1.1 ./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 ./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 ./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 ./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 ./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 ./Platform.Data/Hybrid.cs

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

```
using System.Reflection;
using System.Reflection.Emit;
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<TLinkAddress, ulong> _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLinkAddress> _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        private static readonly Func<object, TLinkAddress> _unboxAbsAndConvert = CompileUnboxAbsAndConvertDelegate();
        private static readonly Func<object, TLinkAddress> _unboxAbsNegateAndConvert = CompileUnboxAbsNegateAndConvertDelegate();

        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 =
        ↪  To.UnsignedAs<TLinkAddress>(Convert.ChangeType(value,
        ↪  NumericType<TLinkAddress>.SignedVersion));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(object value, bool isExternal)
        {
            if (IsDefault(value) && isExternal)
            {
                Value = ExternalZero;
            }
            else
            {
                if (isExternal)
                {
                    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)]
        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);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool IsDefault(object value)
        {
            if (value == null)
            {
                return true;
            }
            var type = value.GetType();
            return type.IsValueType ? value.Equals(Activator.CreateInstance(type)) : false;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<object, TLinkAddress> CompileUnboxAbsNegateAndConvertDelegate()
        {
            return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<TLinkAddress>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<TLinkAddress>.SignedVersion;
                var signedVersionField =
                ↪  typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
                ↪  BindingFlags.Static | BindingFlags.Public);
                emiter.Emit(OpCodes.Ldsfld, signedVersionField);
```

```
                var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                ↪  Types<object, Type>.Array);
                emiter.Call(changeTypeMethod);
                emiter.UnboxValue(signedVersion);
                var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                ↪  signedVersion });
                emiter.Call(absMethod);
                var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate
                ↪  ").MakeGenericMethod(signedVersion);
                emiter.Call(negateMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<object, TLinkAddress> CompileUnboxAbsAndConvertDelegate()
        {
            return DelegateHelpers.Compile<Func<object, TLinkAddress>>(emiter =>
            {
                Ensure.Always.IsUnsignedInteger<TLinkAddress>();
                emiter.LoadArgument(0);
                var signedVersion = NumericType<TLinkAddress>.SignedVersion;
                var signedVersionField =
                ↪  typeof(NumericType<TLinkAddress>).GetTypeInfo().GetField("SignedVersion",
                ↪  BindingFlags.Static | BindingFlags.Public);
                emiter.Emit(OpCodes.Ldsfld, signedVersionField);
                var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
                ↪  Types<object, Type>.Array);
                emiter.Call(changeTypeMethod);
                emiter.UnboxValue(signedVersion);
                var absMethod = typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                ↪  signedVersion });
                emiter.Call(absMethod);
                var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
                ↪  signedVersion });
                emiter.Call(unsignedMethod);
                emiter.Return();
            });
        }
    }
}
```

1.7 ./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>
    /// Представляет интерфейс для работы с данными в формате Links (хранилища взаимосвязей).
    /// </summary>
    /// <remarks>
    /// Этот интерфейс в данный момент не зависит от размера содержимого связи, а значит
    ↪  подходит как для дуплетов, так и для триплетов и т.п.
    /// Возможно этот интерфейс подходит даже для Sequences.
    /// </remarks>
    public interface ILinks<TLinkAddress, TConstants>
        where TConstants : LinksConstants<TLinkAddress>
    {
        #region Constants

        /// <summary>
        /// Возвращает набор констант, который необходим для эффективной коммуникации с методами
        ↪  этого интерфейса.
        /// Эти константы не меняются с момента создания точки доступа к хранилищу.
        /// </summary>
        TConstants Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

```

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

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
```

```
{
    public static class ILinksExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
        ↪  TConstants> links, params TLinkAddress[] restrictions)
            where TConstants : LinksConstants<TLinkAddress>
            => links.Count(restrictions);

        /// <summary>
        /// Возвращает значение, определяющее существует ли связь с указанным индексом в
        ↪  хранилище связей.
        /// </summary>
        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <returns>Значение, определяющее существует ли связь.</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
        ↪  TConstants> links, TLinkAddress link)
            where TConstants : LinksConstants<TLinkAddress>
        {
            var constants = links.Constants;
            return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
            ↪  && Comparer<TLinkAddress>.Default.Compare(links.Count(new
            ↪  LinkAddress<TLinkAddress>(link)), default) > 0);
        }

        /// <param name="links">Хранилище связей.</param>
        /// <param name="link">Индекс проверяемой на существование связи.</param>
        /// <remarks>
        /// TODO: May be move to EnsureExtensions or make it both there and here
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
        ↪  TConstants> links, TLinkAddress link)
            where TConstants : LinksConstants<TLinkAddress>
        {
            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>
```

```
75          /// <param name="links">Хранилище связей.</param>
76          /// <param name="link">Индекс связи.</param>
77          /// <returns>Уникальную связь.</returns>
78          [MethodImpl(MethodImplOptions.AggressiveInlining)]
79          public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
            ↪  ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
80              where TConstants : LinksConstants<TLinkAddress>
81          {
82              var constants = links.Constants;
83              if (constants.IsExternalReference(link))
84              {
85                  return new Point<TLinkAddress>(link, constants.TargetPart + 1);
86              }
87              var linkPartsSetter = new Setter<IList<TLinkAddress>,
                ↪  TLinkAddress>(constants.Continue, constants.Break);
88              links.Each(linkPartsSetter.SetAndReturnTrue, link);
89              return linkPartsSetter.Result;
90          }
91
92          #region Points
93
94          /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
95          /// <param name="links">Хранилище связей.</param>
96          /// <param name="link">Индекс проверяемой связи.</param>
97          /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
98          /// <remarks>
99          /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
            ↪  связь.
100         /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
            ↪  точка и пара существовать одновременно?
101         /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
            ↪  сортировать по индексу в массиве связей?
102         /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
103         /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
            ↪  самой себя любого размера?
104         /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
            ↪  одной ссылки на себя (частичной точки).
105         /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
106         /// самостоятельный цикл через себя? Что если предоставить все варианты использования
            ↪  связей?
107         /// Что если разрешить и нули, а так же частичные варианты?
108         ///
109         /// Что если точка, это только в том случае когда link.Source == link &amp;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;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 ./Platform.Data/ISynchronizedLinks.cs

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

1.10 ./Platform.Data/LinkAddress.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7  
8  namespace Platform.Data
9  {
10     public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
       ↪ IList<TLinkAddress>
11     {
12         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
           ↪ EqualityComparer<TLinkAddress>.Default;
13 
14         public TLinkAddress Index
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19 
20         public TLinkAddress this[int index]
21         {
22             [MethodImpl(MethodImplOptions.AggressiveInlining)]
23             get
24             {
25                 if (index == 0)
26                 {
27                     return Index;
28                 }
29                 else
30                 {
31                     throw new IndexOutOfRangeException();
32                 }
33             }
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             set => throw new NotSupportedException();
36         }
37 
38         public int Count
39         {
40             [MethodImpl(MethodImplOptions.AggressiveInlining)]
41             get => 1;
42         }
43 
44         public bool IsReadOnly
45         {
46             [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
            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();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator()
        {
            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(LinkAddress<TLinkAddress> linkAddress) =>
        ↪  linkAddress.Index;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is LinkAddress<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 ==(LinkAddress<TLinkAddress> left, LinkAddress<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 !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
        ↪  right) => !(left == right);
    }
}
```

1.11 ./Platform.Data/LinksConstants.cs

```
using System.Runtime.CompilerServices;
using Platform.Ranges;
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 class LinksConstants<TLinkAddress> : LinksConstantsBase
    {
        private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
        private static readonly UncheckedConverter<ulong, TLinkAddress>
        ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;

        #region Link parts

        /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
        ↪  самой связи.</summary>
        public int IndexPart
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
        ↪  часть-значение).</summary>
        public int SourcePart
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
        ↪  (последняя часть-значение).</summary>
        public int TargetPart
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        #endregion

        #region Flow control

        /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию
        ↪  Each.</remarks>
        public TLinkAddress Continue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
        public TLinkAddress Skip
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию
        ↪  Each.</remarks>
        public TLinkAddress Break
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

```

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

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

1.12 ./Platform.Data/LinksConstantsBase.cs

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

1.13 ./Platform.Data/LinksConstantsExtensions.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Data
6  {
7      public static class LinksConstantsExtensions
```

```
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪ linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
        ↪ || linksConstants.IsExternalReference(address);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪ linksConstants, TLinkAddress address) =>
        ↪ linksConstants.InternalReferencesRange.Contains(address);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪ linksConstants, TLinkAddress address) =>
        ↪ linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Numbers.Raw
{
    public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Numbers.Raw
{
    public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
    {
        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 ./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;
```

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

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

```
175         {
176             var result = false;
177             for (var i = 1; !result && i < link.Count; i++)
178             {
179                 result = _equalityComparer.Equals(link[0], link[i]);
180             }
181             return result;
182         }
183     }
184 }
```

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkLeft<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
        ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
        ↪  Func<TLinkAddress, bool> isElement, Action<TLinkAddress> visit)
        {
            var stack = new Stack<TLinkAddress>();
            var element = sequence;
            if (isElement(element))
            {
                visit(element);
            }
            else
            {
                while (true)
                {
                    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 ./Platform.Data/Sequences/StopableSequenceWalker.cs

```
using System;
using System.Collections.Generic;
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 StopableSequenceWalker
22      {
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public static bool WalkRight<TLinkAddress>(TLinkAddress sequence, Func<TLinkAddress,
            ↪  TLinkAddress> getSource, Func<TLinkAddress, TLinkAddress> getTarget,
            ↪  Func<TLinkAddress, bool> isElement, Action<TLinkAddress> enter, Action<TLinkAddress>
            ↪  exit, Func<TLinkAddress, bool> canEnter, Func<TLinkAddress, bool> visit)
25          {
26              var exited = 0;
27              var stack = new Stack<TLinkAddress>();
28              var element = sequence;
29              if (isElement(element))
30              {
31                  return visit(element);
32              }
33              while (true)
34              {
35                  if (isElement(element))
36                  {
37                      if (stack.Count == 0)
38                      {
39                          return true;
40                      }
41                      element = stack.Pop();
42                      exit(element);
43                      exited++;
44                      var source = getSource(element);
45                      var target = getTarget(element);
46                      if ((isElement(source) || (exited == 1 && !canEnter(source))) &&
                        ↪  !visit(source))
47                      {
48                          return false;
49                      }
50                      if ((isElement(target) || !canEnter(target)) && !visit(target))
51                      {
52                          return false;
53                      }
54                      element = target;
55                  }
56                  else
57                  {
58                      if (canEnter(element))
59                      {
60                          enter(element);
61                          exited = 0;
62                          stack.Push(element);
63                          element = getSource(element);
64                      }
65                      else
66                      {
67                          if (stack.Count == 0)
68                          {
69                              return true;
70                          }
71                          element = stack.Pop();
72                          exit(element);
73                          exited++;
74                          var source = getSource(element);
75                          var target = getTarget(element);
```

```
                    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 ./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 ./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 ./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 ./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.
```

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

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

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

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

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// Read/Write aliases for IUniLinks.
11      /// </remarks>
12      public interface IUniLinksRW<TLinkAddress>
13      {
14          TLinkAddress Read(int partType, TLinkAddress link);
15          bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
16          TLinkAddress Write(IList<TLinkAddress> before, IList<TLinkAddress> after);
17      }
18  }
```

1.27 ./Platform.Data.Tests/LinksConstantsTests.cs

```
1   using Xunit;
2   using Platform.Reflection;
3   using Platform.Converters;
4   using Platform.Numbers;
5
6   namespace Platform.Data.Tests
7   {
8       public static class LinksConstantsTests
9       {
10          [Fact]
11          public static void ExternalReferencesTest()
12          {
13              TestExternalReferences<ulong, long>();
```

```
            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