences
8  {
9      /// <remarks>
10     /// Реализованный внутри алгоритм наглядно показывает,
11     /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой
12     /// ↪ себя),
13     /// так как стек можно использовать намного эффективнее при ручном управлении.
14     ///
15     /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
16     /// Решить встраивать ли защиту от заикливания.
17     /// Альтернативой защиты от заикливания может быть заранее известное ограничение на
18     /// ↪ погружение вглубь.
19     /// А так же качественное распознавание прохода по циклическому графу.
20     /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер
21     /// ↪ стека.
22     /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
23     /// </remarks>
24     public static class SequenceWalker
25     {
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
28             ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
29             ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
30         {
31             var stack = new Stack<TLinkAddress>();
32             var element = sequence;
33             if (isElement(element))
34             {
35                 visit(element);
36             }
37             else
38             {
39                 while (true)
40                 {
41                     if (isElement(element))
42                     {
43                         if (stack.Count == 0)

```

```

39         {
40             break;
41         }
42         element = stack.Pop();
43         var source = getSource(element);
44         var target = getTarget(element);
45         if (isElement(source))
46         {
47             visit(source);
48         }
49         if (isElement(target))
50         {
51             visit(target);
52         }
53         element = target;
54     }
55     else
56     {
57         stack.Push(element);
58         element = getSource(element);
59     }
60 }
61 }
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static void WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
66 {
67     var stack = new Stack<TLinkAddress>();
68     var element = sequence;
69     if (isElement(element))
70     {
71         visit(element);
72     }
73     else
74     {
75         while (true)
76         {
77             if (isElement(element))
78             {
79                 if (stack.Count == 0)
80                 {
81                     break;
82                 }
83                 element = stack.Pop();
84                 var source = getSource(element);
85                 var target = getTarget(element);
86                 if (isElement(target))
87                 {
88                     visit(target);
89                 }
90                 if (isElement(source))
91                 {
92                     visit(source);
93                 }
94                 element = source;
95             }
96             else
97             {
98                 stack.Push(element);
99                 element = getTarget(element);
100             }
101         }
102     }
103 }
104 }
105 }

```

./Sequences/StopableSequenceWalker.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Sequences
8 {

```

```

9  /// <remarks>
10  /// Реализованный внутри алгоритм наглядно показывает,
11  /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой
    ↪ себя),
12  /// так как стек можно использовать намного эффективнее при ручном управлении.
13  ///
14  /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
15  /// Решить встраивать ли защиту от заикливания.
16  /// Альтернативой защиты от заикливания может быть заранее известное ограничение на
    ↪ погружение вглубь.
17  /// А так же качественное распознавание прохода по циклическому графу.
18  /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер
    ↪ стека.
19  /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
20  /// </remarks>
21  public static class StopableSequenceWalker
22  {
23      [MethodImpl(MethodImplOptions.AggressiveInlining)]
24      public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> enter, Action<TLinkAddress>
    ↪ exit, Func<TLinkAddress, bool> canEnter, Func<TLinkAddress, bool> visit)
25      {
26          var exited = 0;
27          var stack = new Stack<TLinkAddress>();
28          var element = sequence;
29          if (isElement(element))
30          {
31              return visit(element);
32          }
33          while (true)
34          {
35              if (isElement(element))
36              {
37                  if (stack.Count == 0)
38                  {
39                      return true;
40                  }
41                  element = stack.Pop();
42                  exit(element);
43                  exited++;
44                  var source = getSource(element);
45                  var target = getTarget(element);
46                  if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
    ↪ !visit(source))
47                  {
48                      return false;
49                  }
50                  if ((isElement(target) || !canEnter(target)) && !visit(target))
51                  {
52                      return false;
53                  }
54                  element = target;
55              }
56              else
57              {
58                  if (canEnter(element))
59                  {
60                      enter(element);
61                      exited = 0;
62                      stack.Push(element);
63                      element = getSource(element);
64                  }
65                  else
66                  {
67                      if (stack.Count == 0)
68                      {
69                          return true;
70                      }
71                      element = stack.Pop();
72                      exit(element);
73                      exited++;
74                      var source = getSource(element);
75                      var target = getTarget(element);
76                      if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
    ↪ !visit(source))
77                      {
78                          return false;
79                      }

```

```

80         if ((isElement(target) || !canEnter(target)) && !visit(target))
81         {
82             return false;
83         }
84         element = target;
85     }
86 }
87 }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
92 {
93     var stack = new Stack<TLinkAddress>();
94     var element = sequence;
95     if (isElement(element))
96     {
97         return visit(element);
98     }
99     while (true)
100     {
101         if (isElement(element))
102         {
103             if (stack.Count == 0)
104             {
105                 return true;
106             }
107             element = stack.Pop();
108             var source = getSource(element);
109             var target = getTarget(element);
110             if (isElement(source) && !visit(source))
111             {
112                 return false;
113             }
114             if (isElement(target) && !visit(target))
115             {
116                 return false;
117             }
118             element = target;
119         }
120         else
121         {
122             stack.Push(element);
123             element = getSource(element);
124         }
125     }
126 }
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static bool WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
130 {
131     var stack = new Stack<TLinkAddress>();
132     var element = sequence;
133     if (isElement(element))
134     {
135         return visit(element);
136     }
137     while (true)
138     {
139         if (isElement(element))
140         {
141             if (stack.Count == 0)
142             {
143                 return true;
144             }
145             element = stack.Pop();
146             var source = getSource(element);
147             var target = getTarget(element);
148             if (isElement(target) && !visit(target))
149             {
150                 return false;
151             }
152             if (isElement(source) && !visit(source))
153             {
154                 return false;

```

```

155         }
156         element = source;
157     }
158     else
159     {
160         stack.Push(element);
161         element = getTarget(element);
162     }
163 }
164 }
165 }
166 }

```

./Universal/IUniLinksCRUD.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>
10     /// CRUD aliases for IUniLinks.
11     /// </remarks>
12     public interface IUniLinksCRUD<TLinkAddress>
13     {
14         TLinkAddress Read(int partType, TLinkAddress link);
15         TLinkAddress Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
16         TLinkAddress Create(IList<TLinkAddress> parts);
17         TLinkAddress Update(IList<TLinkAddress> before, IList<TLinkAddress> after);
18         void Delete(IList<TLinkAddress> parts);
19     }
20 }

```

./Universal/IUniLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
10     public partial interface IUniLinks<TLinkAddress>
11     {
12         IList<IList<IList<TLinkAddress>>> Trigger(IList<TLinkAddress> condition,
13             ⇨ IList<TLinkAddress> substitution);
14     }
15
16     /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
17     public partial interface IUniLinks<TLinkAddress>
18     {
19         /// <returns>
20         /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
21         ⇨ represents False (was stopped).
22         /// This is done to assure ability to push up stop signal through recursion stack.
23         /// </returns>
24         /// <remarks>
25         /// { 0, 0, 0 } => { itself, itself, itself } // create
26         /// { 1, any, any } => { itself, any, 3 } // update
27         /// { 3, any, any } => { 0, 0, 0 } // delete
28         /// </remarks>
29         TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>,
30             ⇨ TLinkAddress> matchHandler,
31             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>,
32             ⇨ IList<TLinkAddress>, TLinkAddress> substitutionHandler);
33
34         TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>,
35             ⇨ IList<TLinkAddress>, TLinkAddress> matchedHandler,
36             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>,
37             ⇨ TLinkAddress> substitutedHandler);
38     }
39
40     /// <remarks>Extended with small optimization.</remarks>
41     public partial interface IUniLinks<TLinkAddress>
42     {
43         /// <remarks>

```

```

38     /// Something simple should be simple and optimized.
39     /// </remarks>
40     TLinkAddress Count(IList<TLinkAddress> restrictions);
41 }
42 }

```

./Universal/IUniLinksGS.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>
10     /// Get/Set aliases for IUniLinks.
11     /// </remarks>
12     public interface IUniLinksGS<TLinkAddress>
13     {
14         TLinkAddress Get(int partType, TLinkAddress link);
15         TLinkAddress Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
16         TLinkAddress Set(IList<TLinkAddress> before, IList<TLinkAddress> after);
17     }
18 }

```

./Universal/IUniLinksIO.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>
10     /// In/Out aliases for IUniLinks.
11     /// TLinkAddress can be any number type of any size.
12     /// </remarks>
13     public interface IUniLinksIO<TLinkAddress>
14     {
15         /// <remarks>
16         /// default(TLinkAddress) means any link.
17         /// Single element pattern means just element (link).
18         /// Handler gets array of link contents.
19         /// * link[0] is index or identifier.
20         /// * link[1] is source or first.
21         /// * link[2] is target or second.
22         /// * link[3] is linker or third.
23         /// * link[n] is nth part/parent/element/value
24         /// of link (if variable length links used).
25         ///
26         /// Stops and returns false if handler return false.
27         ///
28         /// Acts as Each, Foreach, Select, Search, Match &amp; ...
29         ///
30         /// Handles all links in store if pattern/restrictions is not defined.
31         /// </remarks>
32         bool Out(Func<IList<TLinkAddress>, bool> handler, IList<TLinkAddress> pattern);
33
34         /// <remarks>
35         /// default(TLinkAddress) means itself.
36         /// Equivalent to:
37         /// * creation if before == null
38         /// * deletion if after == null
39         /// * update if before != null &amp;&amp; after != null
40         /// * default(TLinkAddress) if before == null &amp;&amp; after == null
41         ///
42         /// Possible interpretation
43         /// * In(null, new[] { }) creates point (link that points to itself using minimum number
44         ↪ of parts).
45         /// * In(new[] { 4 }, null) deletes 4th link.
46         /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
47         ↪ 5th index.
48         /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with
49         ↪ 2 as source and 3 as target), 0 means it can be placed in any address.
50         /// ...
51         /// </remarks>

```



```

49         TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
50     }
51 }

```

./Universal/IUniLinksIOWithExtensions.cs

```

1  // ReSharper disable TypeParameterCanBeVariant
2  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
4  using System.Collections.Generic;
5
6  namespace Platform.Data.Universal
7  {
8      /// <remarks>Contains some optimizations of Out.</remarks>
9      public interface IUniLinksIOWithExtensions<TLinkAddress> : IUniLinksIO<TLinkAddress>
10     {
11         /// <remarks>
12         /// default(TLinkAddress) means nothing or null.
13         /// Single element pattern means just element (link).
14         /// OutPart(n, null) returns default(TLinkAddress).
15         /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
16         /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
17         /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
18         /// OutPart(3, pattern) ~ GetLinkAddresser(link) or GetLinkAddresser(Search(pattern))
19         /// OutPart(n, pattern) => For any variable length links, returns link or
20         ///     ↪ default(TLinkAddress).
21         ///
22         /// Outs(returns) inner contents of link, its part/parent/element/value.
23         /// </remarks>
24         TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);
25
26         /// <remarks>OutCount() returns total links in store as array.</remarks>
27         IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);
28
29         /// <remarks>OutCount() returns total amount of links in store.</remarks>
30         ulong OutCount(IList<TLinkAddress> pattern);
31     }
32 }

```

./Universal/IUniLinksRW.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>
10     /// Read/Write aliases for IUniLinks.
11     /// </remarks>
12     public interface IUniLinksRW<TLinkAddress>
13     {
14         TLinkAddress Read(int partType, TLinkAddress link);
15         bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
16         TLinkAddress Write(IList<TLinkAddress> before, IList<TLinkAddress> after);
17     }
18 }

```

Index

- ./Exceptions/ArgumentLinkDoesNotExistsException.cs, 1
- ./Exceptions/ArgumentLinkHasDependenciesException.cs, 1
- ./Exceptions/LinkWithSameValueAlreadyExistsException.cs, 1
- ./Exceptions/LinksLimitReachedException.cs, 1
- ./ILinks.cs, 1
- ./ILinksExtensions.cs, 3
- ./ISynchronizedLinks.cs, 5
- ./LinkAddress.cs, 5
- ./LinksConstants.cs, 7
- ./LinksConstantsExtensions.cs, 8
- ./Point.cs, 8
- ./Sequences/ISequenceAppender.cs, 11
- ./Sequences/ISequenceWalker.cs, 11
- ./Sequences/SequenceWalker.cs, 11
- ./Sequences/StopableSequenceWalker.cs, 12
- ./Universal/IUniLinks.cs, 15
- ./Universal/IUniLinksCRUD.cs, 15
- ./Universal/IUniLinksGS.cs, 16
- ./Universal/IUniLinksIO.cs, 16
- ./Universal/IUniLinksIOWithExtensions.cs, 17
- ./Universal/IUniLinksRW.cs, 17