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
{
    /// <summary>
    /// <para>
    /// Represents the argument link does not exists exception.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ArgumentException"/>
    public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>A argument name.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link, string argumentName) :
        ↪  base(FormatMessage(link, argumentName), argumentName) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link))
        ↪  { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(string message, Exception innerException) :
        ↪  base(message, innerException) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
```

```
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public ArgumentLinkDoesNotExistsException(string message) : base(message) { }
76
77          /// <summary>
78          /// <para>
79          /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException"/> instance.
80          /// </para>
81          /// <para></para>
82          /// </summary>
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          public ArgumentLinkDoesNotExistsException() { }
85
86          /// <summary>
87          /// <para>
88          /// Formats the message using the specified link.
89          /// </para>
90          /// <para></para>
91          /// </summary>
92          /// <param name="link">
93          /// <para>The link.</para>
94          /// <para></para>
95          /// </param>
96          /// <param name="argumentName">
97          /// <para>The argument name.</para>
98          /// <para></para>
99          /// </param>
100         /// <returns>
101         /// <para>The string</para>
102         /// <para></para>
103         /// </returns>
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         private static string FormatMessage(TLinkAddress link, string argumentName) => $"Связь
            ↪  [{link}] переданная в аргумент [{argumentName}] не существует.";
106
107         /// <summary>
108         /// <para>
109         /// Formats the message using the specified link.
110         /// </para>
111         /// <para></para>
112         /// </summary>
113         /// <param name="link">
114         /// <para>The link.</para>
115         /// <para></para>
116         /// </param>
117         /// <returns>
118         /// <para>The string</para>
119         /// <para></para>
120         /// </returns>
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
            ↪  качестве аргумента не существует.";
123     }
124 }
```

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

```
1   using System;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Exceptions
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the argument link has dependencies exception.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="ArgumentException"/>
15      public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
16      {
17          /// <summary>
18          /// <para>
19          /// Initializes a new <see cref="ArgumentLinkHasDependenciesException"/> instance.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="link">
```

```
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        /// <param name="paramName">
        /// <para>A param name.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) :
        ↪  base(FormatMessage(link, paramName), paramName) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkHasDependenciesException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(TLinkAddress link) :
        ↪  base(FormatMessage(link)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkHasDependenciesException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(string message, Exception innerException) :
        ↪  base(message, innerException) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkHasDependenciesException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException(string message) : base(message) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkHasDependenciesException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException() { }

        /// <summary>
        /// <para>
        /// Formats the message using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="paramName">
        /// <para>The param name.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [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.";

        /// <summary>
        /// <para>
        /// Formats the message using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [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
{
    /// <summary>
    /// <para>
    /// Represents the link with same value already exists exception.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Exception"/>
    public class LinkWithSameValueAlreadyExistsException : Exception
    {
        /// <summary>
        /// <para>
        /// The default message.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly string DefaultMessage = "Связь с таким же значением уже
        ↪  существует.";

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkWithSameValueAlreadyExistsException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkWithSameValueAlreadyExistsException(string message, Exception innerException)
        ↪  : base(message, innerException) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkWithSameValueAlreadyExistsException"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkWithSameValueAlreadyExistsException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [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
{
    /// <summary>
    /// <para>
    /// Represents the links limit reached exception.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksLimitReachedExceptionBase"/>
    public class LinksLimitReachedException<TLinkAddress> : LinksLimitReachedExceptionBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="limit">
        /// <para>A limit.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(string message, Exception innerException) :
        ↪  base(message, innerException) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
```

```
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(string message) : base(message) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException() : base(DefaultMessage) { }

        /// <summary>
        /// <para>
        /// Formats the message using the specified limit.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="limit">
        /// <para>The limit.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [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
{
    /// <summary>
    /// <para>
    /// Represents the links limit reached exception base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Exception"/>
    public abstract class LinksLimitReachedExceptionBase : Exception
    {
        /// <summary>
        /// <para>
        /// The default message.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly string DefaultMessage = "Достигнут лимит количества связей в
        ↪  хранилище.";

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedExceptionBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksLimitReachedExceptionBase(string message, Exception innerException) :
        ↪  base(message, innerException) { }

        /// <summary>
```

```
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedExceptionBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        [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
{
    /// <summary>
    /// <para>
    /// The hybrid.
    /// </para>
    /// <para></para>
    /// </summary>
    public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedSignExtendingConverter<TLinkAddress, long>
        ↪  _addressToInt64Converter = UncheckedSignExtendingConverter<TLinkAddress,
        ↪  long>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter
        ↪  = UncheckedConverter<long, TLinkAddress>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<TLinkAddress, ulong>
        ↪  _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<ulong, TLinkAddress>
        ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        /// <summary>
        /// <para>
        /// The default.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly UncheckedConverter<object, long> _objectToInt64Converter =
        ↪  UncheckedConverter<object, long>.Default;

        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong HalfOfNumberValuesRange =
        ↪  _addressToUInt64Converter.Convert(NumericType<TLinkAddress>.MaxValue) / 2;
        /// <summary>
        /// <para>
        /// The half of number values range.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TLinkAddress ExternalZero =
        ↪  _uInt64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);

        /// <summary>
        /// <para>
        /// The value.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress Value;

        /// <summary>
        /// <para>
        /// Gets the is nothing value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsNothing
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
        }

        /// <summary>
        /// <para>
        /// Gets the is internal value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsInternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => SignedValue > 0;
        }

        /// <summary>
        /// <para>
        /// Gets the is external value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsExternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
        }

        /// <summary>
        /// <para>
        /// Gets the signed value value.
        /// </para>
        /// <para></para>
        /// </summary>
        public long SignedValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _addressToInt64Converter.Convert(Value);
        }

```

```
135         /// <summary>
136         /// <para>
137         /// Gets the absolute value value.
138         /// </para>
139         /// <para></para>
140         /// </summary>
141         public long AbsoluteValue
142         {
143             [MethodImpl(MethodImplOptions.AggressiveInlining)]
144             get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
                ↪  Platform.Numbers.Math.Abs(SignedValue);
145         }
146 
147         /// <summary>
148         /// <para>
149         /// Initializes a new <see cref="Hybrid"/> instance.
150         /// </para>
151         /// <para></para>
152         /// </summary>
153         /// <param name="value">
154         /// <para>A value.</para>
155         /// <para></para>
156         /// </param>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         public Hybrid(TLinkAddress value)
159         {
160             Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
161             Value = value;
162         }
163 
164         /// <summary>
165         /// <para>
166         /// Initializes a new <see cref="Hybrid"/> instance.
167         /// </para>
168         /// <para></para>
169         /// </summary>
170         /// <param name="value">
171         /// <para>A value.</para>
172         /// <para></para>
173         /// </param>
174         /// <param name="isExternal">
175         /// <para>A is external.</para>
176         /// <para></para>
177         /// </param>
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         public Hybrid(TLinkAddress value, bool isExternal)
180         {
181             if (_equalityComparer.Equals(value, default) && isExternal)
182             {
183                 Value = ExternalZero;
184             }
185             else
186             {
187                 if (isExternal)
188                 {
189                     Value = Math<TLinkAddress>.Negate(value);
190                 }
191                 else
192                 {
193                     Value = value;
194                 }
195             }
196         }
197 
198         /// <summary>
199         /// <para>
200         /// Initializes a new <see cref="Hybrid"/> instance.
201         /// </para>
202         /// <para></para>
203         /// </summary>
204         /// <param name="value">
205         /// <para>A value.</para>
206         /// <para></para>
207         /// </param>
208         [MethodImpl(MethodImplOptions.AggressiveInlining)]
209         public Hybrid(object value) => Value =
            ↪  _int64ToAddressConverter.Convert(_objectToInt64Converter.Convert(value));
210 
```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        /// <param name="isExternal">
        /// <para>A is external.</para>
        /// <para></para>
        /// </param>
        [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;

```

```
276         [MethodImpl(MethodImplOptions.AggressiveInlining)]
277         public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
278
279         [MethodImpl(MethodImplOptions.AggressiveInlining)]
280         public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
            ↪ (int)hybrid.AbsoluteValue;
281
282         [MethodImpl(MethodImplOptions.AggressiveInlining)]
283         public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
284
285         [MethodImpl(MethodImplOptions.AggressiveInlining)]
286         public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
            ↪ (short)hybrid.AbsoluteValue;
287
288         [MethodImpl(MethodImplOptions.AggressiveInlining)]
289         public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);
290
291         [MethodImpl(MethodImplOptions.AggressiveInlining)]
292         public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
            ↪ (sbyte)hybrid.AbsoluteValue;
293
294         /// <summary>
295         /// <para>
296         /// Returns the string.
297         /// </para>
298         /// <para></para>
299         /// </summary>
300         /// <returns>
301         /// <para>The string</para>
302         /// <para></para>
303         /// </returns>
304         [MethodImpl(MethodImplOptions.AggressiveInlining)]
305         public override string ToString() => IsExternal ? $"<{AbsoluteValue}>" :
            ↪ Value.ToString();
306
307         /// <summary>
308         /// <para>
309         /// Determines whether this instance equals.
310         /// </para>
311         /// <para></para>
312         /// </summary>
313         /// <param name="other">
314         /// <para>The other.</para>
315         /// <para></para>
316         /// </param>
317         /// <returns>
318         /// <para>The bool</para>
319         /// <para></para>
320         /// </returns>
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
            ↪ other.Value);
323
324         /// <summary>
325         /// <para>
326         /// Determines whether this instance equals.
327         /// </para>
328         /// <para></para>
329         /// </summary>
330         /// <param name="obj">
331         /// <para>The obj.</para>
332         /// <para></para>
333         /// </param>
334         /// <returns>
335         /// <para>The bool</para>
336         /// <para></para>
337         /// </returns>
338         [MethodImpl(MethodImplOptions.AggressiveInlining)]
339         public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
            ↪ Equals(hybrid) : false;
340
341         /// <summary>
342         /// <para>
343         /// Gets the hash code.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [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>
        /// <para>Returns the set of constants that is necessary for effective communication
        ↪ with the methods of this interface.</para>
        /// <para>Возвращает набор констант, который необходим для эффективной коммуникации с
        ↪ методами этого интерфейса.</para>
        /// </summary>
        /// <remarks>
        /// <para>These constants are not changed since the creation of the links storage access
        ↪ point.</para>
        /// <para>Эти константы не меняются с момента создания точки доступа к хранилищу
        ↪ связей.</para>
        /// </remarks>
        TConstants Constants
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        #endregion

        #region Read

        /// <summary>
        /// <para>Counts and returns the total number of links in the storage that meet the
        ↪ specified restrictions.</para>
        /// <para>Подсчитывает и возвращает общее число связей находящихся в хранилище,
        ↪ соответствующих указанным ограничениям.</para>
        /// </summary>
        /// <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>
94          /// <para>Deletes links that match the specified restrictions.</para>
95          /// <para>Удаляет связи соответствующие указанным ограничениям.</para>
96          /// <param name="restrictions">
97          /// <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>
```

```
98          /// <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>
99          /// </param>
100         /// </summary>
101         [MethodImpl(MethodImplOptions.AggressiveInlining)]
102         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.
103
104         #endregion
105     }
106 }
```

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      /// <summary>
12      /// <para>
13      /// Represents the links extensions.
14      /// </para>
15      /// <para></para>
16      /// </summary>
17      public static class ILinksExtensions
18      {
19          /// <summary>
20          /// <para>
21          /// Counts the links.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          /// <typeparam name="TLinkAddress">
26          /// <para>The link address.</para>
27          /// <para></para>
28          /// </typeparam>
29          /// <typeparam name="TConstants">
30          /// <para>The constants.</para>
31          /// <para></para>
32          /// </typeparam>
33          /// <param name="links">
34          /// <para>The links.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="restrictions">
38          /// <para>The restrictions.</para>
39          /// <para></para>
40          /// </param>
41          /// <returns>
42          /// <para>The link address</para>
43          /// <para></para>
44          /// </returns>
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, params TLinkAddress[] restrictions)
47              where TConstants : LinksConstants<TLinkAddress>
48              => links.Count(restrictions);
49
50          /// <summary>
51          /// Возвращает значение, определяющее существует ли связь с указанным индексом в
            ↪  хранилище связей.
52          /// </summary>
53          /// <param name="links">Хранилище связей.</param>
54          /// <param name="link">Индекс проверяемой на существование связи.</param>
55          /// <returns>Значение, определяющее существует ли связь.</returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
58              where TConstants : LinksConstants<TLinkAddress>
59          {
60              var constants = links.Constants;
```

```
61              return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
                ↪  && Comparer<TLinkAddress>.Default.Compare(links.Count(new
                ↪  LinkAddress<TLinkAddress>(link)), default) > 0);
62          }
63
64          /// <param name="links">Хранилище связей.</param>
65          /// <param name="link">Индекс проверяемой на существование связи.</param>
66          /// <remarks>
67          /// TODO: May be move to EnsureExtensions or make it both there and here
68          /// </remarks>
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
71              where TConstants : LinksConstants<TLinkAddress>
72          {
73              if (!links.Exists(link))
74              {
75                  throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
76              }
77          }
78
79          /// <param name="links">Хранилище связей.</param>
80          /// <param name="link">Индекс проверяемой на существование связи.</param>
81          /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
82          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83          public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link, string argumentName)
84              where TConstants : LinksConstants<TLinkAddress>
85          {
86              if (!links.Exists(link))
87              {
88                  throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
89              }
90          }
91
92          /// <summary>
93          /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
94          /// </summary>
95          /// <param name="links">Хранилище связей.</param>
96          /// <param name="handler">Обработчик каждой подходящей связи.</param>
97          /// <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>
98          /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, Func<IList<TLinkAddress>, TLinkAddress> handler, params
            ↪  TLinkAddress[] restrictions)
101             where TConstants : LinksConstants<TLinkAddress>
102             => links.Each(handler, restrictions);
103
104         /// <summary>
105         /// Возвращает части-значения для связи с указанным индексом.
106         /// </summary>
107         /// <param name="links">Хранилище связей.</param>
108         /// <param name="link">Индекс связи.</param>
109         /// <returns>Уникальную связь.</returns>
110         [MethodImpl(MethodImplOptions.AggressiveInlining)]
111         public static IList<TLinkAddress> GetLink<TLinkAddress, TConstants>(this
            ↪  ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
112             where TConstants : LinksConstants<TLinkAddress>
113         {
114             var constants = links.Constants;
115             if (constants.IsExternalReference(link))
116             {
117                 return new Point<TLinkAddress>(link, constants.TargetPart + 1);
118             }
119             var linkPartsSetter = new Setter<IList<TLinkAddress>,
                ↪  TLinkAddress>(constants.Continue, constants.Break);
120             links.Each(linkPartsSetter.SetAndReturnTrue, link);
121             return linkPartsSetter.Result;
122         }
123
124         #region Points
125
```

```
126         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
127         /// <param name="links">Хранилище связей.</param>
128         /// <param name="link">Индекс проверяемой связи.</param>
129         /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
130         /// <remarks>
131         /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
            ↪  связь.
132         /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
            ↪  точка и пара существовать одновременно?
133         /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
            ↪  сортировать по индексу в массиве связей?
134         /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
135         /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
            ↪  самой себя любого размера?
136         /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
            ↪  одной ссылки на себя (частичной точки).
137         /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
138         /// самостоятельный цикл через себя? Что если предоставить все варианты использования
            ↪  связей?
139         /// Что если разрешить и нули, а так же частичные варианты?
140         ///
141         /// 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.
142         /// 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.
143         ///
144         /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
            ↪  промежуточную связь,
145         /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
146         /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
147         /// </remarks>
148         [MethodImpl(MethodImplOptions.AggressiveInlining)]
149         public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
150             where TConstants : LinksConstants<TLinkAddress>
151         {
152             if (links.Constants.IsExternalReference(link))
153             {
154                 return true;
155             }
156             links.EnsureLinkExists(link);
157             return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
158         }
159 
160         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой частично (связью замкнутой на себе как минимум один раз).</summary>
161         /// <param name="links">Хранилище связей.</param>
162         /// <param name="link">Индекс проверяемой связи.</param>
163         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
164         /// <remarks>
165         /// Достаточно любой одной ссылки на себя.
166         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
            ↪  ссылки на себя (на эту связь).
167         /// </remarks>
168         [MethodImpl(MethodImplOptions.AggressiveInlining)]
169         public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
170             where TConstants : LinksConstants<TLinkAddress>
171         {
172             if (links.Constants.IsExternalReference(link))
173             {
174                 return true;
175             }
176             links.EnsureLinkExists(link);
177             return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
178         }
179 
180         #endregion
181     }
182 }
```

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
```

```
{
    /// <summary>
    /// <para>
    /// Defines the synchronized links.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISynchronized{TLinks}"/>
    /// <seealso cref="ILinks{TLinkAddress, TConstants}"/>
    public interface ISynchronizedLinks<TLinkAddress, TLinks, TConstants> :
    ↪  ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
        where TLinks : ILinks<TLinkAddress, TConstants>
        where TConstants : LinksConstants<TLinkAddress>
    {
    }
}
```

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

```
using System;
using System.Collections;
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 the link address.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IEquatable{LinkAddress{TLinkAddress}}"/>
    /// <seealso cref="IList{TLinkAddress}"/>
    public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
    ↪  IList<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// Gets the index value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// The not supported exception.
        /// </para>
        /// <para></para>
        /// </summary>
        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();
        }

        /// <summary>
        /// <para>
        /// Gets the count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => 1;
        }

        /// <summary>
        /// <para>
        /// Gets the is read only value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkAddress"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkAddress(TLinkAddress index) => Index = index;

        /// <summary>
        /// <para>
        /// Adds the item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Clears this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Determines whether this instance contains.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

        /// <summary>
        /// <para>
        /// Copies the to using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <param name="arrayIndex">
        /// <para>The array index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;

        /// <summary>
        /// <para>
        /// Gets the enumerator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>An enumerator of t link address</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<TLinkAddress> GetEnumerator()
        {
            yield return Index;
        }

        /// <summary>
        /// <para>
        /// Indexes the of using the specified item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
        ↪  0 : -1;

        /// <summary>
        /// <para>
        /// Inserts the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Determines whether this instance remove.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
```

```
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Removes the at using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Gets the enumerator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The enumerator</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator()
        {
            yield return Index;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [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);

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
```

```
289             /// </returns>
290             [MethodImpl(MethodImplOptions.AggressiveInlining)]
291             public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
                ↪  ? Equals(linkAddress) : false;
292
293             /// <summary>
294             /// <para>
295             /// Gets the hash code.
296             /// </para>
297             /// <para></para>
298             /// </summary>
299             /// <returns>
300             /// <para>The int</para>
301             /// <para></para>
302             /// </returns>
303             [MethodImpl(MethodImplOptions.AggressiveInlining)]
304             public override int GetHashCode() => Index.GetHashCode();
305
306             /// <summary>
307             /// <para>
308             /// Returns the string.
309             /// </para>
310             /// <para></para>
311             /// </summary>
312             /// <returns>
313             /// <para>The string</para>
314             /// <para></para>
315             /// </returns>
316             [MethodImpl(MethodImplOptions.AggressiveInlining)]
317             public override string ToString() => Index.ToString();
318
319             [MethodImpl(MethodImplOptions.AggressiveInlining)]
320             public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
                ↪  right)
321             {
322                 if (left == null && right == null)
323                 {
324                     return true;
325                 }
326                 if (left == null)
327                 {
328                     return false;
329                 }
330                 return left.Equals(right);
331             }
332
333             [MethodImpl(MethodImplOptions.AggressiveInlining)]
334             public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
                ↪  right) => !(left == right);
335     }
336 }
```

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     /// <summary>
12     /// <para>
13     /// Represents the links constants.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksConstantsBase"/>
18     public class LinksConstants<TLinkAddress> : LinksConstantsBase
19     {
20         /// <summary>
21         /// <para>
22         /// The increment.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
```

```
27          /// <summary>
28          /// <para>
29          /// The default.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          private static readonly UncheckedConverter<ulong, TLinkAddress>
            ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
34
35          #region Link parts
36
37          /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
            ↪  самой связи.</summary>
38          public int IndexPart
39          {
40              [MethodImpl(MethodImplOptions.AggressiveInlining)]
41              get;
42          }
43
44          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
            ↪  часть-значение).</summary>
45          public int SourcePart
46          {
47              [MethodImpl(MethodImplOptions.AggressiveInlining)]
48              get;
49          }
50
51          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
            ↪  (последняя часть-значение).</summary>
52          public int TargetPart
53          {
54              [MethodImpl(MethodImplOptions.AggressiveInlining)]
55              get;
56          }
57
58          #endregion
59
60          #region Flow control
61
62          /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
63          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
64          public TLinkAddress Continue
65          {
66              [MethodImpl(MethodImplOptions.AggressiveInlining)]
67              get;
68          }
69
70          /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
71          public TLinkAddress Skip
72          {
73              [MethodImpl(MethodImplOptions.AggressiveInlining)]
74              get;
75          }
76
77          /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
78          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
79          public TLinkAddress Break
80          {
81              [MethodImpl(MethodImplOptions.AggressiveInlining)]
82              get;
83          }
84
85          #endregion
86
87          #region Special symbols
88
89          /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
90          public TLinkAddress Null
91          {
92              [MethodImpl(MethodImplOptions.AggressiveInlining)]
93              get;
94          }
95
96          /// <summary>Возвращает значение, обозначающее любую связь.</summary>
97          /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
            ↪  создавать все варианты последовательностей в функции Create.</remarks>
98          public TLinkAddress Any
99          {
```

```
100             [MethodImpl(MethodImplOptions.AggressiveInlining)]
101             get;
102         }
103 
104         /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
105         public TLinkAddress Itself
106         {
107             [MethodImpl(MethodImplOptions.AggressiveInlining)]
108             get;
109         }
110 
111         #endregion
112 
113         #region References
114 
115         /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
            ↪  ссылок).</summary>
116         public Range<TLinkAddress> InternalReferencesRange
117         {
118             [MethodImpl(MethodImplOptions.AggressiveInlining)]
119             get;
120         }
121 
122         /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
            ↪  ссылок).</summary>
123         public Range<TLinkAddress>? ExternalReferencesRange
124         {
125             [MethodImpl(MethodImplOptions.AggressiveInlining)]
126             get;
127         }
128 
129         #endregion
130 
131         /// <summary>
132         /// <para>
133         /// Initializes a new <see cref="LinksConstants"/> instance.
134         /// </para>
135         /// <para></para>
136         /// </summary>
137         /// <param name="targetPart">
138         /// <para>A target part.</para>
139         /// <para></para>
140         /// </param>
141         /// <param name="possibleInternalReferencesRange">
142         /// <para>A possible internal references range.</para>
143         /// <para></para>
144         /// </param>
145         /// <param name="possibleExternalReferencesRange">
146         /// <para>A possible external references range.</para>
147         /// <para></para>
148         /// </param>
149         [MethodImpl(MethodImplOptions.AggressiveInlining)]
150         public LinksConstants(int targetPart, Range<TLinkAddress>
            ↪  possibleInternalReferencesRange, Range<TLinkAddress>?
            ↪  possibleExternalReferencesRange)
151         {
152             IndexPart = 0;
153             SourcePart = 1;
154             TargetPart = targetPart;
155             Null = default;
156             Break = default;
157             var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
158             Continue = currentInternalReferenceIndex;
159             Skip = Arithmetic.Decrement(ref currentInternalReferenceIndex);
160             Any = Arithmetic.Decrement(ref currentInternalReferenceIndex);
161             Itself = Arithmetic.Decrement(ref currentInternalReferenceIndex);
162             Arithmetic.Decrement(ref currentInternalReferenceIndex);
163             InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
                ↪  currentInternalReferenceIndex);
164             ExternalReferencesRange = possibleExternalReferencesRange;
165         }
166 
167         /// <summary>
168         /// <para>
169         /// Initializes a new <see cref="LinksConstants"/> instance.
170         /// </para>
171         /// <para></para>
172         /// </summary>
173         /// <param name="targetPart">
```

```
        /// <para>A target part.</para>
        /// <para></para>
        /// </param>
        /// <param name="enableExternalReferencesSupport">
        /// <para>A enable external references support.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
        ↪  this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
        ↪  GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksConstants"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="possibleInternalReferencesRange">
        /// <para>A possible internal references range.</para>
        /// <para></para>
        /// </param>
        /// <param name="possibleExternalReferencesRange">
        /// <para>A possible external references range.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
        ↪  Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
        ↪  possibleInternalReferencesRange, possibleExternalReferencesRange) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksConstants"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="enableExternalReferencesSupport">
        /// <para>A enable external references support.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(bool enableExternalReferencesSupport) :
        ↪  this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
        ↪  GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksConstants"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="targetPart">
        /// <para>A target part.</para>
        /// <para></para>
        /// </param>
        /// <param name="possibleInternalReferencesRange">
        /// <para>A possible internal references range.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksConstants(int targetPart, Range<TLinkAddress>
        ↪  possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
        ↪  null) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksConstants"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="possibleInternalReferencesRange">
        /// <para>A possible internal references range.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
242         public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
            ↪ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
243
244         /// <summary>
245         /// <para>
246         /// Initializes a new <see cref="LinksConstants"/> instance.
247         /// </para>
248         /// <para></para>
249         /// </summary>
250         [MethodImpl(MethodImplOptions.AggressiveInlining)]
251         public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
            ↪ false) { }
252
253         /// <summary>
254         /// <para>
255         /// Gets the default internal references range using the specified enable external
            ↪ references support.
256         /// </para>
257         /// <para></para>
258         /// </summary>
259         /// <param name="enableExternalReferencesSupport">
260         /// <para>The enable external references support.</para>
261         /// <para></para>
262         /// </param>
263         /// <returns>
264         /// <para>A range of t link address</para>
265         /// <para></para>
266         /// </returns>
267         [MethodImpl(MethodImplOptions.AggressiveInlining)]
268         public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
            ↪ enableExternalReferencesSupport)
269         {
270             if (enableExternalReferencesSupport)
271             {
272                 return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
                    ↪ rValuesRange));
273             }
274             else
275             {
276                 return (_one, NumericType<TLinkAddress>.MaxValue);
277             }
278         }
279
280         /// <summary>
281         /// <para>
282         /// Gets the default external references range using the specified enable external
            ↪ references support.
283         /// </para>
284         /// <para></para>
285         /// </summary>
286         /// <param name="enableExternalReferencesSupport">
287         /// <para>The enable external references support.</para>
288         /// <para></para>
289         /// </param>
290         /// <returns>
291         /// <para>A range of t link address</para>
292         /// <para></para>
293         /// </returns>
294         [MethodImpl(MethodImplOptions.AggressiveInlining)]
295         public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
            ↪ enableExternalReferencesSupport)
296         {
297             if (enableExternalReferencesSupport)
298             {
299                 return (Hybrid<TLinkAddress>.ExternalZero, NumericType<TLinkAddress>.MaxValue);
300             }
301             else
302             {
303                 return null;
304             }
305         }
306     }
307 }
```

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
```

```
{
    /// <summary>
    /// <para>
    /// Represents the links constants base.
    /// </para>
    /// <para></para>
    /// </summary>
    public abstract class LinksConstantsBase
    {
        /// <summary>
        /// <para>
        /// The default target part.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly int DefaultTargetPart = 2;
    }
}
```

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

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

using System.Runtime.CompilerServices;

namespace Platform.Data
{
    /// <summary>
    /// <para>
    /// Represents the links constants extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class LinksConstantsExtensions
    {
        /// <summary>
        /// <para>
        /// Determines whether is reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link address.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="linksConstants">
        /// <para>The links constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="address">
        /// <para>The address.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪  linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
        ↪  || linksConstants.IsExternalReference(address);

        /// <summary>
        /// <para>
        /// Determines whether is internal reference.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link address.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="linksConstants">
        /// <para>The links constants.</para>
        /// <para></para>
        /// </param>
        /// <param name="address">
        /// <para>The address.</para>
        /// <para></para>
```

```
57          /// </param>
58          /// <returns>
59          /// <para>The bool</para>
60          /// <para></para>
61          /// </returns>
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
            ↪  linksConstants, TLinkAddress address) =>
            ↪  linksConstants.InternalReferencesRange.Contains(address);
64
65          /// <summary>
66          /// <para>
67          /// Determines whether is external reference.
68          /// </para>
69          /// <para></para>
70          /// </summary>
71          /// <typeparam name="TLinkAddress">
72          /// <para>The link address.</para>
73          /// <para></para>
74          /// </typeparam>
75          /// <param name="linksConstants">
76          /// <para>The links constants.</para>
77          /// <para></para>
78          /// </param>
79          /// <param name="address">
80          /// <para>The address.</para>
81          /// <para></para>
82          /// </param>
83          /// <returns>
84          /// <para>The bool</para>
85          /// <para></para>
86          /// </returns>
87          [MethodImpl(MethodImplOptions.AggressiveInlining)]
88          public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
            ↪  linksConstants, TLinkAddress address) =>
            ↪  linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
89      }
90  }
```

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       /// <summary>
9       /// <para>
10      /// Represents the address to raw number converter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="IConverter{TLink}"/>
15      public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
16      {
17          /// <summary>
18          /// <para>
19          /// Converts the source.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="source">
24          /// <para>The source.</para>
25          /// <para></para>
26          /// </param>
27          /// <returns>
28          /// <para>The link</para>
29          /// <para></para>
30          /// </returns>
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
33      }
34  }
```

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

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

```

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

namespace Platform.Data.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the raw number to address converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{TLink}"/>
    public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        static private readonly UncheckedConverter<long, TLink> _converter =
        ↪  UncheckedConverter<long, TLink>.Default;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [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
{
    /// <summary>
    /// <para>
    /// Represents the point.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IEquatable{LinkAddress{TLinkAddress}}"/>
    /// <seealso cref="IList{TLinkAddress}"/>
    public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// Gets the index value.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        public TLinkAddress Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// Gets the size value.
        /// </para>
        /// <para></para>
        /// </summary>
        public int Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// The not supported exception.
        /// </para>
        /// <para></para>
        /// </summary>
        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();
        }

        /// <summary>
        /// <para>
        /// Gets the count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Size;
        }

        /// <summary>
        /// <para>
        /// Gets the is read only value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Point"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
```

```
114         /// <para>A size.</para>
115         /// <para></para>
116         /// </param>
117         [MethodImpl(MethodImplOptions.AggressiveInlining)]
118         public Point(TLinkAddress index, int size)
119         {
120             Index = index;
121             Size = size;
122         }
123 
124         /// <summary>
125         /// <para>
126         /// Adds the item.
127         /// </para>
128         /// <para></para>
129         /// </summary>
130         /// <param name="item">
131         /// <para>The item.</para>
132         /// <para></para>
133         /// </param>
134         [MethodImpl(MethodImplOptions.AggressiveInlining)]
135         public void Add(TLinkAddress item) => throw new NotSupportedException();
136 
137         /// <summary>
138         /// <para>
139         /// Clears this instance.
140         /// </para>
141         /// <para></para>
142         /// </summary>
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public void Clear() => throw new NotSupportedException();
145 
146         /// <summary>
147         /// <para>
148         /// Determines whether this instance contains.
149         /// </para>
150         /// <para></para>
151         /// </summary>
152         /// <param name="item">
153         /// <para>The item.</para>
154         /// <para></para>
155         /// </param>
156         /// <returns>
157         /// <para>The bool</para>
158         /// <para></para>
159         /// </returns>
160         [MethodImpl(MethodImplOptions.AggressiveInlining)]
161         public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index)
            ↪  ? true : false;
162 
163         /// <summary>
164         /// <para>
165         /// Copies the to using the specified array.
166         /// </para>
167         /// <para></para>
168         /// </summary>
169         /// <param name="array">
170         /// <para>The array.</para>
171         /// <para></para>
172         /// </param>
173         /// <param name="arrayIndex">
174         /// <para>The array index.</para>
175         /// <para></para>
176         /// </param>
177         [MethodImpl(MethodImplOptions.AggressiveInlining)]
178         public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
179 
180         /// <summary>
181         /// <para>
182         /// Gets the enumerator.
183         /// </para>
184         /// <para></para>
185         /// </summary>
186         /// <returns>
187         /// <para>An enumerator of t link address</para>
188         /// <para></para>
189         /// </returns>
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public IEnumerator<TLinkAddress> GetEnumerator()
        {
            for (int i = 0; i < Size; i++)
            {
                yield return Index;
            }
        }

        /// <summary>
        /// <para>
        /// Indexes the of using the specified item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
        ↪  0 : -1;

        /// <summary>
        /// <para>
        /// Inserts the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Determines whether this instance remove.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Removes the at using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Gets the enumerator.
        /// </para>
        /// <para></para>
```

```
268         /// </summary>
269         /// <returns>
270         /// <para>The enumerator</para>
271         /// <para></para>
272         /// </returns>
273         [MethodImpl(MethodImplOptions.AggressiveInlining)]
274         IEnumerator IEnumerable.GetEnumerator()
275         {
276             for (int i = 0; i < Size; i++)
277             {
278                 yield return Index;
279             }
280         }
281 
282         /// <summary>
283         /// <para>
284         /// Determines whether this instance equals.
285         /// </para>
286         /// <para></para>
287         /// </summary>
288         /// <param name="other">
289         /// <para>The other.</para>
290         /// <para></para>
291         /// </param>
292         /// <returns>
293         /// <para>The bool</para>
294         /// <para></para>
295         /// </returns>
296         [MethodImpl(MethodImplOptions.AggressiveInlining)]
297         public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
            ↪  _equalityComparer.Equals(Index, other.Index);
298 
299         [MethodImpl(MethodImplOptions.AggressiveInlining)]
300         public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
            ↪  linkAddress.Index;
301 
302         /// <summary>
303         /// <para>
304         /// Determines whether this instance equals.
305         /// </para>
306         /// <para></para>
307         /// </summary>
308         /// <param name="obj">
309         /// <para>The obj.</para>
310         /// <para></para>
311         /// </param>
312         /// <returns>
313         /// <para>The bool</para>
314         /// <para></para>
315         /// </returns>
316         [MethodImpl(MethodImplOptions.AggressiveInlining)]
317         public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
            ↪  Equals(linkAddress) : false;
318 
319         /// <summary>
320         /// <para>
321         /// Gets the hash code.
322         /// </para>
323         /// <para></para>
324         /// </summary>
325         /// <returns>
326         /// <para>The int</para>
327         /// <para></para>
328         /// </returns>
329         [MethodImpl(MethodImplOptions.AggressiveInlining)]
330         public override int GetHashCode() => Index.GetHashCode();
331 
332         /// <summary>
333         /// <para>
334         /// Returns the string.
335         /// </para>
336         /// <para></para>
337         /// </summary>
338         /// <returns>
339         /// <para>The string</para>
340         /// <para></para>
341         /// </returns>
342         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
343         public override string ToString() => Index.ToString();
344 
345         [MethodImpl(MethodImplOptions.AggressiveInlining)]
346         public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
347         {
348             if (left == null && right == null)
349             {
350                 return true;
351             }
352             if (left == null)
353             {
354                 return false;
355             }
356             return left.Equals(right);
357         }
358 
359         [MethodImpl(MethodImplOptions.AggressiveInlining)]
360         public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
            ↪  !(left == right);
361 
362         /// <summary>
363         /// <para>
364         /// Determines whether is full point.
365         /// </para>
366         /// <para></para>
367         /// </summary>
368         /// <param name="link">
369         /// <para>The link.</para>
370         /// <para></para>
371         /// </param>
372         /// <returns>
373         /// <para>The bool</para>
374         /// <para></para>
375         /// </returns>
376         [MethodImpl(MethodImplOptions.AggressiveInlining)]
377         public static bool IsFullPoint(params TLinkAddress[] link) =>
            ↪  IsFullPoint((IList<TLinkAddress>)link);
378 
379         /// <summary>
380         /// <para>
381         /// Determines whether is full point.
382         /// </para>
383         /// <para></para>
384         /// </summary>
385         /// <param name="link">
386         /// <para>The link.</para>
387         /// <para></para>
388         /// </param>
389         /// <returns>
390         /// <para>The bool</para>
391         /// <para></para>
392         /// </returns>
393         [MethodImpl(MethodImplOptions.AggressiveInlining)]
394         public static bool IsFullPoint(IList<TLinkAddress> link)
395         {
396             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
397             Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
                ↪  determine link's pointness using only its identifier.");
398             return IsFullPointUnchecked(link);
399         }
400 
401         /// <summary>
402         /// <para>
403         /// Determines whether is full point unchecked.
404         /// </para>
405         /// <para></para>
406         /// </summary>
407         /// <param name="link">
408         /// <para>The link.</para>
409         /// <para></para>
410         /// </param>
411         /// <returns>
412         /// <para>The result.</para>
413         /// <para></para>
414         /// </returns>
415         [MethodImpl(MethodImplOptions.AggressiveInlining)]
416         public static bool IsFullPointUnchecked(IList<TLinkAddress> link)
417         {
```

```
            var result = true;
            for (var i = 1; result && i < link.Count; i++)
            {
                result = _equalityComparer.Equals(link[0], link[i]);
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether is partial point.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPoint(params TLinkAddress[] link) =>
        ↪  IsPartialPoint((IList<TLinkAddress>)link);

        /// <summary>
        /// <para>
        /// Determines whether is partial point.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPoint(IList<TLinkAddress> link)
        {
            Ensure.Always.ArgumentNotEmpty(link, nameof(link));
            Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
            ↪  determine link's pointness using only its identifier.");
            return IsPartialPointUnchecked(link);
        }

        /// <summary>
        /// <para>
        /// Determines whether is partial point unchecked.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPartialPointUnchecked(IList<TLinkAddress> link)
        {
            var result = false;
            for (var i = 1; !result && i < link.Count; i++)
            {
                result = _equalityComparer.Equals(link[0], link[i]);
            }
            return result;
        }
    }
}
```

1.17 ./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>
    {
        /// <summary>
        /// <para>
        /// Triggers the condition.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="condition">
        /// <para>The condition.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of i list i list t link address</para>
        /// <para></para>
        /// </returns>
        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);

        /// <summary>
        /// <para>
        /// Triggers the restriction.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restriction">
        /// <para>The restriction.</para>
        /// <para></para>
        /// </param>
        /// <param name="matchedHandler">
        /// <para>The matched handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitution">
        /// <para>The substitution.</para>
        /// <para></para>
        /// </param>
        /// <param name="substitutedHandler">
        /// <para>The substituted handler.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
```

```
74          TLinkAddress Trigger(IList<TLinkAddress> restriction, Func<IList<TLinkAddress>,
            ↪  IList<TLinkAddress>, TLinkAddress> matchedHandler,
75                IList<TLinkAddress> substitution, Func<IList<TLinkAddress>, IList<TLinkAddress>,
                  ↪  TLinkAddress> substitutedHandler);
76      }
77
78      /// <remarks>Extended with small optimization.</remarks>
79      public partial interface IUniLinks<TLinkAddress>
80      {
81          /// <remarks>
82          /// Something simple should be simple and optimized.
83          /// </remarks>
84          TLinkAddress Count(IList<TLinkAddress> restrictions);
85      }
86  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// CRUD aliases for IUniLinks.
11      /// </remarks>
12      public interface IUniLinksCRUD<TLinkAddress>
13      {
14          /// <summary>
15          /// <para>
16          /// Reads the part type.
17          /// </para>
18          /// <para></para>
19          /// </summary>
20          /// <param name="partType">
21          /// <para>The part type.</para>
22          /// <para></para>
23          /// </param>
24          /// <param name="link">
25          /// <para>The link.</para>
26          /// <para></para>
27          /// </param>
28          /// <returns>
29          /// <para>The link address</para>
30          /// <para></para>
31          /// </returns>
32          TLinkAddress Read(int partType, TLinkAddress link);
33          /// <summary>
34          /// <para>
35          /// Reads the handler.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="handler">
40          /// <para>The handler.</para>
41          /// <para></para>
42          /// </param>
43          /// <param name="pattern">
44          /// <para>The pattern.</para>
45          /// <para></para>
46          /// </param>
47          /// <returns>
48          /// <para>The link address</para>
49          /// <para></para>
50          /// </returns>
51          TLinkAddress Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
52          /// <summary>
53          /// <para>
54          /// Creates the parts.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          /// <param name="parts">
59          /// <para>The parts.</para>
60          /// <para></para>
61          /// </param>
```

```
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Create(IList<TLinkAddress> parts);
        /// <summary>
        /// <para>
        /// Updates the before.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="before">
        /// <para>The before.</para>
        /// <para></para>
        /// </param>
        /// <param name="after">
        /// <para>The after.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Update(IList<TLinkAddress> before, IList<TLinkAddress> after);
        /// <summary>
        /// <para>
        /// Deletes the parts.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="parts">
        /// <para>The parts.</para>
        /// <para></para>
        /// </param>
        void Delete(IList<TLinkAddress> parts);
    }
}
```

1.19 ./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>
    {
        /// <summary>
        /// <para>
        /// Gets the part type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="partType">
        /// <para>The part type.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Get(int partType, TLinkAddress link);
        /// <summary>
        /// <para>
        /// Gets the handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
```

```
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="pattern">
        /// <para>The pattern.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        /// <summary>
        /// <para>
        /// Sets the before.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="before">
        /// <para>The before.</para>
        /// <para></para>
        /// </param>
        /// <param name="after">
        /// <para>The after.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Set(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

1.20 ./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
```

```
43          /// * In(null, new[] { }) creates point (link that points to itself using minimum number
            ↪  of parts).
44          /// * 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  }
```

1.21 ./csharp/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  }
```

1.22 ./csharp/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          /// <summary>
15          /// <para>
16          /// Reads the part type.
17          /// </para>
18          /// <para></para>
19          /// </summary>
20          /// <param name="partType">
21          /// <para>The part type.</para>
22          /// <para></para>
23          /// </param>
24          /// <param name="link">
25          /// <para>The link.</para>
26          /// <para></para>
27          /// </param>
28          /// <returns>
29          /// <para>The link address</para>
30          /// <para></para>
```

```
        /// </returns>
        TLinkAddress Read(int partType, TLinkAddress link);
        /// <summary>
        /// <para>
        /// Determines whether this instance read.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="pattern">
        /// <para>The pattern.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress> pattern);
        /// <summary>
        /// <para>
        /// Writes the before.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="before">
        /// <para>The before.</para>
        /// <para></para>
        /// </param>
        /// <param name="after">
        /// <para>The after.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Write(IList<TLinkAddress> before, IList<TLinkAddress> after);
    }
}
```

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

```
using Xunit;

namespace Platform.Data.Tests
{
    /// <summary>
    /// <para>
    /// Represents the hybrid tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class HybridTests
    {
        /// <summary>
        /// <para>
        /// Tests that object constructor test.
        /// </para>
        /// <para></para>
        /// </summary>
        [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);
        }
    }
}
```

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

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

namespace Platform.Data.Tests
{
    /// <summary>
    /// <para>
    /// Represents the links constants tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class LinksConstantsTests
    {
        /// <summary>
        /// <para>
        /// Tests that constructor test.
        /// </para>
        /// <para></para>
        /// </summary>
        [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);
        }

        /// <summary>
        /// <para>
        /// Tests that external references test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void ExternalReferencesTest()
        {
            TestExternalReferences<ulong, long>();
            TestExternalReferences<uint, int>();
            TestExternalReferences<ushort, short>();
            TestExternalReferences<byte, sbyte>();
        }

        /// <summary>
        /// <para>
        /// Tests the external references.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TUnsigned">
        /// <para>The unsigned.</para>
        /// <para></para>
        /// </typeparam>
        /// <typeparam name="TSigned">
        /// <para>The signed.</para>
        /// <para></para>
        /// </typeparam>
        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);
        }
```

```
76          }
77      }
```

Index