

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.Reflection;
4 using System.Reflection.Emit;
5 using System.Runtime.CompilerServices;
6 using Platform.Exceptions;
7 using Platform.Reflection;
8 using Platform.Converters;
9 using Platform.Numbers;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data
14 {
15     public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
16     {
17         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
18             ↪ EqualityComparer<TLinkAddress>.Default;
19         private static readonly uncheckedConverter<TLinkAddress, long>
20             ↪ _addressToInt64Converter = uncheckedSignExtendingConverter<TLinkAddress,
21             ↪ long>.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 Func<object, TLinkAddress> _unboxAbsAndConvert =
27             ↪ CompileUnboxAbsAndConvertDelegate();
28         private static readonly Func<object, TLinkAddress> _unboxAbsNegateAndConvert =
29             ↪ CompileUnboxAbsNegateAndConvertDelegate();
30
31         public static readonly ulong HalfOfNumberValuesRange =
32             ↪ _addressToUInt64Converter.Convert(NumericType<TLinkAddress>.MaxValue) / 2;
33         public static readonly TLinkAddress ExternalZero =
34             ↪ _uint64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);
35
36         public readonly TLinkAddress Value;
37
38         public bool IsNothing
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
42         }
43
44         public bool IsInternal
45         {
46             [MethodImpl(MethodImplOptions.AggressiveInlining)]
47             get => SignedValue > 0;
48         }
49
50         public bool IsExternal
51         {
52             [MethodImpl(MethodImplOptions.AggressiveInlining)]
53             get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
54         }
55
56         public long SignedValue
57         {
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             get => _addressToInt64Converter.Convert(Value);
60         }
61
62         public long AbsoluteValue
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
66                 ↪ Platform.Numbers.Math.Abs(SignedValue);
67         }
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public Hybrid(TLinkAddress value)
71         {
72             Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
73             Value = value;
74         }
75
76         [MethodImpl(MethodImplOptions.AggressiveInlining)]
77         public Hybrid(TLinkAddress value, bool isExternal)
78         {
79             if (_equalityComparer.Equals(value, default) && isExternal)
80             {
81                 Value = ExternalZero;
82             }
83         }
84     }
85 }

```

```

73     else
74     {
75         if (isExternal)
76         {
77             Value = Math<TLinkAddress>.Negate(value);
78         }
79         else
80         {
81             Value = value;
82         }
83     }
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public Hybrid(object value) => Value =
    ↳ To.UnsignedAs<TLinkAddress>(Convert.ChangeType(value,
    ↳ NumericType<TLinkAddress>.SignedVersion));
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public Hybrid(object value, bool isExternal)
91 {
92     if (IsDefault(value) && isExternal)
93     {
94         Value = ExternalZero;
95     }
96     else
97     {
98         if (isExternal)
99         {
100             Value = _unboxAbsNegateAndConvert(value);
101         }
102         else
103         {
104             Value = _unboxAbsAndConvert(value);
105         }
106     }
107 }
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public static explicit operator Hybrid<TLinkAddress>(long integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static explicit operator Hybrid<TLinkAddress>(int integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
126
127 [MethodImpl(MethodImplOptions.AggressiveInlining)]
128 public static explicit operator Hybrid<TLinkAddress>(short integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
129
130 [MethodImpl(MethodImplOptions.AggressiveInlining)]
131 public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
    ↳ Hybrid<TLinkAddress>(integer);
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
    ↳ hybrid.Value;
138

```

```

139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
    ↳ CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
    ↳ hybrid.AbsoluteValue;
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
    ↳ CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
147
148 [MethodImpl(MethodImplOptions.AggressiveInlining)]
149 public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
    ↳ (int)hybrid.AbsoluteValue;
150
151 [MethodImpl(MethodImplOptions.AggressiveInlining)]
152 public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
    ↳ CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
    ↳ (short)hybrid.AbsoluteValue;
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
    ↳ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);
159
160 [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
    ↳ (sbyte)hybrid.AbsoluteValue;
162
163 [MethodImpl(MethodImplOptions.AggressiveInlining)]
164 public override string ToString() => IsExternal ? $"{<AbsoluteValue>}" :
    ↳ Value.ToString();
165
166 [MethodImpl(MethodImplOptions.AggressiveInlining)]
167 public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
    ↳ other.Value);
168
169 [MethodImpl(MethodImplOptions.AggressiveInlining)]
170 public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
    ↳ Equals(hybrid) : false;
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public override int GetHashCode() => Value.GetHashCode();
174
175 [MethodImpl(MethodImplOptions.AggressiveInlining)]
176 public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
    ↳ left.Equals(right);
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
    ↳ !(left == right);
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 private static bool IsDefault(object value)
183 {
184     if (value == null)
185     {
186         return true;
187     }
188     var type = value.GetType();
189     return type.IsValueType ? value.Equals(Activator.CreateInstance(type)) : false;
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private static Func<object, TLinkAddress> CompileUnboxAbsNegateAndConvertDelegate()
194 {
195     return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emitter =>
196     {
197         Ensure.Always.IsUnsignedInteger<TLinkAddress>();
198         emitter.LoadArgument(0);
199         var signedVersion = NumericType<TLinkAddress>.SignedVersion;
200         var signedVersionField =
            ↳ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
            ↳ BindingFlags.Static | BindingFlags.Public);
        emitter.Emit(OpCodes.Ldsfld, signedVersionField);

```

```

202     var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
203     ↪ Types<object, Type>.Array);
204     emitter.Call(changeTypeMethod);
205     emitter.UnboxValue(signedVersion);
206     var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
207     ↪ signedVersion });
208     emitter.Call(absMethod);
209     var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate",
210     ↪ ").MakeGenericMethod(signedVersion);
211     emitter.Call(negateMethod);
212     var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
213     ↪ signedVersion });
214     emitter.Call(unsignedMethod);
215     emitter.Return();
216     });
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 private static Func<object, TLinkAddress> CompileUnboxAbsAndConvertDelegate()
221 {
222     return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emitter =>
223     {
224         Ensure.Always.IsUnsignedInteger<TLinkAddress>();
225         emitter.LoadArgument(0);
226         var signedVersion = NumericType<TLinkAddress>.SignedVersion;
227         var signedVersionField =
228         ↪ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
229         ↪ BindingFlags.Static | BindingFlags.Public);
230         emitter.Emit(OpCodes.Ldsfld, signedVersionField);
231         var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
232         ↪ Types<object, Type>.Array);
233         emitter.Call(changeTypeMethod);
234         emitter.UnboxValue(signedVersion);
235         var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
236         ↪ signedVersion });
237         emitter.Call(absMethod);
238         var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
239         ↪ signedVersion });
240         emitter.Call(unsignedMethod);
241         emitter.Return();
242     });
243 }
244 }
245 }
246 }

```

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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             get;
31         }
32     }
33 }

```

```

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     ///     принимать 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 }

```

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

```



```

75  /// <param name="links">Хранилище связей.</param>
76  /// <param name="link">Индекс связи.</param>
77  /// <returns>Уникальную связь.</returns>
78  [MethodImpl(MethodImplOptions.AggressiveInlining)]
79  public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
    ↳ ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
80  where TConstants : LinksConstants<TLinkAddress>
81  {
82      var constants = links.Constants;
83      if (constants.IsExternalReference(link))
84      {
85          return new Point<TLinkAddress>(link, constants.TargetPart + 1);
86      }
87      var linkPartsSetter = new Setter<IList<TLinkAddress>,
    ↳ TLinkAddress>(constants.Continue, constants.Break);
88      links.Each(linkPartsSetter.SetAndReturnTrue, link);
89      return linkPartsSetter.Result;
90  }
91
92  #region Points
93
94  /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
    ↳ точкой полностью (связью замкнутой на себе дважды).</summary>
95  /// <param name="links">Хранилище связей.</param>
96  /// <param name="link">Индекс проверяемой связи.</param>
97  /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
98  /// <remarks>
99  /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
    ↳ связь.
100  /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
    ↳ точка и пара существовать одновременно?
101  /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
    ↳ сортировать по индексу в массиве связей?
102  /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
103  /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
    ↳ самой себя любого размера?
104  /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
    ↳ одной ссылки на себя (частичной точки).
105  /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
106  /// самостоятельный цикл через себя? Что если предоставить все варианты использования
    ↳ связей?
107  /// Что если разрешить и нули, а так же частичные варианты?
108  ///
109  /// Что если точка, это только в том случае когда link.Source == link &&
    ↳ link.Target == link , т.е. дважды ссылка на себя.
110  /// А пара это тогда, когда link.Source == link.Target && link.Source != link ,
    ↳ т.е. ссылка не на себя а во вне.
111  ///
112  /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
    ↳ промежуточную связь,
113  /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
114  /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
115  /// </remarks>
116  [MethodImpl(MethodImplOptions.AggressiveInlining)]
117  public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    ↳ TConstants> links, TLinkAddress link)
118  where TConstants : LinksConstants<TLinkAddress>
119  {
120      if (links.Constants.IsExternalReference(link))
121      {
122          return true;
123      }
124      links.EnsureLinkExists(link);
125      return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
126  }
127
128  /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
    ↳ точкой частично (связью замкнутой на себе как минимум один раз).</summary>
129  /// <param name="links">Хранилище связей.</param>
130  /// <param name="link">Индекс проверяемой связи.</param>
131  /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
132  /// <remarks>
133  /// Достаточно любой одной ссылки на себя.
134  /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
    ↳ ссылки на себя (на эту связь).
135  /// </remarks>
136  [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

137     public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    ↪ TConstants> links, TLinkAddress link)
138     where TConstants : LinksConstants<TLinkAddress>
139     {
140         if (links.Constants.IsExternalReference(link))
141         {
142             return true;
143         }
144         links.EnsureLinkExists(link);
145         return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
146     }
147
148     #endregion
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> :
    ↪ ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
8     where TLinks : ILinks<TLinkAddress, TConstants>
9     where TConstants : LinksConstants<TLinkAddress>
10     {
11     }
12 }

```

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>>,
    ↪ 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
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 unchecked<ulong, TLinkAddress>
            ↪ _uInt64ToAddressConverter = unchecked<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 #endregion
93
94 #region References
95
96 /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
97   ↳ ссылок).</summary>
98 public Range<TLinkAddress> InternalReferencesRange
99 {
100     [MethodImpl(MethodImplOptions.AggressiveInlining)]
101     get;
102 }
103
104 /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
105   ↳ ссылок).</summary>
106 public Range<TLinkAddress>? ExternalReferencesRange
107 {
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     get;
110 }
111 #endregion
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public LinksConstants(int targetPart, Range<TLinkAddress>
115   ↳ possibleInternalReferencesRange, Range<TLinkAddress>?
116   ↳ possibleExternalReferencesRange)
117 {
118     IndexPart = 0;
119     SourcePart = 1;
120     TargetPart = targetPart;
121     Null = default;
122     Break = default;
123     var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
124     Continue = currentInternalReferenceIndex;
125     Decrement(ref currentInternalReferenceIndex);
126     Skip = currentInternalReferenceIndex;
127     Decrement(ref currentInternalReferenceIndex);
128     Any = currentInternalReferenceIndex;
129     Decrement(ref currentInternalReferenceIndex);
130     Itself = currentInternalReferenceIndex;
131     Decrement(ref currentInternalReferenceIndex);
132     InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
133   ↳ currentInternalReferenceIndex);
134     ExternalReferencesRange = possibleExternalReferencesRange;
135 }
136
137 [MethodImpl(MethodImplOptions.AggressiveInlining)]
138 public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
139   ↳ this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
140   ↳ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

137     public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
138         ↪ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
139         ↪ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
140
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public LinksConstants(bool enableExternalReferencesSupport) :
143         ↪ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
144         ↪ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
145
146     [MethodImpl(MethodImplOptions.AggressiveInlining)]
147     public LinksConstants(int targetPart, Range<TLinkAddress>
148         ↪ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
149         ↪ null) { }
150
151     [MethodImpl(MethodImplOptions.AggressiveInlining)]
152     public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
153         ↪ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
154
155     [MethodImpl(MethodImplOptions.AggressiveInlining)]
156     public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
157         ↪ false) { }
158
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
161         ↪ enableExternalReferencesSupport)
162     {
163         if (enableExternalReferencesSupport)
164         {
165             return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
166                 ↪ rValuesRange));
167         }
168         else
169         {
170             return (_one, NumericType<TLinkAddress>.MaxValue);
171         }
172     }
173
174     [MethodImpl(MethodImplOptions.AggressiveInlining)]
175     public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
176         ↪ enableExternalReferencesSupport)
177     {
178         if (enableExternalReferencesSupport)
179         {
180             return (_uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumberValue
181                 ↪ sRange + 1UL),
182                 ↪ NumericType<TLinkAddress>.MaxValue);
183         }
184         else
185         {
186             return null;
187         }
188     }
189
190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
191     private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
192         ↪ currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
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         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29     }
30 }

```

```

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++)
105         {
106             yield return Index;
107         }
108     }

```



```

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, new Range<int>(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, new Range<int>(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(IList<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<IList<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 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 зацикливания.
17     /// Альтернативой защиты от заклинания 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             {

```

```

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;

```



```

76         if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
77             ↪ !visit(source))
78         {
79             return false;
80         }
81         if ((isElement(target) || !canEnter(target)) && !visit(target))
82         {
83             return false;
84         }
85         element = target;
86     }
87 }
88 }
89 }
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
92     ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
93     ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
94 {
95     var stack = new Stack<TLinkAddress>();
96     var element = sequence;
97     if (isElement(element))
98     {
99         return visit(element);
100     }
101     while (true)
102     {
103         if (isElement(element))
104         {
105             if (stack.Count == 0)
106             {
107                 return true;
108             }
109             element = stack.Pop();
110             var source = getSource(element);
111             var target = getTarget(element);
112             if (isElement(source) && !visit(source))
113             {
114                 return false;
115             }
116             if (isElement(target) && !visit(target))
117             {
118                 return false;
119             }
120             element = target;
121         }
122         else
123         {
124             stack.Push(element);
125             element = getSource(element);
126         }
127     }
128 }
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static bool WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
131     ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
132     ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
133 {
134     var stack = new Stack<TLinkAddress>();
135     var element = sequence;
136     if (isElement(element))
137     {
138         return visit(element);
139     }
140     while (true)
141     {
142         if (isElement(element))
143         {
144             if (stack.Count == 0)
145             {
146                 return true;
147             }
148             element = stack.Pop();
149             var source = getSource(element);
150             var target = getTarget(element);
151             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.

```

```

45     /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
    ↪ 5th index.
46     /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with
    ↪ 2 as source and 3 as target), 0 means it can be placed in any address.
47     /// ...
48     /// </remarks>
49     TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
50 }
51 }

```

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) ~ GetTLinkAddresser(link) or GetTLinkAddresser(Search(pattern))
19         /// OutPart(n, pattern) => For any variable length links, returns link or
    ↪ default(TLinkAddress).
20         ///
21         /// Outs(returns) inner contents of link, its part/parent/element/value.
22         /// </remarks>
23         TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);
24
25         /// <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 ./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/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         }
15     }
16 }

```



```

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,
        ↪ half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));
25
26     var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
27     var maximum = new Hybrid<TUnsigned>(half, isExternal: true);
28
29     Assert.True(constants.IsExternalReference(minimum));
30     Assert.True(minimum.IsExternal);
31     Assert.False(minimum.IsInternal);
32     Assert.True(constants.IsExternalReference(maximum));
33     Assert.True(maximum.IsExternal);
34     Assert.False(maximum.IsInternal);
35 }
36 }
37 }

```

Index

- ./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, 6
- ./Platform.Data/ILinksExtensions.cs, 7
- ./Platform.Data/ISynchronizedLinks.cs, 10
- ./Platform.Data/LinkAddress.cs, 10
- ./Platform.Data/LinksConstants.cs, 12
- ./Platform.Data/LinksConstantsBase.cs, 14
- ./Platform.Data/LinksConstantsExtensions.cs, 14
- ./Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs, 15
- ./Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs, 15
- ./Platform.Data/Point.cs, 15
- ./Platform.Data/Sequences/ISequenceAppender.cs, 18
- ./Platform.Data/Sequences/ISequenceWalker.cs, 18
- ./Platform.Data/Sequences/SequenceWalker.cs, 18
- ./Platform.Data/Sequences/StopableSequenceWalker.cs, 19
- ./Platform.Data/Universal/IUniLinks.cs, 22
- ./Platform.Data/Universal/IUniLinksCRUD.cs, 22
- ./Platform.Data/Universal/IUniLinksGS.cs, 23
- ./Platform.Data/Universal/IUniLinksIO.cs, 23
- ./Platform.Data/Universal/IUniLinksLOWithExtensions.cs, 24
- ./Platform.Data/Universal/IUniLinksRW.cs, 24