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{TLinkAddress}"/>
        ↪  instance.
        /// </para>
        /// <para>
        /// Инициализирует новый экземпляр класса <see
        ↪  cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para>Связь.</para>
        /// </param>
        /// <param name="argumentName">
        /// <para>A argument name.</para>
        /// <para>Имя аргумента.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link, string argumentName) :
        ↪  base(FormatMessage(link, argumentName), argumentName) { }

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
        ↪  instance.
        /// </para>
        /// <para>
        /// Инициализирует новый экземпляр класса <see
        ↪  cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
        /// </para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para>Сообщение.</para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para>Внутренняя ошибка.</para>
```

```
68          /// </param>
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public ArgumentLinkDoesNotExistsException(string message, Exception innerException) :
            ↪ base(message, innerException) { }
71
72          /// <summary>
73          /// <para>
74          /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
            ↪ instance.
75          /// </para>
76          /// <para>
77          /// Инициализирует новый экземпляр класса <see
            ↪ cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
78          /// </para>
79          /// </summary>
80          /// <param name="message">
81          /// <para>A message.</para>
82          /// <para>Сообщение.</para>
83          /// </param>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public ArgumentLinkDoesNotExistsException(string message) : base(message) { }
86
87          /// <summary>
88          /// <para>
89          /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
            ↪ instance.
90          /// </para>
91          /// <para>
92          /// Инициализирует новый экземпляр класса <see
            ↪ cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
93          /// </para>
94          /// </summary>
95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
96          public ArgumentLinkDoesNotExistsException() { }
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          private static string FormatMessage(TLinkAddress link, string argumentName) => $"Связь
            ↪ [{link}] переданная в аргумент [{argumentName}] не существует.";
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
            ↪ качестве аргумента не существует.";
101     }
102 }
```

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{TLinkAddress}"/>
            ↪ instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="link">
24         /// <para>A link.</para>
25         /// <para>Связь.</para>
26         /// </param>
27         /// <param name="paramName">
28         /// <para>A param name.</para>
29         /// <para>Имя параметра.</para>
30         /// </param>
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) :
            ↪ base(FormatMessage(link, paramName), paramName) { }
```

```

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkHasDependenciesException{TLinkAddress}"/>
        ↪  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{TLinkAddress}"/>
        ↪  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{TLinkAddress}"/>
        ↪  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{TLinkAddress}"/>
        ↪  instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkHasDependenciesException() { }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи
        ↪  [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые
        ↪  препятствуют изменению её внутренней структуры.";
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной
        ↪  в качестве аргумента присутствуют зависимости, которые препятствуют изменению её
        ↪  внутренней структуры.";
    }
}
```

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

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

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

namespace Platform.Data.Exceptions
{
    /// <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{TLinkAddress}"/> 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{TLinkAddress}"/> 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{TLinkAddress}"/> 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{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException() : base(DefaultMessage) { }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества
        ↪  связей в хранилище ({limit}).";
    }
}
```

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

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

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

namespace Platform.Data.Exceptions
{
    /// <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>>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer = EqualityComparer<TLinkAddress>.Default;
        private static readonly UncheckedSignExtendingConverter<TLinkAddress, long> _addressToInt64Converter = UncheckedSignExtendingConverter<TLinkAddress, long>.Default;
        private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter = UncheckedConverter<long, TLinkAddress>.Default;
        private static readonly UncheckedConverter<TLinkAddress, ulong> _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLinkAddress> _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        private static readonly UncheckedConverter<object, long> _objectToInt64Converter = UncheckedConverter<object, long>.Default;

        /// <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);
        }

        /// <summary>
        /// <para>
        /// Gets the absolute value value.
        /// </para>
        /// <para></para>
        /// </summary>
        public long AbsoluteValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
            ↪  Platform.Numbers.Math.Abs(SignedValue);
        }

        /// <summary>
```

```
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value)
        {
            Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
            Value = value;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> 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(TLinkAddress value, bool isExternal)
        {
            if (_equalityComparer.Equals(value, default) && isExternal)
            {
                Value = ExternalZero;
            }
            else
            {
                if (isExternal)
                {
                    Value = Math<TLinkAddress>.Negate(value);
                }
                else
                {
                    Value = value;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(object value) => Value =
        ↪  _int64ToAddressConverter.Convert(_objectToInt64Converter.Convert(value));

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> 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>
```

```
189         [MethodImpl(MethodImplOptions.AggressiveInlining)]
190         public Hybrid(object value, bool isExternal)
191         {
192             var signedValue = value == null ? 0 : _objectToInt64Converter.Convert(value);
193             if (signedValue == 0 && isExternal)
194             {
195                 Value = ExternalZero;
196             }
197             else
198             {
199                 var absoluteValue = System.Math.Abs(signedValue);
200                 Value = isExternal ? _int64ToAddressConverter.Convert(-absoluteValue) :
                    ↪  _int64ToAddressConverter.Convert(absoluteValue);
201             }
202         }
203 
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
206 
207         [MethodImpl(MethodImplOptions.AggressiveInlining)]
208         public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
209 
210         [MethodImpl(MethodImplOptions.AggressiveInlining)]
211         public static explicit operator Hybrid<TLinkAddress>(long integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
212 
213         [MethodImpl(MethodImplOptions.AggressiveInlining)]
214         public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
215 
216         [MethodImpl(MethodImplOptions.AggressiveInlining)]
217         public static explicit operator Hybrid<TLinkAddress>(int integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
218 
219         [MethodImpl(MethodImplOptions.AggressiveInlining)]
220         public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
221 
222         [MethodImpl(MethodImplOptions.AggressiveInlining)]
223         public static explicit operator Hybrid<TLinkAddress>(short integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
224 
225         [MethodImpl(MethodImplOptions.AggressiveInlining)]
226         public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
227 
228         [MethodImpl(MethodImplOptions.AggressiveInlining)]
229         public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
            ↪  Hybrid<TLinkAddress>(integer);
230 
231         [MethodImpl(MethodImplOptions.AggressiveInlining)]
232         public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
            ↪  hybrid.Value;
233 
234         [MethodImpl(MethodImplOptions.AggressiveInlining)]
235         public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
            ↪  CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);
236 
237         [MethodImpl(MethodImplOptions.AggressiveInlining)]
238         public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
            ↪  hybrid.AbsoluteValue;
239 
240         [MethodImpl(MethodImplOptions.AggressiveInlining)]
241         public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
            ↪  CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
242 
243         [MethodImpl(MethodImplOptions.AggressiveInlining)]
244         public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
            ↪  (int)hybrid.AbsoluteValue;
245 
246         [MethodImpl(MethodImplOptions.AggressiveInlining)]
247         public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
            ↪  CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
248 
249         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
250         public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
            ↪ (short)hybrid.AbsoluteValue;
251
252         [MethodImpl(MethodImplOptions.AggressiveInlining)]
253         public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
            ↪ CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);
254
255         [MethodImpl(MethodImplOptions.AggressiveInlining)]
256         public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
            ↪ (sbyte)hybrid.AbsoluteValue;
257
258         /// <summary>
259         /// <para>
260         /// Returns the string.
261         /// </para>
262         /// <para></para>
263         /// </summary>
264         /// <returns>
265         /// <para>The string</para>
266         /// <para></para>
267         /// </returns>
268         [MethodImpl(MethodImplOptions.AggressiveInlining)]
269         public override string ToString() => IsExternal ? $"<{AbsoluteValue}>" :
            ↪ Value.ToString();
270
271         /// <summary>
272         /// <para>
273         /// Determines whether this instance equals.
274         /// </para>
275         /// <para></para>
276         /// </summary>
277         /// <param name="other">
278         /// <para>The other.</para>
279         /// <para></para>
280         /// </param>
281         /// <returns>
282         /// <para>The bool</para>
283         /// <para></para>
284         /// </returns>
285         [MethodImpl(MethodImplOptions.AggressiveInlining)]
286         public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
            ↪ other.Value);
287
288         /// <summary>
289         /// <para>
290         /// Determines whether this instance equals.
291         /// </para>
292         /// <para></para>
293         /// </summary>
294         /// <param name="obj">
295         /// <para>The obj.</para>
296         /// <para></para>
297         /// </param>
298         /// <returns>
299         /// <para>The bool</para>
300         /// <para></para>
301         /// </returns>
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
            ↪ Equals(hybrid) : false;
304
305         /// <summary>
306         /// <para>
307         /// Gets the hash code.
308         /// </para>
309         /// <para></para>
310         /// </summary>
311         /// <returns>
312         /// <para>The int</para>
313         /// <para></para>
314         /// </returns>
315         [MethodImpl(MethodImplOptions.AggressiveInlining)]
316         public override int GetHashCode() => Value.GetHashCode();
317
318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪ left.Equals(right);
320
```

```
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪  !(left == right);
323     }
324 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Delegates;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data
9  {
10     /// <summary>
11     /// <para>Represents an interface for manipulating data in the Links (links storage)
       ↪  format.</para>
12     /// <para>Представляет интерфейс для манипуляции с данными в формате Links (хранилища
       ↪  связей).</para>
13     /// </summary>
14     /// <remarks>
15     /// <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>
16     /// <para>Этот интерфейс не зависит от размера содержимого связи, а значит подходит как для
       ↪  дуплетов, триплетов и последовательностей связей любого размера.</para>
17     /// </remarks>
18     public interface ILinks<TLinkAddress, TConstants>
19         where TConstants : LinksConstants<TLinkAddress>
20     {
21         #region Constants
22
23         /// <summary>
24         /// <para>Returns the set of constants that is necessary for effective communication
           ↪  with the methods of this interface.</para>
25         /// <para>Возвращает набор констант, который необходим для эффективной коммуникации с
           ↪  методами этого интерфейса.</para>
26         /// </summary>
27         /// <remarks>
28         /// <para>These constants are not changed since the creation of the links storage access
           ↪  point.</para>
29         /// <para>Эти константы не меняются с момента создания точки доступа к хранилищу
           ↪  связей.</para>
30         /// </remarks>
31         TConstants Constants
32         {
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             get;
35         }
36
37         #endregion
38
39         #region Read
40
41         /// <summary>
42         /// <para>Counts and returns the total number of links in the storage that meet the
           ↪  specified restriction.</para>
43         /// <para>Подсчитывает и возвращает общее число связей находящихся в хранилище,
           ↪  соответствующих указанному ограничению.</para>
44         /// </summary>
45         /// <param name="restriction"><para>Restriction on the contents of
           ↪  links.</para><para>Ограничение на содержимое связей.</para></param>
46         /// <returns><para>The total number of links in the storage that meet the specified
           ↪  restriction.</para><para>Общее число связей находящихся в хранилище, соответствующих
           ↪  указанному ограничению.</para></returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         TLinkAddress Count(IList<TLinkAddress>? restriction);
49
50         /// <summary>
51         /// <para>Passes through all the links matching the pattern, invoking a handler for each
           ↪  matching link.</para>
52         /// <para>Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
           ↪  (handler) для каждой подходящей связи.</para>
53         /// </summary>
54         /// <param name="restriction">
```

```
55          /// <para>Restriction 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>Ограничение 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          /// <param name="handler"><para>A handler for each matching link.</para><para>Обработчик
            ↪ для каждой подходящей связи.</para></param>
59          /// <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>
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>? handler);
62
63          #endregion
64
65          #region Write
66
67          /// <summary>
68          /// <para>Creates a link.</para>
69          /// <para>Создаёт связь.</para>
70          /// <param name="substitution">
71          /// <para>The content of a new link. This argument is optional, if the null passed as
            ↪ value that means no content of a link is set.</para>
72          /// <para>Содержимое новой связи. Этот аргумент опционален, если null передан в качестве
            ↪ значения это означает, что никакого содержимого для связи не установлено.</para>
73          /// </param>
74          /// <param name="handler">
75          /// <para>A function to handle each executed change. This function can use
            ↪ Constants.Continue to continue proccess each change. Constants.Break can be used to
            ↪ stop receiving of executed changes.</para>
76          /// <para>Функция для обработки каждого выполненного изменения. Эта функция может
            ↪ использовать Constants.Continue чтобы продолжить обрабатывать каждое изменение.
            ↪ Constants.Break может быть использована для остановки получения выполненных
            ↪ изменений.</para>
77          /// </param>
78          /// </summary>
79          /// <returns>
80          /// <para>
81          /// Constants.Continue if all executed changes are handled.
82          /// Constants.Break if proccessing of handled changes is stoped.
83          /// </para>
84          /// <para>
85          /// Constants.Continue если все выполненные изменения обработаны.
86          /// Constants.Break если обработака выполненных изменений остановлена.
87          /// </para>
88          /// </returns>
89          [MethodImpl(MethodImplOptions.AggressiveInlining)]
90          TLinkAddress Create(IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>?
            ↪ handler);
91
92          /// <summary>
93          /// Обновляет связь с указанными restriction[Constants.IndexPart] в адресом связи
94          /// на связь с указанным новым содержимым.
95          /// </summary>
96          /// <param name="restriction">
97          /// Ограничение на содержимое связей.
98          /// Предполагается, что будет указан индекс связи (в restriction[Constants.IndexPart]) и
            ↪ далее за ним будет следовать содержимое связи.
99          /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
            ↪ ссылку на пустоту,
100         /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
            ↪ другой связи.
101         /// </param>
102         /// <param name="substitution"></param>
103         /// <param name="handler">
104         /// <para>A function to handle each executed change. This function can use
            ↪ Constants.Continue to continue proccess each change. Constants.Break can be used to
            ↪ stop receiving of executed changes.</para>
105         /// <para>Функция для обработки каждого выполненного изменения. Эта функция может
            ↪ использовать Constants.Continue чтобы продолжить обрабатывать каждое изменение.
            ↪ Constants.Break может быть использована для остановки получения выполненных
            ↪ изменений.</para>
106         /// </param>
107         /// <returns>
```

```
        /// <para>
        /// Constants.Continue if all executed changes are handled.
        /// Constants.Break if proccessing of handled changes is stoped.
        /// </para>
        /// <para>
        /// Constants.Continue если все выполненные изменения обработаны.
        /// Constants.Break если обработака выполненных изменений остановлена.
        /// </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>? substitution,
        ↪ WriteHandler<TLinkAddress>? handler);

        /// <summary>
        /// <para>Deletes links that match the specified restriction.</para>
        /// <para>Удаляет связи соответствующие указанному ограничению.</para>
        /// </summary>
        /// <param name="restriction">
        /// <para>Restriction on the content of a link. This argument is optional, if the null
        ↪ passed as value that means no restriction on the content of a link are set.</para>
        /// <para>Ограничение на содержимое связи. Этот аргумент опционален, если null передан в
        ↪ качестве значения это означает, что никаких ограничений на содержимое связи не
        ↪ установлено.</para>
        /// </param>
        /// <param name="handler">
        /// <para>A function to handle each executed change. This function can use
        ↪ Constants.Continue to continue proccess each change. Constants.Break can be used to
        ↪ stop receiving of executed changes.</para>
        /// <para>Функция для обработки каждого выполненного изменения. Эта функция может
        ↪ использовать Constants.Continue чтобы продолжить обрабатывать каждое изменение.
        ↪ Constants.Break может быть использована для остановки получения выполненных
        ↪ изменений.</para>
        /// </param>
        /// <returns>
        /// <para>
        /// Constants.Continue if all executed changes are handled.
        /// Constants.Break if proccessing of handled changes is stoped.
        /// </para>
        /// <para>
        /// Constants.Continue если все выполненные изменения обработаны.
        /// Constants.Break если обработака выполненных изменений остановлена.
        /// </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
        ↪ handler);

        #endregion
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Setters;
using Platform.Data.Exceptions;
using Platform.Delegates;

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

namespace Platform.Data
{
    /// <summary>
    /// <para>
    /// Represents the links extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ILinksExtensions
    {
        public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress,
        ↪ LinksConstants<TLinkAddress>> links) => links.Create(null);

        public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress,
        ↪ LinksConstants<TLinkAddress>> links, IList<TLinkAddress>? substitution)
        {
            var constants = links.Constants;
```

```
25              Setter<TLinkAddress, TLinkAddress> setter = new Setter<TLinkAddress,
                ↪  TLinkAddress>(constants.Continue, constants.Break, constants.Null);
26              links.Create(substitution, setter.SetFirstFromSecondListAndReturnTrue);
27              return setter.Result;
28          }
29
30          public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress,
            ↪  LinksConstants<TLinkAddress>> links, IList<TLinkAddress>? restriction,
            ↪  IList<TLinkAddress>? substitution)
31          {
32              var constants = links.Constants;
33              Setter<TLinkAddress, TLinkAddress> setter = new(constants.Continue, constants.Break,
                ↪  constants.Null);
34              links.Update(restriction, substitution, setter.SetFirstFromSecondListAndReturnTrue);
35              return setter.Result;
36          }
37
38          public static TLinkAddress Delete<TLinkAddress>(this ILinks<TLinkAddress,
            ↪  LinksConstants<TLinkAddress>> links, TLinkAddress linkToDelete) => Delete(links,
            ↪  (IList<TLinkAddress>?)new LinkAddress<TLinkAddress>(linkToDelete));
39
40          public static TLinkAddress Delete<TLinkAddress>(this ILinks<TLinkAddress,
            ↪  LinksConstants<TLinkAddress>> links, IList<TLinkAddress>? restriction)
41          {
42              var constants = links.Constants;
43              Setter<TLinkAddress, TLinkAddress> setter = new Setter<TLinkAddress,
                ↪  TLinkAddress>(constants.Continue, constants.Break, constants.Null);
44              links.Delete(restriction, setter.SetFirstFromFirstListAndReturnTrue);
45              return setter.Result;
46          }
47
48          /// <summary>
49          /// <para>
50          /// Counts the links.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          /// <typeparam name="TLinkAddress">
55          /// <para>The link address.</para>
56          /// <para></para>
57          /// </typeparam>
58          /// <typeparam name="TConstants">
59          /// <para>The constants.</para>
60          /// <para></para>
61          /// </typeparam>
62          /// <param name="links">
63          /// <para>The links.</para>
64          /// <para></para>
65          /// </param>
66          /// <param name="restrictions">
67          /// <para>The restrictions.</para>
68          /// <para></para>
69          /// </param>
70          /// <returns>
71          /// <para>The link address</para>
72          /// <para></para>
73          /// </returns>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, params TLinkAddress[] restrictions)
76              where TConstants : LinksConstants<TLinkAddress>
77              => links.Count(restrictions);
78
79          /// <summary>
80          /// Возвращает значение, определяющее существует ли связь с указанным индексом в
            ↪  хранилище связей.
81          /// </summary>
82          /// <param name="links">Хранилище связей.</param>
83          /// <param name="link">Индекс проверяемой на существование связи.</param>
84          /// <returns>Значение, определяющее существует ли связь.</returns>
85          [MethodImpl(MethodImplOptions.AggressiveInlining)]
86          public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
87              where TConstants : LinksConstants<TLinkAddress>
88          {
89              var constants = links.Constants;
```

```
90              return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
                ↪  && Comparer<TLinkAddress>.Default.Compare(links.Count(new
                ↪  LinkAddress<TLinkAddress>(link)), default) > 0);
91          }
92
93          /// <param name="links">Хранилище связей.</param>
94          /// <param name="link">Индекс проверяемой на существование связи.</param>
95          /// <remarks>
96          /// TODO: May be move to EnsureExtensions or make it both there and here
97          /// </remarks>
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
100             where TConstants : LinksConstants<TLinkAddress>
101         {
102             if (!links.Exists(link))
103             {
104                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
105             }
106         }
107
108         /// <param name="links">Хранилище связей.</param>
109         /// <param name="link">Индекс проверяемой на существование связи.</param>
110         /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link, string argumentName)
113             where TConstants : LinksConstants<TLinkAddress>
114         {
115             if (!links.Exists(link))
116             {
117                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
118             }
119         }
120
121         /// <summary>
122         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.
123         /// </summary>
124         /// <param name="links">Хранилище связей.</param>
125         /// <param name="handler">Обработчик каждой подходящей связи.</param>
126         /// <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>
127         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
            ↪  случае.</returns>
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, ReadHandler<TLinkAddress>? handler, params TLinkAddress[]
            ↪  restrictions)
130             where TConstants : LinksConstants<TLinkAddress>
131             => links.Each(restrictions, handler);
132
133         /// <summary>
134         /// Возвращает части-значения для связи с указанным индексом.
135         /// </summary>
136         /// <param name="links">Хранилище связей.</param>
137         /// <param name="link">Индекс связи.</param>
138         /// <returns>Уникальную связь.</returns>
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public static IList<TLinkAddress>? GetLink<TLinkAddress, TConstants>(this
            ↪  ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
141             where TConstants : LinksConstants<TLinkAddress>
142         {
143             var constants = links.Constants;
144             if (constants.IsExternalReference(link))
145             {
146                 return new Point<TLinkAddress>(link, constants.TargetPart + 1);
147             }
148             var linkPartsSetter = new Setter<IList<TLinkAddress>?,
                ↪  TLinkAddress>(constants.Continue, constants.Break);
149             links.Each(linkPartsSetter.SetAndReturnTrue, link);
150             return linkPartsSetter.Result;
151         }
152
153         #region Points
154
```

```
155         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
156         /// <param name="links">Хранилище связей.</param>
157         /// <param name="link">Индекс проверяемой связи.</param>
158         /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
159         /// <remarks>
160         /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
            ↪  связь.
161         /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
            ↪  точка и пара существовать одновременно?
162         /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
            ↪  сортировать по индексу в массиве связей?
163         /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
164         /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
            ↪  самой себя любого размера?
165         /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
            ↪  одной ссылки на себя (частичной точки).
166         /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
167         /// самостоятельный цикл через себя? Что если предоставить все варианты использования
            ↪  связей?
168         /// Что если разрешить и нули, а так же частичные варианты?
169         ///
170         /// 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.
171         /// 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.
172         ///
173         /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
            ↪  промежуточную связь,
174         /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
175         /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
176         /// </remarks>
177         [MethodImpl(MethodImplOptions.AggressiveInlining)]
178         public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
179             where TConstants : LinksConstants<TLinkAddress>
180         {
181             if (links.Constants.IsExternalReference(link))
182             {
183                 return true;
184             }
185             links.EnsureLinkExists(link);
186             return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
187         }
188
189         /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
            ↪  точкой частично (связью замкнутой на себе как минимум один раз).</summary>
190         /// <param name="links">Хранилище связей.</param>
191         /// <param name="link">Индекс проверяемой связи.</param>
192         /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
193         /// <remarks>
194         /// Достаточно любой одной ссылки на себя.
195         /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
            ↪  ссылки на себя (на эту связь).
196         /// </remarks>
197         [MethodImpl(MethodImplOptions.AggressiveInlining)]
198         public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
            ↪  TConstants> links, TLinkAddress link)
199             where TConstants : LinksConstants<TLinkAddress>
200         {
201             if (links.Constants.IsExternalReference(link))
202             {
203                 return true;
204             }
205             links.EnsureLinkExists(link);
206             return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
207         }
208
209         #endregion
210     }
211 }
```

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>
    {
        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{TLinkAddress}"/> 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);

        /// <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>
```

```
215             [MethodImpl(MethodImplOptions.AggressiveInlining)]
216             public bool Remove(TLinkAddress item) => throw new NotSupportedException();
217
218             /// <summary>
219             /// <para>
220             /// Removes the at using the specified index.
221             /// </para>
222             /// <para></para>
223             /// </summary>
224             /// <param name="index">
225             /// <para>The index.</para>
226             /// <para></para>
227             /// </param>
228             [MethodImpl(MethodImplOptions.AggressiveInlining)]
229             public void RemoveAt(int index) => throw new NotSupportedException();
230
231             /// <summary>
232             /// <para>
233             /// Gets the enumerator.
234             /// </para>
235             /// <para></para>
236             /// </summary>
237             /// <returns>
238             /// <para>The enumerator</para>
239             /// <para></para>
240             /// </returns>
241             [MethodImpl(MethodImplOptions.AggressiveInlining)]
242             IEnumerator IEnumerable.GetEnumerator()
243             {
244                 yield return Index;
245             }
246
247             /// <summary>
248             /// <para>
249             /// Determines whether this instance equals.
250             /// </para>
251             /// <para></para>
252             /// </summary>
253             /// <param name="other">
254             /// <para>The other.</para>
255             /// <para></para>
256             /// </param>
257             /// <returns>
258             /// <para>The bool</para>
259             /// <para></para>
260             /// </returns>
261             [MethodImpl(MethodImplOptions.AggressiveInlining)]
262             public virtual bool Equals(LinkAddress<TLinkAddress> other) => other != null &&
                 ↪ _equalityComparer.Equals(Index, other.Index);
263
264             [MethodImpl(MethodImplOptions.AggressiveInlining)]
265             public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
                 ↪ linkAddress.Index;
266
267             [MethodImpl(MethodImplOptions.AggressiveInlining)]
268             public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
                 ↪ new LinkAddress<TLinkAddress>(linkAddress);
269
270             /// <summary>
271             /// <para>
272             /// Determines whether this instance equals.
273             /// </para>
274             /// <para></para>
275             /// </summary>
276             /// <param name="obj">
277             /// <para>The obj.</para>
278             /// <para></para>
279             /// </param>
280             /// <returns>
281             /// <para>The bool</para>
282             /// <para></para>
283             /// </returns>
284             [MethodImpl(MethodImplOptions.AggressiveInlining)]
285             public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
                 ↪ ? Equals(linkAddress) : false;
286
287             /// <summary>
288             /// <para>
```

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

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          private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
21          private static readonly UncheckedConverter<ulong, TLinkAddress>
                ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
22
23          #region Link parts
24
25          /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
                ↪  самой связи.</summary>
26          public int IndexPart
27          {
28              [MethodImpl(MethodImplOptions.AggressiveInlining)]
29              get;
30          }
31
```

```
32          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
            ↪  часть-значение).</summary>
33          public int SourcePart
34          {
35              [MethodImpl(MethodImplOptions.AggressiveInlining)]
36              get;
37          }
38
39          /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
            ↪  (последняя часть-значение).</summary>
40          public int TargetPart
41          {
42              [MethodImpl(MethodImplOptions.AggressiveInlining)]
43              get;
44          }
45
46          #endregion
47
48          #region Flow control
49
50          /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
51          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
52          public TLinkAddress Continue
53          {
54              [MethodImpl(MethodImplOptions.AggressiveInlining)]
55              get;
56          }
57
58          /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
59          public TLinkAddress Skip
60          {
61              [MethodImpl(MethodImplOptions.AggressiveInlining)]
62              get;
63          }
64
65          /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
66          /// <remarks>Используется в функции обработчике, который передаётся в функцию
            ↪  Each.</remarks>
67          public TLinkAddress Break
68          {
69              [MethodImpl(MethodImplOptions.AggressiveInlining)]
70              get;
71          }
72
73          #endregion
74
75          #region Special symbols
76
77          /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
78          public TLinkAddress Null
79          {
80              [MethodImpl(MethodImplOptions.AggressiveInlining)]
81              get;
82          }
83
84          /// <summary>Возвращает значение, обозначающее любую связь.</summary>
85          /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
            ↪  создавать все варианты последовательностей в функции Create.</remarks>
86          public TLinkAddress Any
87          {
88              [MethodImpl(MethodImplOptions.AggressiveInlining)]
89              get;
90          }
91
92          /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
93          public TLinkAddress Itself
94          {
95              [MethodImpl(MethodImplOptions.AggressiveInlining)]
96              get;
97          }
98
99          #endregion
100
101         #region References
102
103         /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
            ↪  ссылок).</summary>
104         public Range<TLinkAddress> InternalReferencesRange
105         {
```

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

```
179         /// <para>A possible internal references range.</para>
180         /// <para></para>
181         /// </param>
182         /// <param name="possibleExternalReferencesRange">
183         /// <para>A possible external references range.</para>
184         /// <para></para>
185         /// </param>
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
            ↪ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
            ↪ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
188
189         /// <summary>
190         /// <para>
191         /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
192         /// </para>
193         /// <para></para>
194         /// </summary>
195         /// <param name="enableExternalReferencesSupport">
196         /// <para>A enable external references support.</para>
197         /// <para></para>
198         /// </param>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         public LinksConstants(bool enableExternalReferencesSupport) :
            ↪ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
            ↪ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
201
202         /// <summary>
203         /// <para>
204         /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
205         /// </para>
206         /// <para></para>
207         /// </summary>
208         /// <param name="targetPart">
209         /// <para>A target part.</para>
210         /// <para></para>
211         /// </param>
212         /// <param name="possibleInternalReferencesRange">
213         /// <para>A possible internal references range.</para>
214         /// <para></para>
215         /// </param>
216         [MethodImpl(MethodImplOptions.AggressiveInlining)]
217         public LinksConstants(int targetPart, Range<TLinkAddress>
            ↪ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
            ↪ null) { }
218
219         /// <summary>
220         /// <para>
221         /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
222         /// </para>
223         /// <para></para>
224         /// </summary>
225         /// <param name="possibleInternalReferencesRange">
226         /// <para>A possible internal references range.</para>
227         /// <para></para>
228         /// </param>
229         [MethodImpl(MethodImplOptions.AggressiveInlining)]
230         public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
            ↪ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
231
232         /// <summary>
233         /// <para>
234         /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
235         /// </para>
236         /// <para></para>
237         /// </summary>
238         [MethodImpl(MethodImplOptions.AggressiveInlining)]
239         public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
            ↪ false) { }
240
241         /// <summary>
242         /// <para>
243         /// Gets the default internal references range using the specified enable external
            ↪ references support.
244         /// </para>
245         /// <para></para>
246         /// </summary>
```

```
247             /// <param name="enableExternalReferencesSupport">
248             /// <para>The enable external references support.</para>
249             /// <para></para>
250             /// </param>
251             /// <returns>
252             /// <para>A range of t link address</para>
253             /// <para></para>
254             /// </returns>
255             [MethodImpl(MethodImplOptions.AggressiveInlining)]
256             public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
                ↪ enableExternalReferencesSupport)
257             {
258                 if (enableExternalReferencesSupport)
259                 {
260                     return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe」
                        ↪ rValuesRange));
261                 }
262                 else
263                 {
264                     return (_one, NumericType<TLinkAddress>.MaxValue);
265                 }
266             }
267 
268             /// <summary>
269             /// <para>
270             /// Gets the default external references range using the specified enable external
                ↪ references support.
271             /// </para>
272             /// <para></para>
273             /// </summary>
274             /// <param name="enableExternalReferencesSupport">
275             /// <para>The enable external references support.</para>
276             /// <para></para>
277             /// </param>
278             /// <returns>
279             /// <para>A range of t link address</para>
280             /// <para></para>
281             /// </returns>
282             [MethodImpl(MethodImplOptions.AggressiveInlining)]
283             public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
                ↪ enableExternalReferencesSupport)
284             {
285                 if (enableExternalReferencesSupport)
286                 {
287                     return (Hybrid<TLinkAddress>.ExternalZero, NumericType<TLinkAddress>.MaxValue);
288                 }
289                 else
290                 {
291                     return null;
292                 }
293             }
294         }
295     }
```

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

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2   
3   namespace Platform.Data
4   {
5       /// <summary>
6       /// <para>
7       /// Represents the links constants base.
8       /// </para>
9       /// <para></para>
10      /// </summary>
11      public abstract class LinksConstantsBase
12      {
13          /// <summary>
14          /// <para>
15          /// The default target part.
16          /// </para>
17          /// <para></para>
18          /// </summary>
19          public static readonly int DefaultTargetPart = 2;
20      }
21  }
```

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

        /// <summary>
        /// <para>
        /// Determines whether is external 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 IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
        ↪  linksConstants, TLinkAddress address) =>
        ↪  linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
    }
}
```

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

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

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

namespace Platform.Data.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the address to raw number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{TLinkAddress}"/>
    public class AddressToRawNumberConverter<TLinkAddress> : IConverter<TLinkAddress>
    {
        /// <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 TLinkAddress Convert(TLinkAddress source) => new Hybrid<TLinkAddress>(source,
        ↪  isExternal: true);
    }
}
```

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

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

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

namespace Platform.Data.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the raw number to address converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{TLinkAddress}"/>
    public class RawNumberToAddressConverter<TLinkAddress> : IConverter<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        static private readonly UncheckedConverter<long, TLinkAddress> _converter =
        ↪  UncheckedConverter<long, TLinkAddress>.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 TLinkAddress Convert(TLinkAddress source) => _converter.Convert(new
        ↪  Hybrid<TLinkAddress>(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>
    {
        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{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>A size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Point(TLinkAddress index, int size)
        {
            Index = index;
            Size = size;
        }

        /// <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);

        /// <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()
        {
            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;
```

```
209
210         /// <summary>
211         /// <para>
212         /// Inserts the index.
213         /// </para>
214         /// <para></para>
215         /// </summary>
216         /// <param name="index">
217         /// <para>The index.</para>
218         /// <para></para>
219         /// </param>
220         /// <param name="item">
221         /// <para>The item.</para>
222         /// <para></para>
223         /// </param>
224         [MethodImpl(MethodImplOptions.AggressiveInlining)]
225         public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
226
227         /// <summary>
228         /// <para>
229         /// Determines whether this instance remove.
230         /// </para>
231         /// <para></para>
232         /// </summary>
233         /// <param name="item">
234         /// <para>The item.</para>
235         /// <para></para>
236         /// </param>
237         /// <returns>
238         /// <para>The bool</para>
239         /// <para></para>
240         /// </returns>
241         [MethodImpl(MethodImplOptions.AggressiveInlining)]
242         public bool Remove(TLinkAddress item) => throw new NotSupportedException();
243
244         /// <summary>
245         /// <para>
246         /// Removes the at using the specified index.
247         /// </para>
248         /// <para></para>
249         /// </summary>
250         /// <param name="index">
251         /// <para>The index.</para>
252         /// <para></para>
253         /// </param>
254         [MethodImpl(MethodImplOptions.AggressiveInlining)]
255         public void RemoveAt(int index) => throw new NotSupportedException();
256
257         /// <summary>
258         /// <para>
259         /// Gets the enumerator.
260         /// </para>
261         /// <para></para>
262         /// </summary>
263         /// <returns>
264         /// <para>The enumerator</para>
265         /// <para></para>
266         /// </returns>
267         [MethodImpl(MethodImplOptions.AggressiveInlining)]
268         IEnumerator IEnumerable.GetEnumerator()
269         {
270             for (int i = 0; i < Size; i++)
271             {
272                 yield return Index;
273             }
274         }
275
276         /// <summary>
277         /// <para>
278         /// Determines whether this instance equals.
279         /// </para>
280         /// <para></para>
281         /// </summary>
282         /// <param name="other">
283         /// <para>The other.</para>
284         /// <para></para>
285         /// </param>
286         /// <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(Point<TLinkAddress> linkAddress) =>
        ↪ linkAddress.Index;

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

        /// <summary>
        /// <para>
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => Index.GetHashCode();

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => Index.ToString();

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

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

        /// <summary>
        /// <para>
        /// Determines whether is full 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 IsFullPoint(params TLinkAddress[] link) =>
        ↪  IsFullPoint((IList<TLinkAddress>?)link);

        /// <summary>
        /// <para>
        /// Determines whether is full 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 IsFullPoint(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 IsFullPointUnchecked(link);
        }

        /// <summary>
        /// <para>
        /// Determines whether is full 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 IsFullPointUnchecked(IList<TLinkAddress>? link)
        {
            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);
```

```
436
437         /// <summary>
438         /// <para>
439         /// Determines whether is partial point.
440         /// </para>
441         /// <para></para>
442         /// </summary>
443         /// <param name="link">
444         /// <para>The link.</para>
445         /// <para></para>
446         /// </param>
447         /// <returns>
448         /// <para>The bool</para>
449         /// <para></para>
450         /// </returns>
451         [MethodImpl(MethodImplOptions.AggressiveInlining)]
452         public static bool IsPartialPoint(IList<TLinkAddress>? link)
453         {
454             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
455             Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
                ↪  determine link's pointness using only its identifier.");
456             return IsPartialPointUnchecked(link);
457         }
458
459         /// <summary>
460         /// <para>
461         /// Determines whether is partial point unchecked.
462         /// </para>
463         /// <para></para>
464         /// </summary>
465         /// <param name="link">
466         /// <para>The link.</para>
467         /// <para></para>
468         /// </param>
469         /// <returns>
470         /// <para>The result.</para>
471         /// <para></para>
472         /// </returns>
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         public static bool IsPartialPointUnchecked(IList<TLinkAddress>? link)
475         {
476             var result = false;
477             for (var i = 1; !result && i < link.Count; i++)
478             {
479                 result = _equalityComparer.Equals(link[0], link[i]);
480             }
481             return result;
482         }
483     }
484 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Delegates;
4
5  // ReSharper disable TypeParameterCanBeVariant
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Universal
9  {
10     /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
11     public partial interface IUniLinks<TLinkAddress>
12     {
13         /// <summary>
14         /// <para>
15         /// Triggers the condition.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         /// <param name="condition">
20         /// <para>The condition.</para>
21         /// <para></para>
22         /// </param>
23         /// <param name="substitution">
24         /// <para>The substitution.</para>
25         /// <para></para>
26         /// </param>
```

```
27          /// <returns>
28          /// <para>A list of i list i list t link address</para>
29          /// <para></para>
30          /// </returns>
31          IList<IList<IList<TLinkAddress>?>> Trigger(IList<TLinkAddress>? condition,
                ↪  IList<TLinkAddress>? substitution);
32      }
33
34      /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
35      public partial interface IUniLinks<TLinkAddress>
36      {
37          /// <returns>
38          /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
                ↪  represents False (was stopped).
39          /// This is done to assure ability to push up stop signal through recursion stack.
40          /// </returns>
41          /// <remarks>
42          /// { 0, 0, 0 } => { itself, itself, itself } // create
43          /// { 1, any, any } => { itself, any, 3 } // update
44          /// { 3, any, any } => { 0, 0, 0 } // delete
45          /// </remarks>
46          TLinkAddress Trigger(IList<TLinkAddress>? patternOrCondition, ReadHandler<TLinkAddress>?
                ↪  matchHandler,
47                               IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>?
                                 ↪  substitutionHandler);
48
49          /// <summary>
50          /// <para>
51          /// Triggers the restriction.
52          /// </para>
53          /// <para></para>
54          /// </summary>
55          /// <param name="restriction">
56          /// <para>The restriction.</para>
57          /// <para></para>
58          /// </param>
59          /// <param name="matchedHandler">
60          /// <para>The matched handler.</para>
61          /// <para></para>
62          /// </param>
63          /// <param name="substitution">
64          /// <para>The substitution.</para>
65          /// <para></para>
66          /// </param>
67          /// <param name="substitutedHandler">
68          /// <para>The substituted handler.</para>
69          /// <para></para>
70          /// </param>
71          /// <returns>
72          /// <para>The link address</para>
73          /// <para></para>
74          /// </returns>
75          TLinkAddress Trigger(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
                ↪  matchedHandler,
76                               IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? substitutedHandler);
77      }
78
79      /// <remarks>Extended with small optimization.</remarks>
80      public partial interface IUniLinks<TLinkAddress>
81      {
82          /// <remarks>
83          /// Something simple should be simple and optimized.
84          /// </remarks>
85          TLinkAddress Count(IList<TLinkAddress>? restrictions);
86      }
87  }
```

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.
```

```
    /// </remarks>
    public interface IUniLinksCRUD<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Reads 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 Read(int partType, TLinkAddress link);
        /// <summary>
        /// <para>
        /// Reads 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 Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress>? pattern);
        /// <summary>
        /// <para>
        /// Creates the parts.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="parts">
        /// <para>The parts.</para>
        /// <para></para>
        /// </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.
```

```
89          /// </para>
90          /// <para></para>
91          /// </summary>
92          /// <param name="parts">
93          /// <para>The parts.</para>
94          /// <para></para>
95          /// </param>
96          TLinkAddress Delete(IList<TLinkAddress>? parts);
97      }
98  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// Get/Set aliases for IUniLinks.
11      /// </remarks>
12      public interface IUniLinksGS<TLinkAddress>
13      {
14          /// <summary>
15          /// <para>
16          /// Gets 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 Get(int partType, TLinkAddress link);
33          /// <summary>
34          /// <para>
35          /// Gets 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 Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress>? pattern);
52          /// <summary>
53          /// <para>
54          /// Sets the before.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          /// <param name="before">
59          /// <para>The before.</para>
60          /// <para></para>
61          /// </param>
62          /// <param name="after">
63          /// <para>The after.</para>
64          /// <para></para>
65          /// </param>
66          /// <returns>
```

```
67          /// <para>The link address</para>
68          /// <para></para>
69          /// </returns>
70          TLinkAddress Set(IList<TLinkAddress>? before, IList<TLinkAddress>? after);
71      }
72  }
```

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

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// In/Out aliases for IUniLinks.
11      /// TLinkAddress can be any number type of any size.
12      /// </remarks>
13      public interface IUniLinksIO<TLinkAddress>
14      {
15          /// <remarks>
16          /// default(TLinkAddress) means any link.
17          /// Single element pattern means just element (link).
18          /// Handler gets array of link contents.
19          /// * link[0] is index or identifier.
20          /// * link[1] is source or first.
21          /// * link[2] is target or second.
22          /// * link[3] is linker or third.
23          /// * link[n] is nth part/parent/element/value
24          /// of link (if variable length links used).
25          ///
26          /// Stops and returns false if handler return false.
27          ///
28          /// Acts as Each, Foreach, Select, Search, Match &amp;amp; ...
29          ///
30          /// Handles all links in store if pattern/restrictions is not defined.
31          /// </remarks>
32          bool Out(Func<IList<TLinkAddress>?, bool> handler, IList<TLinkAddress>? pattern);
33
34          /// <remarks>
35          /// default(TLinkAddress) means itself.
36          /// Equivalent to:
37          /// * creation if before == null
38          /// * deletion if after == null
39          /// * update if before != null &amp;amp;&amp;amp; after != null
40          /// * default(TLinkAddress) if before == null &amp;amp;&amp;amp; after == null
41          ///
42          /// 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).
```

```
        /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
        /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
        /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
        /// OutPart(3, pattern) ~ GeTLinkAddresser(link) or GeTLinkAddresser(Search(pattern))
        /// OutPart(n, pattern) => For any variable length links, returns link or default(TLinkAddress).
        ///
        /// Outs(returns) inner contents of link, its part/parent/element/value.
        /// </remarks>
        TLinkAddress OutOne(int partType, IList<TLinkAddress>? pattern);

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

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

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

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

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

namespace Platform.Data.Universal
{
    /// <remarks>
    /// Read/Write aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksRW<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Reads 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 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">
```

```
59          /// <para>The before.</para>
60          /// <para></para>
61          /// </param>
62          /// <param name="after">
63          /// <para>The after.</para>
64          /// <para></para>
65          /// </param>
66          /// <returns>
67          /// <para>The link address</para>
68          /// <para></para>
69          /// </returns>
70          TLinkAddress Write(IList<TLinkAddress>? before, IList<TLinkAddress>? after);
71      }
72  }
```

1.23 ./csharp/Platform.Data/WriteHandlerState.cs

```
1   using System.Collections.Generic;
2   using Platform.Delegates;
3
4   namespace Platform.Data
5   {
6       public struct WriteHandlerState<TLinkAddress> where TLinkAddress : struct
7       {
8           private readonly EqualityComparer<TLinkAddress> _equalityComparer;
9           public TLinkAddress Result;
10          public WriteHandler<TLinkAddress>? Handler;
11          private TLinkAddress Break;
12
13          public WriteHandlerState(TLinkAddress @continue, TLinkAddress @break,
            ↪  WriteHandler<TLinkAddress>? handler)
14          {
15              _equalityComparer = EqualityComparer<TLinkAddress>.Default;
16              Break = @break;
17              Result = @continue;
18              Handler = handler;
19          }
20
21          public void Apply(TLinkAddress result)
22          {
23              var isAlreadyBreak = _equalityComparer.Equals(Break, Result);
24              var isCurrentlyBreak = _equalityComparer.Equals(Break, result);
25              if (isAlreadyBreak || !isCurrentlyBreak)
26              {
27                  return;
28              }
29              Handler = null;
30              Result = Break;
31
32          }
33
34          public TLinkAddress Handle(IList<TLinkAddress> before, IList<TLinkAddress> after)
35          {
36              Apply(Handler?.Invoke(before, after) ?? Result);
37              return Result;
38          }
39      }
40  }
```

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

```
1   using Xunit;
2
3   namespace Platform.Data.Tests
4   {
5       /// <summary>
6       /// <para>
7       /// Represents the hybrid tests.
8       /// </para>
9       /// <para></para>
10      /// </summary>
11      public static class HybridTests
12      {
13          /// <summary>
14          /// <para>
15          /// Tests that object constructor test.
16          /// </para>
17          /// <para></para>
18          /// </summary>
19          [Fact]
20          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);
        }
    }
}
```

1.25 ./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>();
        }
        private static void TestExternalReferences<TUnsigned, TSigned>()
        {
            var unsingedOne = Arithmetic.Increment(default(TUnsigned));
            var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
            var half = converter.Convert(NumericType<TSigned>.MaxValue);
            LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne,
            ↪  half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));

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

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

Index