LinksPlatform's Platform.Data Class Library

./Exceptions/ArgumentLinkDoesNotExistsException.cs

```
using System;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
    {
        public ArgumentLinkDoesNotExistsException(TLinkAddress link, string paramName) : base(FormatMessage(link, paramName), paramName) { }
        public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link)) { }
        private static string FormatMessage(TLinkAddress link, string paramName) => $"Связь [{link}] переданная в аргумент [{paramName}] не существует.";
        private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в качестве аргумента не существует.";
    }
}
```

./Exceptions/ArgumentLinkHasDependenciesException.cs

```
using System;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
    {
        public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) : base(FormatMessage(link, paramName), paramName) { }
        public ArgumentLinkHasDependenciesException(TLinkAddress link) : base(FormatMessage(link)) { }
        private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые препятствуют изменению её внутренней структуры.";
        private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной в качестве аргумента присутствуют зависимости, которые препятствуют изменению её внутренней структуры.";
    }
}
```

./Exceptions/LinksLimitReachedException.cs

```
using System;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class LinksLimitReachedException<TLinkAddress> : Exception
    {
        public static readonly string DefaultMessage = "Достигнут лимит количества связей в хранилище.";
        public LinksLimitReachedException(string message) : base(message) { }
        public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }
        public LinksLimitReachedException() : base(DefaultMessage) { }
        private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества связей в хранилище ({limit}).";
    }
}
```

./Exceptions/LinkWithSameValueAlreadyExistsException.cs

```
using System;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class LinkWithSameValueAlreadyExistsException : Exception
    {
        public static readonly string DefaultMessage = "Связь с таким же значением уже существует.";
        public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
        public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
    }
}
```

./ILinks.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    /// <summary>
    /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
    /// </summary>
    /// <remarks>
    /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит подходит как для дуплетов, так и для триплетов и т.п.
    /// Возможно этот интерфейс подходит даже для Sequences.
    /// </remarks>
    public interface ILinks<TLinkAddress, TConstants>
        where TConstants : LinksConstants<TLinkAddress>
    {
        #region Constants

        /// <summary>
        /// Возвращает набор констант, который необходим для эффективной коммуникации с методами этого интерфейса.
        /// Эти константы не меняются с момента создания точки доступа к хранилищу.
        /// </summary>
        TConstants Constants { get; }

        #endregion

        #region Read

        /// <summary>
        /// Подсчитывает и возвращает общее число связей находящихся в хранилище, соответствующих указанным ограничениям.
        /// </summary>
        /// <param name="restriction">Ограничения на содержимое связей.</param>
        /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным ограничениям.</returns>
        TLinkAddress Count(IList<TLinkAddress> restriction);

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном случае.</returns>
        TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress> restrictions);

        #endregion

        #region Write

        /// <summary>
        /// Создаёт связь.
        /// </summary>
        /// <returns>Индекс созданной связи.</returns>
        TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно принимать restrictions, возможно и возвращать связь нужно целиком.

        /// <summary>
        /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
        /// на связь с указанным новым содержимым.
        /// </summary>
        /// <param name="restrictions">
        /// Ограничения на содержимое связей.
        /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart]) и далее за ним будет следовать содержимое связи.
        /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
        /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс другой связи.
        /// </param>
        /// <param name="substitution"></param>
```

```
        /// <returns>Индекс обновлённой связи.</returns>
        TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
        // TODO: Возможно и возвращать связь нужно целиком.

        /// <summary>Удаляет связь с указанным индексом.</summary>
        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.

        #endregion
    }
}
```

./ILinksExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Setters;
using Platform.Data.Exceptions;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    public static class ILinksExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress, TConstants> links, params TLinkAddress[] restrictions)
            where TConstants : LinksConstants<TLinkAddress>
            => links.Count(restrictions);

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанным индексом в хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
            where TConstants : LinksConstants<TLinkAddress>
        {
            var constants = links.Constants;
            return constants.IsExternalReference(link) || (constants.IsInternalReference(link) && Comparer<TLinkAddress>.Default.Compare(links.Count(new LinkAddress<TLinkAddress>(link)), default) > 0);
        }

        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <remarks>
        /// TODO: May be move to EnsureExtensions or make it both there and here
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
            where TConstants : LinksConstants<TLinkAddress>
        {
            if (!links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress, TConstants> links, TLinkAddress link, string argumentName)
            where TConstants : LinksConstants<TLinkAddress>
        {
            if (!links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
            }
        }
```

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

```
        {
            if (links.Constants.IsExternalReference(link))
            {
                return true;
            }
            links.EnsureLinkExists(link);
            return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
        }

        /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом точкой частично (связью замкнутой на себе как минимум один раз).</summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой связи.</param>
        /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
        /// <remarks>
        /// Достаточно любой одной ссылки на себя.
        /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли ссылки на себя (на эту связь).
        /// </remarks>
        public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
            where TConstants : LinksConstants<TLinkAddress>
        {
            if (links.Constants.IsExternalReference(link))
            {
                return true;
            }
            links.EnsureLinkExists(link);
            return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
        }

        #endregion
    }
}
```

./ISynchronizedLinks.cs

```
using Platform.Threading.Synchronization;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    public interface ISynchronizedLinks<TLinkAddress, TLinks, TConstants> : ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
        where TLinks : ILinks<TLinkAddress, TConstants>
        where TConstants : LinksConstants<TLinkAddress>
    {
    }
}
```

./LinkAddress.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer = EqualityComparer<TLinkAddress>.Default;

        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index == 0)
                {
```

```
                return Index;
            }
            else
            {
                throw new IndexOutOfRangeException();
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        set => throw new NotSupportedException();
    }

    public int Count => 1;

    public bool IsReadOnly => true;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public LinkAddress(TLinkAddress index) => Index = index;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Add(TLinkAddress item) => throw new NotSupportedException();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Clear() => throw new NotSupportedException();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
    ↪  ? true : false;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public IEnumerator<TLinkAddress> GetEnumerator()
    {
        yield return Index;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
    ↪  0 : -1;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool Remove(TLinkAddress item) => throw new NotSupportedException();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void RemoveAt(int index) => throw new NotSupportedException();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    IEnumerator IEnumerable.GetEnumerator()
    {
        yield return Index;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public virtual bool Equals(LinkAddress<TLinkAddress> other) =>
    ↪  _equalityComparer.Equals(Index, other.Index);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
    ↪  linkAddress.Index;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
    ↪  new LinkAddress<TLinkAddress>(linkAddress);

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
    ↪  ? Equals(linkAddress) : false;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override int GetHashCode() => Index.GetHashCode();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override string ToString() => Index.ToString();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

./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)
           ↪  самой связи.</summary>
16         public int IndexPart { get; }
17
18         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
           ↪  часть-значение).</summary>
19         public int SourcePart { get; }
20
21         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
           ↪  (последняя часть-значение).</summary>
22         public int TargetPart { get; }
23
24         #endregion
25
26         #region Flow control
27
28         /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
29         /// <remarks>Используется в функции обработчике, который передаётся в функцию
           ↪  Each.</remarks>
30         public TLinkAddress Continue { get; }
31
32         /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
33         public TLinkAddress Skip { get; }
34
35         /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
36         /// <remarks>Используется в функции обработчике, который передаётся в функцию
           ↪  Each.</remarks>
37         public TLinkAddress Break { get; }
38
39         #endregion
40
41         #region Special symbols
42
43         /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
44         public TLinkAddress Null { get; }
45
46         /// <summary>Возвращает значение, обозначающее любую связь.</summary>
47         /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
           ↪  создавать все варианты последовательностей в функции Create.</remarks>
48         public TLinkAddress Any { get; }
49
50         /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
51         public TLinkAddress Itself { get; }
52
53         #endregion
54
55         #region References
56
57         /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
           ↪  ссылок).</summary>
58         public Range<TLinkAddress> InternalReferencesRange { get; }
59
60         /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
           ↪  ссылок).</summary>
61         public Range<TLinkAddress>? ExternalReferencesRange { get; }
62
```

```
63          #endregion
64
65          public LinksConstants(int targetPart, Range<TLinkAddress>
            ↪  possibleInternalReferencesRange, Range<TLinkAddress>?
            ↪  possibleExternalReferencesRange)
66          {
67              IndexPart = 0;
68              SourcePart = 1;
69              TargetPart = targetPart;
70              Null = Integer<TLinkAddress>.Zero;
71              Break = Integer<TLinkAddress>.Zero;
72              var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
73              Continue = currentInternalReferenceIndex;
74              Decrement(ref currentInternalReferenceIndex);
75              Skip = currentInternalReferenceIndex;
76              Decrement(ref currentInternalReferenceIndex);
77              Any = currentInternalReferenceIndex;
78              Decrement(ref currentInternalReferenceIndex);
79              Itself = currentInternalReferenceIndex;
80              Decrement(ref currentInternalReferenceIndex);
81              InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
                ↪  currentInternalReferenceIndex);
82              ExternalReferencesRange = possibleExternalReferencesRange;
83          }
84
85          public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
            ↪  this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
            ↪  GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) {}
86
87          public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
            ↪  Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
            ↪  possibleInternalReferencesRange, possibleExternalReferencesRange) { }
88
89          public LinksConstants(bool enableExternalReferencesSupport) :
            ↪  this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
            ↪  GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
90
91          public LinksConstants(int targetPart, Range<TLinkAddress>
            ↪  possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
            ↪  null) { }
92
93          public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
            ↪  this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
94
95          public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
            ↪  false) { }
96
97          public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
            ↪  enableExternalReferencesSupport)
98          {
99              if (enableExternalReferencesSupport)
100             {
101                 return (Integer<TLinkAddress>.One,
                    ↪  (Integer<TLinkAddress>)GetHalfOfNumberValuesRange());
102             }
103             else
104             {
105                 return (Integer<TLinkAddress>.One, NumericType<TLinkAddress>.MaxValue);
106             }
107         }
108
109         public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
            ↪  enableExternalReferencesSupport)
110         {
111             if (enableExternalReferencesSupport)
112             {
113                 return ((Integer<TLinkAddress>)(GetHalfOfNumberValuesRange() + 1UL),
                    ↪  NumericType<TLinkAddress>.MaxValue);
114             }
115             else
116             {
117                 return null;
118             }
119         }
120
121         private static ulong GetHalfOfNumberValuesRange() =>
            ↪  ((ulong)(Integer<TLinkAddress>)NumericType<TLinkAddress>.MaxValue) / 2;
122
```

```
123         private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
            ↪ currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
124     }
125 }
```

./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>
           ↪ linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
           ↪ || linksConstants.IsExternalReference(address);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
           ↪ linksConstants, TLinkAddress address) =>
           ↪ linksConstants.InternalReferencesRange.ContainsValue(address);
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
           ↪ linksConstants, TLinkAddress address) =>
           ↪ linksConstants.ExternalReferencesRange?.ContainsValue(address) ?? false;
17     }
18 }
```

./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 =
           ↪ EqualityComparer<TLinkAddress>.Default;
16
17         public TLinkAddress Index
18         {
19             [MethodImpl(MethodImplOptions.AggressiveInlining)]
20             get;
21         }
22
23         public int Size
24         {
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             get;
27         }
28
29         public TLinkAddress this[int index]
30         {
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             get
33             {
34                 if (index < Size)
35                 {
36                     return Index;
37                 }
38                 else
39                 {
40                     throw new IndexOutOfRangeException();
41                 }
42             }
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             set => throw new NotSupportedException();
45         }
46
47         public int Count => int.MaxValue;
```

```
48
49          public bool IsReadOnly => true;
50
51          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52          public Point(TLinkAddress index, int size)
53          {
54              Index = index;
55              Size = size;
56          }
57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public void Add(TLinkAddress item) => throw new NotSupportedException();
60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public void Clear() => throw new NotSupportedException();
63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
            ↪  ? true : false;
66
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public IEnumerator<TLinkAddress> GetEnumerator()
72          {
73              for (int i = 0; i < Size; i++)
74              {
75                  yield return Index;
76              }
77          }
78
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
            ↪  0 : -1;
81
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
84
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          public bool Remove(TLinkAddress item) => throw new NotSupportedException();
87
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) =>
            ↪  _equalityComparer.Equals(Index, other.Index);
102
103         [MethodImpl(MethodImplOptions.AggressiveInlining)]
104         public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
            ↪  linkAddress.Index;
105
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
            ↪  Equals(linkAddress) : false;
108
109         [MethodImpl(MethodImplOptions.AggressiveInlining)]
110         public override int GetHashCode() => Index.GetHashCode();
111
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         public override string ToString() => Index.ToString();
114
115         [MethodImpl(MethodImplOptions.AggressiveInlining)]
116         public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
            ↪  left.Equals(right);
117
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
            ↪  !(left == right);
```

```
120
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         public static bool IsFullPoint(params TLinkAddress[] link) =>
            ↪ IsFullPoint((IList<TLinkAddress>)link);
123
124         [MethodImpl(MethodImplOptions.AggressiveInlining)]
125         public static bool IsFullPoint(IList<TLinkAddress> link)
126         {
127             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
128             Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
                ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
129             return IsFullPointUnchecked(link);
130         }
131
132         [MethodImpl(MethodImplOptions.AggressiveInlining)]
133         public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
134         {
135             var result = true;
136             for (var i = 1; result && i < link.Count; i++)
137             {
138                 result = _equalityComparer.Equals(link[0], link[i]);
139             }
140             return result;
141         }
142
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static bool IsPartialPoint(params TLinkAddress[] link) =>
            ↪ IsPartialPoint((IList<TLinkAddress>)link);
145
146         [MethodImpl(MethodImplOptions.AggressiveInlining)]
147         public static bool IsPartialPoint(IList<TLinkAddress> link)
148         {
149             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
150             Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
                ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
151             return IsPartialPointUnchecked(link);
152         }
153
154         [MethodImpl(MethodImplOptions.AggressiveInlining)]
155         public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
156         {
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
       ↪  себя),
12     /// так как стэк 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 зацикливания.
16     /// Альтернативой защиты от закливания 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 SequenceWalker
22     {
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
           ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
           ↪  Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
25         {
26             var stack = new Stack<TLinkAddress>();
27             var element = sequence;
28             if (isElement(element))
29             {
30                 visit(element);
31             }
32             else
33             {
34                 while (true)
35                 {
36                     if (isElement(element))
37                     {
38                         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
```

```
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        if (isElement(target))
                        {
                            visit(target);
                        }
                        if (isElement(source))
                        {
                            visit(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        element = getTarget(element);
                    }
                }
            }
        }
    }
```

./Sequences/StopableSequenceWalker.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Sequences
{
    /// <remarks>
    /// Реализованный внутри алгоритм наглядно показывает,
    /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой себя),
    /// так как стэк можно использовать намного эффективнее при ручном управлении.
    ///
    /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
    /// Решить встраивать ли защиту от зацикливания.
    /// Альтернативой защиты от закливания может быть заранее известное ограничение на погружение вглубь.
    /// А так же качественное распознавание прохода по циклическому графу.
    /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер стека.
    /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
    /// </remarks>
    public static class StopableSequenceWalker
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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)
        {
            var exited = 0;
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
```

```
                    }
                    element = stack.Pop();
                    exit(element);
                    exited++;
                    var source = getSource(element);
                    var target = getTarget(element);
                    if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                    ↪  !visit(source))
                    {
                        return false;
                    }
                    if ((isElement(target) || !canEnter(target)) && !visit(target))
                    {
                        return false;
                    }
                    element = target;
                }
                else
                {
                    if (canEnter(element))
                    {
                        enter(element);
                        exited = 0;
                        stack.Push(element);
                        element = getSource(element);
                    }
                    else
                    {
                        if (stack.Count == 0)
                        {
                            return true;
                        }
                        element = stack.Pop();
                        exit(element);
                        exited++;
                        var source = getSource(element);
                        var target = getTarget(element);
                        if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                        ↪  !visit(source))
                        {
                            return false;
                        }
                        if ((isElement(target) || !canEnter(target)) && !visit(target))
                        {
                            return false;
                        }
                        element = target;
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
        {
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    var source = getSource(element);
                    var target = getTarget(element);
                    if (isElement(source) && !visit(source))
                    {
                        return false;
                    }
                    if (isElement(target) && !visit(target))
```

```
                    {
                        return false;
                    }
                    element = target;
                }
                else
                {
                    stack.Push(element);
                    element = getSource(element);
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
        {
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    var source = getSource(element);
                    var target = getTarget(element);
                    if (isElement(target) && !visit(target))
                    {
                        return false;
                    }
                    if (isElement(source) && !visit(source))
                    {
                        return false;
                    }
                    element = source;
                }
                else
                {
                    stack.Push(element);
                    element = getTarget(element);
                }
            }
        }
    }
}
```

./Universal/IUniLinksCRUD.cs

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

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// CRUD aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksCRUD<TLinkAddress>
    {
        TLinkAddress Read(int partType, TLinkAddress link);
        TLinkAddress Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Create(IList<TLinkAddress> parts);
        TLinkAddress Update(IList<TLinkAddress> before, IList<TLinkAddress> after);
        void Delete(IList<TLinkAddress> parts);
    }
}
```

./Universal/IUniLinks.cs

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

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        IList<IList<IList<TLinkAddress>>> Trigger(IList<TLinkAddress> condition,
        ↪  IList<TLinkAddress> substitution);
    }

    /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        /// <returns>
        /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
        ↪  represents False (was stopped).
        /// This is done to assure ability to push up stop signal through recursion stack.
        /// </returns>
        /// <remarks>
        /// { 0, 0, 0 } => { itself, itself, itself } // create
        /// { 1, any, any } => { itself, any, 3 } // update
        /// { 3, any, any } => { 0, 0, 0 } // delete
        /// </remarks>
        TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>,
        ↪  TLinkAddress> matchHandler,
                             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>,
                             ↪  IList<TLinkAddress>, TLinkAddress> substitutionHandler);

        TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>,
        ↪  IList<TLinkAddress>, TLinkAddress> matchedHandler,
              IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>,
              ↪  TLinkAddress> substitutedHandler);
    }

    /// <remarks>Extended with small optimization.</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        /// <remarks>
        /// Something simple should be simple and optimized.
        /// </remarks>
        TLinkAddress Count(IList<TLinkAddress> restrictions);
    }
}
```

./Universal/IUniLinksGS.cs

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

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// Get/Set aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksGS<TLinkAddress>
    {
        TLinkAddress Get(int partType, TLinkAddress link);
        TLinkAddress Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Set(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

./Universal/IUniLinksIO.cs

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

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
```

```
    /// <remarks>
    /// In/Out aliases for IUniLinks.
    /// TLinkAddress can be any number type of any size.
    /// </remarks>
    public interface IUniLinksIO<TLinkAddress>
    {
        /// <remarks>
        /// default(TLinkAddress) means any link.
        /// Single element pattern means just element (link).
        /// Handler gets array of link contents.
        /// * link[0] is index or identifier.
        /// * link[1] is source or first.
        /// * link[2] is target or second.
        /// * link[3] is linker or third.
        /// * link[n] is nth part/parent/element/value
        /// of link (if variable length links used).
        ///
        /// Stops and returns false if handler return false.
        ///
        /// Acts as Each, Foreach, Select, Search, Match &amp;amp; ...
        ///
        /// Handles all links in store if pattern/restrictions is not defined.
        /// </remarks>
        bool Out(Func<IList<TLinkAddress>, bool> handler, IList<TLinkAddress> pattern);

        /// <remarks>
        /// default(TLinkAddress) means itself.
        /// Equivalent to:
        /// * creation if before == null
        /// * deletion if after == null
        /// * update if before != null &amp;amp;&amp;amp; after != null
        /// * default(TLinkAddress) if before == null &amp;amp;&amp;amp; after == null
        ///
        /// Possible interpretation
        /// * In(null, new[] { }) creates point (link that points to itself using minimum number of parts).
        /// * In(new[] { 4 }, null) deletes 4th link.
        /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to 5th index.
        /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with 2 as source and 3 as target), 0 means it can be placed in any address.
        /// ...
        /// </remarks>
        TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

./Universal/IUniLinksIOWithExtensions.cs

```
// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System.Collections.Generic;

namespace Platform.Data.Universal
{
    /// <remarks>Contains some optimizations of Out.</remarks>
    public interface IUniLinksIOWithExtensions<TLinkAddress> : IUniLinksIO<TLinkAddress>
    {
        /// <remarks>
        /// default(TLinkAddress) means nothing or null.
        /// Single element pattern means just element (link).
        /// OutPart(n, null) returns default(TLinkAddress).
        /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
        /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
        /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
        /// OutPart(3, pattern) ~ GeTLinkAddresser(link) or GeTLinkAddresser(Search(pattern))
        /// OutPart(n, pattern) => For any variable length links, returns link or default(TLinkAddress).
        ///
        /// Outs(returns) inner contents of link, its part/parent/element/value.
        /// </remarks>
        TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);

        /// <remarks>OutCount() returns total links in store as array.</remarks>
        IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);

        /// <remarks>OutCount() returns total amount of links in store.</remarks>
        ulong OutCount(IList<TLinkAddress> pattern);
```

```
    }
}
```

./Universal/IUniLinksRW.cs

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

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// Read/Write aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksRW<TLinkAddress>
    {
        TLinkAddress Read(int partType, TLinkAddress link);
        bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Write(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

Index