LinksPlatform's Platform.Data Class Library

./Platform.Data/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.";
    }
}
```

./Platform.Data/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.";
    }
}
```

./Platform.Data/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}).";
    }
}
```

./Platform.Data/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) { }
    }
}
```

./Platform.Data/Hybrid.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data
{
    public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
    {
        public static readonly ulong HalfOfNumberValuesRange =
        ↪  ((ulong)(Integer<TLinkAddress>)NumericType<TLinkAddress>.MaxValue) / 2;
        public static readonly TLinkAddress ExternalZero =
        ↪  (Integer<TLinkAddress>)(HalfOfNumberValuesRange + 1UL);

        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly Func<object, TLinkAddress> _negateAndConvert =
        ↪  CompileNegateAndConvertDelegate();
        private static readonly Func<object, TLinkAddress> _unboxAbsAndConvert =
        ↪  CompileUnboxAbsAndConvertDelegate();
        private static readonly Func<object, TLinkAddress> _unboxAbsNegateAndConvert =
        ↪  CompileUnboxAbsNegateAndConvertDelegate();

        public readonly TLinkAddress Value;

        public bool IsNothing
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || Convert.ToInt64(Value) == 0;
        }

        public bool IsInternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Convert.ToInt64(To.Signed(Value)) > 0;
        }

        public bool IsExternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) ||
            ↪  Convert.ToInt64(To.Signed(Value)) < 0;
        }

        public long AbsoluteValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
            ↪  Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value)
        {
            Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
            Value = value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value, bool isExternal)
        {
            if (_equalityComparer.Equals(value, default) && isExternal)
            {
                Value = ExternalZero;
            }
            else
            {
                if (isExternal)
                {
                    Value = _negateAndConvert(value);
                }
                else
```

```
72                 {
73                     Value = value;
74                 }
75             }
76         }
77 
78         [MethodImpl(MethodImplOptions.AggressiveInlining)]
79         public Hybrid(object value) => Value =
           ↪  To.UnsignedAs<TLinkAddress>(Convert.ChangeType(value,
           ↪  NumericType<TLinkAddress>.SignedVersion));
80 
81         [MethodImpl(MethodImplOptions.AggressiveInlining)]
82         public Hybrid(object value, bool isExternal)
83         {
84             if (IsDefault(value) && isExternal)
85             {
86                 Value = ExternalZero;
87             }
88             else
89             {
90                 if (isExternal)
91                 {
92                     Value = _unboxAbsNegateAndConvert(value);
93                 }
94                 else
95                 {
96                     Value = _unboxAbsAndConvert(value);
97                 }
98             }
99         }
100 
101        [MethodImpl(MethodImplOptions.AggressiveInlining)]
102        public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
103 
104        [MethodImpl(MethodImplOptions.AggressiveInlining)]
105        public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
106 
107        [MethodImpl(MethodImplOptions.AggressiveInlining)]
108        public static explicit operator Hybrid<TLinkAddress>(long integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
109 
110        [MethodImpl(MethodImplOptions.AggressiveInlining)]
111        public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
112 
113        [MethodImpl(MethodImplOptions.AggressiveInlining)]
114        public static explicit operator Hybrid<TLinkAddress>(int integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
115 
116        [MethodImpl(MethodImplOptions.AggressiveInlining)]
117        public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
118 
119        [MethodImpl(MethodImplOptions.AggressiveInlining)]
120        public static explicit operator Hybrid<TLinkAddress>(short integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
121 
122        [MethodImpl(MethodImplOptions.AggressiveInlining)]
123        public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
124 
125        [MethodImpl(MethodImplOptions.AggressiveInlining)]
126        public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
           ↪  Hybrid<TLinkAddress>(integer);
127 
128        [MethodImpl(MethodImplOptions.AggressiveInlining)]
129        public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
           ↪  hybrid.Value;
130 
131        [MethodImpl(MethodImplOptions.AggressiveInlining)]
132        public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
           ↪  Convert.ToUInt64(hybrid.Value);
133 
134        [MethodImpl(MethodImplOptions.AggressiveInlining)]
135        public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
           ↪  hybrid.AbsoluteValue;
```

```
136
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
            ↪ Convert.ToUInt32(hybrid.Value);
139
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
            ↪ (int)hybrid.AbsoluteValue;
142
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
            ↪ Convert.ToUInt16(hybrid.Value);
145
146         [MethodImpl(MethodImplOptions.AggressiveInlining)]
147         public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
            ↪ (short)hybrid.AbsoluteValue;
148
149         [MethodImpl(MethodImplOptions.AggressiveInlining)]
150         public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
            ↪ Convert.ToByte(hybrid.Value);
151
152         [MethodImpl(MethodImplOptions.AggressiveInlining)]
153         public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
            ↪ (sbyte)hybrid.AbsoluteValue;
154
155         [MethodImpl(MethodImplOptions.AggressiveInlining)]
156         public override string ToString() => IsNothing ? default(TLinkAddress) == null ?
            ↪ "Nothing" : default(TLinkAddress).ToString() : IsExternal ? $"<{AbsoluteValue}>" :
            ↪ Value.ToString();
157
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
            ↪ other.Value);
160
161         [MethodImpl(MethodImplOptions.AggressiveInlining)]
162         public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
            ↪ Equals(hybrid) : false;
163
164         [MethodImpl(MethodImplOptions.AggressiveInlining)]
165         public override int GetHashCode() => Value.GetHashCode();
166
167         [MethodImpl(MethodImplOptions.AggressiveInlining)]
168         public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪ left.Equals(right);
169
170         [MethodImpl(MethodImplOptions.AggressiveInlining)]
171         public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪ !(left == right);
172
173         private static bool IsDefault(object value)
174         {
175             var type = value.GetType();
176             if (type.IsValueType)
177             {
178                 return value.Equals(Activator.CreateInstance(type));
179             }
180             return value == null;
181         }
182
183         private static Func<object, TLinkAddress> CompileUnboxAbsNegateAndConvertDelegate()
184         {
185             return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
186             {
187                 Ensure.Always.IsUnsignedInteger<TLinkAddress>();
188                 emiter.LoadArgument(0);
189                 var signedVersion = NumericType<TLinkAddress>.SignedVersion;
190                 var signedVersionField =
                    ↪ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
                    ↪ BindingFlags.Static | BindingFlags.Public);
191                 //emiter.LoadField(signedVersionField);
192                 emiter.Emit(OpCodes.Ldsfld, signedVersionField);
193                 var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                    ↪ Types<object, Type>.Array);
194                 emiter.Call(changeTypeMethod);
195                 emiter.UnboxValue(signedVersion);
196                 var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                    ↪ signedVersion });
```

```
                emiter.Call(absMethod);
                var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                ↪  ").MakeGenericMethod(signedVersion);
                emiter.Call(negateMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
        }

        private static Func<object, TLinkAddress> CompileUnboxAbsAndConvertDelegate()
        {
            return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<TLinkAddress>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<TLinkAddress>.SignedVersion;
                var signedVersionField =
                ↪  typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
                ↪  BindingFlags.Static | BindingFlags.Public);
                //emiter.LoadField(signedVersionField);
                emiter.Emit(OpCodes.Ldsfld, signedVersionField);
                var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                ↪  Types<object, Type>.Array);
                emiter.Call(changeTypeMethod);
                emiter.UnboxValue(signedVersion);
                var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                ↪  signedVersion });
                emiter.Call(absMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
        }

        // TODO: Use directed negation instead provided by the next version of
        ↪  Platform.Numbers.Math.Negate with no conversions required
        private static Func<object, TLinkAddress> CompileNegateAndConvertDelegate()
        {
            return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<TLinkAddress>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<TLinkAddress>.SignedVersion;
                var signedMethod = typeof(To).GetTypeInfo().GetMethod("Signed", new[] {
                ↪  typeof(TLinkAddress) });
                emiter.Call(signedMethod);
                var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                ↪  ").MakeGenericMethod(signedVersion);
                emiter.Call(negateMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
        }
    }
}
```

./Platform.Data/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>
```

```
16          where TConstants : LinksConstants<TLinkAddress>
17      {
18          #region Constants
19
20          /// <summary>
21          /// Возвращает набор констант, который необходим для эффективной коммуникации с методами
                ↪ этого интерфейса.
22          /// Эти константы не меняются с момента создания точки доступа к хранилищу.
23          /// </summary>
24          TConstants Constants { get; }
25
26          #endregion
27
28          #region Read
29
30          /// <summary>
31          /// Подсчитывает и возвращает общее число связей находящихся в хранилище,
                ↪ соответствующих указанным ограничениям.
32          /// </summary>
33          /// <param name="restriction">Ограничения на содержимое связей.</param>
34          /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным
                ↪ ограничениям.</returns>
35          TLinkAddress Count(IList<TLinkAddress> restriction);
36
37          /// <summary>
38          /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
                ↪ (handler) для каждой подходящей связи.
39          /// </summary>
40          /// <param name="handler">Обработчик каждой подходящей связи.</param>
41          /// <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>
42          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
                ↪ случае.</returns>
43          TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress>
                ↪ restrictions);
44
45          #endregion
46
47          #region Write
48
49          /// <summary>
50          /// Создаёт связь.
51          /// </summary>
52          /// <returns>Индекс созданной связи.</returns>
53          TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно
                ↪ принимать restrictions, возможно и возвращать связь нужно целиком.
54
55          /// <summary>
56          /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
57          /// на связь с указанным новым содержимым.
58          /// </summary>
59          /// <param name="restrictions">
60          /// Ограничения на содержимое связей.
61          /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
                ↪ и далее за ним будет следовать содержимое связи.
62          /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
                ↪ ссылку на пустоту,
63          /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
                ↪ другой связи.
64          /// </param>
65          /// <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  }
```

./Platform.Data/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.IsInternalReference(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         /// <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>
118         {
119             if (links.Constants.IsExternalReference(link))
120             {
121                 return true;
122             }
123             links.EnsureLinkExists(link);
124             return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
125         }
126
127         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪ точкой частично (связью замкнутой на себе как минимум один раз).</summary>
128         /// <param name="links">Хранилище связей.</param>
129         /// <param name="link">Индекс проверяемой связи.</param>
130         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
131         /// <remarks>
132         /// Достаточно любой одной ссылки на себя.
```

```
133         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
            ↪ ссылки на себя (на эту связь).
134         /// </remarks>
135         public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, TLinkAddress link)
136             where TConstants : LinksConstants<TLinkAddress>
137         {
138             if (links.Constants.IsExternalReference(link))
139             {
140                 return true;
141             }
142             links.EnsureLinkExists(link);
143             return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
144         }
145
146         #endregion
147     }
148 }
```

./Platform.Data/ISynchronizedLinks.cs

```
1  using Platform.Threading.Synchronization;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data
6  {
7      public interface ISynchronizedLinks<TLinkAddress, TLinks, TConstants> :
       ↪ ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
8          where TLinks : ILinks<TLinkAddress, TConstants>
9          where TConstants : LinksConstants<TLinkAddress>
10     {
11     }
12 }
```

./Platform.Data/LinkAddress.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data
9  {
10     public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
       ↪ IList<TLinkAddress>
11     {
12         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
           ↪ EqualityComparer<TLinkAddress>.Default;
13
14         public TLinkAddress Index
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19
20         public TLinkAddress this[int index]
21         {
22             [MethodImpl(MethodImplOptions.AggressiveInlining)]
23             get
24             {
25                 if (index == 0)
26                 {
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)
            ↪  ? true : false;
53
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          public IEnumerator<TLinkAddress> GetEnumerator()
59          {
60              yield return Index;
61          }
62
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
            ↪  0 : -1;
65
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public bool Remove(TLinkAddress item) => throw new NotSupportedException();
71
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public void RemoveAt(int index) => throw new NotSupportedException();
74
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          IEnumerator IEnumerable.GetEnumerator()
77          {
78              yield return Index;
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          public virtual bool Equals(LinkAddress<TLinkAddress> other) =>
            ↪  _equalityComparer.Equals(Index, other.Index);
83
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
            ↪  linkAddress.Index;
86
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
            ↪  new LinkAddress<TLinkAddress>(linkAddress);
89
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
            ↪  ? Equals(linkAddress) : false;
92
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          public override int GetHashCode() => Index.GetHashCode();
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public override string ToString() => Index.ToString();
98
99          [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 }
```

./Platform.Data/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
```

```
{
    public class LinksConstants<TLinkAddress>
    {
        public static readonly int DefaultTargetPart = 2;

        #region Link parts

        /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор) самой связи.</summary>
        public int IndexPart { get; }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая часть-значение).</summary>
        public int SourcePart { get; }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец (последняя часть-значение).</summary>
        public int TargetPart { get; }

        #endregion

        #region Flow control

        /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию Each.</remarks>
        public TLinkAddress Continue { get; }

        /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
        public TLinkAddress Skip { get; }

        /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию Each.</remarks>
        public TLinkAddress Break { get; }

        #endregion

        #region Special symbols

        /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
        public TLinkAddress Null { get; }

        /// <summary>Возвращает значение, обозначающее любую связь.</summary>
        /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет создавать все варианты последовательностей в функции Create.</remarks>
        public TLinkAddress Any { get; }

        /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
        public TLinkAddress Itself { get; }

        #endregion

        #region References

        /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних ссылок).</summary>
        public Range<TLinkAddress> InternalReferencesRange { get; }

        /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних ссылок).</summary>
        public Range<TLinkAddress>? ExternalReferencesRange { get; }

        #endregion

        public LinksConstants(int targetPart, Range<TLinkAddress> possibleInternalReferencesRange, Range<TLinkAddress>? possibleExternalReferencesRange)
        {
            IndexPart = 0;
            SourcePart = 1;
            TargetPart = targetPart;
            Null = Integer<TLinkAddress>.Zero;
            Break = Integer<TLinkAddress>.Zero;
            var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
            Continue = currentInternalReferenceIndex;
            Decrement(ref currentInternalReferenceIndex);
            Skip = currentInternalReferenceIndex;
            Decrement(ref currentInternalReferenceIndex);
            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>)Hybrid<TLinkAddress>.HalfOfNumberValuesRange);
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>)(Hybrid<TLinkAddress>.HalfOfNumberValuesRange +
                    ↪  1UL), NumericType<TLinkAddress>.MaxValue);
114             }
115             else
116             {
117                 return null;
118             }
119         }
120
121         private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
            ↪  currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
122     }
123 }
```

./Platform.Data/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)]
```

```
        public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪  linksConstants, TLinkAddress address) =>
        ↪  linksConstants.InternalReferencesRange.ContainsValue(address);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪  linksConstants, TLinkAddress address) =>
        ↪  linksConstants.ExternalReferencesRange?.ContainsValue(address) ?? false;
    }
}
```

./Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs

```
using Platform.Interfaces;

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

namespace Platform.Data.Numbers.Raw
{
    public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
    {
        public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
    }
}
```

./Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs

```
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Numbers.Raw
{
    public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
    {
        public TLink Convert(TLink source) => (Integer<TLink>)new
        ↪  Hybrid<TLink>(source).AbsoluteValue;
    }
}
```

./Platform.Data/Point.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Collections;

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

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

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

        public int Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index < Size)
                {
                    return Index;
                }
                else
                {
```

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

    public int Count => int.MaxValue;

    public bool IsReadOnly => true;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Point(TLinkAddress index, int size)
    {
        Index = index;
        Size = size;
    }

    [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()
    {
        for (int i = 0; i < Size; i++)
        {
            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()
    {
        for (int i = 0; i < Size; i++)
        {
            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(Point<TLinkAddress> linkAddress) =>
    ↪  linkAddress.Index;

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

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

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

```

```
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 }
```

./Platform.Data/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  }
```

./Platform.Data/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      {
```

```
        IEnumerable<IList<TLinkAddress>> Walk(TLinkAddress sequence);
    }
}
```

./Platform.Data/Sequences/SequenceWalker.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 SequenceWalker
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress, TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget, Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
        {
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                visit(element);
            }
            else
            {
                while (true)
                {
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        if (isElement(source))
                        {
                            visit(source);
                        }
                        if (isElement(target))
                        {
                            visit(target);
                        }
                        element = target;
                    }
                    else
                    {
                        stack.Push(element);
                        element = getSource(element);
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress, TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget, Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
        {
            var stack = new Stack<TLinkAddress>();
```

```
            var element = sequence;
            if (isElement(element))
            {
                visit(element);
            }
            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);
                    }
                }
            }
        }
    }
}
```

./Platform.Data/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);
                }
            }
        }
    }
}
```

./Platform.Data/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);
    }
}
```

./Platform.Data/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);
    }
}
```

./Platform.Data/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);
    }
}
```

./Platform.Data/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);
    }
}
```

./Platform.Data/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);
    }
}
```

./Platform.Data/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);
    }
}
```

./Platform.Data.Tests/LinksConstantsTests.cs

```
using Xunit;

namespace Platform.Data.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ExternalReferencesTest()
        {
            LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
            ↪  (long.MaxValue + 1UL, ulong.MaxValue));

            var minimum = new Hybrid<ulong>(0, isExternal: true);
            var maximum = new Hybrid<ulong>(long.MaxValue, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(constants.IsExternalReference(maximum));
        }
    }
}
```

Index