

## LinksPlatform's Platform.Data Class Library

### 1.1 ./csharp/Platform.Data/Exceptions/ArgumentLinkDoesNotExistsException.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Exceptions
7 {
8     public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ArgumentLinkDoesNotExistsException(TLinkAddress link, string argumentName) :
12             → base(FormatMessage(link, argumentName), argumentName) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link))
16             → { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public ArgumentLinkDoesNotExistsException(string message, Exception innerException) :
20             → base(message, innerException) { }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public ArgumentLinkDoesNotExistsException(string message) : base(message) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public ArgumentLinkDoesNotExistsException() { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         private static string FormatMessage(TLinkAddress link, string argumentName) => $"Связь
30             → [{link}] переданная в аргумент [{argumentName}] не существует.";
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
34             → качестве аргумента не существует.";
35     }
36 }
```

### 1.2 ./csharp/Platform.Data/Exceptions/ArgumentLinkHasDependenciesException.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Exceptions
7 {
8     public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) :
12             → base(FormatMessage(link, paramName), paramName) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public ArgumentLinkHasDependenciesException(TLinkAddress link) :
16             → base(FormatMessage(link)) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public ArgumentLinkHasDependenciesException(string message, Exception innerException) :
20             → base(message, innerException) { }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public ArgumentLinkHasDependenciesException(string message) : base(message) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public ArgumentLinkHasDependenciesException() { }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи
30             → [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые
31             → препятствуют изменению её внутренней структуры.";
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной
35             → в качестве аргумента присутствуют зависимости, которые препятствуют изменению её
36             → внутренней структуры.";
37     }
38 }
```

### 1.3 ./csharp/Platform.Data/Exceptions/LinkWithSameValueAlreadyExistsException.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Exceptions
7 {
8     public class LinkWithSameValueAlreadyExistsException : Exception
9     {
10         public static readonly string DefaultMessage = "Связь с таким же значением уже
11             ↳ существует.";
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinkWithSameValueAlreadyExistsException(string message, Exception innerException)
15             ↳ : base(message, innerException) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
```

### 1.4 ./csharp/Platform.Data/Exceptions/LinksLimitReachedException.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Exceptions
7 {
8     public class LinksLimitReachedException<TLinkAddress> : LinksLimitReachedExceptionBase
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinksLimitReachedException(string message, Exception innerException) :
15             ↳ base(message, innerException) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public LinksLimitReachedException(string message) : base(message) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinksLimitReachedException() : base(DefaultMessage) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества
25             ↳ связей в хранилище ({limit}).";
26     }
```

### 1.5 ./csharp/Platform.Data/Exceptions/LinksLimitReachedExceptionBase.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Exceptions
7 {
8     public abstract class LinksLimitReachedExceptionBase : Exception
9     {
10         public static readonly string DefaultMessage = "Достигнут лимит количества связей в
11             ↳ хранилище.";
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected LinksLimitReachedExceptionBase(string message, Exception innerException) :
15             ↳ base(message, innerException) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected LinksLimitReachedExceptionBase(string message) : base(message) { }
19     }
```

### 1.6 ./csharp/Platform.Data/Hybrid.cs

```
1 using System;
2 using System.Collections.Generic;
```

```

3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Reflection;
6 using Platform.Converters;
7 using Platform.Numbers;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data
12 {
13     public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
14     {
15         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
16             ↪ EqualityComparer<TLinkAddress>.Default;
17         private static readonly UncheckedSignExtendingConverter<TLinkAddress, long>
18             ↪ _addressToInt64Converter = UncheckedSignExtendingConverter<TLinkAddress,
19             ↪ long>.Default;
20         private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter
21             ↪ = UncheckedConverter<long, TLinkAddress>.Default;
22         private static readonly UncheckedConverter<TLinkAddress, ulong>
23             ↪ _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
24         private static readonly UncheckedConverter<ulong, TLinkAddress>
25             ↪ _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
26         private static readonly UncheckedConverter<object, long> _objectToInt64Converter =
27             ↪ UncheckedConverter<object, long>.Default;
28
29         public static readonly ulong HalfOfNumberValuesRange =
30             ↪ _addressToUInt64Converter.Convert(NumericType<TLinkAddress>.MaxValue) / 2;
31         public static readonly TLinkAddress ExternalZero =
32             ↪ _uInt64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);
33
34         public readonly TLinkAddress Value;
35
36         public bool IsNothing
37         {
38             [MethodImpl(MethodImplOptions.AggressiveInlining)]
39             get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
40         }
41
42         public bool IsInternal
43         {
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             get => SignedValue > 0;
46         }
47
48         public bool IsExternal
49         {
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
52         }
53
54         public long SignedValue
55         {
56             [MethodImpl(MethodImplOptions.AggressiveInlining)]
57             get => _addressToInt64Converter.Convert(Value);
58         }
59
60         public long AbsoluteValue
61         {
62             [MethodImpl(MethodImplOptions.AggressiveInlining)]
63             get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
64                 ↪ Platform.Numbers.Math.Abs(SignedValue);
65         }
66
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public Hybrid(TLinkAddress value)
69         {
70             Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
71             Value = value;
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         public Hybrid(TLinkAddress value, bool isExternal)
76         {
77             if (_equalityComparer.Equals(value, default) && isExternal)
78             {
79                 Value = ExternalZero;
80             }
81             else
82             {
83

```

```

73         if (isExternal)
74         {
75             Value = Math<TLinkAddress>.Negate(value);
76         }
77         else
78         {
79             Value = value;
80         }
81     }
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public Hybrid(object value) => Value =
86     ↪ _int64ToAddressConverter.Convert(_objectToInt64Converter.Convert(value));
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public Hybrid(object value, bool isExternal)
90 {
91     var signedValue = value == null ? 0 : _objectToInt64Converter.Convert(value);
92     if (signedValue == 0 && isExternal)
93     {
94         Value = ExternalZero;
95     }
96     else
97     {
98         var absoluteValue = System.Math.Abs(signedValue);
99         Value = isExternal ? _int64ToAddressConverter.Convert(-absoluteValue) :
100             ↪ _int64ToAddressConverter.Convert(absoluteValue);
101     }
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
106     ↪ Hybrid<TLinkAddress>(integer);
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
110     ↪ Hybrid<TLinkAddress>(integer);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public static explicit operator Hybrid<TLinkAddress>(long integer) => new
114     ↪ Hybrid<TLinkAddress>(integer);
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
118     ↪ Hybrid<TLinkAddress>(integer);
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 public static explicit operator Hybrid<TLinkAddress>(int integer) => new
122     ↪ Hybrid<TLinkAddress>(integer);
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
126     ↪ Hybrid<TLinkAddress>(integer);
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static explicit operator Hybrid<TLinkAddress>(short integer) => new
130     ↪ Hybrid<TLinkAddress>(integer);
131
132 [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
134     ↪ Hybrid<TLinkAddress>(integer);
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
138     ↪ Hybrid<TLinkAddress>(integer);
139
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
142     ↪ hybrid.Value;
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
146     ↪ CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);
147
148 [MethodImpl(MethodImplOptions.AggressiveInlining)]
149 public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
150     ↪ hybrid.AbsoluteValue;

```

```

137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
139         ↪ CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
140
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
143         ↪ (int)hybrid.AbsoluteValue;
144
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
147         ↪ CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
148
149     [MethodImpl(MethodImplOptions.AggressiveInlining)]
150     public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
151         ↪ (short)hybrid.AbsoluteValue;
152
153     [MethodImpl(MethodImplOptions.AggressiveInlining)]
154     public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
155         ↪ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);
156
157     [MethodImpl(MethodImplOptions.AggressiveInlining)]
158     public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
159         ↪ (sbyte)hybrid.AbsoluteValue;
160
161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
162     public override string ToString() => IsExternal ? $"{<AbsoluteValue>}" :
163         ↪ Value.ToString();
164
165     [MethodImpl(MethodImplOptions.AggressiveInlining)]
166     public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
167         ↪ other.Value);
168
169     [MethodImpl(MethodImplOptions.AggressiveInlining)]
170     public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
171         ↪ Equals(hybrid) : false;
172
173     [MethodImpl(MethodImplOptions.AggressiveInlining)]
174     public override int GetHashCode() => Value.GetHashCode();
175
176     [MethodImpl(MethodImplOptions.AggressiveInlining)]
177     public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
178         ↪ left.Equals(right);
179
180     [MethodImpl(MethodImplOptions.AggressiveInlining)]
181     public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
182         ↪ !(left == right);
183 }
184 }

```

## 1.7 ./csharp/Platform.Data/ILinks.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
8  {
9      /// <summary>
10      /// <para>Represents an interface for manipulating data in the Links (links storage)
11      ↪ format.</para>
12      /// <para>Представляет интерфейс для манипуляции с данными в формате Links (хранилища
13      ↪ связей).</para>
14      /// </summary>
15      /// <remarks>
16      /// <para>This interface is independent of the size of the content of the link, meaning it
17      ↪ is suitable for both doublets, triplets, and link sequences of any size.</para>
18      /// <para>Этот интерфейс не зависит от размера содержимого связи, а значит подходит как для
19      ↪ дуплетов, триплетов и последовательностей связей любого размера.</para>
20      /// </remarks>
21      public interface ILinks<TLinkAddress, TConstants>
22      where TConstants : LinksConstants<TLinkAddress>
23      {
24          #region Constants
25
26          /// <summary>

```

```

23    /// <para>Returns the set of constants that is necessary for effective communication
24    → with the methods of this interface.</para>
25    /// <para>Возвращает набор констант, который необходим для эффективной коммуникации с
26    → методами этого интерфейса.</para>
27    /// </summary>
28    /// <remarks>
29    /// <para>These constants are not changed since the creation of the links storage access
30    → point.</para>
31    /// <para>Эти константы не меняются с момента создания точки доступа к хранилищу
32    → связей.</para>
33    /// </remarks>
34    TConstants Constants
35    {
36        [MethodImpl(MethodImplOptions.AggressiveInlining)]
37        get;
38    }
39
40    #endregion
41
42    #region Read
43
44    /// <summary>
45    /// <para>Counts and returns the total number of links in the storage that meet the
46    → specified restrictions.</para>
47    /// <para>Подсчитывает и возвращает общее число связей находящихся в хранилище,
48    → соответствующих указанным ограничениям.</para>
49    /// </summary>
50    /// <param name="restriction"><para>Restrictions on the contents of
51    → links.</para><para>Ограничения на содержимое связей.</para></param>
52    /// <returns><para>The total number of links in the storage that meet the specified
53    → restrictions.</para><para>Общее число связей находящихся в хранилище,
54    → соответствующих указанным ограничениям.</para></returns>
55    [MethodImpl(MethodImplOptions.AggressiveInlining)]
56    TLinkAddress Count(IList<TLinkAddress> restriction);
57
58    /// <summary>
59    /// <para>Passes through all the links matching the pattern, invoking a handler for each
60    → matching link.</para>
61    /// <para>Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
62    → (handler) для каждой подходящей связи.</para>
63    /// </summary>
64    /// <param name="handler"><para>A handler for each matching link.</para><para>Обработчик
65    → для каждой подходящей связи.</para></param>
66    /// <param name="restrictions">
67    /// <para>Restrictions on the contents of links. Each constraint can have values:
68    → Constants.Null - the 0th link denoting a reference to the void, Any - the absence of
69    → a constraint, 1..∞ a specific link index.</para>
70    /// <para>Ограничения на содержимое связей. Каждое ограничение может иметь значения:
71    → Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Ану - отсутствие
72    → ограничения, 1..∞ конкретный индекс связи.</para>
73    /// </param>
74    /// <returns><para>Constants.Continue, if the pass through the links was not
75    → interrupted, and Constants.Break otherwise.</para><para>Constants.Continue, в случае
76    → если проход по связям не был прерван и Constants.Break в обратном
77    → случае.</para></returns>
78    [MethodImpl(MethodImplOptions.AggressiveInlining)]
79    TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress>
80    → restrictions);
81
82    #endregion
83
84    #region Write
85
86    /// <summary>
87    /// <para>Creates a link.</para>
88    /// <para>Создаёт связь.</para>
89    /// <param name="restrictions">
90    /// <para>Restrictions on the content of a link. This argument is optional, if the null
91    → passed as value that means no restrictions on the content of a link are set.</para>
92    /// <para>Ограничения на содержимое связи. Этот аргумент опционален, если null передан в
93    → качестве значения это означает, что никаких ограничений на содержимое связи не
94    → установлено.</para>
95    /// </param>
96    /// </summary>
97    /// <returns><para>Index of the created link.</para><para>Индекс созданной
98    → связи.</para></returns>
99    [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

76 TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возвращать связь
    ↳ возвращать нужно целиком.
77
78 /// <summary>
79 /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
80 /// на связь с указанным новым содержимым.
81 /// </summary>
82 /// <param name="restrictions">
83 /// Ограничения на содержимое связей.
84 /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
    ↳ и далее за ним будет следовать содержимое связи.
85 /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
    ↳ ссылку на пустоту,
86 /// Constants.Itself - требование установить ссылку на себя, 1.. $\infty$  конкретный индекс
    ↳ другой связи.
87 /// </param>
88 /// <param name="substitution"></param>
89 /// <returns>Индекс обновлённой связи.</returns>
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
    ↳ // TODO: Возможно и возвращать связь нужно целиком.
92
93 /// <summary>
94 /// <para>Deletes links that match the specified restrictions.</para>
95 /// <para>Удаляет связи соответствующие указанным ограничениям.</para>
96 /// <param name="restrictions">
97 /// <para>Restrictions on the content of a link. This argument is optional, if the null
    ↳ passed as value that means no restrictions on the content of a link are set.</para>
98 /// <para>Ограничения на содержимое связи. Этот аргумент опционален, если null передан в
    ↳ качестве значения это означает, что никаких ограничений на содержимое связи не
    ↳ установлено.</para>
99 /// </param>
100 /// </summary>
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 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.
103
104 #endregion
105 }
106 }

```

## 1.8 ./csharp/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
100  → связь.
101  /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
102  → точка и пара существовать одновременно?
103  /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
104  → сортировать по индексу в массиве связей?
105  /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
106  /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
107  → самой себя любого размера?
108  /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
109  → одной ссылки на себя (частичной точки).
110  /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
111  /// самостоятельный цикл через себя? Что если предоставить все варианты использования
112  → связей?
113  /// Что если разрешить и нули, а так же частичные варианты?
114  ///
115  /// Что если точка, это только в том случае когда link.Source == link && link.Target == link, т.е. дважды ссылка на себя.
116  /// А пара это тогда, когда link.Source == link.Target && link.Source != link, т.е. ссылка не на себя а во вне.
117  ///
118  /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
119  → промежуточную связь,
120  /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
121  /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
122  /// </remarks>
123  [MethodImpl(MethodImplOptions.AggressiveInlining)]
124  public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
125  → TConstants> links, TLinkAddress link)
126  where TConstants : LinksConstants<TLinkAddress>
127  {
128      if (links.Constants.IsExternalReference(link))
129      {
130          return true;
131      }
132      links.EnsureLinkExists(link);
133      return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
134  }
135
136  /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
137  → точкой частично (связью замкнутой на себе как минимум один раз).</summary>
138  /// <param name="links">Хранилище связей.</param>
139  /// <param name="link">Индекс проверяемой связи.</param>
140  /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
141  /// <remarks>
142  /// Достаточно любой одной ссылки на себя.
143  /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
144  → ссылки на себя (на эту связь).
145  /// </remarks>
146  [MethodImpl(MethodImplOptions.AggressiveInlining)]
147  public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
148  → TConstants> links, TLinkAddress link)
149  where TConstants : LinksConstants<TLinkAddress>
150  {
151      if (links.Constants.IsExternalReference(link))
152      {
153          return true;
154      }
155      links.EnsureLinkExists(link);
156      return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
157  }
158
159  #endregion
160  }

```

## 1.9 ./csharp/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> :
8      → ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
9      where TLinks : ILinks<TLinkAddress, TConstants>
10     where TConstants : LinksConstants<TLinkAddress>
11     {

```

```

11     }
12 }

```

## 1.10 ./csharp/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
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get => 1;
42         }
43
44         public bool IsReadOnly
45         {
46             [MethodImpl(MethodImplOptions.AggressiveInlining)]
47             get => true;
48         }
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public LinkAddress(TLinkAddress index) => Index = index;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public void Add(TLinkAddress item) => throw new NotSupportedException();
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public void Clear() => throw new NotSupportedException();
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
        ↳ ? true : false;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public IEnumerator<TLinkAddress> GetEnumerator()
67         {
68             yield return Index;
69         }
70
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
        ↳ 0 : -1;

```

```

73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     public bool Remove(TLinkAddress item) => throw new NotSupportedException();
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public void RemoveAt(int index) => throw new NotSupportedException();
81
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     IEnumerator IEnumerable.GetEnumerator()
84     {
85         yield return Index;
86     }
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
90         ↪ _equalityComparer.Equals(Index, other.Index);
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
94         ↪ linkAddress.Index;
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
98         ↪ new LinkAddress<TLinkAddress>(linkAddress);
99
100     [MethodImpl(MethodImplOptions.AggressiveInlining)]
101     public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
102         ↪ ? Equals(linkAddress) : false;
103
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public override int GetHashCode() => Index.GetHashCode();
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     public override string ToString() => Index.ToString();
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
112         ↪ right)
113     {
114         if (left == null && right == null)
115         {
116             return true;
117         }
118         if (left == null)
119         {
120             return false;
121         }
122         return left.Equals(right);
123     }
124
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
127         ↪ right) => !(left == right);
128 }

```

### 1.11 ./csharp/Platform.Data/LinksConstants.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Ranges;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data
10 {
11     public class LinksConstants<TLinkAddress> : LinksConstantsBase
12     {
13         private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
14         private static readonly UncheckedConverter<ulong, TLinkAddress>
15             ↪ _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
16
17         #region Link parts
18
19         /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
20         ↪ самой связи.</summary>

```

```

19 public int IndexPart
20 {
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     get;
23 }
24
25 /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
    ↳ часть-значение).</summary>
26 public int SourcePart
27 {
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     get;
30 }
31
32 /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
    ↳ (последняя часть-значение).</summary>
33 public int TargetPart
34 {
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     get;
37 }
38
39 #endregion
40
41 #region Flow control
42
43 /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
44 /// <remarks>Используется в функции обработчике, который передаётся в функцию
    ↳ Each.</remarks>
45 public TLinkAddress Continue
46 {
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     get;
49 }
50
51 /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
52 public TLinkAddress Skip
53 {
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     get;
56 }
57
58 /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
59 /// <remarks>Используется в функции обработчике, который передаётся в функцию
    ↳ Each.</remarks>
60 public TLinkAddress Break
61 {
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     get;
64 }
65
66 #endregion
67
68 #region Special symbols
69
70 /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
71 public TLinkAddress Null
72 {
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     get;
75 }
76
77 /// <summary>Возвращает значение, обозначающее любую связь.</summary>
78 /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
    ↳ создавать все варианты последовательностей в функции Create.</remarks>
79 public TLinkAddress Any
80 {
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     get;
83 }
84
85 /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
86 public TLinkAddress Itself
87 {
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     get;
90 }
91
92 #endregion
93

```

```

94 #region References
95
96 /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
    ↳ ссылок).</summary>
97 public Range<TLinkAddress> InternalReferencesRange
98 {
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100     get;
101 }
102
103 /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
    ↳ ссылок).</summary>
104 public Range<TLinkAddress>? ExternalReferencesRange
105 {
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     get;
108 }
109
110 #endregion
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public LinksConstants(int targetPart, Range<TLinkAddress>
    ↳ possibleInternalReferencesRange, Range<TLinkAddress>?
    ↳ possibleExternalReferencesRange)
114 {
115     IndexPart = 0;
116     SourcePart = 1;
117     TargetPart = targetPart;
118     Null = default;
119     Break = default;
120     var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
121     Continue = currentInternalReferenceIndex;
122     Skip = Arithmetic.Decrement(ref currentInternalReferenceIndex);
123     Any = Arithmetic.Decrement(ref currentInternalReferenceIndex);
124     Itself = Arithmetic.Decrement(ref currentInternalReferenceIndex);
125     Arithmetic.Decrement(ref currentInternalReferenceIndex);
126     InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
    ↳ currentInternalReferenceIndex);
127     ExternalReferencesRange = possibleExternalReferencesRange;
128 }
129
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
    ↳ this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
    ↳ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
    ↳ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
    ↳ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public LinksConstants(bool enableExternalReferencesSupport) :
    ↳ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
    ↳ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public LinksConstants(int targetPart, Range<TLinkAddress>
    ↳ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
    ↳ null) { }
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
    ↳ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
    ↳ false) { }
147
148 [MethodImpl(MethodImplOptions.AggressiveInlining)]
149 public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
    ↳ enableExternalReferencesSupport)
150 {
151     if (enableExternalReferencesSupport)
152     {
153         return (_one, _UInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
    ↳ rValuesRange));
154     }

```

```

155         else
156         {
157             return (_one, NumericType<TLinkAddress>.MaxValue);
158         }
159     }
160
161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
162     public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
    ↪ enableExternalReferencesSupport)
163     {
164         if (enableExternalReferencesSupport)
165         {
166             return (Hybrid<TLinkAddress>.ExternalZero, NumericType<TLinkAddress>.MaxValue);
167         }
168         else
169         {
170             return null;
171         }
172     }
173 }
174 }

```

#### 1.12 ./csharp/Platform.Data/LinksConstantsBase.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data
4  {
5      public abstract class LinksConstantsBase
6      {
7          public static readonly int DefaultTargetPart = 2;
8      }
9  }

```

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

```

#### 1.14 ./csharp/Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Numbers.Raw
7  {
8      public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
12     }
13 }

```

#### 1.15 ./csharp/Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3

```

```

4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Numbers.Raw
7  {
8      public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9      {
10         static private readonly UncheckedConverter<long, TLink> _converter =
11             ↳ UncheckedConverter<long, TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TLink Convert(TLink source) => _converter.Convert(new
15             ↳ Hybrid<TLink>(source).AbsoluteValue);
16     }
17 }

```

## 1.16 ./csharp/Platform.Data/Point.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Exceptions;
6  using Platform.Ranges;
7  using Platform.Collections;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data
12 {
13     public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
14     {
15         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
16             ↳ EqualityComparer<TLinkAddress>.Default;
17
18         public TLinkAddress Index
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public int Size
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public TLinkAddress this[int index]
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get
34             {
35                 if (index < Size)
36                 {
37                     return Index;
38                 }
39                 else
40                 {
41                     throw new IndexOutOfRangeException();
42                 }
43             }
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set => throw new NotSupportedException();
46         }
47
48         public int Count
49         {
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             get => Size;
52         }
53
54         public bool IsReadOnly
55         {
56             [MethodImpl(MethodImplOptions.AggressiveInlining)]
57             get => true;
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public Point(TLinkAddress index, int size)
62         {
63             Index = index;
64             Size = size;
65         }
66     }
67 }

```

```

65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public void Add(TLinkAddress item) => throw new NotSupportedException();
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public void Clear() => throw new NotSupportedException();
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
73     ↪ ? true : false;
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public IEnumerator<TLinkAddress> GetEnumerator()
80 {
81     for (int i = 0; i < Size; i++)
82     {
83         yield return Index;
84     }
85 }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
89     ↪ 0 : -1;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 public bool Remove(TLinkAddress item) => throw new NotSupportedException();
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 public void RemoveAt(int index) => throw new NotSupportedException();
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 IEnumerator IEnumerable.GetEnumerator()
102 {
103     for (int i = 0; i < Size; i++)
104     {
105         yield return Index;
106     }
107 }
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
111     ↪ _equalityComparer.Equals(Index, other.Index);
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
115     ↪ linkAddress.Index;
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
119     ↪ Equals(linkAddress) : false;
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public override int GetHashCode() => Index.GetHashCode();
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public override string ToString() => Index.ToString();
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
129 {
130     if (left == null && right == null)
131     {
132         return true;
133     }
134     if (left == null)
135     {
136         return false;
137     }
138     return left.Equals(right);
139 }
140
141 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

138     public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
139         ↪ !(left == right);
140
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public static bool IsFullPoint(params TLinkAddress[] link) =>
143         ↪ IsFullPoint((IList<TLinkAddress>)link);
144
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     public static bool IsFullPoint(IList<TLinkAddress> link)
147     {
148         Ensure.Always.ArgumentNotEmpty(link, nameof(link));
149         Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
150             ↪ determine link's pointness using only its identifier.");
151         return IsFullPointUnchecked(link);
152     }
153
154     [MethodImpl(MethodImplOptions.AggressiveInlining)]
155     public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
156     {
157         var result = true;
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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
166     public static bool IsPartialPoint(params TLinkAddress[] link) =>
167         ↪ IsPartialPoint((IList<TLinkAddress>)link);
168
169     [MethodImpl(MethodImplOptions.AggressiveInlining)]
170     public static bool IsPartialPoint(IList<TLinkAddress> link)
171     {
172         Ensure.Always.ArgumentNotEmpty(link, nameof(link));
173         Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
174             ↪ determine link's pointness using only its identifier.");
175         return IsPartialPointUnchecked(link);
176     }
177
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
180     {
181         var result = false;
182         for (var i = 1; !result && i < link.Count; i++)
183         {
184             result = _equalityComparer.Equals(link[0], link[i]);
185         }
186         return result;
187     }
188 }

```

### 1.17 ./csharp/Platform.Data/Sequences/ISequenceAppender.cs

```

1 using System.Runtime.CompilerServices;
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 ISequenceAppender<TLinkAddress>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant);
11     }
12 }

```

### 1.18 ./csharp/Platform.Data/Sequences/ISequenceWalker.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Sequences
7 {
8     public interface ISequenceWalker<TLinkAddress>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         IEnumerable<IList<TLinkAddress>> Walk(TLinkAddress sequence);
12     }
13 }

```

```
12     }
13 }
```

### 1.19 ./csharp/Platform.Data/Sequences/SequenceWalker.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Sequences
8 {
9     /// <remarks>
10    /// Реализованный внутри алгоритм наглядно показывает,
11    /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой
12    /// ↪ себя),
13    /// так как стек можно использовать намного эффективнее при ручном управлении.
14    ///
15    /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
16    /// Решить встраивать ли защиту от заикливания.
17    /// Альтернативой защиты от заикливания может быть заранее известное ограничение на
18    /// ↪ погружение вглубь.
19    /// А так же качественное распознавание прохода по циклическому графу.
20    /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер
21    /// ↪ стека.
22    /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
23    /// </remarks>
24    public static class SequenceWalker
25    {
26        [MethodImpl(MethodImplOptions.AggressiveInlining)]
27        public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
28        ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
29        ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
30        {
31            var stack = new Stack<TLinkAddress>();
32            var element = sequence;
33            if (isElement(element))
34            {
35                visit(element);
36            }
37            else
38            {
39                while (true)
40                {
41                    if (isElement(element))
42                    {
43                        if (stack.Count == 0)
44                        {
45                            break;
46                        }
47                        element = stack.Pop();
48                        var source = getSource(element);
49                        var target = getTarget(element);
50                        if (isElement(source))
51                        {
52                            visit(source);
53                        }
54                        if (isElement(target))
55                        {
56                            visit(target);
57                        }
58                        element = target;
59                    }
60                    else
61                    {
62                        stack.Push(element);
63                        element = getSource(element);
64                    }
65                }
66            }
67        }
68
69        [MethodImpl(MethodImplOptions.AggressiveInlining)]
70        public static void WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
71        ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
72        ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
73        {
74            var stack = new Stack<TLinkAddress>();
75            var element = sequence;
```

```

69         if (isElement(element))
70         {
71             visit(element);
72         }
73     else
74     {
75         while (true)
76         {
77             if (isElement(element))
78             {
79                 if (stack.Count == 0)
80                 {
81                     break;
82                 }
83                 element = stack.Pop();
84                 var source = getSource(element);
85                 var target = getTarget(element);
86                 if (isElement(target))
87                 {
88                     visit(target);
89                 }
90                 if (isElement(source))
91                 {
92                     visit(source);
93                 }
94                 element = source;
95             }
96             else
97             {
98                 stack.Push(element);
99                 element = getTarget(element);
100             }
101         }
102     }
103 }
104 }
105 }

```

## 1.20 ./csharp/Platform.Data/Sequences/StopableSequenceWalker.cs

```

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

```

```

35     if (isElement(element))
36     {
37         if (stack.Count == 0)
38         {
39             return true;
40         }
41         element = stack.Pop();
42         exit(element);
43         exited++;
44         var source = getSource(element);
45         var target = getTarget(element);
46         if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
            ↪ !visit(source))
47         {
48             return false;
49         }
50         if ((isElement(target) || !canEnter(target)) && !visit(target))
51         {
52             return false;
53         }
54         element = target;
55     }
56     else
57     {
58         if (canEnter(element))
59         {
60             enter(element);
61             exited = 0;
62             stack.Push(element);
63             element = getSource(element);
64         }
65         else
66         {
67             if (stack.Count == 0)
68             {
69                 return true;
70             }
71             element = stack.Pop();
72             exit(element);
73             exited++;
74             var source = getSource(element);
75             var target = getTarget(element);
76             if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
            ↪ !visit(source))
77             {
78                 return false;
79             }
80             if ((isElement(target) || !canEnter(target)) && !visit(target))
81             {
82                 return false;
83             }
84             element = target;
85         }
86     }
87 }
88
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
92 {
93     var stack = new Stack<TLinkAddress>();
94     var element = sequence;
95     if (isElement(element))
96     {
97         return visit(element);
98     }
99     while (true)
100     {
101         if (isElement(element))
102         {
103             if (stack.Count == 0)
104             {
105                 return true;
106             }
107             element = stack.Pop();
108             var source = getSource(element);
109             var target = getTarget(element);

```

```

110         if (isElement(source) && !visit(source))
111         {
112             return false;
113         }
114         if (isElement(target) && !visit(target))
115         {
116             return false;
117         }
118         element = target;
119     }
120     else
121     {
122         stack.Push(element);
123         element = getSource(element);
124     }
125 }
126 }
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static bool WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
130 {
131     var stack = new Stack<TLinkAddress>();
132     var element = sequence;
133     if (isElement(element))
134     {
135         return visit(element);
136     }
137     while (true)
138     {
139         if (isElement(element))
140         {
141             if (stack.Count == 0)
142             {
143                 return true;
144             }
145             element = stack.Pop();
146             var source = getSource(element);
147             var target = getTarget(element);
148             if (isElement(target) && !visit(target))
149             {
150                 return false;
151             }
152             if (isElement(source) && !visit(source))
153             {
154                 return false;
155             }
156             element = source;
157         }
158         else
159         {
160             stack.Push(element);
161             element = getTarget(element);
162         }
163     }
164 }
165 }
166 }

```

## 1.21 ./csharp/Platform.Data/Universal/IUniLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
10     public partial interface IUniLinks<TLinkAddress>
11     {
12         IList<IList<IList<TLinkAddress>>> Trigger(IList<TLinkAddress> condition,
            ↪ IList<TLinkAddress> substitution);
13     }
14
15     /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
16     public partial interface IUniLinks<TLinkAddress>
17     {

```

```

18     /// <returns>
19     /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
    ↪ represents False (was stopped).
20     /// This is done to assure ability to push up stop signal through recursion stack.
21     /// </returns>
22     /// <remarks>
23     /// { 0, 0, 0 } => { itself, itself, itself } // create
24     /// { 1, any, any } => { itself, any, 3 } // update
25     /// { 3, any, any } => { 0, 0, 0 } // delete
26     /// </remarks>
27     TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>,
    ↪ TLinkAddress> matchHandler,
28         IList<TLinkAddress> substitution, Func<IList<TLinkAddress>,
    ↪ IList<TLinkAddress>, TLinkAddress> substitutionHandler);
29
30     TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>,
    ↪ IList<TLinkAddress>, TLinkAddress> matchedHandler,
31         IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>,
    ↪ TLinkAddress> substitutedHandler);
32 }
33
34 /// <remarks>Extended with small optimization.</remarks>
35 public partial interface IUniLinks<TLinkAddress>
36 {
37     /// <remarks>
38     /// Something simple should be simple and optimized.
39     /// </remarks>
40     TLinkAddress Count(IList<TLinkAddress> restrictions);
41 }
42 }

```

## 1.22 ./csharp/Platform.Data/Universal/IUniLinksCRUD.cs

```

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

```

## 1.23 ./csharp/Platform.Data/Universal/IUniLinksGS.cs

```

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

```

## 1.24 ./csharp/Platform.Data/Universal/IUniLinksIO.cs

```

1 using System;
2 using System.Collections.Generic;
3
4 // ReSharper disable TypeParameterCanBeVariant

```

```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Universal
8 {
9     /// <remarks>
10    /// In/Out aliases for IUniLinks.
11    /// TLinkAddress can be any number type of any size.
12    /// </remarks>
13    public interface IUniLinksIO<TLinkAddress>
14    {
15        /// <remarks>
16        /// default(TLinkAddress) means any link.
17        /// Single element pattern means just element (link).
18        /// Handler gets array of link contents.
19        /// * link[0] is index or identifier.
20        /// * link[1] is source or first.
21        /// * link[2] is target or second.
22        /// * link[3] is linker or third.
23        /// * link[n] is nth part/parent/element/value
24        /// of link (if variable length links used).
25        ///
26        /// Stops and returns false if handler return false.
27        ///
28        /// Acts as Each, Foreach, Select, Search, Match &amp; ...
29        ///
30        /// Handles all links in store if pattern/restrictions is not defined.
31        /// </remarks>
32        bool Out(Func<IList<TLinkAddress>, bool> handler, IList<TLinkAddress> pattern);
33
34        /// <remarks>
35        /// default(TLinkAddress) means itself.
36        /// Equivalent to:
37        /// * creation if before == null
38        /// * deletion if after == null
39        /// * update if before != null &amp;&amp; after != null
40        /// * default(TLinkAddress) if before == null &amp;&amp; after == null
41        ///
42        /// Possible interpretation
43        /// * In(null, new[] { }) creates point (link that points to itself using minimum number
44        ///   ↳ of parts).
45        /// * In(new[] { 4 }, null) deletes 4th link.
46        /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
47        ///   ↳ 5th index.
48        /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with
49        ///   ↳ 2 as source and 3 as target), 0 means it can be placed in any address.
50        /// ...
51        /// </remarks>
52        TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
53    }
54 }

```

## 1.25 ./csharp/Platform.Data/Universal/IUniLinksIOWithExtensions.cs

```

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

```

```

26         IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);
27
28         /// <remarks>OutCount() returns total amount of links in store.</remarks>
29         ulong OutCount(IList<TLinkAddress> pattern);
30     }
31 }

```

#### 1.26 ./csharp/Platform.Data/Universal/IUniLinksRW.cs

```

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

```

#### 1.27 ./csharp/Platform.Data.Tests/HybridTests.cs

```

1  using Xunit;
2
3  namespace Platform.Data.Tests
4  {
5      public static class HybridTests
6      {
7          [Fact]
8          public static void ObjectConstructorTest()
9          {
10             Assert.Equal(0, new Hybrid<byte>(unchecked((byte)128)).AbsoluteValue);
11             Assert.Equal(0, new Hybrid<byte>((object)128).AbsoluteValue);
12             Assert.Equal(1, new Hybrid<byte>(unchecked((byte)-1)).AbsoluteValue);
13             Assert.Equal(1, new Hybrid<byte>((object)-1).AbsoluteValue);
14             Assert.Equal(0, new Hybrid<byte>(unchecked((byte)0)).AbsoluteValue);
15             Assert.Equal(0, new Hybrid<byte>((object)0).AbsoluteValue);
16             Assert.Equal(1, new Hybrid<byte>(unchecked((byte)1)).AbsoluteValue);
17             Assert.Equal(1, new Hybrid<byte>((object)1).AbsoluteValue);
18         }
19     }
20 }

```

#### 1.28 ./csharp/Platform.Data.Tests/LinksConstantsTests.cs

```

1  using Xunit;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Tests
7  {
8      public static class LinksConstantsTests
9      {
10         [Fact]
11         public static void ConstructorTest()
12         {
13             var constants = new LinksConstants<ulong>(enableExternalReferencesSupport: true);
14             Assert.Equal(Hybrid<ulong>.ExternalZero,
15                 ↪ constants.ExternalReferencesRange.Value.Minimum);
16             Assert.Equal(ulong.MaxValue, constants.ExternalReferencesRange.Value.Maximum);
17         }
18
19         [Fact]
20         public static void ExternalReferencesTest()
21         {
22             TestExternalReferences<ulong, long>();
23             TestExternalReferences<uint, int>();
24             TestExternalReferences<ushort, short>();
25             TestExternalReferences<byte, sbyte>();
26         }
27
28         private static void TestExternalReferences<TUnsigned, TSigned>()
29         {

```



```

29     var unsingedOne = Arithmetic.Increment(default(TUnsigned));
30     var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
31     var half = converter.Convert(NumericType<TSigned>.MaxValue);
32     LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne,
    ↪ half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));
33
34     var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
35     var maximum = new Hybrid<TUnsigned>(half, isExternal: true);
36
37     Assert.True(constants.IsExternalReference(minimum));
38     Assert.True(minimum.IsExternal);
39     Assert.False(minimum.IsInternal);
40     Assert.True(constants.IsExternalReference(maximum));
41     Assert.True(maximum.IsExternal);
42     Assert.False(maximum.IsInternal);
43 }
44 }
45 }

```

## Index

- ./csharp/Platform.Data.Tests/HybridTests.cs, 24
- ./csharp/Platform.Data.Tests/LinksConstantsTests.cs, 24
- ./csharp/Platform.Data/Exceptions/ArgumentLinkDoesNotExistException.cs, 1
- ./csharp/Platform.Data/Exceptions/ArgumentLinkHasDependenciesException.cs, 1
- ./csharp/Platform.Data/Exceptions/LinkWithSameValueAlreadyExistsException.cs, 1
- ./csharp/Platform.Data/Exceptions/LinksLimitReachedException.cs, 2
- ./csharp/Platform.Data/Exceptions/LinksLimitReachedExceptionBase.cs, 2
- ./csharp/Platform.Data/Hybrid.cs, 2
- ./csharp/Platform.Data/ILinks.cs, 5
- ./csharp/Platform.Data/ILinksExtensions.cs, 7
- ./csharp/Platform.Data/ISynchronizedLinks.cs, 9
- ./csharp/Platform.Data/LinkAddress.cs, 10
- ./csharp/Platform.Data/LinksConstants.cs, 11
- ./csharp/Platform.Data/LinksConstantsBase.cs, 14
- ./csharp/Platform.Data/LinksConstantsExtensions.cs, 14
- ./csharp/Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs, 14
- ./csharp/Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs, 14
- ./csharp/Platform.Data/Point.cs, 15
- ./csharp/Platform.Data/Sequences/ISequenceAppender.cs, 17
- ./csharp/Platform.Data/Sequences/ISequenceWalker.cs, 17
- ./csharp/Platform.Data/Sequences/SequenceWalker.cs, 18
- ./csharp/Platform.Data/Sequences/StopableSequenceWalker.cs, 19
- ./csharp/Platform.Data/Universal/IUniLinks.cs, 21
- ./csharp/Platform.Data/Universal/IUniLinksCRUD.cs, 22
- ./csharp/Platform.Data/Universal/IUniLinksGS.cs, 22
- ./csharp/Platform.Data/Universal/IUniLinksIO.cs, 22
- ./csharp/Platform.Data/Universal/IUniLinksIOWithExtensions.cs, 23
- ./csharp/Platform.Data/Universal/IUniLinksRW.cs, 24