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

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link, string argumentName) : base(FormatMessage(link, argumentName), argumentName) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link)) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(string message, Exception innerException) : base(message, innerException) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(string message) : base(message) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress link, string argumentName) => $"Связь [{link}] переданная в аргумент [{argumentName}] не существует.";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в качестве аргумента не существует.";
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) : base(FormatMessage(link, paramName), paramName) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(TLinkAddress link) : base(FormatMessage(link)) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(string message, Exception innerException) : base(message, innerException) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(string message) : base(message) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException() { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые препятствуют изменению её внутренней структуры.";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной в качестве аргумента присутствуют зависимости, которые препятствуют изменению её внутренней структуры.";
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class LinkWithSameValueAlreadyExistsException : Exception
    {
        public static readonly string DefaultMessage = "Связь с таким же значением уже существует.";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkWithSameValueAlreadyExistsException(string message, Exception innerException) : base(message, innerException) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public class LinksLimitReachedException<TLinkAddress> : LinksLimitReachedExceptionBase
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(string message, Exception innerException) : base(message, innerException) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(string message) : base(message) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException() : base(DefaultMessage) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества связей в хранилище ({limit}).";
    }
}
```

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

```
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    public abstract class LinksLimitReachedExceptionBase : Exception
    {
        public static readonly string DefaultMessage = "Достигнут лимит количества связей в хранилище.";

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksLimitReachedExceptionBase(string message, Exception innerException) : base(message, innerException) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksLimitReachedExceptionBase(string message) : base(message) { }
    }
}
```

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

```
using System;
using System.Collections.Generic;
```

```
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly UncheckedSignExtendingConverter<TLinkAddress, long>
        ↪  _addressToInt64Converter = UncheckedSignExtendingConverter<TLinkAddress,
        ↪  long>.Default;
        private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter
        ↪  = UncheckedConverter<long, TLinkAddress>.Default;
        private static readonly UncheckedConverter<TLinkAddress, ulong>
        ↪  _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLinkAddress>
        ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        private static readonly UncheckedConverter<object, long> _objectToInt64Converter =
        ↪  UncheckedConverter<object, long>.Default;

        public static readonly ulong HalfOfNumberValuesRange =
        ↪  _addressToUInt64Converter.Convert(NumericType<TLinkAddress>.MaxValue) / 2;
        public static readonly TLinkAddress ExternalZero =
        ↪  _uInt64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);

        public readonly TLinkAddress Value;

        public bool IsNothing
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
        }

        public bool IsInternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => SignedValue > 0;
        }

        public bool IsExternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
        }

        public long SignedValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _addressToInt64Converter.Convert(Value);
        }

        public long AbsoluteValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
            ↪  Platform.Numbers.Math.Abs(SignedValue);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value)
        {
            Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
            Value = value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value, bool isExternal)
        {
            if (_equalityComparer.Equals(value, default) && isExternal)
            {
                Value = ExternalZero;
            }
            else
            {
```

```
            if (isExternal)
            {
                Value = Math<TLinkAddress>.Negate(value);
            }
            else
            {
                Value = value;
            }
        }
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Hybrid(object value) => Value =
    ↪  _int64ToAddressConverter.Convert(_objectToInt64Converter.Convert(value));

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public Hybrid(object value, bool isExternal)
    {
        var signedValue = value == null ? 0 : _objectToInt64Converter.Convert(value);
        if (signedValue == 0 && isExternal)
        {
            Value = ExternalZero;
        }
        else
        {
            var absoluteValue = System.Math.Abs(signedValue);
            Value = isExternal ? _int64ToAddressConverter.Convert(-absoluteValue) :
            ↪  _int64ToAddressConverter.Convert(absoluteValue);
        }
    }

    [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)]
        public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
        ↪ (sbyte)hybrid.AbsoluteValue;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => IsExternal ? $"<{AbsoluteValue}>" :
        ↪ Value.ToString();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
        ↪ other.Value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
        ↪ Equals(hybrid) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => Value.GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
        ↪ left.Equals(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
        ↪ !(left == right);
    }
}
```

1.7 ./csharp/Platform.Data/ILinks.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    /// <summary>
    /// <para>Represents an interface for manipulating data in the Links (links storage)
    ↪ format.</para>
    /// <para>Представляет интерфейс для манипуляции с данными в формате Links (хранилища
    ↪ связей).</para>
    /// </summary>
    /// <remarks>
    /// <para>This interface is independent of the size of the content of the link, meaning it
    ↪ is suitable for both doublets, triplets, and link sequences of any size.</para>
    /// <para>Этот интерфейс не зависит от размера содержимого связи, а значит подходит как для
    ↪ дуплетов, триплетов и последовательностей связей любого размера.</para>
    /// </remarks>
    public interface ILinks<TLinkAddress, TConstants>
        where TConstants : LinksConstants<TLinkAddress>
    {
        #region Constants

        /// <summary>
```

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

```
76          TLinkAddress Create(IList<TLinkAddress> restrictions); // TODO: Возвращать связь
            ↪ возвращать нужно целиком.
77
78          /// <summary>
79          /// Обновляет связь с указанными restrictions[Constants.IndexPart] в адресом связи
80          /// на связь с указанным новым содержимым.
81          /// </summary>
82          /// <param name="restrictions">
83          /// Ограничения на содержимое связей.
84          /// Предполагается, что будет указан индекс связи (в restrictions[Constants.IndexPart])
            ↪ и далее за ним будет следовать содержимое связи.
85          /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
            ↪ ссылку на пустоту,
86          /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
            ↪ другой связи.
87          /// </param>
88          /// <param name="substitution"></param>
89          /// <returns>Индекс обновлённой связи.</returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress> substitution);
            ↪ // TODO: Возможно и возвращать связь нужно целиком.
92
93          /// <summary>Удаляет связь с указанным индексом.</summary>
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          void Delete(IList<TLinkAddress> restrictions); // TODO: Возможно всегда нужно принимать
            ↪ restrictions, а так же возвращать удалённую связь, если удаление было реально
            ↪ выполнено, и Null, если нет.
96
97          #endregion
98      }
99  }
```

1.8 ./csharp/Platform.Data/ILinksExtensions.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Setters;
5   using Platform.Data.Exceptions;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data
10  {
11      public static class ILinksExtensions
12      {
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, params TLinkAddress[] restrictions)
15              where TConstants : LinksConstants<TLinkAddress>
16              => links.Count(restrictions);
17
18          /// <summary>
19          /// Возвращает значение, определяющее существует ли связь с указанным индексом в
            ↪ хранилище связей.
20          /// </summary>
21          /// <param name="links">Хранилище связей.</param>
22          /// <param name="link">Индекс проверяемой на существование связи.</param>
23          /// <returns>Значение, определяющее существует ли связь.</returns>
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, TLinkAddress link)
26              where TConstants : LinksConstants<TLinkAddress>
27          {
28              var constants = links.Constants;
29              return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
                ↪ && Comparer<TLinkAddress>.Default.Compare(links.Count(new
                ↪ LinkAddress<TLinkAddress>(link)), default) > 0);
30          }
31
32          /// <param name="links">Хранилище связей.</param>
33          /// <param name="link">Индекс проверяемой на существование связи.</param>
34          /// <remarks>
35          /// TODO: May be move to EnsureExtensions or make it both there and here
36          /// </remarks>
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, TLinkAddress link)
39              where TConstants : LinksConstants<TLinkAddress>
40          {
```

```
            if (!links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
            }
        }

        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
        ↪  TConstants> links, TLinkAddress link, string argumentName)
            where TConstants : LinksConstants<TLinkAddress>
        {
            if (!links.Exists(link))
            {
                throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
            }
        }

        /// <summary>
        /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
        ↪  (handler) для каждой подходящей связи.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="handler">Обработчик каждой подходящей связи.</param>
        /// <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>
        /// <returns>True, в случае если проход по связям не был прерван и False в обратном
        ↪  случае.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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);

        /// <summary>
        /// Возвращает части-значения для связи с указанным индексом.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс связи.</param>
        /// <returns>Уникальную связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
        ↪  ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
            where TConstants : LinksConstants<TLinkAddress>
        {
            var constants = links.Constants;
            if (constants.IsExternalReference(link))
            {
                return new Point<TLinkAddress>(link, constants.TargetPart + 1);
            }
            var linkPartsSetter = new Setter<IList<TLinkAddress>,
            ↪  TLinkAddress>(constants.Continue, constants.Break);
            links.Each(linkPartsSetter.SetAndReturnTrue, link);
            return linkPartsSetter.Result;
        }

        #region Points

        /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
        ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой связи.</param>
        /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
        /// <remarks>
        /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
        ↪  связь.
        /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
        ↪  точка и пара существовать одновременно?
        /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
        ↪  сортировать по индексу в массиве связей?
        /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
        /// 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 &amp;amp;&amp;
            ↪ link.Target == link , т.е. дважды ссылка на себя.
110         /// А пара это тогда, когда link.Source == link.Target &amp;amp;&amp; link.Source != link ,
            ↪ т.е. ссылка не на себя а во вне.
111         ///
112         /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
            ↪ промежуточную связь,
113         /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
114         /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
115         /// </remarks>
116         [MethodImpl(MethodImplOptions.AggressiveInlining)]
117         public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, TLinkAddress link)
118             where TConstants : LinksConstants<TLinkAddress>
119         {
120             if (links.Constants.IsExternalReference(link))
121             {
122                 return true;
123             }
124             links.EnsureLinkExists(link);
125             return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
126         }
127 
128         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪ точкой частично (связью замкнутой на себе как минимум один раз).</summary>
129         /// <param name="links">Хранилище связей.</param>
130         /// <param name="link">Индекс проверяемой связи.</param>
131         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
132         /// <remarks>
133         /// Достаточно любой одной ссылки на себя.
134         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
            ↪ ссылки на себя (на эту связь).
135         /// </remarks>
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪ TConstants> links, TLinkAddress link)
138             where TConstants : LinksConstants<TLinkAddress>
139         {
140             if (links.Constants.IsExternalReference(link))
141             {
142                 return true;
143             }
144             links.EnsureLinkExists(link);
145             return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
146         }
147 
148         #endregion
149     }
150 }
```

1.9 ./csharp/Platform.Data/ISynchronizedLinks.cs

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

1.10 ./csharp/Platform.Data/LinkAddress.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  
```

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
    ↪  IList<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index == 0)
                {
                    return Index;
                }
                else
                {
                    throw new IndexOutOfRangeException();
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => 1;
        }

        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkAddress(TLinkAddress index) => Index = index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
        ↪  ? true : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLinkAddress> GetEnumerator()
        {
            yield return Index;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
        ↪  0 : -1;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();
```

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

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Ranges;
3  using Platform.Reflection;
4  using Platform.Converters;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data
10 {
11     public class LinksConstants<TLinkAddress> : LinksConstantsBase
12     {
13         private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
14         private static readonly UncheckedConverter<ulong, TLinkAddress>
            ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
15
16         #region Link parts
17
18         /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
            ↪  самой связи.</summary>
19         public int IndexPart
20         {
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             get;
23         }
24
25         /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
            ↪  часть-значение).</summary>
```

```
26          public int SourcePart
27          {
28              [MethodImpl(MethodImplOptions.AggressiveInlining)]
29              get;
30          }
31
32          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
            ↪  (последняя часть-значение).</summary>
33          public int TargetPart
34          {
35              [MethodImpl(MethodImplOptions.AggressiveInlining)]
36              get;
37          }
38
39          #endregion
40
41          #region Flow control
42
43          /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
44          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
45          public TLinkAddress Continue
46          {
47              [MethodImpl(MethodImplOptions.AggressiveInlining)]
48              get;
49          }
50
51          /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
52          public TLinkAddress Skip
53          {
54              [MethodImpl(MethodImplOptions.AggressiveInlining)]
55              get;
56          }
57
58          /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
59          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
60          public TLinkAddress Break
61          {
62              [MethodImpl(MethodImplOptions.AggressiveInlining)]
63              get;
64          }
65
66          #endregion
67
68          #region Special symbols
69
70          /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
71          public TLinkAddress Null
72          {
73              [MethodImpl(MethodImplOptions.AggressiveInlining)]
74              get;
75          }
76
77          /// <summary>Возвращает значение, обозначающее любую связь.</summary>
78          /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
            ↪  создавать все варианты последовательностей в функции Create.</remarks>
79          public TLinkAddress Any
80          {
81              [MethodImpl(MethodImplOptions.AggressiveInlining)]
82              get;
83          }
84
85          /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
86          public TLinkAddress Itself
87          {
88              [MethodImpl(MethodImplOptions.AggressiveInlining)]
89              get;
90          }
91
92          #endregion
93
94          #region References
95
96          /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
            ↪  ссылок).</summary>
97          public Range<TLinkAddress> InternalReferencesRange
98          {
99              [MethodImpl(MethodImplOptions.AggressiveInlining)]
100             get;
```

```
        }

        /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних ссылок).</summary>
        public Range<TLinkAddress>? ExternalReferencesRange
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        #endregion

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(int targetPart, Range<TLinkAddress> possibleInternalReferencesRange, Range<TLinkAddress>? possibleExternalReferencesRange)
        {
            IndexPart = 0;
            SourcePart = 1;
            TargetPart = targetPart;
            Null = default;
            Break = default;
            var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
            Continue = currentInternalReferenceIndex;
            Skip = Arithmetic.Decrement(ref currentInternalReferenceIndex);
            Any = Arithmetic.Decrement(ref currentInternalReferenceIndex);
            Itself = Arithmetic.Decrement(ref currentInternalReferenceIndex);
            Arithmetic.Decrement(ref currentInternalReferenceIndex);
            InternalReferencesRange = (possibleInternalReferencesRange.Minimum, currentInternalReferenceIndex);
            ExternalReferencesRange = possibleExternalReferencesRange;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(int targetPart, bool enableExternalReferencesSupport) : this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport), GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange, Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart, possibleInternalReferencesRange, possibleExternalReferencesRange) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(bool enableExternalReferencesSupport) : this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport), GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(int targetPart, Range<TLinkAddress> possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange, null) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) : this(DefaultTargetPart, possibleInternalReferencesRange, null) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport: false) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool enableExternalReferencesSupport)
        {
            if (enableExternalReferencesSupport)
            {
                return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
                    rValuesRange));
            }
            else
            {
                return (_one, NumericType<TLinkAddress>.MaxValue);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
162         public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
            ↪ enableExternalReferencesSupport)
163         {
164             if (enableExternalReferencesSupport)
165             {
166                 return (Hybrid<TLinkAddress>.ExternalZero, NumericType<TLinkAddress>.MaxValue);
167             }
168             else
169             {
170                 return null;
171             }
172         }
173     }
174 }
```

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

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

1.13 ./csharp/Platform.Data/LinksConstantsExtensions.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data
6  {
7      public static class LinksConstantsExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static bool IsReference<TLinkAddress>(this LinksConstants<TLinkAddress>
           ↪ linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
           ↪ || linksConstants.IsExternalReference(address);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
           ↪ linksConstants, TLinkAddress address) =>
           ↪ linksConstants.InternalReferencesRange.Contains(address);
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
           ↪ linksConstants, TLinkAddress address) =>
           ↪ linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
17     }
18 }
```

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

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

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

```
1  using System.Runtime.CompilerServices;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Numbers.Raw
7  {
8      public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9      {
10         static private readonly UncheckedConverter<long, TLink> _converter =
           ↪ UncheckedConverter<long, TLink>.Default;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink source) => _converter.Convert(new
        ↪  Hybrid<TLink>(source).AbsoluteValue);
    }
}
```

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

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Ranges;
using Platform.Collections;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public int Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index < Size)
                {
                    return Index;
                }
                else
                {
                    throw new IndexOutOfRangeException();
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => int.MaxValue;
        }

        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Point(TLinkAddress index, int size)
        {
            Index = index;
            Size = size;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
        ↪  ? true : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLinkAddress> GetEnumerator()
        {
            for (int i = 0; i < Size; i++)
            {
                yield return Index;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
        ↪  0 : -1;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator()
        {
            for (int i = 0; i < Size; i++)
            {
                yield return Index;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
        ↪  _equalityComparer.Equals(Index, other.Index);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
        ↪  linkAddress.Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
        ↪  Equals(linkAddress) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => Index.GetHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => Index.ToString();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
        {
            if (left == null && right == null)
            {
                return true;
            }
            if (left == null)
            {
                return false;
            }
            return left.Equals(right);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
        ↪  !(left == right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFullPoint(params TLinkAddress[] link) =>
        ↪  IsFullPoint((IList<TLinkAddress>)link);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsFullPoint(IList<TLinkAddress> link)
```

```
145         {
146             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
147             Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
                ↪  determine link's pointness using only its identifier.");
148             return IsFullPointUnchecked(link);
149         }
150 
151         [MethodImpl(MethodImplOptions.AggressiveInlining)]
152         public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
153         {
154             var result = true;
155             for (var i = 1; result && i < link.Count; i++)
156             {
157                 result = _equalityComparer.Equals(link[0], link[i]);
158             }
159             return result;
160         }
161 
162         [MethodImpl(MethodImplOptions.AggressiveInlining)]
163         public static bool IsPartialPoint(params TLinkAddress[] link) =>
            ↪  IsPartialPoint((IList<TLinkAddress>)link);
164 
165         [MethodImpl(MethodImplOptions.AggressiveInlining)]
166         public static bool IsPartialPoint(IList<TLinkAddress> link)
167         {
168             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
169             Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
                ↪  determine link's pointness using only its identifier.");
170             return IsPartialPointUnchecked(link);
171         }
172 
173         [MethodImpl(MethodImplOptions.AggressiveInlining)]
174         public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
175         {
176             var result = false;
177             for (var i = 1; !result && i < link.Count; i++)
178             {
179                 result = _equalityComparer.Equals(link[0], link[i]);
180             }
181             return result;
182         }
183     }
184 }
```

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

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

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

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4 
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```
6
7   namespace Platform.Data.Sequences
8   {
9       /// <remarks>
10      /// Реализованный внутри алгоритм наглядно показывает,
11      /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой
        ↪  себя),
12      /// так как стэк 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 зацикливания.
16      /// Альтернативой защиты от закливания 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                  {
```

```
                    if (isElement(element))
                    {
                        if (stack.Count == 0)
                        {
                            break;
                        }
                        element = stack.Pop();
                        var source = getSource(element);
                        var target = getTarget(element);
                        if (isElement(target))
                        {
                            visit(target);
                        }
                        if (isElement(source))
                        {
                            visit(source);
                        }
                        element = source;
                    }
                    else
                    {
                        stack.Push(element);
                        element = getTarget(element);
                    }
                }
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Sequences
{
    /// <remarks>
    /// Реализованный внутри алгоритм наглядно показывает,
    /// что совершенно не обязательна рекурсивная реализация (с вложенным вызовом функцией самой себя),
    /// так как стэк можно использовать намного эффективнее при ручном управлении.
    ///
    /// Решить объединять ли логику в одну функцию, или оставить 4 отдельных реализации?
    /// Решить встраивать ли защиту от зацикливания.
    /// Альтернативой защиты от закливания может быть заранее известное ограничение на погружение вглубь.
    /// А так же качественное распознавание прохода по циклическому графу.
    /// Ограничение на уровень глубины рекурсии может позволить использовать уменьшенный размер стека.
    /// Можно использовать глобальный стек (или несколько глобальных стеков на каждый поток).
    /// </remarks>
    public static class StopableSequenceWalker
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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)
        {
            var exited = 0;
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    exit(element);
```

```
                    exited++;
                    var source = getSource(element);
                    var target = getTarget(element);
                    if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                    ↪  !visit(source))
                    {
                        return false;
                    }
                    if ((isElement(target) || !canEnter(target)) && !visit(target))
                    {
                        return false;
                    }
                    element = target;
                }
                else
                {
                    if (canEnter(element))
                    {
                        enter(element);
                        exited = 0;
                        stack.Push(element);
                        element = getSource(element);
                    }
                    else
                    {
                        if (stack.Count == 0)
                        {
                            return true;
                        }
                        element = stack.Pop();
                        exit(element);
                        exited++;
                        var source = getSource(element);
                        var target = getTarget(element);
                        if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                        ↪  !visit(source))
                        {
                            return false;
                        }
                        if ((isElement(target) || !canEnter(target)) && !visit(target))
                        {
                            return false;
                        }
                        element = target;
                    }
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
        {
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    var source = getSource(element);
                    var target = getTarget(element);
                    if (isElement(source) && !visit(source))
                    {
                        return false;
                    }
                    if (isElement(target) && !visit(target))
                    {
                        return false;
                    }
```

```
                    element = target;
                }
                else
                {
                    stack.Push(element);
                    element = getSource(element);
                }
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Func<TLinkAddress, bool> visit)
        {
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                return visit(element);
            }
            while (true)
            {
                if (isElement(element))
                {
                    if (stack.Count == 0)
                    {
                        return true;
                    }
                    element = stack.Pop();
                    var source = getSource(element);
                    var target = getTarget(element);
                    if (isElement(target) && !visit(target))
                    {
                        return false;
                    }
                    if (isElement(source) && !visit(source))
                    {
                        return false;
                    }
                    element = source;
                }
                else
                {
                    stack.Push(element);
                    element = getTarget(element);
                }
            }
        }
    }
}
```

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

```
using System;
using System.Collections.Generic;

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        IList<IList<IList<TLinkAddress>>> Trigger(IList<TLinkAddress> condition,
        ↪  IList<TLinkAddress> substitution);
    }

    /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        /// <returns>
        /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
        ↪  represents False (was stopped).
        /// This is done to assure ability to push up stop signal through recursion stack.
        /// </returns>
        /// <remarks>
        /// { 0, 0, 0 } => { itself, itself, itself } // create
        /// { 1, any, any } => { itself, any, 3 } // update
```

```
        /// { 3, any, any } => { 0, 0, 0 } // delete
        /// </remarks>
        TLinkAddress Trigger(IList<TLinkAddress> patternOrCondition, Func<IList<TLinkAddress>, TLinkAddress> matchHandler,
                             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>, TLinkAddress> substitutionHandler);

        TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>, IList<TLinkAddress>, TLinkAddress> matchedHandler,
                             IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>, TLinkAddress> substitutedHandler);
    }

    /// <remarks>Extended with small optimization.</remarks>
    public partial interface IUniLinks<TLinkAddress>
    {
        /// <remarks>
        /// Something simple should be simple and optimized.
        /// </remarks>
        TLinkAddress Count(IList<TLinkAddress> restrictions);
    }
}
```

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

```
using System;
using System.Collections.Generic;

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// CRUD aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksCRUD<TLinkAddress>
    {
        TLinkAddress Read(int partType, TLinkAddress link);
        TLinkAddress Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Create(IList<TLinkAddress> parts);
        TLinkAddress Update(IList<TLinkAddress> before, IList<TLinkAddress> after);
        void Delete(IList<TLinkAddress> parts);
    }
}
```

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

```
using System;
using System.Collections.Generic;

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// Get/Set aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksGS<TLinkAddress>
    {
        TLinkAddress Get(int partType, TLinkAddress link);
        TLinkAddress Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Set(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

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

```
using System;
using System.Collections.Generic;

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// In/Out aliases for IUniLinks.
    /// TLinkAddress can be any number type of any size.
    /// </remarks>
    public interface IUniLinksIO<TLinkAddress>
```

```
    {
        /// <remarks>
        /// default(TLinkAddress) means any link.
        /// Single element pattern means just element (link).
        /// Handler gets array of link contents.
        /// * link[0] is index or identifier.
        /// * link[1] is source or first.
        /// * link[2] is target or second.
        /// * link[3] is linker or third.
        /// * link[n] is nth part/parent/element/value
        /// of link (if variable length links used).
        ///
        /// Stops and returns false if handler return false.
        ///
        /// Acts as Each, Foreach, Select, Search, Match &amp;amp; ...
        ///
        /// Handles all links in store if pattern/restrictions is not defined.
        /// </remarks>
        bool Out(Func<IList<TLinkAddress>, bool> handler, IList<TLinkAddress> pattern);

        /// <remarks>
        /// default(TLinkAddress) means itself.
        /// Equivalent to:
        /// * creation if before == null
        /// * deletion if after == null
        /// * update if before != null &amp;amp;&amp;amp; after != null
        /// * default(TLinkAddress) if before == null &amp;amp;&amp;amp; after == null
        ///
        /// Possible interpretation
        /// * In(null, new[] { }) creates point (link that points to itself using minimum number of parts).
        /// * In(new[] { 4 }, null) deletes 4th link.
        /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to 5th index.
        /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with 2 as source and 3 as target), 0 means it can be placed in any address.
        /// ...
        /// </remarks>
        TLinkAddress In(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

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

```
// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

using System.Collections.Generic;

namespace Platform.Data.Universal
{
    /// <remarks>Contains some optimizations of Out.</remarks>
    public interface IUniLinksIOWithExtensions<TLinkAddress> : IUniLinksIO<TLinkAddress>
    {
        /// <remarks>
        /// default(TLinkAddress) means nothing or null.
        /// Single element pattern means just element (link).
        /// OutPart(n, null) returns default(TLinkAddress).
        /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
        /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
        /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
        /// OutPart(3, pattern) ~ GeTLinkAddresser(link) or GeTLinkAddresser(Search(pattern))
        /// OutPart(n, pattern) => For any variable length links, returns link or default(TLinkAddress).
        ///
        /// Outs(returns) inner contents of link, its part/parent/element/value.
        /// </remarks>
        TLinkAddress OutOne(int partType, IList<TLinkAddress> pattern);

        /// <remarks>OutCount() returns total links in store as array.</remarks>
        IList<IList<TLinkAddress>> OutAll(IList<TLinkAddress> pattern);

        /// <remarks>OutCount() returns total amount of links in store.</remarks>
        ulong OutCount(IList<TLinkAddress> pattern);
    }
}
```

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

```
using System;
using System.Collections.Generic;

// ReSharper disable TypeParameterCanBeVariant
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Universal
{
    /// <remarks>
    /// Read/Write aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksRW<TLinkAddress>
    {
        TLinkAddress Read(int partType, TLinkAddress link);
        bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        TLinkAddress Write(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

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

```
using Xunit;

namespace Platform.Data.Tests
{
    public static class HybridTests
    {
        [Fact]
        public static void ObjectConstructorTest()
        {
            Assert.Equal(0, new Hybrid<byte>(unchecked((byte)128)).AbsoluteValue);
            Assert.Equal(0, new Hybrid<byte>((object)128).AbsoluteValue);
            Assert.Equal(1, new Hybrid<byte>(unchecked((byte)-1)).AbsoluteValue);
            Assert.Equal(1, new Hybrid<byte>((object)-1).AbsoluteValue);
            Assert.Equal(0, new Hybrid<byte>(unchecked((byte)0)).AbsoluteValue);
            Assert.Equal(0, new Hybrid<byte>((object)0).AbsoluteValue);
            Assert.Equal(1, new Hybrid<byte>(unchecked((byte)1)).AbsoluteValue);
            Assert.Equal(1, new Hybrid<byte>((object)1).AbsoluteValue);
        }
    }
}
```

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

```
using Xunit;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

namespace Platform.Data.Tests
{
    public static class LinksConstantsTests
    {
        [Fact]
        public static void ConstructorTest()
        {
            var constants = new LinksConstants<ulong>(enableExternalReferencesSupport: true);
            Assert.Equal(Hybrid<ulong>.ExternalZero,
            ↪  constants.ExternalReferencesRange.Value.Minimum);
            Assert.Equal(ulong.MaxValue, constants.ExternalReferencesRange.Value.Maximum);
        }

        [Fact]
        public static void ExternalReferencesTest()
        {
            TestExternalReferences<ulong, long>();
            TestExternalReferences<uint, int>();
            TestExternalReferences<ushort, short>();
            TestExternalReferences<byte, sbyte>();
        }

        private static void TestExternalReferences<TUnsigned, TSigned>()
        {
            var unsingedOne = Arithmetic.Increment(default(TUnsigned));
            var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
            var half = converter.Convert(NumericType<TSigned>.MaxValue);
            LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne,
            ↪  half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));

            var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
```

```
            var maximum = new Hybrid<TUnsigned>(half, isExternal: true);

            Assert.True(constants.IsExternalReference(minimum));
            Assert.True(minimum.IsExternal);
            Assert.False(minimum.IsInternal);
            Assert.True(constants.IsExternalReference(maximum));
            Assert.True(maximum.IsExternal);
            Assert.False(maximum.IsInternal);
        }
    }
}
```

Index