

# LinksPlatform's Platform.Data Class Library

## 1.1 ./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 ./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 ./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) { }
22     }
23 }
```

### 1.4 ./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     }
27 }
```

### 1.5 ./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     }
20 }
```

### 1.6 ./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 }
185 }

```

## 1.7 ./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     /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
11     /// </summary>
12     /// <remarks>
13     /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит
14     ↳ подходит как для дуплетов, так и для триплетов и т.п.
15     /// Возможно этот интерфейс подходит даже для Sequences.
16     /// </remarks>
17     public interface ILinks<TLinkAddress, TConstants>
18     where TConstants : LinksConstants<TLinkAddress>
19     {
20         #region Constants
21
22         /// <summary>
23         /// Возвращает набор констант, который необходим для эффективной коммуникации с методами
24         ↳ этого интерфейса.
25         /// Эти константы не меняются с момента создания точки доступа к хранилищу.
26         /// </summary>
27         TConstants Constants
28         {
29             get;
30         }
31     }
32 }

```

```

27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         get;
29     }
30
31 #endregion
32
33 #region Read
34
35     /// <summary>
36     /// Подсчитывает и возвращает общее число связей находящихся в хранилище,
37     /// соответствующих указанным ограничениям.
38     /// </summary>
39     /// <param name="restriction">Ограничения на содержимое связей.</param>
40     /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным
41     ///     ограничениям.</returns>
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     TLinkAddress Count(IList<TLinkAddress> restriction);
44
45     /// <summary>
46     /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
47     ///     (handler) для каждой подходящей связи.
48     /// </summary>
49     /// <param name="handler">Обработчик каждой подходящей связи.</param>
50     /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
51     ///     может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
52     ///     Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
53     /// <returns>True, в случае если проход по связям не был прерван и False в обратном
54     ///     случае.</returns>
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress>
57     restrictions);
58
59 #endregion
60
61 #region Write
62
63     /// <summary>
64     /// Создаёт связь.
65     /// </summary>
66     /// <returns>Индекс созданной связи.</returns>
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно
69     restrictions, возможно и возвращать связь нужно целиком.
70
71     /// <summary>
72     /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
73     ///     на связь с указанным новым содержимым.
74     /// </summary>
75     /// <param name="restrictions">
76     ///     Ограничения на содержимое связей.
77     ///     Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
78     ///     и далее за ним будет следовать содержимое связи.
79     ///     Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
80     ///     ссылку на пустоту,
81     ///     Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
82     ///     другой связи.
83     /// </param>
84     /// <param name="substitution"></param>
85     /// <returns>Индекс обновлённой связи.</returns>
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
88     // TODO: Возможно и возвращать связь нужно целиком.
89
90     /// <summary>Удаляет связь с указанным индексом.</summary>
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     void Delete(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно принимать
93     restrictions, а так же возвращать удалённую связь, если удаление было реально
94     выполнено, и Null, если нет.
95
96 #endregion
97
98 }
99
100 }
```

## 1.8 ./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,
15             ↳ TConstants> links, params TLinkAddress[] restrictions)
16             => links.Count(restrictions);
17
18         /// <summary>
19         /// Возвращает значение, определяющее существует ли связь с указанным индексом в
20         ↳ хранилище связей.
21         /// </summary>
22         /// <param name="links">Хранилище связей.</param>
23         /// <param name="link">Индекс проверяемой на существование связи.</param>
24         /// <returns>Значение, определяющее существует ли связь.</returns>
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
27             ↳ TConstants> links, TLinkAddress link)
28             where TConstants : LinksConstants<TLinkAddress>
29         {
30             var constants = links.Constants;
31             return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
32                 ↳ && Comparer<TLinkAddress>.Default.Compare(links.Count(new
33                     ↳ LinkAddress<TLinkAddress>(link)), default) > 0);
34         }
35
36         /// <param name="links">Хранилище связей.</param>
37         /// <param name="link">Индекс проверяемой на существование связи.</param>
38         /// <remarks>
39         /// TODO: May be move to EnsureExtensions or make it both there and here
40         /// </remarks>
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
43             ↳ TConstants> links, TLinkAddress link)
44             where TConstants : LinksConstants<TLinkAddress>
45         {
46             if (!links.Exists(link))
47             {
48                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
49             }
50         }
51
52         /// <param name="links">Хранилище связей.</param>
53         /// <param name="link">Индекс проверяемой на существование связи.</param>
54         /// <param name="argumentName">Имя аргумента, в который передается индекс связи.</param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
57             ↳ TConstants> links, TLinkAddress link, string argumentName)
58             where TConstants : LinksConstants<TLinkAddress>
59         {
60             if (!links.Exists(link))
61             {
62                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
63             }
64         }
65
66         /// <summary>
67         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
68         ↳ (handler) для каждой подходящей связи.
69         /// </summary>
70         /// <param name="links">Хранилище связей.</param>
71         /// <param name="handler">Обработчик каждой подходящей связи.</param>
72         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
73         ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
74         ↳ Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
75         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
76         ↳ случае.</returns>
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
79             ↳ TConstants> links, Func<IList<TLinkAddress>, TLinkAddress> handler, params
80             ↳ TLinkAddress[] restrictions)
81             where TConstants : LinksConstants<TLinkAddress>
82             => links.Each(handler, restrictions);
83     }
84 }

```

```

71  /// <summary>
72  /// Возвращает части-значения для связи с указанным индексом.
73  /// </summary>
74  /// <param name="links">Хранилище связей.</param>
75  /// <param name="link">Индекс связи.</param>
76  /// <returns>Уникальную связь.</returns>
77  [MethodImpl(MethodImplOptions.AggressiveInlining)]
78  public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
79  → 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>,
88  → TLinkAddress>(constants.Continue, constants.Break);
89  links.Each(linkPartsSetter.SetAndReturnTrue, link);
90  return linkPartsSetter.Result;
91  }
92  #region Points
93
94  /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
95  → точкой полностью (связью замкнутой на себе дважды).</summary>
96  /// <param name="links">Хранилище связей.</param>
97  /// <param name="link">Индекс проверяемой связи.</param>
98  /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
99  /// <remarks>
100  /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
101  → связь.
102  /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
103  → точка и пара существовать одновременно?
104  /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
105  → сортировать по индексу в массиве связей?
106  /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
107  /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
108  → самой себя любого размера?
109  /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
110  → одной ссылки на себя (частичной точки).
111  /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
112  → самостоятельный цикл через себя? Что если предоставить все варианты использования
113  → связей?
114  /// Что если разрешить и нули, а так же частичные варианты?
115  ///
116  /// Что если точка, это только в том случае когда link.Source == link && link.Target == link,
117  → т.е. дважды ссылка на себя.
118  /// А пара это тогда, когда link.Source == link.Target && link.Source != link,
119  → т.е. ссылка не на себя а во вне.
120  ///
121  /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
122  → промежуточную связь,
123  /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
124  /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
125  /// </remarks>
126  [MethodImpl(MethodImplOptions.AggressiveInlining)]
127  public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
128  → TConstants> links, TLinkAddress link)
129  where TConstants : LinksConstants<TLinkAddress>
130  {
131  if (links.Constants.IsExternalReference(link))
132  {
133  return true;
134  }
135  links.EnsureLinkExists(link);
136  return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
137  }
138
139  /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
140  → точкой частично (связью замкнутой на себе как минимум один раз).</summary>
141  /// <param name="links">Хранилище связей.</param>
142  /// <param name="link">Индекс проверяемой связи.</param>
143  /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
144  /// <remarks>
145  /// Достаточно любой одной ссылки на себя.

```



```

134     /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
135     ↪ ссылки на себя (на эту связь).
136     /// </remarks>
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
139     ↪ TConstants> links, TLinkAddress link)
140     where TConstants : LinksConstants<TLinkAddress>
141     {
142         if (links.Constants.IsExternalReference(link))
143         {
144             return true;
145         }
146         links.EnsureLinkExists(link);
147         return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
148     }
149 }
150 }

```

## 1.9 ./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     {
12     }
13 }

```

## 1.10 ./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>>,
11     ↪ IList<TLinkAddress>
12     {
13         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
14         ↪ EqualityComparer<TLinkAddress>.Default;
15
16         public TLinkAddress Index
17         {
18             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19             get;
20         }
21
22         public TLinkAddress this[int index]
23         {
24             [MethodImpl(MethodImplOptions.AggressiveInlining)]
25             get
26             {
27                 if (index == 0)
28                 {
29                     return Index;
30                 }
31                 else
32                 {
33                     throw new IndexOutOfRangeException();
34                 }
35             }
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             set => throw new NotSupportedException();
38         }
39
40         public int Count
41         {
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             get => 1;
44         }
45     }
46 }

```

```

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

```

```

115         {
116             return false;
117         }
118         return left.Equals(right);
119     }
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
        ↪ right) => !(left == right);
123 }
124 }

```

### 1.11 ./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>
            ↪ _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
15
16         #region Link parts
17
18         /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
            ↪ 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
79   ↳ создавать все варианты последовательностей в функции Create.</remarks>
80 public TLinkAddress Any
81 {
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     get;
84 }
85
86 /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
87 public TLinkAddress Itself
88 {
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     get;
91 }
92
93 #endregion
94
95 #region References
96
97 /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
98   ↳ ссылок).</summary>
99 public Range<TLinkAddress> InternalReferencesRange
100 {
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     get;
103 }
104
105 /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
106   ↳ ссылок).</summary>
107 public Range<TLinkAddress>? ExternalReferencesRange
108 {
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     get;
111 }
112
113 #endregion
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public LinksConstants(int targetPart, Range<TLinkAddress>
117   ↳ possibleInternalReferencesRange, Range<TLinkAddress>?
118   ↳ possibleExternalReferencesRange)
119 {
120     IndexPart = 0;
121     SourcePart = 1;
122     TargetPart = targetPart;
123     Null = default;
124     Break = default;
125     var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
126     Continue = currentInternalReferenceIndex;
127     Decrement(ref currentInternalReferenceIndex);
128     Skip = currentInternalReferenceIndex;
129     Decrement(ref currentInternalReferenceIndex);
130     Any = currentInternalReferenceIndex;
131     Decrement(ref currentInternalReferenceIndex);
132     Itself = currentInternalReferenceIndex;
133     Decrement(ref currentInternalReferenceIndex);
134     InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
135   ↳ currentInternalReferenceIndex);
136     ExternalReferencesRange = possibleExternalReferencesRange;
137 }
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
141   ↳ this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
142   ↳ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }

```

```

135 [MethodImpl(MethodImplOptions.AggressiveInlining)]
136 public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
137     ↪ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
138     ↪ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
139
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 public LinksConstants(bool enableExternalReferencesSupport) :
142     ↪ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
143     ↪ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public LinksConstants(int targetPart, Range<TLinkAddress>
147     ↪ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
148     ↪ null) { }
149
150 [MethodImpl(MethodImplOptions.AggressiveInlining)]
151 public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
152     ↪ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
156     ↪ false) { }
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
160     ↪ enableExternalReferencesSupport)
161 {
162     if (enableExternalReferencesSupport)
163     {
164         return (_one, _UInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
165             ↪ rValuesRange));
166     }
167     else
168     {
169         return (_one, NumericType<TLinkAddress>.MaxValue);
170     }
171 }
172
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
175     ↪ enableExternalReferencesSupport)
176 {
177     if (enableExternalReferencesSupport)
178     {
179         return (_UInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumberValue
180             ↪ sRange + 1UL),
181             ↪ NumericType<TLinkAddress>.MaxValue);
182     }
183     else
184     {
185         return null;
186     }
187 }
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
191     ↪ currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
192 }
193 }

```

## 1.12 ./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 ./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>
11        ↪ linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
12        ↪ || linksConstants.IsExternalReference(address);
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
16        ↪ linksConstants, TLinkAddress address) =>
17        ↪ linksConstants.InternalReferencesRange.Contains(address);
18
19        [MethodImpl(MethodImplOptions.AggressiveInlining)]
20        public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
21        ↪ linksConstants, TLinkAddress address) =>
22        ↪ linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
23    }
24 }

```

#### 1.14 ./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 ./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 ./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        {

```

```

25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     get;
27 }
28
29 public TLinkAddress this[int index]
30 {
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     get
33     {
34         if (index < Size)
35         {
36             return Index;
37         }
38         else
39         {
40             throw new IndexOutOfRangeException();
41         }
42     }
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     set => throw new NotSupportedException();
45 }
46
47 public int Count
48 {
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     get => int.MaxValue;
51 }
52
53 public bool IsReadOnly
54 {
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     get => true;
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 public Point(TLinkAddress index, int size)
61 {
62     Index = index;
63     Size = size;
64 }
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public void Add(TLinkAddress item) => throw new NotSupportedException();
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public void Clear() => throw new NotSupportedException();
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
74     ↪ ? true : false;
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public IEnumerator<TLinkAddress> GetEnumerator()
81 {
82     for (int i = 0; i < Size; i++)
83     {
84         yield return Index;
85     }
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
90     ↪ 0 : -1;
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 public bool Remove(TLinkAddress item) => throw new NotSupportedException();
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 public void RemoveAt(int index) => throw new NotSupportedException();
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 IEnumerator IEnumerable.GetEnumerator()
103 {
104     for (int i = 0; i < Size; i++)

```

```

103     {
104         yield return Index;
105     }
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
    ↳ _equalityComparer.Equals(Index, other.Index);
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
    ↳ linkAddress.Index;
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
    ↳ Equals(linkAddress) : false;
116
117 [MethodImpl(MethodImplOptions.AggressiveInlining)]
118 public override int GetHashCode() => Index.GetHashCode();
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 public override string ToString() => Index.ToString();
122
123 [MethodImpl(MethodImplOptions.AggressiveInlining)]
124 public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
125 {
126     if (left == null && right == null)
127     {
128         return true;
129     }
130     if (left == null)
131     {
132         return false;
133     }
134     return left.Equals(right);
135 }
136
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
    ↳ !(left == right);
139
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 public static bool IsFullPoint(params TLinkAddress[] link) =>
    ↳ IsFullPoint((IList<TLinkAddress>)link);
142
143 [MethodImpl(MethodImplOptions.AggressiveInlining)]
144 public static bool IsFullPoint(IList<TLinkAddress> link)
145 {
146     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
147     Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
    ↳ determine link's pointness using only its identifier.");
148     return IsFullPointUnchecked(link);
149 }
150
151 [MethodImpl(MethodImplOptions.AggressiveInlining)]
152 public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
153 {
154     var result = true;
155     for (var i = 1; result && i < link.Count; i++)
156     {
157         result = _equalityComparer.Equals(link[0], link[i]);
158     }
159     return result;
160 }
161
162 [MethodImpl(MethodImplOptions.AggressiveInlining)]
163 public static bool IsPartialPoint(params TLinkAddress[] link) =>
    ↳ IsPartialPoint((IList<TLinkAddress>)link);
164
165 [MethodImpl(MethodImplOptions.AggressiveInlining)]
166 public static bool IsPartialPoint(IList<TLinkAddress> link)
167 {
168     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
169     Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
    ↳ determine link's pointness using only its identifier.");
170     return IsPartialPointUnchecked(link);
171 }
172

```



```

173     [MethodImpl(MethodImplOptions.AggressiveInlining)]
174     public static bool IsPartialPointUnchecked(ICollection<TLinkAddress> link)
175     {
176         var result = false;
177         for (var i = 1; !result && i < link.Count; i++)
178         {
179             result = _equalityComparer.Equals(link[0], link[i]);
180         }
181         return result;
182     }
183 }
184 }

```

### 1.17 ./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 ./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<ICollection<TLinkAddress>> Walk(TLinkAddress sequence);
12     }
13 }

```

### 1.19 ./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 обязательно рекурсивная реализация (с вложенным вызовом функцией самой
12     /// ↪ себя),
13     /// так как стек можно использовать намного эффективнее при ручном управлении.
14     /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
15     /// Решить встраивать ли защиту от заикливания.
16     /// Альтернативой защиты от заикливания может быть заранее известное ограничение на
17     /// ↪ погружение вглубь.
18     /// А так же качественное распознавание прохода по циклическому графу.
19     /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер
20     /// ↪ стека.
21     /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
22     /// </remarks>
23     public static class SequenceWalker
24     {
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
27             ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
28             ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
29         {
30             var stack = new Stack<TLinkAddress>();
31             var element = sequence;
32             if (isElement(element))
33             {
34                 visit(element);
35             }
36         }
37     }
38 }

```

```

32     else
33     {
34         while (true)
35         {
36             if (isElement(element))
37             {
38                 if (stack.Count == 0)
39                 {
40                     break;
41                 }
42                 element = stack.Pop();
43                 var source = getSource(element);
44                 var target = getTarget(element);
45                 if (isElement(source))
46                 {
47                     visit(source);
48                 }
49                 if (isElement(target))
50                 {
51                     visit(target);
52                 }
53                 element = target;
54             }
55             else
56             {
57                 stack.Push(element);
58                 element = getSource(element);
59             }
60         }
61     }
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static void WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
66 {
67     var stack = new Stack<TLinkAddress>();
68     var element = sequence;
69     if (isElement(element))
70     {
71         visit(element);
72     }
73     else
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 ./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 обязательно 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 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             {
41                 if (isElement(element))
42                 {
43                     {
44                         if (stack.Count == 0)
45                         {
46                             return true;
47                         }
48                         element = stack.Pop();
49                         exit(element);
50                         exited++;
51                         var source = getSource(element);
52                         var target = getTarget(element);
53                         if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
54                             ↪ !visit(source))
55                         {
56                             return false;
57                         }
58                         if ((isElement(target) || !canEnter(target)) && !visit(target))
59                         {
60                             return false;
61                         }
62                         element = target;
63                     }
64                 }
65                 else
66                 {
67                     if (canEnter(element))
68                     {
69                         enter(element);
70                         exited = 0;
71                         stack.Push(element);
72                         element = getSource(element);
73                     }
74                     else
75                     {
76                         if (stack.Count == 0)
77                         {
78                             return true;
79                         }
80                         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 ./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,
13             ⇨ IList<TLinkAddress> substitution);
14     }
15
16     /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
17     public partial interface IUniLinks<TLinkAddress>
18     {
19         /// <returns>
20         /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
21         /// ⇨ represents False (was stopped).
22         /// This is done to assure ability to push up stop signal through recursion stack.
23         /// </returns>
24         /// <remarks>
25         /// { 0, 0, 0 } => { itself, itself, itself } // create
26         /// { 1, any, any } => { itself, any, 3 } // update
27         /// { 3, any, any } => { 0, 0, 0 } // delete
28         /// </remarks>
29         TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>,
30             ⇨ TLinkAddress> matchHandler,
31             ⇨ IList<TLinkAddress> substitution, Func<IList<TLinkAddress>,
32             ⇨ IList<TLinkAddress>, TLinkAddress> substitutionHandler);
33
34         TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>,
35             ⇨ IList<TLinkAddress>, TLinkAddress> matchedHandler,
36             ⇨ IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>,
37             ⇨ TLinkAddress> substitutedHandler);
38     }
39
40     /// <remarks>Extended with small optimization.</remarks>
41     public partial interface IUniLinks<TLinkAddress>
42     {
43         /// <remarks>
44         /// Something simple should be simple and optimized.
45         /// </remarks>
46         TLinkAddress Count(IList<TLinkAddress> restrictions);
47     }
48 }

```

## 1.22 ./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 ./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 ./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 & ...
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 & & after != null
40        /// * default(TLinkAddress) if before == null & & 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>
    TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
}
}

```

## 1.25 ./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) ~ GetLinkAddresser(link) or GetLinkAddresser(Search(pattern))
19         /// OutPart(n, pattern) => For any variable length links, returns link or
20         ↪ default(TLinkAddress).
21         ///
22         /// Outs(returns) inner contents of link, its part/parent/element/value.
23         /// </remarks>
24         TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);
25
26         /// <remarks>OutCount() returns total links in store as array.</remarks>
27         IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);
28
29         /// <remarks>OutCount() returns total amount of links in store.</remarks>
30         ulong OutCount(IList<TLinkAddress> pattern);
31     }
}

```

## 1.26 ./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 ./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 ./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 ExternalReferencesTest()
12         {
13             TestExternalReferences<ulong, long>();
14             TestExternalReferences<uint, int>();
15             TestExternalReferences<ushort, short>();
16             TestExternalReferences<byte, sbyte>();
17         }
18
19         private static void TestExternalReferences<TUnsigned, TSigned>()
20         {
21             var unsingedOne = Arithmetic.Increment(default(TUnsigned));
22             var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
23             var half = converter.Convert(NumericType<TSigned>.MaxValue);
24             LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne,
25                 ↪ half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));
26
27             var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
28             var maximum = new Hybrid<TUnsigned>(half, isExternal: true);
29
30             Assert.True(constants.IsExternalReference(minimum));
31             Assert.True(minimum.IsExternal);
32             Assert.False(minimum.IsInternal);
33             Assert.True(constants.IsExternalReference(maximum));
34             Assert.True(maximum.IsExternal);
35             Assert.False(maximum.IsInternal);
36         }
37     }
38 }

```



## Index

- ./Platform.Data.Tests/HybridTests.cs, 23
- ./Platform.Data.Tests/LinksConstantsTests.cs, 24
- ./Platform.Data/Exceptions/ArgumentLinkDoesNotExistsException.cs, 1
- ./Platform.Data/Exceptions/ArgumentLinkHasDependenciesException.cs, 1
- ./Platform.Data/Exceptions/LinkWithSameValueAlreadyExistsException.cs, 1
- ./Platform.Data/Exceptions/LinksLimitReachedException.cs, 2
- ./Platform.Data/Exceptions/LinksLimitReachedExceptionBase.cs, 2
- ./Platform.Data/Hybrid.cs, 2
- ./Platform.Data/ILinks.cs, 5
- ./Platform.Data/ILinksExtensions.cs, 6
- ./Platform.Data/ISynchronizedLinks.cs, 9
- ./Platform.Data/LinkAddress.cs, 9
- ./Platform.Data/LinksConstants.cs, 11
- ./Platform.Data/LinksConstantsBase.cs, 13
- ./Platform.Data/LinksConstantsExtensions.cs, 13
- ./Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs, 14
- ./Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs, 14
- ./Platform.Data/Point.cs, 14
- ./Platform.Data/Sequences/ISequenceAppender.cs, 17
- ./Platform.Data/Sequences/ISequenceWalker.cs, 17
- ./Platform.Data/Sequences/SequenceWalker.cs, 17
- ./Platform.Data/Sequences/StopableSequenceWalker.cs, 18
- ./Platform.Data/Universal/IUniLinks.cs, 21
- ./Platform.Data/Universal/IUniLinksCRUD.cs, 21
- ./Platform.Data/Universal/IUniLinksGS.cs, 22
- ./Platform.Data/Universal/IUniLinksIO.cs, 22
- ./Platform.Data/Universal/IUniLinksIOWithExtensions.cs, 23
- ./Platform.Data/Universal/IUniLinksRW.cs, 23