

## 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 argumentName) :
10             ↪ base(FormatMessage(link, argumentName), argumentName) { }
11
12         public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link))
13             ↪ { }
14
15         public ArgumentLinkDoesNotExistsException(string message, Exception innerException) :
16             ↪ base(message, innerException) { }
17
18         public ArgumentLinkDoesNotExistsException(string message) : base(message) { }
19
20         public ArgumentLinkDoesNotExistsException() { }
21
22         private static string FormatMessage(TLinkAddress link, string argumentName) => $"Связь
23             ↪ [{link}] переданная в аргумент [{argumentName}] не существует.";
24
25         private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
26             ↪ качестве аргумента не существует.";
27     }
28 }
```

### 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
12         public ArgumentLinkHasDependenciesException(TLinkAddress link) :
13             ↪ base(FormatMessage(link)) { }
14
15         public ArgumentLinkHasDependenciesException(string message, Exception innerException) :
16             ↪ base(message, innerException) { }
17
18         public ArgumentLinkHasDependenciesException(string message) : base(message) { }
19
20         public ArgumentLinkHasDependenciesException() { }
21
22         private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи
23             ↪ [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые
24             ↪ препятствуют изменению её внутренней структуры.";
25
26         private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной
27             ↪ в качестве аргумента присутствуют зависимости, которые препятствуют изменению её
28             ↪ внутренней структуры.";
29     }
30 }
```

### 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 const string DefaultMessage = "Связь с таким же значением уже существует.";
10
11         public LinkWithSameValueAlreadyExistsException(string message, Exception innerException)
12             ↪ : base(message, innerException) { }
13
14         public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }
15     }
16 }
```

```

15         public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
16     }
17 }

```

#### 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> : LinksLimitReachedExceptionBase
8     {
9         public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }
10
11         public LinksLimitReachedException(string message, Exception innerException) :
12             → base(message, innerException) { }
13
14         public LinksLimitReachedException(string message) : base(message) { }
15
16         public LinksLimitReachedException() : base(DefaultMessage) { }
17
18         private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества
19             → связей в хранилище ({limit}).";
20     }
21 }

```

#### 1.5 ./Platform.Data/Exceptions/LinksLimitReachedExceptionBase.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 abstract class LinksLimitReachedExceptionBase : Exception
8     {
9         public static readonly string DefaultMessage = "Достигнут лимит количества связей в
10             → хранилище.";
11
12         protected LinksLimitReachedExceptionBase(string message, Exception innerException) :
13             → base(message, innerException) { }
14
15         protected LinksLimitReachedExceptionBase(string message) : base(message) { }
16     }
17 }

```

#### 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 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     }
34 }

```

```

25     public static readonly TLinkAddress ExternalZero =
26         ↪ _uInt64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);
27
28     public readonly TLinkAddress Value;
29
30     public bool IsNothing
31     {
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
34     }
35
36     public bool IsInternal
37     {
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         get => SignedValue > 0;
40     }
41
42     public bool IsExternal
43     {
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
46     }
47
48     public long SignedValue
49     {
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         get => _addressToInt64Converter.Convert(Value);
52     }
53
54     public long AbsoluteValue
55     {
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
58             ↪ Platform.Numbers.Math.Abs(SignedValue);
59     }
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public Hybrid(TLinkAddress value)
63     {
64         Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
65         Value = value;
66     }
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public Hybrid(TLinkAddress value, bool isExternal)
70     {
71         if (_equalityComparer.Equals(value, default) && isExternal)
72         {
73             Value = ExternalZero;
74         }
75         else
76         {
77             if (isExternal)
78             {
79                 Value = Math<TLinkAddress>.Negate(value);
80             }
81             else
82             {
83                 Value = value;
84             }
85         }
86     }
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     public Hybrid(object value) => Value =
90         ↪ To.UnsignedAs<TLinkAddress>(Convert.ChangeType(value,
91         ↪ NumericType<TLinkAddress>.SignedVersion));
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public Hybrid(object value, bool isExternal)
95     {
96         if (IsDefault(value) && isExternal)
97         {
98             Value = ExternalZero;
99         }
100         else
101         {
102             if (isExternal)
103             {

```

```

        Value = _unboxAbsNegateAndConvert(value);
    }
    else
    {
        Value = _unboxAbsAndConvert(value);
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(long integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(int integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(short integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
    ↪ Hybrid<TLinkAddress>(integer);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
    ↪ hybrid.Value;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
    ↪ CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
    ↪ hybrid.AbsoluteValue;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
    ↪ CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
    ↪ (int)hybrid.AbsoluteValue;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
    ↪ CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
    ↪ (short)hybrid.AbsoluteValue;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
    ↪ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);

[MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

161 public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
162     ↪ (sbyte)hybrid.AbsoluteValue;
163
164 [MethodImpl(MethodImplOptions.AggressiveInlining)]
165 public override string ToString() => IsExternal ? $"<{AbsoluteValue}>" :
166     ↪ Value.ToString();
167
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
170     ↪ other.Value);
171
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
174     ↪ Equals(hybrid) : false;
175
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public override int GetHashCode() => Value.GetHashCode();
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
181     ↪ left.Equals(right);
182
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
185     ↪ !(left == right);
186
187 private static bool IsDefault(object value)
188 {
189     if (value == null)
190     {
191         return true;
192     }
193     var type = value.GetType();
194     return type.IsValueType ? value.Equals(Activator.CreateInstance(type)) : false;
195 }
196
197 private static Func<object, TLinkAddress> CompileUnboxAbsNegateAndConvertDelegate()
198 {
199     return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emitter =>
200     {
201         Ensure.Always.IsUnsignedInteger<TLinkAddress>();
202         emitter.LoadArgument(0);
203         var signedVersion = NumericType<TLinkAddress>.SignedVersion;
204         var signedVersionField =
205             ↪ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
206             ↪ BindingFlags.Static | BindingFlags.Public);
207         emitter.Emit(OpCodes.Ldsfld, signedVersionField);
208         var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
209             ↪ Types<object, Type>.Array);
210         emitter.Call(changeTypeMethod);
211         emitter.UnboxValue(signedVersion);
212         var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
213             ↪ signedVersion });
214         emitter.Call(absMethod);
215         var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate",
216             ↪ ").MakeGenericMethod(signedVersion);
217         emitter.Call(negateMethod);
218         var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
219             ↪ signedVersion });
220         emitter.Call(unsignedMethod);
221         emitter.Return();
222     }));
223 }
224
225 private static Func<object, TLinkAddress> CompileUnboxAbsAndConvertDelegate()
226 {
227     return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emitter =>
228     {
229         Ensure.Always.IsUnsignedInteger<TLinkAddress>();
230         emitter.LoadArgument(0);
231         var signedVersion = NumericType<TLinkAddress>.SignedVersion;
232         var signedVersionField =
233             ↪ typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
234             ↪ BindingFlags.Static | BindingFlags.Public);
235         emitter.Emit(OpCodes.Ldsfld, signedVersionField);
236         var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
237             ↪ Types<object, Type>.Array);

```

```

223         emitter.Call(changeTypeMethod);
224         emitter.UnboxValue(signedVersion);
225         var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
226             ↳ signedVersion });
227         emitter.Call(absMethod);
228         var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
229             ↳ signedVersion });
230         emitter.Call(unsignedMethod);
231         emitter.Return();
232     });
233 }

```

## 1.7 ./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; }
27
28         #endregion
29
30         #region Read
31
32         /// <summary>
33         /// Подсчитывает и возвращает общее число связей находящихся в хранилище,
34         ↳ соответствующих указанным ограничениям.
35         /// </summary>
36         /// <param name="restriction">Ограничения на содержимое связей.</param>
37         /// <returns>Общее число связей находящихся в хранилище, соответствующих указанным
38         ↳ ограничениям.</returns>
39         TLinkAddress Count(IList<TLinkAddress> restriction);
40
41         /// <summary>
42         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
43         ↳ (handler) для каждой подходящей связи.
44         /// </summary>
45         /// <param name="handler">Обработчик каждой подходящей связи.</param>
46         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
47         ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
48         ↳ Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
49         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
50         ↳ случае.</returns>
51         TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler, IList<TLinkAddress>
52             ↳ restrictions);
53
54         #endregion
55
56         #region Write
57
58         /// <summary>
59         /// Создаёт связь.
60         /// </summary>
61         /// <returns>Индекс созданной связи.</returns>
62         TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно
63             ↳ принимать restrictions, возможно и возвращать связь нужно целиком.
64     }
65 }

```

```

55     /// <summary>
56     /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
57     /// на связь с указанным новым содержимым.
58     /// </summary>
59     /// <param name="restrictions">
60     /// Ограничения на содержимое связей.
61     /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
62     /// → и далее за ним будет следовать содержимое связи.
63     /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
64     /// → ссылку на пустоту,
65     /// Constants.Itself - требование установить ссылку на себя, 1.. $\infty$  конкретный индекс
66     /// → другой связи.
67     /// </param>
68     /// <param name="substitution"></param>
69     /// <returns>Индекс обновлённой связи.</returns>
70     TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
71     → // TODO: Возможно и возвращать связь нужно целиком.
72
73     /// <summary>Удаляет связь с указанным индексом.</summary>
74     void Delete(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно принимать
75     → restrictions, а так же возвращать удалённую связь, если удаление было реально
76     → выполнено, и Null, если нет.
77
78     #endregion
79 }
80 }

```

## 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 }

```

```

46
47 /// <param name="links">Хранилище связей.</param>
48 /// <param name="link">Индекс проверяемой на существование связи.</param>
49 /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    ↳ TConstants> links, TLinkAddress link, string argumentName)
52     where TConstants : LinksConstants<TLinkAddress>
53 {
54     if (!links.Exists(link))
55     {
56         throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
57     }
58 }
59
60 /// <summary>
61 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
62 /// </summary>
63 /// <param name="links">Хранилище связей.</param>
64 /// <param name="handler">Обработчик каждой подходящей связи.</param>
65 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
66 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
    ↳ TConstants> links, Func<IList<TLinkAddress>, TLinkAddress> handler, params
    ↳ TLinkAddress[] restrictions)
    where TConstants : LinksConstants<TLinkAddress>
    => links.Each(handler, restrictions);
69
70
71
72 /// <summary>
73 /// Возвращает части-значения для связи с указанным индексом.
74 /// </summary>
75 /// <param name="links">Хранилище связей.</param>
76 /// <param name="link">Индекс связи.</param>
77 /// <returns>Уникальную связь.</returns>
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
    ↳ ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
    where TConstants : LinksConstants<TLinkAddress>
80 {
81     var constants = links.Constants;
82     if (constants.IsExternalReference(link))
83     {
84         return new Point<TLinkAddress>(link, constants.TargetPart + 1);
85     }
86     var linkPartsSetter = new Setter<IList<TLinkAddress>,
    ↳ TLinkAddress>(constants.Continue, constants.Break);
87     links.Each(linkPartsSetter.SetAndReturnTrue, link);
88     return linkPartsSetter.Result;
89 }
90
91 #region Points
92
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 ,
111     /// т.е. ссылка не на себя а во вне.
112     ///
113     /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
114     /// промежуточную связь,
115     /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
116     /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
117     /// </remarks>
118     public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
119     ↪ TConstants> links, TLinkAddress link)
120     where TConstants : LinksConstants<TLinkAddress>
121     {
122         if (links.Constants.IsExternalReference(link))
123         {
124             return true;
125         }
126         links.EnsureLinkExists(link);
127         return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
128     }
129
130     /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
131     ↪ точкой частично (связью замкнутой на себе как минимум один раз).</summary>
132     /// <param name="links">Хранилище связей.</param>
133     /// <param name="link">Индекс проверяемой связи.</param>
134     /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
135     /// <remarks>
136     /// Достаточно любой одной ссылки на себя.
137     /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
138     ↪ ссылки на себя (на эту связь).
139     /// </remarks>
140     public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
141     ↪ TConstants> links, TLinkAddress link)
142     where TConstants : LinksConstants<TLinkAddress>
143     {
144         if (links.Constants.IsExternalReference(link))
145         {
146             return true;
147         }
148         links.EnsureLinkExists(link);
149         return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
150     }
151
152     #endregion
153 }
154 }

```

## 1.9 ./Platform.Data/ISynchronizedLinks.cs

```

1 using Platform.Threading.Synchronization;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data
6 {
7     public interface ISynchronizedLinks<TLinkAddress, TLinks, TConstants> :
8     ↪ ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
9     where TLinks : ILinks<TLinkAddress, TConstants>
10     where TConstants : LinksConstants<TLinkAddress>
11     {
12     }
13 }

```

## 1.10 ./Platform.Data/LinkAddress.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data
9 {
10     public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
11     ↪ IList<TLinkAddress>
12     {
13         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
14         ↪ EqualityComparer<TLinkAddress>.Default;
15     }
16 }

```

```

13
14 public TLinkAddress Index
15 {
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     get;
18 }
19
20 public TLinkAddress this[int index]
21 {
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     get
24     {
25         if (index == 0)
26         {
27             return Index;
28         }
29         else
30         {
31             throw new IndexOutOfRangeException();
32         }
33     }
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     set => throw new NotSupportedException();
36 }
37
38 public int Count => 1;
39
40 public bool IsReadOnly => true;
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public LinkAddress(TLinkAddress index) => Index = index;
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public void Add(TLinkAddress item) => throw new NotSupportedException();
47
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public void Clear() => throw new NotSupportedException();
50
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
53     ↪ ? true : false;
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public IEnumerator<TLinkAddress> GetEnumerator()
60 {
61     yield return Index;
62 }
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
66     ↪ 0 : -1;
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public bool Remove(TLinkAddress item) => throw new NotSupportedException();
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public void RemoveAt(int index) => throw new NotSupportedException();
76
77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 IEnumerator IEnumerable.GetEnumerator()
79 {
80     yield return Index;
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
85     ↪ _equalityComparer.Equals(Index, other.Index);
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
89     ↪ linkAddress.Index;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

88     public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
89         ↪ new LinkAddress<TLinkAddress>(linkAddress);
90
91     [MethodImpl(MethodImplOptions.AggressiveInlining)]
92     public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
93         ↪ ? Equals(linkAddress) : false;
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public override int GetHashCode() => Index.GetHashCode();
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public override string ToString() => Index.ToString();
100
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
103         ↪ right)
104     {
105         if (left == null && right == null)
106         {
107             return true;
108         }
109         if (left == null)
110         {
111             return false;
112         }
113         return left.Equals(right);
114     }
115
116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
117     public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
118         ↪ right) => !(left == right);
119 }

```

### 1.11 ./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 const int DefaultTargetPart = 2;
13
14         private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
15         private static readonly UncheckedConverter<ulong, TLinkAddress>
16             ↪ _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
17
18         #region Link parts
19
20         /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
21         ↪ самой связи.</summary>
22         public int IndexPart { get; }
23
24         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
25         ↪ часть-значение).</summary>
26         public int SourcePart { get; }
27
28         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
29         ↪ (последняя часть-значение).</summary>
30         public int TargetPart { get; }
31
32         #endregion
33
34         #region Flow control
35
36         /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
37         /// <remarks>Используется в функции обработчике, который передаётся в функцию
38         ↪ Each.</remarks>
39         public TLinkAddress Continue { get; }
40
41         /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
42         public TLinkAddress Skip { get; }
43
44         /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>

```

```

40  /// <remarks>Используется в функции обработчике, который передаётся в функцию
    ↳ Each.</remarks>
41  public TLinkAddress Break { get; }
42
43  #endregion
44
45  #region Special symbols
46
47  /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
48  public TLinkAddress Null { get; }
49
50  /// <summary>Возвращает значение, обозначающее любую связь.</summary>
51  /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
    ↳ создавать все варианты последовательностей в функции Create.</remarks>
52  public TLinkAddress Any { get; }
53
54  /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
55  public TLinkAddress Itself { get; }
56
57  #endregion
58
59  #region References
60
61  /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
    ↳ ссылок).</summary>
62  public Range<TLinkAddress> InternalReferencesRange { get; }
63
64  /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
    ↳ ссылок).</summary>
65  public Range<TLinkAddress>? ExternalReferencesRange { get; }
66
67  #endregion
68
69  public LinksConstants(int targetPart, Range<TLinkAddress>
    ↳ possibleInternalReferencesRange, Range<TLinkAddress>?
    ↳ possibleExternalReferencesRange)
70  {
71      IndexPart = 0;
72      SourcePart = 1;
73      TargetPart = targetPart;
74      Null = default;
75      Break = default;
76      var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
77      Continue = currentInternalReferenceIndex;
78      Decrement(ref currentInternalReferenceIndex);
79      Skip = currentInternalReferenceIndex;
80      Decrement(ref currentInternalReferenceIndex);
81      Any = currentInternalReferenceIndex;
82      Decrement(ref currentInternalReferenceIndex);
83      Itself = currentInternalReferenceIndex;
84      Decrement(ref currentInternalReferenceIndex);
85      InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
    ↳ currentInternalReferenceIndex);
86      ExternalReferencesRange = possibleExternalReferencesRange;
87  }
88
89  public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
    ↳ this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
    ↳ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
90
91  public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
    ↳ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
    ↳ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
92
93  public LinksConstants(bool enableExternalReferencesSupport) :
    ↳ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
    ↳ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
94
95  public LinksConstants(int targetPart, Range<TLinkAddress>
    ↳ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
    ↳ null) { }
96
97  public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
    ↳ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
98
99  public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
    ↳ false) { }

```

```

101     public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
        ↪ enableExternalReferencesSupport)
102     {
103         if (enableExternalReferencesSupport)
104         {
105             return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
        ↪ rValuesRange));
106         }
107         else
108         {
109             return (_one, NumericType<TLinkAddress>.MaxValue);
110         }
111     }
112
113     public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
        ↪ enableExternalReferencesSupport)
114     {
115         if (enableExternalReferencesSupport)
116         {
117             return (_uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumberValue
        ↪ sRange + 1UL),
        ↪ NumericType<TLinkAddress>.MaxValue);
118         }
119         else
120         {
121             return null;
122         }
123     }
124
125     private static void Decrement(ref TLinkAddress currentInternalReferenceIndex) =>
        ↪ currentInternalReferenceIndex = Arithmetic.Decrement(currentInternalReferenceIndex);
126 }
127 }

```

## 1.12 ./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 }

```

## 1.13 ./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 }

```

## 1.14 ./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 =
10             ↪ UncheckedConverter<long, TLink>.Default;
11
12         public TLink Convert(TLink source) => _converter.Convert(new
13             ↪ Hybrid<TLink>(source).AbsoluteValue);
14     }
15 }

```

### 1.15 ./Platform.Data/Point.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Ranges;
7 using Platform.Collections;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data
12 {
13     public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
14     {
15         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
16             ↪ EqualityComparer<TLinkAddress>.Default;
17
18         public TLinkAddress Index
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23
24         public int Size
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         public TLinkAddress this[int index]
31         {
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             get
34             {
35                 if (index < Size)
36                 {
37                     return Index;
38                 }
39                 else
40                 {
41                     throw new IndexOutOfRangeException();
42                 }
43             }
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set => throw new NotSupportedException();
46         }
47
48         public int Count => int.MaxValue;
49
50         public bool IsReadOnly => true;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public Point(TLinkAddress index, int size)
54         {
55             Index = index;
56             Size = size;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public void Add(TLinkAddress item) => throw new NotSupportedException();
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public void Clear() => throw new NotSupportedException();
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
67             ↪ ? true : false;
68     }
69 }

```

```

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

```

```

139     Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
140         ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
141     return IsFullPointUnchecked(link);
142 }
143 [MethodImpl(MethodImplOptions.AggressiveInlining)]
144 public static bool IsFullPointUnchecked(ICollection<TLinkAddress> link)
145 {
146     var result = true;
147     for (var i = 1; result && i < link.Count; i++)
148     {
149         result = _equalityComparer.Equals(link[0], link[i]);
150     }
151     return result;
152 }
153
154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 public static bool IsPartialPoint(params TLinkAddress[] link) =>
156     IsPartialPoint((ICollection<TLinkAddress>)link);
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static bool IsPartialPoint(ICollection<TLinkAddress> link)
160 {
161     Ensure.Always.ArgumentNotEmpty(link, nameof(link));
162     Ensure.Always.ArgumentInRange(link.Count, new Range<int>(2, int.MaxValue),
163         ↪ nameof(link), "Cannot determine link's pointness using only its identifier.");
164     return IsPartialPointUnchecked(link);
165 }
166
167 [MethodImpl(MethodImplOptions.AggressiveInlining)]
168 public static bool IsPartialPointUnchecked(ICollection<TLinkAddress> link)
169 {
170     var result = false;
171     for (var i = 1; !result && i < link.Count; i++)
172     {
173         result = _equalityComparer.Equals(link[0], link[i]);
174     }
175     return result;
176 }

```

#### 1.16 ./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  }

```

#### 1.17 ./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<ICollection<TLinkAddress>> Walk(TLinkAddress sequence);
10     }
11 }

```

#### 1.18 ./Platform.Data/Sequences/SequenceWalker.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Sequences
8  {
9      /// <remarks>
10     /// Реализованный внутри алгоритм наглядно показывает,
11     /// что совершенно не обязательно рекурсивная реализация (с вложенным вызовом функцией самой
12     /// ↪ себя),

```



```

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 SequenceWalker
22  {
23      [MethodImpl(MethodImplOptions.AggressiveInlining)]
24      public static void WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↳ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↳ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
25      {
26          var stack = new Stack<TLinkAddress>();
27          var element = sequence;
28          if (isElement(element))
29          {
30              visit(element);
31          }
32          else
33          {
34              while (true)
35              {
36                  if (isElement(element))
37                  {
38                      if (stack.Count == 0)
39                      {
40                          break;
41                      }
42                      element = stack.Pop();
43                      var source = getSource(element);
44                      var target = getTarget(element);
45                      if (isElement(source))
46                      {
47                          visit(source);
48                      }
49                      if (isElement(target))
50                      {
51                          visit(target);
52                      }
53                      element = target;
54                  }
55                  else
56                  {
57                      stack.Push(element);
58                      element = getSource(element);
59                  }
60              }
61          }
62      }
63
64      [MethodImpl(MethodImplOptions.AggressiveInlining)]
65      public static void WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
    ↳ TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
    ↳ Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
66      {
67          var stack = new Stack<TLinkAddress>();
68          var element = sequence;
69          if (isElement(element))
70          {
71              visit(element);
72          }
73          else
74          {
75              while (true)
76              {
77                  if (isElement(element))
78                  {
79                      if (stack.Count == 0)
80                      {
81                          break;
82                      }
83                      element = stack.Pop();

```

```

84         var source = getSource(element);
85         var target = getTarget(element);
86         if (isElement(target))
87         {
88             visit(target);
89         }
90         if (isElement(source))
91         {
92             visit(source);
93         }
94         element = source;
95     }
96     else
97     {
98         stack.Push(element);
99         element = getTarget(element);
100     }
101 }
102 }
103 }
104 }
105 }

```

### 1.19 ./Platform.Data/Sequences/StopableSequenceWalker.cs

```

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

```

```

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 }

```

## 1.20 ./Platform.Data/Universal/IUniLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
10     public partial interface IUniLinks<TLinkAddress>
11     {
12         IList<IList<IList<TLinkAddress>>> Trigger(IList<TLinkAddress> condition,
            ↪ IList<TLinkAddress> substitution);
13     }
14
15     /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
16     public partial interface IUniLinks<TLinkAddress>
17     {
18         /// <returns>
19         /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
20         ↪ represents False (was stopped).
21         /// This is done to assure ability to push up stop signal through recursion stack.
22         /// </returns>
23         /// <remarks>
24         /// { 0, 0, 0 } => { itself, itself, itself } // create
25         /// { 1, any, any } => { itself, any, 3 } // update
26         /// { 3, any, any } => { 0, 0, 0 } // delete
27         /// </remarks>
28         TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>,
            ↪ TLinkAddress> matchHandler,
29             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>,
            ↪ IList<TLinkAddress>, TLinkAddress> substitutionHandler);

```

```

30         TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>,
    ↪      IList<TLinkAddress>, TLinkAddress> matchedHandler,
31         IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>,
    ↪      TLinkAddress> substitutedHandler);
32     }
33
34     /// <remarks>Extended with small optimization.</remarks>
35     public partial interface IUniLinks<TLinkAddress>
36     {
37         /// <remarks>
38         /// Something simple should be simple and optimized.
39         /// </remarks>
40         TLinkAddress Count(IList<TLinkAddress> restrictions);
41     }
42 }

```

#### 1.21 ./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 }

```

#### 1.22 ./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 }

```

#### 1.23 ./Platform.Data/Universal/IUniLinksIO.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>
10     /// In/Out aliases for IUniLinks.
11     /// TLinkAddress can be any number type of any size.
12     /// </remarks>
13     public interface IUniLinksIO<TLinkAddress>
14     {
15         /// <remarks>
16         /// default(TLinkAddress) means any link.
17         /// Single element pattern means just element (link).
18         /// Handler gets array of link contents.
19         /// * link[0] is index or identifier.
20         /// * link[1] is source or first.

```

```

21     /// * link[2] is target or second.
22     /// * link[3] is linker or third.
23     /// * link[n] is nth part/parent/element/value
24     /// of link (if variable length links used).
25     ///
26     /// Stops and returns false if handler return false.
27     ///
28     /// Acts as Each, Foreach, Select, Search, Match &...
29     ///
30     /// Handles all links in store if pattern/restrictions is not defined.
31     /// </remarks>
32     bool Out(Func<IList<TLinkAddress>, bool> handler, IList<TLinkAddress> pattern);
33
34     /// <remarks>
35     /// default(TLinkAddress) means itself.
36     /// Equivalent to:
37     /// * creation if before == null
38     /// * deletion if after == null
39     /// * update if before != null && after != null
40     /// * default(TLinkAddress) if before == null && after == null
41     ///
42     /// Possible interpretation
43     /// * In(null, new[] { }) creates point (link that points to itself using minimum number
44     ///   ↳ of parts).
45     /// * In(new[] { 4 }, null) deletes 4th link.
46     /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
47     ///   ↳ 5th index.
48     /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with
49     ///   ↳ 2 as source and 3 as target), 0 means it can be placed in any address.
50     /// ...
51     /// </remarks>
    TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
}
}

```

#### 1.24 ./Platform.Data/Universal/IUniLinksIOWithExtensions.cs

```

1  // ReSharper disable TypeParameterCanBeVariant
2  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
4  using System.Collections.Generic;
5
6  namespace Platform.Data.Universal
7  {
8      /// <remarks>Contains some optimizations of Out.</remarks>
9      public interface IUniLinksIOWithExtensions<TLinkAddress> : IUniLinksIO<TLinkAddress>
10     {
11         /// <remarks>
12         /// default(TLinkAddress) means nothing or null.
13         /// Single element pattern means just element (link).
14         /// OutPart(n, null) returns default(TLinkAddress).
15         /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
16         /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
17         /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
18         /// OutPart(3, pattern) ~ GetLinkAddresser(link) or GetLinkAddresser(Search(pattern))
19         /// OutPart(n, pattern) => For any variable length links, returns link or
20         ///   ↳ default(TLinkAddress).
21         ///
22         /// Outs(returns) inner contents of link, its part/parent/element/value.
23         /// </remarks>
24         TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);
25
26         /// <remarks>OutCount() returns total links in store as array.</remarks>
27         IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);
28
29         /// <remarks>OutCount() returns total amount of links in store.</remarks>
30         ulong OutCount(IList<TLinkAddress> pattern);
31     }
}

```

#### 1.25 ./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 }

```

## 1.26 ./Platform.Data.Tests/LinksConstantsTests.cs

```

1  using Xunit;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Tests
7  {
8      public static class LinksConstantsTests
9      {
10         [Fact]
11         public static void ExternalReferencesTest()
12         {
13             TestExternalReferences<ulong, long>();
14             TestExternalReferences<uint, int>();
15             TestExternalReferences<ushort, short>();
16             TestExternalReferences<byte, sbyte>();
17         }
18
19         private static void TestExternalReferences<TUnsigned, TSigned>()
20         {
21             var unsingedOne = Arithmetic.Increment(default(TUnsigned));
22             var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
23             var half = converter.Convert(NumericType<TSigned>.MaxValue);
24             LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne,
25                 ↪ half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));
26
27             var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
28             var maximum = new Hybrid<TUnsigned>(half, isExternal: true);
29
30             Assert.True(constants.IsExternalReference(minimum));
31             Assert.True(minimum.IsExternal);
32             Assert.False(minimum.IsInternal);
33             Assert.True(constants.IsExternalReference(maximum));
34             Assert.True(maximum.IsExternal);
35             Assert.False(maximum.IsInternal);
36         }
37     }

```

## Index

- ./Platform.Data.Tests/LinksConstantsTests.cs, 23
- ./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, 9
- ./Platform.Data/LinkAddress.cs, 9
- ./Platform.Data/LinksConstants.cs, 11
- ./Platform.Data/LinksConstantsExtensions.cs, 13
- ./Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs, 13
- ./Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs, 13
- ./Platform.Data/Point.cs, 14
- ./Platform.Data/Sequences/ISequenceAppender.cs, 16
- ./Platform.Data/Sequences/ISequenceWalker.cs, 16
- ./Platform.Data/Sequences/SequenceWalker.cs, 16
- ./Platform.Data/Sequences/StopableSequenceWalker.cs, 18
- ./Platform.Data/Universal/IUniLinks.cs, 20
- ./Platform.Data/Universal/IUniLinksCRUD.cs, 21
- ./Platform.Data/Universal/IUniLinksGS.cs, 21
- ./Platform.Data/Universal/IUniLinksIO.cs, 21
- ./Platform.Data/Universal/IUniLinksIOWithExtensions.cs, 22
- ./Platform.Data/Universal/IUniLinksRW.cs, 22