

LinksPlatform's Platform.Data Class Library

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

```
1 using System;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Exceptions
6 {
7     public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
8     {
9         public ArgumentLinkDoesNotExistsException(TLinkAddress link, string paramName) :
10             ↪ base(FormatMessage(link, paramName), paramName) { }
11         public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link))
12             ↪ { }
13         private static string FormatMessage(TLinkAddress link, string paramName) => $"Связь
14             ↪ [{link}] переданная в аргумент [{paramName}] не существует.";
15         private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
16             ↪ качестве аргумента не существует.";
17     }
18 }
```

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

```
1 using System;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Exceptions
6 {
7     public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
8     {
9         public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) :
10             ↪ base(FormatMessage(link, paramName), paramName) { }
11         public ArgumentLinkHasDependenciesException(TLinkAddress link) :
12             ↪ base(FormatMessage(link)) { }
13         private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи
14             ↪ [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые
15             ↪ препятствуют изменению её внутренней структуры.";
16         private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной
17             ↪ в качестве аргумента присутствуют зависимости, которые препятствуют изменению её
18             ↪ внутренней структуры.";
19     }
20 }
```

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

```
1 using System;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Exceptions
6 {
7     public class LinkWithSameValueAlreadyExistsException : Exception
8     {
9         public static readonly string DefaultMessage = "Связь с таким же значением уже
10             ↪ существует.";
11         public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
12         public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
13     }
14 }
```

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

```
1 using System;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Exceptions
6 {
7     public class LinksLimitReachedException<TLinkAddress> : Exception
8     {
9         public static readonly string DefaultMessage = "Достигнут лимит количества связей в
10             ↪ хранилище.";
11         public LinksLimitReachedException(string message) : base(message) { }
12         public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }
13         public LinksLimitReachedException() : base(DefaultMessage) { }
14         private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества
15             ↪ связей в хранилище ({limit}).";
16     }
17 }
```

1.5 ./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 UncheckedSignExtendingConverter<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)
```

```

70     {
71         Value = ExternalZero;
72     }
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) =>
138     ↪ hybrid.Value;
139
140 [MethodImpl(MethodImplOptions.AggressiveInlining)]
141 public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
142     ↪ CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);
143
144 [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
146     ↪ hybrid.AbsoluteValue;
147
148 [MethodImpl(MethodImplOptions.AggressiveInlining)]
149 public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
150     ↪ CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
151
152 [MethodImpl(MethodImplOptions.AggressiveInlining)]
153 public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
154     ↪ (int)hybrid.AbsoluteValue;
155
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
158     ↪ CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
159
160 [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
162     ↪ (short)hybrid.AbsoluteValue;
163
164 [MethodImpl(MethodImplOptions.AggressiveInlining)]
165 public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
166     ↪ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);
167
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
170     ↪ (sbyte)hybrid.AbsoluteValue;
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public override string ToString() => IsNothing ? default(TLinkAddress) == null ?
174     ↪ "Nothing" : default(TLinkAddress).ToString() : IsExternal ? $"<{AbsoluteValue}>" :
175     ↪ Value.ToString();
176
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
179     ↪ other.Value);
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
183     ↪ Equals(hybrid) : false;
184
185 [MethodImpl(MethodImplOptions.AggressiveInlining)]
186 public override int GetHashCode() => Value.GetHashCode();
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
190     ↪ left.Equals(right);
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
194     ↪ !(left == right);
195
196 private static bool IsDefault(object value)
197 {
198     if (value == null)
199     {
200         return true;
201     }
202     var type = value.GetType();
203     return type.IsValueType ? value.Equals(Activator.CreateInstance(type)) : false;
204 }
205
206 private static Func<object, TLinkAddress> CompileUnboxAbsNegateAndConvertDelegate()
207 {
208     return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emitter =>
209     {
210         Ensure.Always.IsUnsignedInteger<TLinkAddress>();
211         emitter.LoadArgument(0);
212         var signedVersion = NumericType<TLinkAddress>.SignedVersion;
213     });
214 }

```

```

198     var signedVersionField =
199         ↳ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
200         ↳ BindingFlags.Static | BindingFlags.Public);
201     //emitter.LoadField(signedVersionField);
202     emitter.Emit(OpCodes.Ldsfld, signedVersionField);
203     var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
204         ↳ Types<object, Type>.Array);
205     emitter.Call(changeTypeMethod);
206     emitter.UnboxValue(signedVersion);
207     var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
208         ↳ signedVersion });
209     emitter.Call(absMethod);
210     var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate",
211         ↳ ").MakeGenericMethod(signedVersion);
212     emitter.Call(negateMethod);
213     var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
214         ↳ signedVersion });
215     emitter.Call(unsignedMethod);
216     emitter.Return();
217 });
218 }
219
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.LoadField(signedVersionField);
231         emitter.Emit(OpCodes.Ldsfld, signedVersionField);
232         var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
233             ↳ Types<object, Type>.Array);
234         emitter.Call(changeTypeMethod);
235         emitter.UnboxValue(signedVersion);
236         var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
237             ↳ signedVersion });
238         emitter.Call(absMethod);
239         var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
240             ↳ signedVersion });
241         emitter.Call(unsignedMethod);
242         emitter.Return();
243     });
244 }
245 }

```

1.6 ./Platform.Data/ILinks.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data
7  {
8      /// <summary>
9      /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
10     /// </summary>
11     /// <remarks>
12     /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит
13     ↳ подходит как для дуплетов, так и для триплетов и т.п.
14     /// Возможно этот интерфейс подходит даже для Sequences.
15     /// </remarks>
16     public interface ILinks<TLinkAddress, TConstants>
17     where TConstants : LinksConstants<TLinkAddress>
18     {
19         #region Constants
20
21         /// <summary>
22         /// Возвращает набор констант, который необходим для эффективной коммуникации с методами
23         ↳ этого интерфейса.
24         /// Эти константы не меняются с момента создания точки доступа к хранилищу.
25         /// </summary>
26         TConstants Constants { get; }

```

```

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

```

1.7 ./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>
88 /// <param name="links">Хранилище связей.</param>
89 /// <param name="link">Индекс связи.</param>
90 /// <returns>Уникальную связь.</returns>
91 [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 public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    ↳ TConstants> links, TLinkAddress link)
117 where TConstants : LinksConstants<TLinkAddress>
118 {
119     if (links.Constants.IsExternalReference(link))
120     {
121         return true;
122     }
123     links.EnsureLinkExists(link);
124     return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
125 }
126
127 /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
    ↳ точкой частично (связью замкнутой на себе как минимум один раз).</summary>
128 /// <param name="links">Хранилище связей.</param>
129 /// <param name="link">Индекс проверяемой связи.</param>
130 /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
131 /// <remarks>
132 /// Достаточно любой одной ссылки на себя.
133 /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
    ↳ ссылки на себя (на эту связь).
134 /// </remarks>
135 public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    ↳ TConstants> links, TLinkAddress link)
136 where TConstants : LinksConstants<TLinkAddress>
137 {
138     if (links.Constants.IsExternalReference(link))
139     {
140         return true;

```



```

141     }
142     links.EnsureLinkExists(link);
143     return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
144 }
145
146 #endregion
147 }
148 }

```

1.8 ./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 }

```

1.9 ./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 => 1;
41
42         public bool IsReadOnly => true;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public LinkAddress(TLinkAddress index) => Index = index;
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public void Add(TLinkAddress item) => throw new NotSupportedException();
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public void Clear() => throw new NotSupportedException();
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
55             ↳ ? true : false;
56     }
57 }

```

```

53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public IEnumerator<TLinkAddress> GetEnumerator()
58     {
59         yield return Index;
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
64         ↪ 0 : -1;
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     public bool Remove(TLinkAddress item) => throw new NotSupportedException();
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public void RemoveAt(int index) => throw new NotSupportedException();
74
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     IEnumerator IEnumerable.GetEnumerator()
77     {
78         yield return Index;
79     }
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
83         ↪ _equalityComparer.Equals(Index, other.Index);
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
87         ↪ linkAddress.Index;
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
91         ↪ new LinkAddress<TLinkAddress>(linkAddress);
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
95         ↪ ? Equals(linkAddress) : false;
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     public override int GetHashCode() => Index.GetHashCode();
99
100    [MethodImpl(MethodImplOptions.AggressiveInlining)]
101    public override string ToString() => Index.ToString();
102
103    [MethodImpl(MethodImplOptions.AggressiveInlining)]
104    public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
105        ↪ right)
106    {
107        if (left == null && right == null)
108            return true;
109        if (left == null)
110            return false;
111        return left.Equals(right);
112    }
113
114    [MethodImpl(MethodImplOptions.AggressiveInlining)]
115    public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
116        ↪ right) => !(left == right);
117
118 }

```

1.10 ./Platform.Data/LinksConstants.cs

```

1  using Platform.Ranges;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data
9  {
10     public class LinksConstants<TLinkAddress>

```

```

11 {
12     public static readonly int DefaultTargetPart = 2;
13
14     private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
15     private static readonly unchecked Converter<ulong, TLinkAddress>
16         ↳ _uInt64ToAddressConverter = unchecked Converter<ulong, TLinkAddress>.Default;
17
18     #region Link parts
19     /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
20     ↳ самой связи.</summary>
21     public int IndexPart { get; }
22
23     /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
24     ↳ часть-значение).</summary>
25     public int SourcePart { get; }
26
27     /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
28     ↳ (последняя часть-значение).</summary>
29     public int TargetPart { get; }
30
31     #endregion
32
33     #region Flow control
34     /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
35     /// <remarks>Используется в функции обработчике, который передаётся в функцию
36     ↳ Each.</remarks>
37     public TLinkAddress Continue { get; }
38
39     /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
40     public TLinkAddress Skip { get; }
41
42     /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
43     /// <remarks>Используется в функции обработчике, который передаётся в функцию
44     ↳ Each.</remarks>
45     public TLinkAddress Break { get; }
46
47     #endregion
48
49     #region Special symbols
50     /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
51     public TLinkAddress Null { get; }
52
53     /// <summary>Возвращает значение, обозначающее любую связь.</summary>
54     /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
55     ↳ создавать все варианты последовательностей в функции Create.</remarks>
56     public TLinkAddress Any { get; }
57
58     /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
59     public TLinkAddress Itself { get; }
60
61     #endregion
62
63     #region References
64     /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
65     ↳ ссылок).</summary>
66     public Range<TLinkAddress> InternalReferencesRange { get; }
67
68     /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
69     ↳ ссылок).</summary>
70     public Range<TLinkAddress>? ExternalReferencesRange { get; }
71
72     #endregion
73
74     public LinksConstants(int targetPart, Range<TLinkAddress>
75         ↳ possibleInternalReferencesRange, Range<TLinkAddress>?
76         ↳ possibleExternalReferencesRange)
77     {
78         IndexPart = 0;
79         SourcePart = 1;
80         TargetPart = targetPart;
81         Null = default;
82         Break = default;
83         var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
84         Continue = currentInternalReferenceIndex;
85         Decrement(ref currentInternalReferenceIndex);
86         Skip = currentInternalReferenceIndex;
87     }
88 }

```

```

80     Decrement(ref currentInternalReferenceIndex);
81     Any = currentInternalReferenceIndex;
82     Decrement(ref currentInternalReferenceIndex);
83     Itself = currentInternalReferenceIndex;
84     Decrement(ref currentInternalReferenceIndex);
85     InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
86     ↪ currentInternalReferenceIndex);
87     ExternalReferencesRange = possibleExternalReferencesRange;
88 }
89
90 public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
91     ↪ this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
92     ↪ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
93
94 public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
95     ↪ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
96     ↪ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
97
98 public LinksConstants(bool enableExternalReferencesSupport) :
99     ↪ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
100     ↪ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
101
102 public LinksConstants(int targetPart, Range<TLinkAddress>
103     ↪ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
104     ↪ null) { }
105
106 public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
107     ↪ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
108
109 public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
110     ↪ false) { }
111
112 public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
113     ↪ enableExternalReferencesSupport)
114 {
115     if (enableExternalReferencesSupport)
116     {
117         return (_one, _UInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumberValue,
118         ↪ rValuesRange));
119     }
120     else
121     {
122         return (_one, NumericType<TLinkAddress>.MaxValue);
123     }
124 }
125
126 public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
127     ↪ enableExternalReferencesSupport)
128 {
129     if (enableExternalReferencesSupport)
130     {
131         return (_UInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumberValue,
132         ↪ sRange + 1UL),
133         ↪ NumericType<TLinkAddress>.MaxValue);
134     }
135     else
136     {
137         return null;
138     }
139 }
140
141 private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
142     ↪ currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
143 }
144 }

```

1.11 ./Platform.Data/LinksConstantsExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data
6  {
7      public static class LinksConstantsExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```

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

```

1 using Platform.Converters;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Numbers.Raw
6 {
7     public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
8     {
9         static private readonly UncheckedConverter<long, TLink> _converter =
    ↪ UncheckedConverter<long, TLink>.Default;
10
11         public TLink Convert(TLink source) => _converter.Convert(new
    ↪ Hybrid<TLink>(source).AbsoluteValue);
12    }
13 }

```

1.14 ./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 =
    ↪ EqualityComparer<TLinkAddress>.Default;
16
17         public TLinkAddress Index
18         {
19             [MethodImpl(MethodImplOptions.AggressiveInlining)]
20             get;
21         }
22
23         public int Size
24         {
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             get;
27         }
28
29         public TLinkAddress this[int index]
30         {
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             get

```

```

33     {
34         if (index < Size)
35         {
36             return Index;
37         }
38         else
39         {
40             throw new IndexOutOfRangeException();
41         }
42     }
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     set => throw new NotSupportedException();
45 }
46
47 public int Count => int.MaxValue;
48
49 public bool IsReadOnly => true;
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public Point(TLinkAddress index, int size)
53 {
54     Index = index;
55     Size = size;
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public void Add(TLinkAddress item) => throw new NotSupportedException();
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public void Clear() => throw new NotSupportedException();
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
66     ↪ ? true : false;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public IEnumerator<TLinkAddress> GetEnumerator()
73 {
74     for (int i = 0; i < Size; i++)
75     {
76         yield return Index;
77     }
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
82     ↪ 0 : -1;
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public bool Remove(TLinkAddress item) => throw new NotSupportedException();
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public void RemoveAt(int index) => throw new NotSupportedException();
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 IEnumerator IEnumerable.GetEnumerator()
95 {
96     for (int i = 0; i < Size; i++)
97     {
98         yield return Index;
99     }
100 }
101
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
104     ↪ _equalityComparer.Equals(Index, other.Index);
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
108     ↪ linkAddress.Index;
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
112     ↪ Equals(linkAddress) : false;

```

```

108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public override int GetHashCode() => Index.GetHashCode();
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public override string ToString() => Index.ToString();
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
116 {
117     if (left == null && right == null)
118         return true;
119     if (left == null)
120         return false;
121     return left.Equals(right);
122 }
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
126     ↪ !(left == right);
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static bool IsFullPoint(params TLinkAddress[] link) =>
130     ↪ IsFullPoint((IList<TLinkAddress>)link);
131
132 [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 public static bool IsFullPoint(IList<TLinkAddress> link)
134 {
135     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
136     Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
137         ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
138     return IsFullPointUnchecked(link);
139 }
140
141 [MethodImpl(MethodImplOptions.AggressiveInlining)]
142 public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
143 {
144     var result = true;
145     for (var i = 1; result && i < link.Count; i++)
146     {
147         result = _equalityComparer.Equals(link[0], link[i]);
148     }
149     return result;
150 }
151
152 [MethodImpl(MethodImplOptions.AggressiveInlining)]
153 public static bool IsPartialPoint(params TLinkAddress[] link) =>
154     ↪ IsPartialPoint((IList<TLinkAddress>)link);
155
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 public static bool IsPartialPoint(IList<TLinkAddress> link)
158 {
159     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
160     Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
161         ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
162     return IsPartialPointUnchecked(link);
163 }
164
165 [MethodImpl(MethodImplOptions.AggressiveInlining)]
166 public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
167 {
168     var result = false;
169     for (var i = 1; !result && i < link.Count; i++)
170     {
171         result = _equalityComparer.Equals(link[0], link[i]);
172     }
173     return result;
174 }
175 }

```

1.15 ./Platform.Data/Sequences/ISequenceAppender.cs

```

1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 namespace Platform.Data.Sequences
4 {
5     public interface ISequenceAppender<TLinkAddress>
6     {
7         TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant);
8     }
9 }

```

```
8     }
9 }
```

1.16 ./Platform.Data/Sequences/ISequenceWalker.cs

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

1.17 ./Platform.Data/Sequences/SequenceWalker.cs

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



```

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 }

```

1.18 ./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
    ↪ себя),
12     /// так как стек можно использовать намного эффективнее при ручном управлении.
13     ///
14     /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
15     /// Решить встраивать ли защиту от заикливания.
16     /// Альтернативой защиты от заикливания может быть заранее известное ограничение на
    ↪ погружение вглубь.
17     /// А так же качественное распознавание прохода по циклическому графу.
18     /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер
    ↪ стека.
19     /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
20     /// </remarks>
21     public static class StopableSequenceWalker
22     {
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

24 public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↳ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↳ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> enter, Action<TLinkAddress>
    ↳ exit, Func<TLinkAddress, bool> canEnter, Func<TLinkAddress, bool> visit)
25 {
26     var exited = 0;
27     var stack = new Stack<TLinkAddress>();
28     var element = sequence;
29     if (isElement(element))
30     {
31         return visit(element);
32     }
33     while (true)
34     {
35         if (isElement(element))
36         {
37             if (stack.Count == 0)
38             {
39                 return true;
40             }
41             element = stack.Pop();
42             exit(element);
43             exited++;
44             var source = getSource(element);
45             var target = getTarget(element);
46             if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
    ↳ !visit(source))
47             {
48                 return false;
49             }
50             if ((isElement(target) || !canEnter(target)) && !visit(target))
51             {
52                 return false;
53             }
54             element = target;
55         }
56         else
57         {
58             if (canEnter(element))
59             {
60                 enter(element);
61                 exited = 0;
62                 stack.Push(element);
63                 element = getSource(element);
64             }
65             else
66             {
67                 if (stack.Count == 0)
68                 {
69                     return true;
70                 }
71                 element = stack.Pop();
72                 exit(element);
73                 exited++;
74                 var source = getSource(element);
75                 var target = getTarget(element);
76                 if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
    ↳ !visit(source))
77                 {
78                     return false;
79                 }
80                 if ((isElement(target) || !canEnter(target)) && !visit(target))
81                 {
82                     return false;
83                 }
84                 element = target;
85             }
86         }
87     }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↳ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↳ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
92 {
93     var stack = new Stack<TLinkAddress>();
94     var element = sequence;
95     if (isElement(element))

```

```

96     {
97         return visit(element);
98     }
99     while (true)
100     {
101         if (isElement(element))
102         {
103             if (stack.Count == 0)
104             {
105                 return true;
106             }
107             element = stack.Pop();
108             var source = getSource(element);
109             var target = getTarget(element);
110             if (isElement(source) && !visit(source))
111             {
112                 return false;
113             }
114             if (isElement(target) && !visit(target))
115             {
116                 return false;
117             }
118             element = target;
119         }
120         else
121         {
122             stack.Push(element);
123             element = getSource(element);
124         }
125     }
126 }
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static bool WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↪ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↪ Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
130 {
131     var stack = new Stack<TLinkAddress>();
132     var element = sequence;
133     if (isElement(element))
134     {
135         return visit(element);
136     }
137     while (true)
138     {
139         if (isElement(element))
140         {
141             if (stack.Count == 0)
142             {
143                 return true;
144             }
145             element = stack.Pop();
146             var source = getSource(element);
147             var target = getTarget(element);
148             if (isElement(target) && !visit(target))
149             {
150                 return false;
151             }
152             if (isElement(source) && !visit(source))
153             {
154                 return false;
155             }
156             element = source;
157         }
158         else
159         {
160             stack.Push(element);
161             element = getTarget(element);
162         }
163     }
164 }
165 }
166 }

```

1.19 ./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 }

```

1.20 ./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 }

```

1.21 ./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 }

```

1.22 ./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>
52         TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
53     }
54 }

```

1.23 ./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
    ↪ 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 }

```

1.24 ./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 }

```

1.25 ./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,
    ↪ 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, 22
- ./Platform.Data/Exceptions/ArgumentLinkDoesNotExistsException.cs, 1
- ./Platform.Data/Exceptions/ArgumentLinkHasDependenciesException.cs, 1
- ./Platform.Data/Exceptions/LinkWithSameValueAlreadyExistsException.cs, 1
- ./Platform.Data/Exceptions/LinksLimitReachedException.cs, 1
- ./Platform.Data/Hybrid.cs, 1
- ./Platform.Data/ILinks.cs, 5
- ./Platform.Data/ILinksExtensions.cs, 6
- ./Platform.Data/ISynchronizedLinks.cs, 9
- ./Platform.Data/LinkAddress.cs, 9
- ./Platform.Data/LinksConstants.cs, 10
- ./Platform.Data/LinksConstantsExtensions.cs, 12
- ./Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs, 13
- ./Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs, 13
- ./Platform.Data/Point.cs, 13
- ./Platform.Data/Sequences/ISequenceAppender.cs, 15
- ./Platform.Data/Sequences/ISequenceWalker.cs, 16
- ./Platform.Data/Sequences/SequenceWalker.cs, 16
- ./Platform.Data/Sequences/StopableSequenceWalker.cs, 17
- ./Platform.Data/Universal/IUniLinks.cs, 19
- ./Platform.Data/Universal/IUniLinksCRUD.cs, 20
- ./Platform.Data/Universal/IUniLinksGS.cs, 20
- ./Platform.Data/Universal/IUniLinksIO.cs, 21
- ./Platform.Data/Universal/IUniLinksIOWithExtensions.cs, 21
- ./Platform.Data/Universal/IUniLinksRW.cs, 22