

LinksPlatform's Platform.Collections Class Library

1.1 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Arrays
5  {
6      /// <summary>
7      /// <para>Represents <see cref="T:TElement[]"> array filler with additional methods that
8      ///     ↪ return a given constant of type <typeparamref cref="TReturnConstant"/>.</para>
9      /// <para>Представляет заполнитель массива <see cref="T:TElement[]"> с дополнительными
10     ↪ методами, возвращающими заданную константу типа <typeparamref
11     ↪ cref="TReturnConstant"/>.</para>
12     /// </summary>
13     /// <typeparam name="TElement"><para>The elements' type.</para><para>Тип элементов
14     ↪ массива.</para></typeparam>
15     /// <typeparam name="TReturnConstant"><para>The return constant's type.</para><para>Тип
16     ↪ возвращаемой константы.</para></typeparam>
17     public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
18     {
19         /// <summary>
20         /// <para>
21         ///     The return constant.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         protected readonly TReturnConstant _returnConstant;
26
27         /// <summary>
28         /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
29         ↪ specified array, the offset from which filling will start and the constant returned
30         ↪ when elements are being filled.</para>
31         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
32         ↪ указанный массив, смещение с которого начнётся заполнение и константу возвращаемую
33         ↪ при заполнении элементов.</para>
34         /// </summary>
35         /// <param name="array"><para>The array to fill.</para><para>Массив для
36         ↪ заполнения.</para></param>
37         /// <param name="offset"><para>The offset from which to start the array
38         ↪ filling.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
39         /// <param name="returnConstant"><para>The constant's value.</para><para>Значение
40         ↪ константы.</para></param>
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
43             ↪ base(array, offset) => _returnConstant = returnConstant;
44
45         /// <summary>
46         /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
47         ↪ specified array and the constant returned when elements are being filled. Filling
48         ↪ will start from the beginning of the array.</para>
49         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
50         ↪ указанный массив и константу возвращаемую при заполнении элементов. Заполнение
51         ↪ начнётся с начала массива.</para>
52         /// </summary>
53         /// <param name="array"><para>The array to fill.</para><para>Массив для
54         ↪ заполнения.</para></param>
55         /// <param name="returnConstant"><para>The constant's value.</para><para>Значение
56         ↪ константы.</para></param>
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
59             ↪ returnConstant) { }
60
61         /// <summary>
62         /// <para>Adds an item into the array and returns the constant.</para>
63         /// <para>Добавляет элемент в массив и возвращает константу.</para>
64         /// </summary>
65         /// <param name="element"><para>The element to add.</para><para>Добавляемый
66         ↪ элемент.</para></param>
67         /// <returns>
68         /// <para>The constant's value.</para>
69         /// <para>Значение константы.</para>
70         /// </returns>
71         [MethodImpl(MethodImplOptions.AggressiveInlining)]
72         public TReturnConstant AddAndReturnConstant(TElement element) =>
73             ↪ _array.AddAndReturnConstant(ref _position, element, _returnConstant);
74
75         /// <summary>

```

```

54     /// <para>Adds the first element from the specified list to the filled array and returns
    → the constant.</para>
55     /// <para>Добавляет первый элемент из указанного списка в заполняемый массив и
    → возвращает константу.</para>
56     /// </summary>
57     /// <param name="element"><para>The list from which the first item will be
    → added.</para><para>Список из которого будет добавлен первый элемент.</para></param>
58     /// <returns>
59     /// <para>The constant's value.</para>
60     /// <para>Значение константы.</para>
61     /// </returns>
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements) =>
    → _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);
64
65     /// <summary>
66     /// <para>Adds all elements from the specified list to the filled array and returns the
    → constant.</para>
67     /// <para>Добавляет все элементы из указанного списка в заполняемый массив и возвращает
    → константу.</para>
68     /// </summary>
69     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → для добавления.</para></param>
70     /// <returns>
71     /// <para>The constant's value.</para>
72     /// <para>Значение константы.</para>
73     /// </returns>
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements) =>
    → _array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);
76
77     /// <summary>
78     /// <para>Adds the elements of the list to the array, skipping the first element and
    → returns the constant.</para>
79     /// <para>Добавляет элементы списка в массив пропуская первый элемент и возвращает
    → константу.</para>
80     /// </summary>
81     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → для добавления.</para></param>
82     /// <returns>
83     /// <para>The constant's value.</para>
84     /// <para>Значение константы.</para>
85     /// </returns>
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements) =>
    → _array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
88 }
89 }

```

1.2 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Arrays
5  {
6      /// <summary>
7      /// <para>Represents an <see cref="T:TElement[]"> array filler.</para>
8      /// <para>Представляет заполнитель массива <see cref="T:TElement[]">.</para>
9      /// </summary>
10     /// <typeparam name="TElement"><para>The elements' type.</para><para>Тип элементов
    → массива.</para></typeparam>
11     public class ArrayFiller<TElement>
12     {
13         /// <summary>
14         /// <para>
15         /// The array.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         protected readonly TElement[] _array;
20         /// <summary>
21         /// <para>
22         /// The position.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         protected long _position;

```

```

27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84

```

```

/// <summary>
/// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
→ specified array as the array to fill and the offset from which to start
→ filling.</para>
/// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
→ указанный массив в качестве заполняемого и смещение с которого начнётся
→ заполнение.</para>
/// </summary>
/// <param name="array"><para>The array to fill.</para><para>Массив для
→ заполнения.</para></param>
/// <param name="offset"><para>The offset from which to start filling the
→ array.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public ArrayFiller(TElement[] array, long offset)
{
    _array = array;
    _position = offset;
}

/// <summary>
/// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
→ specified array. Filling will start from the beginning of the array.</para>
/// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
→ указанный массив. Заполнение начнётся с начала массива.</para>
/// </summary>
/// <param name="array"><para>The array to fill.</para><para>Массив для
→ заполнения.</para></param>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public ArrayFiller(TElement[] array) : this(array, 0) { }

/// <summary>
/// <para>Adds an item into the array.</para>
/// <para>Добавляет элемент в массив.</para>
/// </summary>
/// <param name="element"><para>The element to add.</para><para>Добавляемый
→ элемент.</para></param>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Add(TElement element) => _array[_position++] = element;

/// <summary>
/// <para>Adds an item into the array and returns <see langword="true"/>.</para>
/// <para>Добавляет элемент в массив и возвращает <see langword="true"/>.</para>
/// </summary>
/// <param name="element"><para>The element to add.</para><para>Добавляемый
→ элемент.</para></param>
/// <returns>
/// <para>The <see langword="true"/> value.</para>
/// <para>Значение <see langword="true"/>.</para>
/// </returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
→ _position, element, true);

/// <summary>
/// <para>Adds the first element from the specified list to the array to fill and
→ returns <see langword="true"/>.</para>
/// <para>Добавляет первый элемент из указанного списка в заполняемый массив и
→ возвращает <see langword="true"/>.</para>
/// </summary>
/// <param name="elements"><para>The list from which the first item will be
→ added.</para><para>Список из которого будет добавлен первый элемент.</para></param>
/// <returns>
/// <para>The <see langword="true"/> value.</para>
/// <para>Значение <see langword="true"/>.</para>
/// </returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
→ _array.AddFirstAndReturnConstant(ref _position, elements, true);

/// <summary>
/// <para>Adds all elements from the specified list to the array to fill and returns
→ <see langword="true"/>.</para>
/// <para>Добавляет все элементы из указанного списка в заполняемый массив и возвращает
→ <see langword="true"/>.</para>
/// </summary>

```

```

85     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    ↪ которые необходимо добавить.</para></param>
86     /// <returns>
87     /// <para>The <see langword="true"/> value.</para>
88     /// <para>Значение <see langword="true"/>.</para>
89     /// </returns>
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public bool AddAllAndReturnTrue(IList<TElement> elements) =>
    ↪ _array.AddAllAndReturnConstant(ref _position, elements, true);
92
93     /// <summary>
94     /// <para>Adds values to the array skipping the first element and returns <see
    ↪ langword="true"/>.</para>
95     /// <para>Добавляет значения в массив пропуская первый элемент и возвращает <see
    ↪ langword="true"/>.</para>
96     /// </summary>
97     /// <param name="elements"><para>A list from which elements will be added except the
    ↪ first.</para><para>Список из которого будут добавлены элементы кроме
    ↪ первого.</para></param>
98     /// <returns>
99     /// <para>The <see langword="true"/> value.</para>
100    /// <para>Значение <see langword="true"/>.</para>
101    /// </returns>
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
    ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
104 }
105 }

```

1.3 ./csharp/Platform.Collections/Arrays/ArrayPool.cs

```

1  using System.Runtime.CompilerServices;
2
3  namespace Platform.Collections.Arrays
4  {
5      /// <summary>
6      /// <para>Represents a set of wrapper methods over <see cref="ArrayPool{T}"/> class methods
    ↪ to simplify access to them.</para>
7      /// <para>Представляет набор методов обёрток над методами класса <see cref="ArrayPool{T}"/>
    ↪ для упрощения доступа к ним.</para>
8      /// </summary>
9      public static class ArrayPool
10     {
11         /// <summary>
12         /// <para>
13         /// The default sizes amount.
14         /// </para>
15         /// <para></para>
16         /// </summary>
17         public static readonly int DefaultSizesAmount = 512;
18         /// <summary>
19         /// <para>
20         /// The default max arrays per size.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public static readonly int DefaultMaxArraysPerSize = 32;
25
26         /// <summary>
27         /// <para>Allocation of an array of a specified size from the array pool.</para>
28         /// <para>Выделение массива указанного размера из пула массивов.</para>
29         /// </summary>
30         /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>
31         /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
    ↪ массива.</para></param>
32         /// <returns>
33         /// <para>The array from a pool of arrays.</para>
34         /// <para>Массив из пулла массивов.</para>
35         /// </returns>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
38
39         /// <summary>
40         /// <para>Freeing an array into an array pool.</para>
41         /// <para>Освобождение массива в пул массивов.</para>
42         /// </summary>
43         /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>

```

```

44     /// <param name="array"><para>The array to be freed into the pull.</para><para>Массив
    ↪ который нужно освободить в пулл.</para></param>
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
47 }
48 }

```

1.4 ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Disposables;
5 using Platform.Collections.Stacks;
6
7 namespace Platform.Collections.Arrays
8 {
9     /// <summary>
10    /// <para>Represents a set of arrays ready for reuse.</para>
11    /// <para>Представляет собой набор массивов готовых к повторному использованию.</para>
12    /// </summary>
13    /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>
14    /// <remarks>
15    /// Original idea from
    ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
16    /// </remarks>
17    public class ArrayPool<T>
18    {
19        // May be use Default class for that later.
20        /// <summary>
21        /// <para>
22        /// The thread instance.
23        /// </para>
24        /// <para></para>
25        /// </summary>
26        [ThreadStatic]
27        private static ArrayPool<T> _threadInstance;
28        /// <summary>
29        /// <para>
30        /// Gets the thread instance value.
31        /// </para>
32        /// <para></para>
33        /// </summary>
34        internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
    ↪ ArrayPool<T>());
35        private readonly int _maxArraysPerSize;
36        private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
    ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
37
38        /// <summary>
39        /// <para>Initializes a new instance of the ArrayPool class using the specified maximum
    ↪ number of arrays per size.</para>
40        /// <para>Инициализирует новый экземпляр класса ArrayPool, используя указанное
    ↪ максимальное количество массивов на каждый размер.</para>
41        /// </summary>
42        /// <param name="maxArraysPerSize"><para>The maximum number of arrays in the pool per
    ↪ size.</para><para>Максимальное количество массивов в пуле на каждый
    ↪ размер.</para></param>
43        [MethodImpl(MethodImplOptions.AggressiveInlining)]
44        public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
45
46        /// <summary>
47        /// <para>Initializes a new instance of the ArrayPool class using the default maximum
    ↪ number of arrays per size.</para>
48        /// <para>Инициализирует новый экземпляр класса ArrayPool, используя максимальное
    ↪ количество массивов на каждый размер по умолчанию.</para>
49        /// </summary>
50        [MethodImpl(MethodImplOptions.AggressiveInlining)]
51        public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
52
53        /// <summary>
54        /// <para>Retrieves an array from the pool, which will automatically return to the pool
    ↪ when the container is disposed.</para>
55        /// <para>Извлекает из пула массив, который автоматически вернётся в пул при
    ↪ высвобождении контейнера.</para>
56        /// </summary>
57        /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
    ↪ массива.</para></param>

```

```

58     /// <returns>
59     /// <para>The disposable container containing either a new array or an array from the
    → pool.</para>
60     /// <para>Высвобождаемый контейнер содержащий либо новый массив, либо массив из
    → пула.</para>
61     /// </returns>
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
64
65     /// <summary>
66     /// <para>Replaces the array with another array from the pool with the specified
    → size.</para>
67     /// <para>Заменяет массив на другой массив из пула с указанным размером.</para>
68     /// </summary>
69     /// <param name="source"><para>The source array.</para><para>Исходный
    → массив.</para></param>
70     /// <param name="size"><para>A new array size.</para><para>Новый размер
    → массива.</para></param>
71     /// <returns>
72     /// <para>An array with a new size.</para>
73     /// <para>Массив с новым размером.</para>
74     /// </returns>
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     public Disposable<T[]> Resize(Disposable<T[]> source, long size)
77     {
78         var destination = AllocateDisposable(size);
79         T[] sourceArray = source;
80         if (!sourceArray.IsNullOrEmpty())
81         {
82             T[] destinationArray = destination;
83             Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
    → sourceArray.LongLength);
84             source.Dispose();
85         }
86         return destination;
87     }
88
89     /// <summary>
90     /// <para>Clears the pool.</para>
91     /// <para>Очищает пул.</para>
92     /// </summary>
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public virtual void Clear() => _pool.Clear();
95
96     /// <summary>
97     /// <para>Retrieves an array with the specified size from the pool.</para>
98     /// <para>Извлекает из пула массив с указанным размером.</para>
99     /// </summary>
100    /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
    → массива.</para></param>
101    /// <returns>
102    /// <para>An array from the pool or a new array.</para>
103    /// <para>Массив из пула или новый массив.</para>
104    /// </returns>
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
    → _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
107
108    /// <summary>
109    /// <para>Frees the array to the pool for later reuse.</para>
110    /// <para>Освобождает массив в пул для последующего повторного использования.</para>
111    /// </summary>
112    /// <param name="array"><para>The array to be freed into the pool.</para><para>Массив
    → который нужно освободить в пул.</para></param>
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    public virtual void Free(T[] array)
115    {
116        if (array.IsNullOrEmpty())
117        {
118            return;
119        }
120        var stack = _pool.GetOrAdd(array.LongLength, size => new
    → Stack<T[]>(_maxArraysPerSize));
121        if (stack.Count == _maxArraysPerSize) // Stack is full
122        {
123            return;
124        }

```

```

125         stack.Push(array);
126     }
127 }
128 }

```

1.5 ./csharp/Platform.Collections/Arrays/ArrayString.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Segments;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Arrays
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the array string.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="Segment{T}" />
15     public class ArrayString<T> : Segment<T>
16     {
17         /// <summary>
18         /// <para>
19         /// Initializes a new <see cref="ArrayString" /> instance.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="length">
24         /// <para>A length.</para>
25         /// <para></para>
26         /// </param>
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public ArrayString(int length) : base(new T[length], 0, length) { }
29
30         /// <summary>
31         /// <para>
32         /// Initializes a new <see cref="ArrayString" /> instance.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         /// <param name="array">
37         /// <para>A array.</para>
38         /// <para></para>
39         /// </param>
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public ArrayString(T[] array) : base(array, 0, array.Length) { }
42
43         /// <summary>
44         /// <para>
45         /// Initializes a new <see cref="ArrayString" /> instance.
46         /// </para>
47         /// <para></para>
48         /// </summary>
49         /// <param name="array">
50         /// <para>A array.</para>
51         /// <para></para>
52         /// </param>
53         /// <param name="length">
54         /// <para>A length.</para>
55         /// <para></para>
56         /// </param>
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public ArrayString(T[] array, int length) : base(array, 0, length) { }
59     }
60 }

```

1.6 ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs

```

1  using System.Runtime.CompilerServices;
2
3  namespace Platform.Collections.Arrays
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the char array extensions.
8      /// </para>
9      /// <para></para>
10     /// </summary>

```

```

11 public static unsafe class CharArrayExtensions
12 {
13     /// <summary>
14     /// <para>Generates a hash code for an array segment with the specified offset and
15     → length. The hash code is generated based on the values of the array elements
16     → included in the specified segment.</para>
17     /// <para>Генерирует хэш-код сегмента массива с указанным смещением и длиной. Хэш-код
18     → генерируется на основе значений элементов массива входящих в указанный
19     → сегмент.</para>
20     /// </summary>
21     /// <param name="array"><para>The array to hash.</para><para>Массив для
22     → хеширования.</para></param>
23     /// <param name="offset"><para>The offset from which reading of the specified number of
24     → elements in the array starts.</para><para>Смещение, с которого начинается чтение
25     → указанного количества элементов в массиве.</para></param>
26     /// <param name="length"><para>The number of array elements used to calculate the
27     → hash.</para><para>Количество элементов массива, на основе которых будет вычислен
28     → хэш.</para></param>
29     /// <returns>
30     /// <para>The hash code of the segment in the array.</para>
31     /// <para>Хэш-код сегмента в массиве.</para>
32     /// </returns>
33     /// <remarks>
34     /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
35     → a3eda37d3d4cd10/mscorlib/system/string.cs#L833
36     /// </remarks>
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public static int GenerateHashCode(this char[] array, int offset, int length)
39     {
40         var hashSeed = 5381;
41         var hashAccumulator = hashSeed;
42         fixed (char* arrayPointer = &array[offset])
43         {
44             for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
45             → < last; charPointer++)
46             {
47                 hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
48             }
49             return hashAccumulator + (hashSeed * 1566083941);
50         }
51     }
52     /// <summary>
53     /// <para>Checks if all elements of two lists are equal.</para>
54     /// <para>Проверяет равны ли все элементы двух списков.</para>
55     /// </summary>
56     /// <param name="left"><para>The first compared array.</para><para>Первый массив для
57     → сравнения.</para></param>
58     /// <param name="leftOffset"><para>The offset from which reading of the specified number
59     → of elements in the first array starts.</para><para>Смещение, с которого начинается
60     → чтение элементов в первом массиве.</para></param>
61     /// <param name="length"><para>The number of checked elements.</para><para>Количество
62     → проверяемых элементов.</para></param>
63     /// <param name="right"><para>The second compared array.</para><para>Второй массив для
64     → сравнения.</para></param>
65     /// <param name="rightOffset"><para>The offset from which reading of the specified
66     → number of elements in the second array starts.</para><para>Смещение, с которого
67     → начинается чтение элементов в втором массиве.</para></param>
68     /// <returns>
69     /// <para><see langword="true"/> if the segments of the passed arrays are equal to each
70     → other otherwise <see langword="false"/>.</para>
71     /// <para><see langword="true"/>, если сегменты переданных массивов равны друг другу,
72     → иначе же <see langword="false"/>.</para>
73     /// </returns>
74     /// <remarks>
75     /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
76     → a3eda37d3d4cd10/mscorlib/system/string.cs#L364
77     /// </remarks>
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
80     → right, int rightOffset)
81     {
82         fixed (char* leftPointer = &left[leftOffset])
83         {
84             fixed (char* rightPointer = &right[rightOffset])
85             {

```



```

65         char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
66         if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
        ↪ rightPointerCopy, ref length))
67         {
68             return false;
69         }
70         CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
        ↪ ref length);
71         return length <= 0;
72     }
73 }
74 }
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
        ↪ int length)
77 {
78     while (length >= 10)
79     {
80         if ((* (int*)left != * (int*)right)
81             || (* (int*)(left + 2) != * (int*)(right + 2))
82             || (* (int*)(left + 4) != * (int*)(right + 4))
83             || (* (int*)(left + 6) != * (int*)(right + 6))
84             || (* (int*)(left + 8) != * (int*)(right + 8)))
85         {
86             return false;
87         }
88         left += 10;
89         right += 10;
90         length -= 10;
91     }
92     return true;
93 }
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
        ↪ int length)
96 {
97     // This depends on the fact that the String objects are
98     // always zero terminated and that the terminating zero is not included
99     // in the length. For odd string sizes, the last compare will include
100    // the zero terminator.
101    while (length > 0)
102    {
103        if ((* (int*)left != * (int*)right)
104            {
105            break;
106        }
107        left += 2;
108        right += 2;
109        length -= 2;
110    }
111 }
112 }
113 }

```

1.7 ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Collections.Arrays
6  {
7      /// <summary>
8      /// <para>Represents a set of extension methods for a <see cref="T:T[]" /> array.</para>
9      /// <para>Представляет набор методов расширения для массива <see cref="T:T[]" />.</para>
10     /// </summary>
11     public static class GenericArrayExtensions
12     {
13         /// <summary>
14         /// <para>Checks if an array exists, if so, checks the array length using the index
        ↪ variable type int, and if the array length is greater than the index - return
        ↪ array[index], otherwise - default value.</para>
15         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
        ↪ помощью переменной index, и если длина массива больше индекса - возвращает
        ↪ array[index], иначе - значение по умолчанию.</para>
16         /// </summary>
17         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ↪ массива.</para></typeparam>

```

```

18  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
19  /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
    → сравнения.</para></param>
20  /// <returns><para>Array element or default value.</para><para>Элемент массива или же
    → значение по умолчанию.</para></returns>
21  [MethodImpl(MethodImplOptions.AggressiveInlining)]
22  public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
    → array.Length > index ? array[index] : default;
23
24  /// <summary>
25  /// <para>Checks whether the array exists, if so, checks the array length using the
    → index variable type long, and if the array length is greater than the index - return
    → array[index], otherwise - default value.</para>
26  /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
    → помощью переменной index, и если длина массива больше индекса - возвращает
    → array[index], иначе - значение по умолчанию.</para>
27  /// </summary>
28  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
29  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
30  /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
    → для сравнения.</para></param>
31  /// <returns><para>Array element or default value.</para><para>Элемент массива или же
    → значение по умолчанию.</para></returns>
32  [MethodImpl(MethodImplOptions.AggressiveInlining)]
33  public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
    → array.LongLength > index ? array[index] : default;
34
35  /// <summary>
36  /// <para>Checks whether the array exist, if so, checks the array length using the index
    → variable type int, and if the array length is greater than the index, set the element
    → variable to array[index] and return <see langword="true"/>.</para>
37  /// <para>Проверяет, существует ли массив, если да, то идет проверка длины массива с
    → помощью переменной index типа int, и если длина массива больше значения index,
    → устанавливает значение переменной element - array[index] и возвращает <see
    → langword="true"/>.</para>
38  /// </summary>
39  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
40  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
41  /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
    → сравнения.</para></param>
42  /// <param name="element"><para>Passing the argument by reference, if successful, it
    → will take the value array[index] otherwise default value.</para><para>Передаёт
    → аргумент по ссылке, в случае успеха он примет значение array[index] в противном
    → случае значение по умолчанию.</para></param>
43  /// <returns><para><see langword="true"/> if successful otherwise <see
    → langword="false"/>.</para><para><see langword="true"/> в случае успеха, в противном
    → случае <see langword="false"/>.</para></returns>
44  [MethodImpl(MethodImplOptions.AggressiveInlining)]
45  public static bool TryGetElement<T>(this T[] array, int index, out T element)
46  {
47      if (array != null && array.Length > index)
48      {
49          element = array[index];
50          return true;
51      }
52      else
53      {
54          element = default;
55          return false;
56      }
57  }
58
59  /// <summary>
60  /// <para>Checks whether the array exist, if so, checks the array length using the
    → index variable type long, and if the array length is greater than the index, set the
    → element variable to array[index] and return <see langword="true"/>.</para>

```

```

61  /// <para>Проверяет, существует ли массив, если да, то идет проверка длины массива с
    → помощью переменной index типа long, и если длина массива больше значения index,
    → устанавливает значение переменной element - array[index] и возвращает <see
    → langword="true"/>.</para>
62  /// </summary>
63  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
64  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
65  /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
    → для сравнения.</para></param>
66  /// <param name="element"><para>Passing the argument by reference, if successful, it
    → will take the value array[index] otherwise default value.</para><para>Передаёт
    → аргумент по ссылке, в случае успеха он примет значение array[index] в противном
    → случае значение по умолчанию.</para></param>
67  /// <returns><para><see langword="true"/> if successful otherwise <see
    → langword="false"/>.</para><para><see langword="true"/> в случае успеха, в противном
    → случае <see langword="false"/>.</para></returns>
68  [MethodImpl(MethodImplOptions.AggressiveInlining)]
69  public static bool TryGetElement<T>(this T[] array, long index, out T element)
70  {
71      if (array != null && array.LongLength > index)
72      {
73          element = array[index];
74          return true;
75      }
76      else
77      {
78          element = default;
79          return false;
80      }
81  }
82
83  /// <summary>
84  /// <para>Copying of elements from one array to another array.</para>
85  /// <para>Копирует элементы из одного массива в другой массив.</para>
86  /// </summary>
87  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
88  /// <param name="array"><para>The array to copy.</para><para>Массив который необходимо
    → скопировать.</para></param>
89  /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
90  [MethodImpl(MethodImplOptions.AggressiveInlining)]
91  public static T[] Clone<T>(this T[] array)
92  {
93      var copy = new T[array.LongLength];
94      Array.Copy(array, 0L, copy, 0L, array.LongLength);
95      return copy;
96  }
97
98  /// <summary>
99  /// <para>Shifts all the elements of the array by one position to the right.</para>
100  /// <para>Сдвигает вправо все элементы массива на одну позицию.</para>
101  /// </summary>
102  /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
    → массива.</para></typeparam>
103  /// <param name="array"><para>The array to copy from.</para><para>Массив для
    → копирования.</para></param>
104  /// <returns>
105  /// <para>Array with a shift of elements by one position.</para>
106  /// <para>Массив со сдвигом элементов на одну позицию.</para>
107  /// </returns>
108  [MethodImpl(MethodImplOptions.AggressiveInlining)]
109  public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
110
111  /// <summary>
112  /// <para>Shifts all elements of the array to the right by the specified number of
    → elements.</para>
113  /// <para>Сдвигает вправо все элементы массива на указанное количество элементов.</para>
114  /// </summary>
115  /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
    → массива.</para></typeparam>
116  /// <param name="array"><para>The array to copy from.</para><para>Массив для
    → копирования.</para></param>
117  /// <param name="shift"><para>The number of items to shift.</para><para>Количество
    → сдвигаемых элементов.</para></param>

```

```

118 /// <returns>
119 /// <para>If the value of the shift variable is less than zero - an <see
    → cref="NotImplementedException"/> exception is thrown, but if the value of the shift
    → variable is 0 - an exact copy of the array is returned. Otherwise, an array is
    → returned with the shift of the elements.</para>
120 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
    → cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
    → возвращается точная копия массива. Иначе возвращается массив со сдвигом
    → элементов.</para>
121 /// </returns>
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static IList<T> ShiftRight<T>(this T[] array, long shift)
124 {
125     if (shift < 0)
126     {
127         throw new NotImplementedException();
128     }
129     if (shift == 0)
130     {
131         return array.Clone<T>();
132     }
133     else
134     {
135         var restrictions = new T[array.LongLength + shift];
136         Array.Copy(array, 0L, restrictions, shift, array.LongLength);
137         return restrictions;
138     }
139 }
140
141 /// <summary>
142 /// <para>Adding in array the passed element at the specified position and increments
    → position value by one.</para>
143 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
    → значение position на единицу.</para>
144 /// </summary>
145 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
146 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
147 /// <param name="position"><para>A reference to the position of type int where the
    → element will be added.</para><para>Ссылка на позицию типа int, в которую будет
    → добавлен элемент.</para></param>
148 /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
    → который нужно добавить в массив.</para></param>
149 [MethodImpl(MethodImplOptions.AggressiveInlining)]
150 public static void Add<T>(this T[] array, ref int position, T element) =>
    → array[position++] = element;
151
152 /// <summary>
153 /// <para>Adding in array the passed element at the specified position and increments
    → position value by one.</para>
154 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
    → значение position на единицу.</para>
155 /// </summary>
156 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
157 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
158 /// <param name="position"><para>A reference to the position of type long where the
    → element will be added.</para><para>Ссылка на позицию типа long, в которую будет
    → добавлен элемент.</para></param>
159 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
    → который необходимо добавить в массив.</para></param>
160 [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 public static void Add<T>(this T[] array, ref long position, T element) =>
    → array[position++] = element;
162
163 /// <summary>
164 /// <para>Adding in array the passed element, at the specified position, increments
    → position value by one and returns the value of the passed constant.</para>
165 /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
    → значение position на единицу и возвращает значение переданной константы.</para>
166 /// </summary>
167 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
    → массива.</para></typeparam>

```

```

168  /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
    → возвращаемой константы.</para></typeparam>
169  /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
170  /// <param name="position"><para>Reference to the position to which the element will be
    → added.</para><para>Ссылка на позицию, в которую будет добавлен
    → элемент.</para></param>
171  /// <param name="element"><para>The element to add to the array.</para><para>Элемент
    → который необходимо добавить в массив.</para></param>
172  /// <param name="returnConstant"><para>The constant value that will be
    → returned.</para><para>Значение константы, которое будет возвращено.</para></param>
173  /// </returns>
174  /// <para>The constant value passed as an argument.</para>
175  /// <para>Значение константы, переданное в качестве аргумента.</para>
176  /// </returns>
177  [MethodImpl(MethodImplOptions.AggressiveInlining)]
178  public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, TElement element, TReturnConstant
    → returnConstant)
179  {
180      array.Add(ref position, element);
181      return returnConstant;
182  }
183
184  /// <summary>
185  /// <para>Adds the first element from the passed collection to the array, at the
    → specified position and increments position value by one.</para>
186  /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
    → позицию и увеличивает значение position на единицу.</para>
187  /// </summary>
188  /// <typeparam name="T"><para>Array element type.</para><para>Тип элементов
    → массива.</para></typeparam>
189  /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
190  /// <param name="position"><para>Reference to the position to which the element will be
    → added.</para><para>Ссылка на позицию, в которую будет добавлен
    → элемент.</para></param>
191  /// <param name="elements"><para>List, the first element of which will be added to the
    → array.</para><para>Список, первый элемент которого будет добавлен в
    → массив.</para></param>
192  [MethodImpl(MethodImplOptions.AggressiveInlining)]
193  public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
    → array[position++] = elements[0];
194
195  /// <summary>
196  /// <para>Adds the first element from the passed collection to the array, at the
    → specified position, increments position value by one and returns the value of the
    → passed constant.</para>
197  /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
    → позицию, увеличивает значение position на единицу и возвращает значение переданной
    → константы.</para>
198  /// </summary>
199  /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
    → массива.</para></typeparam>
200  /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
    → возвращаемой константы.</para></typeparam>
201  /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
202  /// <param name="position"><para>Reference to the position to which the element will be
    → added.</para><para>Ссылка на позицию, в которую будет добавлен
    → элемент.</para></param>
203  /// <param name="elements"><para>List, the first element of which will be added to the
    → array.</para><para>Список, первый элемент которого будет добавлен в
    → массив.</para></param>
204  /// <param name="returnConstant"><para>The constant value that will be
    → returned.</para><para>Значение константы, которое будет возвращено.</para></param>
205  /// </returns>
206  /// <para>The constant value passed as an argument.</para>
207  /// <para>Значение константы, переданное в качестве аргумента.</para>
208  /// </returns>
209  [MethodImpl(MethodImplOptions.AggressiveInlining)]
210  public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
    → returnConstant)
211  {

```

```

212     array.AddFirst(ref position, elements);
213     return returnConstant;
214 }
215
216 /// <summary>
217 /// <para>Adding in array all elements from the passed collection, at the specified
    → position, increases the position value by the number of elements added and returns
    → the value of the passed constant.</para>
218 /// <para>Добавляет в массив все элементы из переданной коллекции, на указанную позицию,
    → увеличивает значение position на количество добавленных элементов и возвращает
    → значение переданной константы.</para>
219 /// </summary>
220 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
    → массива.</para></typeparam>
221 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
    → возвращаемой константы.</para></typeparam>
222 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элементы.</para></param>
223 /// <param name="position"><para>Reference to the position from which elements will be
    → added to the array.</para><para>Ссылка на позицию, начиная с которой будут
    → добавляться элементы в массив.</para></param>
224 /// <param name="elements"><para>List, whose elements will be added to the
    → array.</para><para>Список, элементы которого будут добавлены в
    → массив.</para></param>
225 /// <param name="returnConstant"><para>The constant value that will be
    → returned.</para><para>Значение константы, которое будет возвращено.</para></param>
226 /// <returns>
227 /// <para>The constant value passed as an argument.</para>
228 /// <para>Значение константы, переданное в качестве аргумента.</para>
229 /// </returns>
230 [MethodImpl(MethodImplOptions.AggressiveInlining)]
231 public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
    → returnConstant)
232 {
233     array.AddAll(ref position, elements);
234     return returnConstant;
235 }
236
237 /// <summary>
238 /// <para>Adding in array a collection of elements, starting from a specific position
    → and increases the position value by the number of elements added.</para>
239 /// <para>Добавляет в массив все элементы коллекции, начиная с определенной позиции и
    → увеличивает значение position на количество добавленных элементов.</para>
240 /// </summary>
241 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
242 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элементы.</para></param>
243 /// <param name="position"><para>Reference to the position from which elements will be
    → added to the array.</para><para>Ссылка на позицию, начиная с которой будут
    → добавляться элементы в массив.</para></param>
244 /// <param name="elements"><para>List, whose elements will be added to the
    → array.</para><para>Список, элементы которого будут добавлены в
    → массив.</para></param>
245 [MethodImpl(MethodImplOptions.AggressiveInlining)]
246 public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
247 {
248     for (var i = 0; i < elements.Count; i++)
249     {
250         array.Add(ref position, elements[i]);
251     }
252 }
253
254 /// <summary>
255 /// <para>Adding in array all elements of the collection, skipping the first position,
    → increments position value by one and returns the value of the passed constant.</para>
256 /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию,
    → увеличивает значение position на единицу и возвращает значение переданной
    → константы.</para>
257 /// </summary>
258 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
    → массива.</para></typeparam>
259 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
    → возвращаемой константы.</para></typeparam>

```

```

260  /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    ↳ необходимо добавить элементы.</para></param>
261  /// <param name="position"><para>Reference to the position from which to start adding
    ↳ elements.</para><para>Ссылка на позицию, с которой начинается добавление
    ↳ элементов.</para></param>
262  /// <param name="elements"><para>List, whose elements will be added to the
    ↳ array.</para><para>Список, элементы которого будут добавлены в
    ↳ массив.</para></param>
263  /// <param name="returnConstant"><para>The constant value that will be
    ↳ returned.</para><para>Значение константы, которое будет возвращено.</para></param>
264  /// <returns>
265  /// <para>The constant value passed as an argument.</para>
266  /// <para>Значение константы, переданное в качестве аргумента.</para>
267  /// </returns>
268  [MethodImpl(MethodImplOptions.AggressiveInlining)]
269  public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
    ↳ TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
    ↳ TReturnConstant returnConstant)
270  {
271      array.AddSkipFirst(ref position, elements);
272      return returnConstant;
273  }
274
275  /// <summary>
276  /// <para>Adding in array all elements of the collection, skipping the first position
    ↳ and increments position value by one.</para>
277  /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию и
    ↳ увеличивает значение position на единицу.</para>
278  /// </summary>
279  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    ↳ массива.</para></typeparam>
280  /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    ↳ необходимо добавить элементы.</para></param>
281  /// <param name="position"><para>Reference to the position from which to start adding
    ↳ elements.</para><para>Ссылка на позицию, с которой начинается добавление
    ↳ элементов.</para></param>
282  /// <param name="elements"><para>List, whose elements will be added to the
    ↳ array.</para><para>Список, элементы которого будут добавлены в
    ↳ массив.</para></param>
283  [MethodImpl(MethodImplOptions.AggressiveInlining)]
284  public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
    ↳ => array.AddSkipFirst(ref position, elements, 1);
285
286  /// <summary>
287  /// <para>Adding in array all but the first element, skipping a specified number of
    ↳ positions and increments position value by one.</para>
288  /// <para>Добавляет в массив все элементы коллекции, кроме первого, пропуская
    ↳ определенное количество позиций и увеличивает значение position на единицу.</para>
289  /// </summary>
290  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    ↳ массива.</para></typeparam>
291  /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    ↳ необходимо добавить элементы.</para></param>
292  /// <param name="position"><para>Reference to the position from which to start adding
    ↳ elements.</para><para>Ссылка на позицию, с которой начинается добавление
    ↳ элементов.</para></param>
293  /// <param name="elements"><para>List, whose elements will be added to the
    ↳ array.</para><para>Список, элементы которого будут добавлены в
    ↳ массив.</para></param>
294  /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
    ↳ пропускаемых элементов.</para></param>
295  [MethodImpl(MethodImplOptions.AggressiveInlining)]
296  public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
    ↳ int skip)
297  {
298      for (var i = skip; i < elements.Count; i++)
299      {
300          array.Add(ref position, elements[i]);
301      }
302  }
303  }
304  }

```

1.8 ./csharp/Platform.Collections/BitString.cs

```

1  using System;
2  using System.Collections.Concurrent;

```

```

3 using System.Collections.Generic;
4 using System.Numerics;
5 using System.Runtime.CompilerServices;
6 using System.Threading.Tasks;
7 using Platform.Exceptions;
8 using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>
16     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
17     ///   ↳ 64 бит в массиве значений.
18     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
19     ///   ↳ байт в 8 байт.
20     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
21     ///   ↳ помощью которой можно быстро
22     /// проверять есть ли значения непосредственно далее (ниже по уровню).
23     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
24     /// </remarks>
25     public class BitString : IEquatable<BitString>
26     {
27         private static readonly byte[] [] _bitsSetIn16Bits;
28         private long[] _array;
29         private long _length;
30         private long _minPositiveWord;
31         private long _maxPositiveWord;
32
33         /// <summary>
34         /// <para>
35         /// The value.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         public bool this[long index]
40         {
41             [MethodImpl(MethodImplOptions.AggressiveInlining)]
42             get => Get(index);
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             set => Set(index, value);
45         }
46
47         /// <summary>
48         /// <para>
49         /// Gets or sets the length value.
50         /// </para>
51         /// <para></para>
52         /// </summary>
53         public long Length
54         {
55             [MethodImpl(MethodImplOptions.AggressiveInlining)]
56             get => _length;
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             set
59             {
60                 if (_length == value)
61                 {
62                     return;
63                 }
64                 Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
65                 // Currently we never shrink the array
66                 if (value > _length)
67                 {
68                     var words = GetWordsCountFromIndex(value);
69                     var oldWords = GetWordsCountFromIndex(_length);
70                     if (words > _array.LongLength)
71                     {
72                         var copy = new long[words];
73                         Array.Copy(_array, copy, _array.LongLength);
74                         _array = copy;
75                     }
76                     else
77                     {
78                         // What is going on here?
79                         Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
80                     }
81                     // What is going on here?
82                 }
83             }
84         }
85     }
86 }

```



```

79         var mask = (int)(_length % 64);
80         if (mask > 0)
81         {
82             _array[oldWords - 1] &= (1L << mask) - 1;
83         }
84     }
85     else
86     {
87         // Looks like minimum and maximum positive words are not updated
88         throw new NotImplementedException();
89     }
90     _length = value;
91 }
92 }
93
94 #region Constructors
95
96 /// <summary>
97 /// <para>
98 /// Initializes a new <see cref="BitString"/> instance.
99 /// </para>
100 /// <para></para>
101 /// </summary>
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 static BitString()
104 {
105     _bitsSetIn16Bits = new byte[65536][];
106     int i, c, k;
107     byte bitIndex;
108     for (i = 0; i < 65536; i++)
109     {
110         // Calculating size of array (number of positive bits)
111         for (c = 0, k = 1; k <= 65536; k <= 1)
112         {
113             if ((i & k) == k)
114             {
115                 c++;
116             }
117         }
118         var array = new byte[c];
119         // Adding positive bits indices into array
120         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
121         {
122             if ((i & k) == k)
123             {
124                 array[c++] = bitIndex;
125             }
126             bitIndex++;
127         }
128         _bitsSetIn16Bits[i] = array;
129     }
130 }
131
132 /// <summary>
133 /// <para>
134 /// Initializes a new <see cref="BitString"/> instance.
135 /// </para>
136 /// <para></para>
137 /// </summary>
138 /// <param name="other">
139 /// <para>A other.</para>
140 /// <para></para>
141 /// </param>
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public BitString(BitString other)
144 {
145     Ensure.Always.ArgumentNotNull(other, nameof(other));
146     _length = other._length;
147     _array = new long[GetWordsCountFromIndex(_length)];
148     _minPositiveWord = other._minPositiveWord;
149     _maxPositiveWord = other._maxPositiveWord;
150     Array.Copy(other._array, _array, _array.LongLength);
151 }
152
153 /// <summary>
154 /// <para>
155 /// Initializes a new <see cref="BitString"/> instance.
156 /// </para>
157 /// <para></para>

```

```

158     /// </summary>
159     /// <param name="length">
160     /// <para>A length.</para>
161     /// <para></para>
162     /// </param>
163     [MethodImpl(MethodImplOptions.AggressiveInlining)]
164     public BitString(long length)
165     {
166         Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
167         _length = length;
168         _array = new long[GetWordsCountFromIndex(_length)];
169         MarkBordersAsAllBitsReset();
170     }
171
172     /// <summary>
173     /// <para>
174     /// Initializes a new <see cref="BitString"/> instance.
175     /// </para>
176     /// <para></para>
177     /// </summary>
178     /// <param name="length">
179     /// <para>A length.</para>
180     /// <para></para>
181     /// </param>
182     /// <param name="defaultValue">
183     /// <para>A default value.</para>
184     /// <para></para>
185     /// </param>
186     [MethodImpl(MethodImplOptions.AggressiveInlining)]
187     public BitString(long length, bool defaultValue)
188         : this(length)
189     {
190         if (defaultValue)
191         {
192             SetAll();
193         }
194     }
195
196 #endregion
197
198     /// <summary>
199     /// <para>
200     /// Nots this instance.
201     /// </para>
202     /// <para></para>
203     /// </summary>
204     /// <returns>
205     /// <para>The bit string</para>
206     /// <para></para>
207     /// </returns>
208     [MethodImpl(MethodImplOptions.AggressiveInlining)]
209     public BitString Not()
210     {
211         for (var i = 0L; i < _array.LongLength; i++)
212         {
213             _array[i] = ~_array[i];
214             RefreshBordersByWord(i);
215         }
216         return this;
217     }
218
219     /// <summary>
220     /// <para>
221     /// Parallels the not.
222     /// </para>
223     /// <para></para>
224     /// </summary>
225     /// <returns>
226     /// <para>The bit string</para>
227     /// <para></para>
228     /// </returns>
229     [MethodImpl(MethodImplOptions.AggressiveInlining)]
230     public BitString ParallelNot()
231     {
232         var threads = Environment.ProcessorCount / 2;
233         if (threads <= 1)
234         {
235             return Not();
236         }
237     }

```

```

236     }
237     var partitioner = Partitioner.Create(OL, _array.LongLength, _array.LongLength /
    ↪ threads);
238     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    ↪ MaxDegreeOfParallelism = threads }, range =>
    {
239         var maximum = range.Item2;
240         for (var i = range.Item1; i < maximum; i++)
241         {
242             _array[i] = ~_array[i];
243         }
244     });
245     MarkBordersAsAllBitsSet();
246     TryShrinkBorders();
247     return this;
248 }
249
250
251 /// <summary>
252 /// <para>
253 /// Vectors the not.
254 /// </para>
255 /// <para></para>
256 /// </summary>
257 /// <returns>
258 /// <para>The bit string</para>
259 /// <para></para>
260 /// </returns>
261 [MethodImpl(MethodImplOptions.AggressiveInlining)]
262 public BitString VectorNot()
263 {
264     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
265     {
266         return Not();
267     }
268     var step = Vector<long>.Count;
269     if (_array.Length < step)
270     {
271         return Not();
272     }
273     VectorNotLoop(_array, step, 0, _array.Length);
274     MarkBordersAsAllBitsSet();
275     TryShrinkBorders();
276     return this;
277 }
278
279 /// <summary>
280 /// <para>
281 /// Parallels the vector not.
282 /// </para>
283 /// <para></para>
284 /// </summary>
285 /// <returns>
286 /// <para>The bit string</para>
287 /// <para></para>
288 /// </returns>
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public BitString ParallelVectorNot()
291 {
292     var threads = Environment.ProcessorCount / 2;
293     if (threads <= 1)
294     {
295         return VectorNot();
296     }
297     if (!Vector.IsHardwareAccelerated)
298     {
299         return ParallelNot();
300     }
301     var step = Vector<long>.Count;
302     if (_array.Length < (step * threads))
303     {
304         return VectorNot();
305     }
306     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
307     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
    ↪ range.Item1, range.Item2));
308     MarkBordersAsAllBitsSet();
309     TryShrinkBorders();

```

```

310         return this;
311     }
312
313     /// <summary>
314     /// <para>
315     /// Vectors the not loop using the specified array.
316     /// </para>
317     /// <para></para>
318     /// </summary>
319     /// <param name="array">
320     /// <para>The array.</para>
321     /// <para></para>
322     /// </param>
323     /// <param name="step">
324     /// <para>The step.</para>
325     /// <para></para>
326     /// </param>
327     /// <param name="start">
328     /// <para>The start.</para>
329     /// <para></para>
330     /// </param>
331     /// <param name="maximum">
332     /// <para>The maximum.</para>
333     /// <para></para>
334     /// </param>
335     [MethodImpl(MethodImplOptions.AggressiveInlining)]
336     static private void VectorNotLoop(long[] array, int step, int start, int maximum)
337     {
338         var i = start;
339         var range = maximum - start - 1;
340         var stop = range - (range % step);
341         for (; i < stop; i += step)
342         {
343             (~new Vector<long>(array, i)).CopyTo(array, i);
344         }
345         for (; i < maximum; i++)
346         {
347             array[i] = ~array[i];
348         }
349     }
350
351     /// <summary>
352     /// <para>
353     /// Ands the other.
354     /// </para>
355     /// <para></para>
356     /// </summary>
357     /// <param name="other">
358     /// <para>The other.</para>
359     /// <para></para>
360     /// </param>
361     /// <returns>
362     /// <para>The bit string</para>
363     /// <para></para>
364     /// </returns>
365     [MethodImpl(MethodImplOptions.AggressiveInlining)]
366     public BitString And(BitString other)
367     {
368         EnsureBitStringHasTheSameSize(other, nameof(other));
369         GetCommonOuterBorders(this, other, out long from, out long to);
370         var otherArray = other._array;
371         for (var i = from; i <= to; i++)
372         {
373             _array[i] &= otherArray[i];
374             RefreshBordersByWord(i);
375         }
376         return this;
377     }
378
379     /// <summary>
380     /// <para>
381     /// Parallels the and using the specified other.
382     /// </para>
383     /// <para></para>
384     /// </summary>
385     /// <param name="other">
386     /// <para>The other.</para>
387     /// <para></para>

```

```

388 /// </param>
389 /// <returns>
390 /// <para>The bit string</para>
391 /// <para></para>
392 /// </returns>
393 [MethodImpl(MethodImplOptions.AggressiveInlining)]
394 public BitString ParallelAnd(BitString other)
395 {
396     var threads = Environment.ProcessorCount / 2;
397     if (threads <= 1)
398     {
399         return And(other);
400     }
401     EnsureBitStringHasTheSameSize(other, nameof(other));
402     GetCommonOuterBorders(this, other, out long from, out long to);
403     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
404     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
405         ↪ MaxDegreeOfParallelism = threads }, range =>
406     {
407         var maximum = range.Item2;
408         for (var i = range.Item1; i < maximum; i++)
409         {
410             _array[i] &= other._array[i];
411         }
412     });
413     MarkBordersAsAllBitsSet();
414     TryShrinkBorders();
415     return this;
416 }
417
418 /// <summary>
419 /// <para>
420 /// Vectors the and using the specified other.
421 /// </para>
422 /// <para></para>
423 /// </summary>
424 /// <param name="other">
425 /// <para>The other.</para>
426 /// <para></para>
427 /// </param>
428 /// <returns>
429 /// <para>The bit string</para>
430 /// <para></para>
431 /// </returns>
432 [MethodImpl(MethodImplOptions.AggressiveInlining)]
433 public BitString VectorAnd(BitString other)
434 {
435     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
436     {
437         return And(other);
438     }
439     var step = Vector<long>.Count;
440     if (_array.Length < step)
441     {
442         return And(other);
443     }
444     EnsureBitStringHasTheSameSize(other, nameof(other));
445     GetCommonOuterBorders(this, other, out int from, out int to);
446     VectorAndLoop(_array, other._array, step, from, to + 1);
447     MarkBordersAsAllBitsSet();
448     TryShrinkBorders();
449     return this;
450 }
451
452 /// <summary>
453 /// <para>
454 /// Parallels the vector and using the specified other.
455 /// </para>
456 /// <para></para>
457 /// </summary>
458 /// <param name="other">
459 /// <para>The other.</para>
460 /// <para></para>
461 /// </param>
462 /// <returns>
463 /// <para>The bit string</para>
464 /// <para></para>
465 /// </returns>

```

```

465 [MethodImpl(MethodImplOptions.AggressiveInlining)]
466 public BitString ParallelVectorAnd(BitString other)
467 {
468     var threads = Environment.ProcessorCount / 2;
469     if (threads <= 1)
470     {
471         return VectorAnd(other);
472     }
473     if (!Vector.IsHardwareAccelerated)
474     {
475         return ParallelAnd(other);
476     }
477     var step = Vector<long>.Count;
478     if (_array.Length < (step * threads))
479     {
480         return VectorAnd(other);
481     }
482     EnsureBitStringHasTheSameSize(other, nameof(other));
483     GetCommonOuterBorders(this, other, out int from, out int to);
484     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
485     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
486         ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
487         ↪ step, range.Item1, range.Item2));
488     MarkBordersAsAllBitsSet();
489     TryShrinkBorders();
490     return this;
491 }
492
493 /// <summary>
494 /// <para>
495 /// Vectors the and loop using the specified array.
496 /// </para>
497 /// <para></para>
498 /// </summary>
499 /// <param name="array">
500 /// <para>The array.</para>
501 /// </param>
502 /// <param name="otherArray">
503 /// <para>The other array.</para>
504 /// </param>
505 /// <param name="step">
506 /// <para>The step.</para>
507 /// </param>
508 /// <param name="start">
509 /// <para>The start.</para>
510 /// </param>
511 /// <param name="maximum">
512 /// <para>The maximum.</para>
513 /// </param>
514 [MethodImpl(MethodImplOptions.AggressiveInlining)]
515 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
516 ↪ int maximum)
517 {
518     var i = start;
519     var range = maximum - start - 1;
520     var stop = range - (range % step);
521     for (; i < stop; i += step)
522     {
523         (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
524     }
525     for (; i < maximum; i++)
526     {
527         array[i] &= otherArray[i];
528     }
529 }
530
531 /// <summary>
532 /// <para>
533 /// Ors the other.
534 /// </para>
535 /// <para></para>
536 /// </summary>
537 /// <param name="other">

```

```

540 /// <para>The other.</para>
541 /// <para></para>
542 /// </param>
543 /// <returns>
544 /// <para>The bit string</para>
545 /// <para></para>
546 /// </returns>
547 [MethodImpl(MethodImplOptions.AggressiveInlining)]
548 public BitString Or(BitString other)
549 {
550     EnsureBitStringHasTheSameSize(other, nameof(other));
551     GetCommonOuterBorders(this, other, out long from, out long to);
552     for (var i = from; i <= to; i++)
553     {
554         _array[i] |= other._array[i];
555         RefreshBordersByWord(i);
556     }
557     return this;
558 }
559
560 /// <summary>
561 /// <para>
562 /// Parallels the or using the specified other.
563 /// </para>
564 /// <para></para>
565 /// </summary>
566 /// <param name="other">
567 /// <para>The other.</para>
568 /// <para></para>
569 /// </param>
570 /// <returns>
571 /// <para>The bit string</para>
572 /// <para></para>
573 /// </returns>
574 [MethodImpl(MethodImplOptions.AggressiveInlining)]
575 public BitString ParallelOr(BitString other)
576 {
577     var threads = Environment.ProcessorCount / 2;
578     if (threads <= 1)
579     {
580         return Or(other);
581     }
582     EnsureBitStringHasTheSameSize(other, nameof(other));
583     GetCommonOuterBorders(this, other, out long from, out long to);
584     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
585     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
586         ↪ MaxDegreeOfParallelism = threads }, range =>
587     {
588         var maximum = range.Item2;
589         for (var i = range.Item1; i < maximum; i++)
590         {
591             _array[i] |= other._array[i];
592         }
593     });
594     MarkBordersAsAllBitsSet();
595     TryShrinkBorders();
596     return this;
597 }
598
599 /// <summary>
600 /// <para>
601 /// Vectors the or using the specified other.
602 /// </para>
603 /// <para></para>
604 /// </summary>
605 /// <param name="other">
606 /// <para>The other.</para>
607 /// <para></para>
608 /// </param>
609 /// <returns>
610 /// <para>The bit string</para>
611 /// <para></para>
612 /// </returns>
613 [MethodImpl(MethodImplOptions.AggressiveInlining)]
614 public BitString VectorOr(BitString other)
615 {
616     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)

```

```

617         return Or(other);
618     }
619     var step = Vector<long>.Count;
620     if (_array.Length < step)
621     {
622         return Or(other);
623     }
624     EnsureBitStringHasTheSameSize(other, nameof(other));
625     GetCommonOuterBorders(this, other, out int from, out int to);
626     VectorOrLoop(_array, other._array, step, from, to + 1);
627     MarkBordersAsAllBitsSet();
628     TryShrinkBorders();
629     return this;
630 }
631
632 /// <summary>
633 /// <para>
634 /// Parallels the vector or using the specified other.
635 /// </para>
636 /// <para></para>
637 /// </summary>
638 /// <param name="other">
639 /// <para>The other.</para>
640 /// <para></para>
641 /// </param>
642 /// <returns>
643 /// <para>The bit string</para>
644 /// <para></para>
645 /// </returns>
646 [MethodImpl(MethodImplOptions.AggressiveInlining)]
647 public BitString ParallelVectorOr(BitString other)
648 {
649     var threads = Environment.ProcessorCount / 2;
650     if (threads <= 1)
651     {
652         return VectorOr(other);
653     }
654     if (!Vector.IsHardwareAccelerated)
655     {
656         return ParallelOr(other);
657     }
658     var step = Vector<long>.Count;
659     if (_array.Length < (step * threads))
660     {
661         return VectorOr(other);
662     }
663     EnsureBitStringHasTheSameSize(other, nameof(other));
664     GetCommonOuterBorders(this, other, out int from, out int to);
665     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
666     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↳ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
        ↳ step, range.Item1, range.Item2));
667     MarkBordersAsAllBitsSet();
668     TryShrinkBorders();
669     return this;
670 }
671
672 /// <summary>
673 /// <para>
674 /// Vectors the or loop using the specified array.
675 /// </para>
676 /// <para></para>
677 /// </summary>
678 /// <param name="array">
679 /// <para>The array.</para>
680 /// <para></para>
681 /// </param>
682 /// <param name="otherArray">
683 /// <para>The other array.</para>
684 /// <para></para>
685 /// </param>
686 /// <param name="step">
687 /// <para>The step.</para>
688 /// <para></para>
689 /// </param>
690 /// <param name="start">
691 /// <para>The start.</para>
692 /// <para></para>

```



```

693     /// </param>
694     /// <param name="maximum">
695     /// <para>The maximum.</para>
696     /// <para></para>
697     /// </param>
698     [MethodImpl(MethodImplOptions.AggressiveInlining)]
699     static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
700     ↪ int maximum)
701     {
702         var i = start;
703         var range = maximum - start - 1;
704         var stop = range - (range % step);
705         for (; i < stop; i += step)
706         {
707             (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
708         }
709         for (; i < maximum; i++)
710         {
711             array[i] |= otherArray[i];
712         }
713     }
714     /// <summary>
715     /// <para>
716     /// Xors the other.
717     /// </para>
718     /// <para></para>
719     /// </summary>
720     /// <param name="other">
721     /// <para>The other.</para>
722     /// <para></para>
723     /// </param>
724     /// <returns>
725     /// <para>The bit string</para>
726     /// <para></para>
727     /// </returns>
728     [MethodImpl(MethodImplOptions.AggressiveInlining)]
729     public BitString Xor(BitString other)
730     {
731         EnsureBitStringHasTheSameSize(other, nameof(other));
732         GetCommonOuterBorders(this, other, out long from, out long to);
733         for (var i = from; i <= to; i++)
734         {
735             _array[i] ^= other._array[i];
736             RefreshBordersByWord(i);
737         }
738         return this;
739     }
740     /// <summary>
741     /// <para>
742     /// Parallels the xor using the specified other.
743     /// </para>
744     /// <para></para>
745     /// </summary>
746     /// <param name="other">
747     /// <para>The other.</para>
748     /// <para></para>
749     /// </param>
750     /// <returns>
751     /// <para>The bit string</para>
752     /// <para></para>
753     /// </returns>
754     [MethodImpl(MethodImplOptions.AggressiveInlining)]
755     public BitString ParallelXor(BitString other)
756     {
757         var threads = Environment.ProcessorCount / 2;
758         if (threads <= 1)
759         {
760             return Xor(other);
761         }
762         EnsureBitStringHasTheSameSize(other, nameof(other));
763         GetCommonOuterBorders(this, other, out long from, out long to);
764         var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
765         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
766             ↪ MaxDegreeOfParallelism = threads }, range =>
767         {
768             var maximum = range.Item2;

```

```

769         for (var i = range.Item1; i < maximum; i++)
770         {
771             _array[i] ^= other._array[i];
772         }
773     });
774     MarkBordersAsAllBitsSet();
775     TryShrinkBorders();
776     return this;
777 }
778
779 /// <summary>
780 /// <para>
781 /// Vectors the xor using the specified other.
782 /// </para>
783 /// <para></para>
784 /// </summary>
785 /// <param name="other">
786 /// <para>The other.</para>
787 /// <para></para>
788 /// </param>
789 /// <returns>
790 /// <para>The bit string</para>
791 /// <para></para>
792 /// </returns>
793 [MethodImpl(MethodImplOptions.AggressiveInlining)]
794 public BitString VectorXor(BitString other)
795 {
796     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
797     {
798         return Xor(other);
799     }
800     var step = Vector<long>.Count;
801     if (_array.Length < step)
802     {
803         return Xor(other);
804     }
805     EnsureBitStringHasTheSameSize(other, nameof(other));
806     GetCommonOuterBorders(this, other, out int from, out int to);
807     VectorXorLoop(_array, other._array, step, from, to + 1);
808     MarkBordersAsAllBitsSet();
809     TryShrinkBorders();
810     return this;
811 }
812
813 /// <summary>
814 /// <para>
815 /// Parallels the vector xor using the specified other.
816 /// </para>
817 /// <para></para>
818 /// </summary>
819 /// <param name="other">
820 /// <para>The other.</para>
821 /// <para></para>
822 /// </param>
823 /// <returns>
824 /// <para>The bit string</para>
825 /// <para></para>
826 /// </returns>
827 [MethodImpl(MethodImplOptions.AggressiveInlining)]
828 public BitString ParallelVectorXor(BitString other)
829 {
830     var threads = Environment.ProcessorCount / 2;
831     if (threads <= 1)
832     {
833         return VectorXor(other);
834     }
835     if (!Vector.IsHardwareAccelerated)
836     {
837         return ParallelXor(other);
838     }
839     var step = Vector<long>.Count;
840     if (_array.Length < (step * threads))
841     {
842         return VectorXor(other);
843     }
844     EnsureBitStringHasTheSameSize(other, nameof(other));
845     GetCommonOuterBorders(this, other, out int from, out int to);
846     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);

```

```

847         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
848             ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
849             ↪ step, range.Item1, range.Item2));
850         MarkBordersAsAllBitsSet();
851         TryShrinkBorders();
852         return this;
853     }
854
855     /// <summary>
856     /// <para>
857     /// Vectors the xor loop using the specified array.
858     /// </para>
859     /// </summary>
860     /// <param name="array">
861     /// <para>The array.</para>
862     /// </param>
863     /// <param name="otherArray">
864     /// <para>The other array.</para>
865     /// </param>
866     /// <param name="step">
867     /// <para>The step.</para>
868     /// </param>
869     /// <param name="start">
870     /// <para>The start.</para>
871     /// </param>
872     /// <param name="maximum">
873     /// <para>The maximum.</para>
874     /// </param>
875     [MethodImpl(MethodImplOptions.AggressiveInlining)]
876     static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
877     ↪ int maximum)
878     {
879         var i = start;
880         var range = maximum - start - 1;
881         var stop = range - (range % step);
882         for (; i < stop; i += step)
883         {
884             (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
885         }
886         for (; i < maximum; i++)
887         {
888             array[i] ^= otherArray[i];
889         }
890     }
891
892     [MethodImpl(MethodImplOptions.AggressiveInlining)]
893     private void RefreshBordersByWord(long wordIndex)
894     {
895         if (_array[wordIndex] == 0)
896         {
897             if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
898             {
899                 _minPositiveWord++;
900             }
901             if (wordIndex == _maxPositiveWord && wordIndex != 0)
902             {
903                 _maxPositiveWord--;
904             }
905         }
906         else
907         {
908             if (wordIndex < _minPositiveWord)
909             {
910                 _minPositiveWord = wordIndex;
911             }
912             if (wordIndex > _maxPositiveWord)
913             {
914                 _maxPositiveWord = wordIndex;
915             }
916         }
917     }
918 }
919
920 /// <summary>

```

```

922     /// <para>
923     /// Determines whether this instance try shrink borders.
924     /// </para>
925     /// <para></para>
926     /// </summary>
927     /// <returns>
928     /// <para>The borders updated.</para>
929     /// <para></para>
930     /// </returns>
931     [MethodImpl(MethodImplOptions.AggressiveInlining)]
932     public bool TryShrinkBorders()
933     {
934         GetBorders(out long from, out long to);
935         while (from <= to && _array[from] == 0)
936         {
937             from++;
938         }
939         if (from > to)
940         {
941             MarkBordersAsAllBitsReset();
942             return true;
943         }
944         while (to >= from && _array[to] == 0)
945         {
946             to--;
947         }
948         if (to < from)
949         {
950             MarkBordersAsAllBitsReset();
951             return true;
952         }
953         var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
954         if (bordersUpdated)
955         {
956             SetBorders(from, to);
957         }
958         return bordersUpdated;
959     }
960
961     /// <summary>
962     /// <para>
963     /// Determines whether this instance get.
964     /// </para>
965     /// <para></para>
966     /// </summary>
967     /// <param name="index">
968     /// <para>The index.</para>
969     /// <para></para>
970     /// </param>
971     /// <returns>
972     /// <para>The bool</para>
973     /// <para></para>
974     /// </returns>
975     [MethodImpl(MethodImplOptions.AggressiveInlining)]
976     public bool Get(long index)
977     {
978         Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
979         return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
980     }
981
982     /// <summary>
983     /// <para>
984     /// Sets the index.
985     /// </para>
986     /// <para></para>
987     /// </summary>
988     /// <param name="index">
989     /// <para>The index.</para>
990     /// <para></para>
991     /// </param>
992     /// <param name="value">
993     /// <para>The value.</para>
994     /// <para></para>
995     /// </param>
996     [MethodImpl(MethodImplOptions.AggressiveInlining)]
997     public void Set(long index, bool value)
998     {
999         if (value)

```

```

1000     {
1001         Set(index);
1002     }
1003     else
1004     {
1005         Reset(index);
1006     }
1007 }
1008
1009 /// <summary>
1010 /// <para>
1011 /// Sets the index.
1012 /// </para>
1013 /// <para></para>
1014 /// </summary>
1015 /// <param name="index">
1016 /// <para>The index.</para>
1017 /// <para></para>
1018 /// </param>
1019 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1020 public void Set(long index)
1021 {
1022     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
1023     var wordIndex = GetWordIndexFromIndex(index);
1024     var mask = GetBitMaskFromIndex(index);
1025     _array[wordIndex] |= mask;
1026     RefreshBordersByWord(wordIndex);
1027 }
1028
1029 /// <summary>
1030 /// <para>
1031 /// Resets the index.
1032 /// </para>
1033 /// <para></para>
1034 /// </summary>
1035 /// <param name="index">
1036 /// <para>The index.</para>
1037 /// <para></para>
1038 /// </param>
1039 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1040 public void Reset(long index)
1041 {
1042     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
1043     var wordIndex = GetWordIndexFromIndex(index);
1044     var mask = GetBitMaskFromIndex(index);
1045     _array[wordIndex] &= ~mask;
1046     RefreshBordersByWord(wordIndex);
1047 }
1048
1049 /// <summary>
1050 /// <para>
1051 /// Determines whether this instance add.
1052 /// </para>
1053 /// <para></para>
1054 /// </summary>
1055 /// <param name="index">
1056 /// <para>The index.</para>
1057 /// <para></para>
1058 /// </param>
1059 /// <returns>
1060 /// <para>The bool</para>
1061 /// <para></para>
1062 /// </returns>
1063 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1064 public bool Add(long index)
1065 {
1066     var wordIndex = GetWordIndexFromIndex(index);
1067     var mask = GetBitMaskFromIndex(index);
1068     if ((_array[wordIndex] & mask) == 0)
1069     {
1070         _array[wordIndex] |= mask;
1071         RefreshBordersByWord(wordIndex);
1072         return true;
1073     }
1074     else
1075     {
1076         return false;
1077     }

```

```

1078     }
1079
1080     /// <summary>
1081     /// <para>
1082     /// Sets the all using the specified value.
1083     /// </para>
1084     /// <para></para>
1085     /// </summary>
1086     /// <param name="value">
1087     /// <para>The value.</para>
1088     /// <para></para>
1089     /// </param>
1090     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1091     public void SetAll(bool value)
1092     {
1093         if (value)
1094         {
1095             SetAll();
1096         }
1097         else
1098         {
1099             ResetAll();
1100         }
1101     }
1102
1103     /// <summary>
1104     /// <para>
1105     /// Sets the all.
1106     /// </para>
1107     /// <para></para>
1108     /// </summary>
1109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1110     public void SetAll()
1111     {
1112         const long fillValue = unchecked((long)0xffffffffffffffff);
1113         var words = GetWordsCountFromIndex(_length);
1114         for (var i = 0; i < words; i++)
1115         {
1116             _array[i] = fillValue;
1117         }
1118         MarkBordersAsAllBitsSet();
1119     }
1120
1121     /// <summary>
1122     /// <para>
1123     /// Resets the all.
1124     /// </para>
1125     /// <para></para>
1126     /// </summary>
1127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1128     public void ResetAll()
1129     {
1130         const long fillValue = 0;
1131         GetBorders(out long from, out long to);
1132         for (var i = from; i <= to; i++)
1133         {
1134             _array[i] = fillValue;
1135         }
1136         MarkBordersAsAllBitsReset();
1137     }
1138
1139     /// <summary>
1140     /// <para>
1141     /// Gets the set indices.
1142     /// </para>
1143     /// <para></para>
1144     /// </summary>
1145     /// <returns>
1146     /// <para>The result.</para>
1147     /// <para></para>
1148     /// </returns>
1149     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1150     public List<long> GetSetIndices()
1151     {
1152         var result = new List<long>();
1153         GetBorders(out long from, out long to);
1154         for (var i = from; i <= to; i++)
1155         {

```

```

1156         var word = _array[i];
1157         if (word != 0)
1158         {
1159             AppendAllSetBitIndices(result, i, word);
1160         }
1161     }
1162     return result;
1163 }
1164
1165 /// <summary>
1166 /// <para>
1167 /// Gets the set u int 64 indices.
1168 /// </para>
1169 /// <para></para>
1170 /// </summary>
1171 /// <returns>
1172 /// <para>The result.</para>
1173 /// <para></para>
1174 /// </returns>
1175 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1176 public List<ulong> GetSetUInt64Indices()
1177 {
1178     var result = new List<ulong>();
1179     GetBorders(out ulong from, out ulong to);
1180     for (var i = from; i <= to; i++)
1181     {
1182         var word = _array[i];
1183         if (word != 0)
1184         {
1185             AppendAllSetBitIndices(result, i, word);
1186         }
1187     }
1188     return result;
1189 }
1190
1191 /// <summary>
1192 /// <para>
1193 /// Gets the first set bit index.
1194 /// </para>
1195 /// <para></para>
1196 /// </summary>
1197 /// <returns>
1198 /// <para>The long</para>
1199 /// <para></para>
1200 /// </returns>
1201 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1202 public long GetFirstSetBitIndex()
1203 {
1204     var i = _minPositiveWord;
1205     var word = _array[i];
1206     if (word != 0)
1207     {
1208         return GetFirstSetBitForWord(i, word);
1209     }
1210     return -1;
1211 }
1212
1213 /// <summary>
1214 /// <para>
1215 /// Gets the last set bit index.
1216 /// </para>
1217 /// <para></para>
1218 /// </summary>
1219 /// <returns>
1220 /// <para>The long</para>
1221 /// <para></para>
1222 /// </returns>
1223 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1224 public long GetLastSetBitIndex()
1225 {
1226     var i = _maxPositiveWord;
1227     var word = _array[i];
1228     if (word != 0)
1229     {
1230         return GetLastSetBitForWord(i, word);
1231     }
1232     return -1;
1233 }

```

```

1234
1235 /// <summary>
1236 /// <para>
1237 /// Counts the set bits.
1238 /// </para>
1239 /// <para></para>
1240 /// </summary>
1241 /// <returns>
1242 /// <para>The total.</para>
1243 /// <para></para>
1244 /// </returns>
1245 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1246 public long CountSetBits()
1247 {
1248     var total = 0L;
1249     GetBorders(out long from, out long to);
1250     for (var i = from; i <= to; i++)
1251     {
1252         var word = _array[i];
1253         if (word != 0)
1254         {
1255             total += CountSetBitsForWord(word);
1256         }
1257     }
1258     return total;
1259 }
1260
1261 /// <summary>
1262 /// <para>
1263 /// Determines whether this instance have common bits.
1264 /// </para>
1265 /// <para></para>
1266 /// </summary>
1267 /// <param name="other">
1268 /// <para>The other.</para>
1269 /// <para></para>
1270 /// </param>
1271 /// <returns>
1272 /// <para>The bool</para>
1273 /// <para></para>
1274 /// </returns>
1275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1276 public bool HaveCommonBits(BitString other)
1277 {
1278     EnsureBitStringHasTheSameSize(other, nameof(other));
1279     GetCommonInnerBorders(this, other, out long from, out long to);
1280     var otherArray = other._array;
1281     for (var i = from; i <= to; i++)
1282     {
1283         var left = _array[i];
1284         var right = otherArray[i];
1285         if (left != 0 && right != 0 && (left & right) != 0)
1286         {
1287             return true;
1288         }
1289     }
1290     return false;
1291 }
1292
1293 /// <summary>
1294 /// <para>
1295 /// Counts the common bits using the specified other.
1296 /// </para>
1297 /// <para></para>
1298 /// </summary>
1299 /// <param name="other">
1300 /// <para>The other.</para>
1301 /// <para></para>
1302 /// </param>
1303 /// <returns>
1304 /// <para>The total.</para>
1305 /// <para></para>
1306 /// </returns>
1307 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1308 public long CountCommonBits(BitString other)
1309 {
1310     EnsureBitStringHasTheSameSize(other, nameof(other));
1311     GetCommonInnerBorders(this, other, out long from, out long to);

```



```

1312     var total = 0L;
1313     var otherArray = other._array;
1314     for (var i = from; i <= to; i++)
1315     {
1316         var left = _array[i];
1317         var right = otherArray[i];
1318         var combined = left & right;
1319         if (combined != 0)
1320         {
1321             total += CountSetBitsForWord(combined);
1322         }
1323     }
1324     return total;
1325 }
1326
1327 /// <summary>
1328 /// <para>
1329 /// Gets the common indices using the specified other.
1330 /// </para>
1331 /// <para></para>
1332 /// </summary>
1333 /// <param name="other">
1334 /// <para>The other.</para>
1335 /// <para></para>
1336 /// </param>
1337 /// <returns>
1338 /// <para>The result.</para>
1339 /// <para></para>
1340 /// </returns>
1341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1342 public List<long> GetCommonIndices(BitString other)
1343 {
1344     EnsureBitStringHasTheSameSize(other, nameof(other));
1345     GetCommonInnerBorders(this, other, out long from, out long to);
1346     var result = new List<long>();
1347     var otherArray = other._array;
1348     for (var i = from; i <= to; i++)
1349     {
1350         var left = _array[i];
1351         var right = otherArray[i];
1352         var combined = left & right;
1353         if (combined != 0)
1354         {
1355             AppendAllSetBitIndices(result, i, combined);
1356         }
1357     }
1358     return result;
1359 }
1360
1361 /// <summary>
1362 /// <para>
1363 /// Gets the common u int 64 indices using the specified other.
1364 /// </para>
1365 /// <para></para>
1366 /// </summary>
1367 /// <param name="other">
1368 /// <para>The other.</para>
1369 /// <para></para>
1370 /// </param>
1371 /// <returns>
1372 /// <para>The result.</para>
1373 /// <para></para>
1374 /// </returns>
1375 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1376 public List<ulong> GetCommonUInt64Indices(BitString other)
1377 {
1378     EnsureBitStringHasTheSameSize(other, nameof(other));
1379     GetCommonBorders(this, other, out ulong from, out ulong to);
1380     var result = new List<ulong>();
1381     var otherArray = other._array;
1382     for (var i = from; i <= to; i++)
1383     {
1384         var left = _array[i];
1385         var right = otherArray[i];
1386         var combined = left & right;
1387         if (combined != 0)
1388         {
1389             AppendAllSetBitIndices(result, i, combined);

```

```

1390     }
1391 }
1392 return result;
1393 }
1394
1395 /// <summary>
1396 /// <para>
1397 /// Gets the first common bit index using the specified other.
1398 /// </para>
1399 /// <para></para>
1400 /// </summary>
1401 /// <param name="other">
1402 /// <para>The other.</para>
1403 /// <para></para>
1404 /// </param>
1405 /// <returns>
1406 /// <para>The long</para>
1407 /// <para></para>
1408 /// </returns>
1409 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1410 public long GetFirstCommonBitIndex(BitString other)
1411 {
1412     EnsureBitStringHasTheSameSize(other, nameof(other));
1413     GetCommonInnerBorders(this, other, out long from, out long to);
1414     var otherArray = other._array;
1415     for (var i = from; i <= to; i++)
1416     {
1417         var left = _array[i];
1418         var right = otherArray[i];
1419         var combined = left & right;
1420         if (combined != 0)
1421         {
1422             return GetFirstSetBitForWord(i, combined);
1423         }
1424     }
1425     return -1;
1426 }
1427
1428 /// <summary>
1429 /// <para>
1430 /// Gets the last common bit index using the specified other.
1431 /// </para>
1432 /// <para></para>
1433 /// </summary>
1434 /// <param name="other">
1435 /// <para>The other.</para>
1436 /// <para></para>
1437 /// </param>
1438 /// <returns>
1439 /// <para>The long</para>
1440 /// <para></para>
1441 /// </returns>
1442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1443 public long GetLastCommonBitIndex(BitString other)
1444 {
1445     EnsureBitStringHasTheSameSize(other, nameof(other));
1446     GetCommonInnerBorders(this, other, out long from, out long to);
1447     var otherArray = other._array;
1448     for (var i = to; i >= from; i--)
1449     {
1450         var left = _array[i];
1451         var right = otherArray[i];
1452         var combined = left & right;
1453         if (combined != 0)
1454         {
1455             return GetLastSetBitForWord(i, combined);
1456         }
1457     }
1458     return -1;
1459 }
1460
1461 /// <summary>
1462 /// <para>
1463 /// Determines whether this instance equals.
1464 /// </para>
1465 /// <para></para>
1466 /// </summary>
1467 /// <param name="obj">

```

```

1468     /// <para>The obj.</para>
1469     /// <para></para>
1470     /// </param>
1471     /// <returns>
1472     /// <para>The bool</para>
1473     /// <para></para>
1474     /// </returns>
1475     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1476     public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
        => false;

1477
1478     /// <summary>
1479     /// <para>
1480     /// Determines whether this instance equals.
1481     /// </para>
1482     /// <para></para>
1483     /// </summary>
1484     /// <param name="other">
1485     /// <para>The other.</para>
1486     /// <para></para>
1487     /// </param>
1488     /// <returns>
1489     /// <para>The bool</para>
1490     /// <para></para>
1491     /// </returns>
1492     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1493     public bool Equals(BitString other)
1494     {
1495         if (_length != other._length)
1496         {
1497             return false;
1498         }
1499         var otherArray = other._array;
1500         if (_array.Length != otherArray.Length)
1501         {
1502             return false;
1503         }
1504         if (_minPositiveWord != other._minPositiveWord)
1505         {
1506             return false;
1507         }
1508         if (_maxPositiveWord != other._maxPositiveWord)
1509         {
1510             return false;
1511         }
1512         GetCommonBorders(this, other, out ulong from, out ulong to);
1513         for (var i = from; i <= to; i++)
1514         {
1515             if (_array[i] != otherArray[i])
1516             {
1517                 return false;
1518             }
1519         }
1520         return true;
1521     }
1522     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1523     private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
1524     {
1525         Ensure.Always.ArgumentNotNull(other, argumentName);
1526         if (_length != other._length)
1527         {
1528             throw new ArgumentException("Bit string must be the same size.", argumentName);
1529         }
1530     }
1531     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1532     private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
1533     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1534     private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
1535     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1536     private void GetBorders(out long from, out long to)
1537     {
1538         from = _minPositiveWord;
1539         to = _maxPositiveWord;
1540     }
1541     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1542     private void GetBorders(out ulong from, out ulong to)
1543     {
1544         from = (ulong)_minPositiveWord;

```

```

1545         to = (ulong)_maxPositiveWord;
1546     }
1547 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1548     private void SetBorders(long from, long to)
1549     {
1550         _minPositiveWord = from;
1551         _maxPositiveWord = to;
1552     }
1553 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1554     private Range<long> GetValidIndexRange() => (0, _length - 1);
1555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1556     private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
1557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1558     private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
1559         ↪ wordValue)
1560     {
1561         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
1562             ↪ bits32to47, out byte[] bits48to63);
1563         AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
1564             ↪ bits48to63);
1565     }
1566 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1567     private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
1568         ↪ wordValue)
1569     {
1570         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
1571             ↪ bits32to47, out byte[] bits48to63);
1572         AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
1573             ↪ bits48to63);
1574     }
1575 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1576     private static long CountSetBitsForWord(long word)
1577     {
1578         GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
1579             ↪ out byte[] bits48to63);
1580         return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
1581             ↪ bits48to63.LongLength;
1582     }
1583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1584     private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
1585     {
1586         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
1587             ↪ bits32to47, out byte[] bits48to63);
1588         return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
1589     }
1590 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1591     private static long GetLastSetBitForWord(long wordIndex, long wordValue)
1592     {
1593         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
1594             ↪ bits32to47, out byte[] bits48to63);
1595         return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
1596     }
1597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1598     private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
1599         ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1600     {
1601         for (var j = 0; j < bits00to15.Length; j++)
1602         {
1603             result.Add(bits00to15[j] + (i * 64));
1604         }
1605         for (var j = 0; j < bits16to31.Length; j++)
1606         {
1607             result.Add(bits16to31[j] + 16 + (i * 64));
1608         }
1609         for (var j = 0; j < bits32to47.Length; j++)
1610         {
1611             result.Add(bits32to47[j] + 32 + (i * 64));
1612         }
1613         for (var j = 0; j < bits48to63.Length; j++)
1614         {
1615             result.Add(bits48to63[j] + 48 + (i * 64));
1616         }
1617     }
1618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1619     private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1620         ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1621     {

```

```

1610         for (var j = 0; j < bits00to15.Length; j++)
1611         {
1612             result.Add(bits00to15[j] + (i * 64));
1613         }
1614         for (var j = 0; j < bits16to31.Length; j++)
1615         {
1616             result.Add(bits16to31[j] + 16UL + (i * 64));
1617         }
1618         for (var j = 0; j < bits32to47.Length; j++)
1619         {
1620             result.Add(bits32to47[j] + 32UL + (i * 64));
1621         }
1622         for (var j = 0; j < bits48to63.Length; j++)
1623         {
1624             result.Add(bits48to63[j] + 48UL + (i * 64));
1625         }
1626     }
1627 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1628     private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1629     ↪ bits32to47, byte[] bits48to63)
1630     {
1631         if (bits00to15.Length > 0)
1632         {
1633             return bits00to15[0] + (i * 64);
1634         }
1635         if (bits16to31.Length > 0)
1636         {
1637             return bits16to31[0] + 16 + (i * 64);
1638         }
1639         if (bits32to47.Length > 0)
1640         {
1641             return bits32to47[0] + 32 + (i * 64);
1642         }
1643         return bits48to63[0] + 48 + (i * 64);
1644     }
1645 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1646     private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1647     ↪ bits32to47, byte[] bits48to63)
1648     {
1649         if (bits48to63.Length > 0)
1650         {
1651             return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1652         }
1653         if (bits32to47.Length > 0)
1654         {
1655             return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1656         }
1657         if (bits16to31.Length > 0)
1658         {
1659             return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1660         }
1661         return bits00to15[bits00to15.Length - 1] + (i * 64);
1662     }
1663 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1664     private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
1665     ↪ byte[] bits32to47, out byte[] bits48to63)
1666     {
1667         bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1668         bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
1669         bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1670         bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1671     }
1672
1673     /// <summary>
1674     /// <para>
1675     /// Gets the common inner borders using the specified left.
1676     /// </para>
1677     /// <para></para>
1678     /// </summary>
1679     /// <param name="left">
1680     /// <para>The left.</para>
1681     /// <para></para>
1682     /// </param>
1683     /// <param name="right">
1684     /// <para>The right.</para>
1685     /// <para></para>
1686     /// </param>

```

```

1684    /// <param name="from">
1685    /// <para>The from.</para>
1686    /// <para></para>
1687    /// </param>
1688    /// <param name="to">
1689    /// <para>The to.</para>
1690    /// <para></para>
1691    /// </param>
1692    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1693    public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1694    {
1695        from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1696        to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1697    }
1698
1699    /// <summary>
1700    /// <para>
1701    /// Gets the common outer borders using the specified left.
1702    /// </para>
1703    /// <para></para>
1704    /// </summary>
1705    /// <param name="left">
1706    /// <para>The left.</para>
1707    /// <para></para>
1708    /// </param>
1709    /// <param name="right">
1710    /// <para>The right.</para>
1711    /// <para></para>
1712    /// </param>
1713    /// <param name="from">
1714    /// <para>The from.</para>
1715    /// <para></para>
1716    /// </param>
1717    /// <param name="to">
1718    /// <para>The to.</para>
1719    /// <para></para>
1720    /// </param>
1721    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1722    public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1723    {
1724        from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1725        to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1726    }
1727
1728    /// <summary>
1729    /// <para>
1730    /// Gets the common outer borders using the specified left.
1731    /// </para>
1732    /// <para></para>
1733    /// </summary>
1734    /// <param name="left">
1735    /// <para>The left.</para>
1736    /// <para></para>
1737    /// </param>
1738    /// <param name="right">
1739    /// <para>The right.</para>
1740    /// <para></para>
1741    /// </param>
1742    /// <param name="from">
1743    /// <para>The from.</para>
1744    /// <para></para>
1745    /// </param>
1746    /// <param name="to">
1747    /// <para>The to.</para>
1748    /// <para></para>
1749    /// </param>
1750    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1751    public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
    ↪ out int to)
1752    {
1753        from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1754        to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1755    }
1756
1757    /// <summary>
1758    /// <para>

```

```

1759     /// Gets the common borders using the specified left.
1760     /// </para>
1761     /// <para></para>
1762     /// </summary>
1763     /// <param name="left">
1764     /// <para>The left.</para>
1765     /// <para></para>
1766     /// </param>
1767     /// <param name="right">
1768     /// <para>The right.</para>
1769     /// <para></para>
1770     /// </param>
1771     /// <param name="from">
1772     /// <para>The from.</para>
1773     /// <para></para>
1774     /// </param>
1775     /// <param name="to">
1776     /// <para>The to.</para>
1777     /// <para></para>
1778     /// </param>
1779     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1780     public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
        → ulong to)
1781     {
1782         from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1783         to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1784     }
1785
1786     /// <summary>
1787     /// <para>
1788     /// Gets the words count from index using the specified index.
1789     /// </para>
1790     /// <para></para>
1791     /// </summary>
1792     /// <param name="index">
1793     /// <para>The index.</para>
1794     /// <para></para>
1795     /// </param>
1796     /// <returns>
1797     /// <para>The long</para>
1798     /// <para></para>
1799     /// </returns>
1800     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1801     public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1802
1803     /// <summary>
1804     /// <para>
1805     /// Gets the word index from index using the specified index.
1806     /// </para>
1807     /// <para></para>
1808     /// </summary>
1809     /// <param name="index">
1810     /// <para>The index.</para>
1811     /// <para></para>
1812     /// </param>
1813     /// <returns>
1814     /// <para>The long</para>
1815     /// <para></para>
1816     /// </returns>
1817     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1818     public static long GetWordIndexFromIndex(long index) => index >> 6;
1819
1820     /// <summary>
1821     /// <para>
1822     /// Gets the bit mask from index using the specified index.
1823     /// </para>
1824     /// <para></para>
1825     /// </summary>
1826     /// <param name="index">
1827     /// <para>The index.</para>
1828     /// <para></para>
1829     /// </param>
1830     /// <returns>
1831     /// <para>The long</para>
1832     /// <para></para>
1833     /// </returns>
1834     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1835     public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);

```

```

1836
1837     /// <summary>
1838     /// <para>
1839     /// Gets the hash code.
1840     /// </para>
1841     /// <para></para>
1842     /// </summary>
1843     /// <returns>
1844     /// <para>The int</para>
1845     /// <para></para>
1846     /// </returns>
1847     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1848     public override int GetHashCode() => base.GetHashCode();
1849
1850     /// <summary>
1851     /// <para>
1852     /// Returns the string.
1853     /// </para>
1854     /// <para></para>
1855     /// </summary>
1856     /// <returns>
1857     /// <para>The string</para>
1858     /// <para></para>
1859     /// </returns>
1860     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1861     public override string ToString() => base.ToString();
1862 }
1863 }

```

1.9 ./csharp/Platform.Collections/BitStringExtensions.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Random;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the bit string extensions.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public static class BitStringExtensions
15    {
16        /// <summary>
17        /// <para>
18        /// Sets the random bits using the specified string.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        /// <param name="@string">
23        /// <para>The string.</para>
24        /// <para></para>
25        /// </param>
26        [MethodImpl(MethodImplOptions.AggressiveInlining)]
27        public static void SetRandomBits(this BitString @string)
28        {
29            for (var i = 0; i < @string.Length; i++)
30            {
31                var value = RandomHelpers.Default.NextBoolean();
32                @string.Set(i, value);
33            }
34        }
35    }
36 }

```

1.10 ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs

```

1 using System.Collections.Concurrent;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Concurrent
8 {
9     /// <summary>
10    /// <para>

```



```

11  /// Represents the concurrent queue extensions.
12  /// </para>
13  /// <para></para>
14  /// </summary>
15  public static class ConcurrentQueueExtensions
16  {
17      /// <summary>
18      /// <para>
19      /// Dequeues the all using the specified queue.
20      /// </para>
21      /// <para></para>
22      /// </summary>
23      /// <typeparam name="T">
24      /// <para>The .</para>
25      /// <para></para>
26      /// </typeparam>
27      /// <param name="queue">
28      /// <para>The queue.</para>
29      /// <para></para>
30      /// </param>
31      /// <returns>
32      /// <para>An enumerable of t</para>
33      /// <para></para>
34      /// </returns>
35      [MethodImpl(MethodImplOptions.AggressiveInlining)]
36      public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
37      {
38          while (queue.TryDequeue(out T item))
39          {
40              yield return item;
41          }
42      }
43  }
44  }

```

1.11 ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs

```

1  using System.Collections.Concurrent;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Concurrent
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the concurrent stack extensions.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class ConcurrentStackExtensions
15     {
16         /// <summary>
17         /// <para>
18         /// Pops the or default using the specified stack.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <typeparam name="T">
23         /// <para>The .</para>
24         /// <para></para>
25         /// </typeparam>
26         /// <param name="stack">
27         /// <para>The stack.</para>
28         /// <para></para>
29         /// </param>
30         /// <returns>
31         /// <para>The</para>
32         /// <para></para>
33         /// </returns>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
            ↪ value) ? value : default;
36
37         /// <summary>
38         /// <para>
39         /// Peeks the or default using the specified stack.
40         /// </para>
41         /// <para></para>

```

```

42     /// </summary>
43     /// <typeparam name="T">
44     /// <para>The .</para>
45     /// <para></para>
46     /// </typeparam>
47     /// <param name="stack">
48     /// <para>The stack.</para>
49     /// <para></para>
50     /// </param>
51     /// <returns>
52     /// <para>The</para>
53     /// <para></para>
54     /// </returns>
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
        ↪ value) ? value : default;
57 }
58 }

```

1.12 ./csharp/Platform.Collections/EnsureExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Runtime.CompilerServices;
5  using Platform.Exceptions;
6  using Platform.Exceptions.ExtensionRoots;
7
8  #pragma warning disable IDE0060 // Remove unused parameter
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the ensure extensions.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     public static class EnsureExtensions
20     {
21         #region Always
22
23         /// <summary>
24         /// <para>
25         /// Arguments the not empty using the specified root.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         /// <typeparam name="T">
30         /// <para>The .</para>
31         /// <para></para>
32         /// </typeparam>
33         /// <param name="root">
34         /// <para>The root.</para>
35         /// <para></para>
36         /// </param>
37         /// <param name="argument">
38         /// <para>The argument.</para>
39         /// <para></para>
40         /// </param>
41         /// <param name="argumentName">
42         /// <para>The argument name.</para>
43         /// <para></para>
44         /// </param>
45         /// <param name="message">
46         /// <para>The message.</para>
47         /// <para></para>
48         /// </param>
49         /// <exception cref="ArgumentException">
50         /// <para></para>
51         /// <para></para>
52         /// </exception>
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
        ↪ ICollection<T> argument, string argumentName, string message)
55         {
56             if (argument.IsNullOrEmpty())
57             {

```

```

58         throw new ArgumentException(message, argumentName);
59     }
60 }
61
62 /// <summary>
63 /// <para>
64 /// Arguments the not empty using the specified root.
65 /// </para>
66 /// <para></para>
67 /// </summary>
68 /// <typeparam name="T">
69 /// <para>The .</para>
70 /// <para></para>
71 /// </typeparam>
72 /// <param name="root">
73 /// <para>The root.</para>
74 /// <para></para>
75 /// </param>
76 /// <param name="argument">
77 /// <para>The argument.</para>
78 /// <para></para>
79 /// </param>
80 /// <param name="argumentName">
81 /// <para>The argument name.</para>
82 /// <para></para>
83 /// </param>
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
    ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
    ↪ argumentName, null);
86
87 /// <summary>
88 /// <para>
89 /// Arguments the not empty using the specified root.
90 /// </para>
91 /// <para></para>
92 /// </summary>
93 /// <typeparam name="T">
94 /// <para>The .</para>
95 /// <para></para>
96 /// </typeparam>
97 /// <param name="root">
98 /// <para>The root.</para>
99 /// <para></para>
100 /// </param>
101 /// <param name="argument">
102 /// <para>The argument.</para>
103 /// <para></para>
104 /// </param>
105 [MethodImpl(MethodImplOptions.AggressiveInlining)]
106 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
    ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
107
108 /// <summary>
109 /// <para>
110 /// Arguments the not empty and not white space using the specified root.
111 /// </para>
112 /// <para></para>
113 /// </summary>
114 /// <param name="root">
115 /// <para>The root.</para>
116 /// <para></para>
117 /// </param>
118 /// <param name="argument">
119 /// <para>The argument.</para>
120 /// <para></para>
121 /// </param>
122 /// <param name="argumentName">
123 /// <para>The argument name.</para>
124 /// <para></para>
125 /// </param>
126 /// <param name="message">
127 /// <para>The message.</para>
128 /// <para></para>
129 /// </param>
130 /// <exception cref="ArgumentException">
131 /// <para></para>
132 /// <para></para>

```

```

133     /// </exception>
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
136     ↪     string argument, string argumentName, string message)
137     {
138         if (string.IsNullOrEmpty(argument))
139         {
140             throw new ArgumentException(message, argumentName);
141         }
142     }
143     /// <summary>
144     /// <para>
145     /// Arguments the not empty and not white space using the specified root.
146     /// </para>
147     /// <para></para>
148     /// </summary>
149     /// <param name="root">
150     /// <para>The root.</para>
151     /// <para></para>
152     /// </param>
153     /// <param name="argument">
154     /// <para>The argument.</para>
155     /// <para></para>
156     /// </param>
157     /// <param name="argumentName">
158     /// <para>The argument name.</para>
159     /// <para></para>
160     /// </param>
161     [MethodImpl(MethodImplOptions.AggressiveInlining)]
162     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
163     ↪     string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
164     ↪     argument, argumentName, null);
165     /// <summary>
166     /// <para>
167     /// Arguments the not empty and not white space using the specified root.
168     /// </para>
169     /// <para></para>
170     /// </summary>
171     /// <param name="root">
172     /// <para>The root.</para>
173     /// <para></para>
174     /// </param>
175     /// <param name="argument">
176     /// <para>The argument.</para>
177     /// <para></para>
178     /// </param>
179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
180     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
181     ↪     string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
182
183     #endregion
184
185     #region OnDebug
186     /// <summary>
187     /// <para>
188     /// Arguments the not empty using the specified root.
189     /// </para>
190     /// <para></para>
191     /// </summary>
192     /// <typeparam name="T">
193     /// <para>The .</para>
194     /// <para></para>
195     /// </typeparam>
196     /// <param name="root">
197     /// <para>The root.</para>
198     /// <para></para>
199     /// </param>
200     /// <param name="argument">
201     /// <para>The argument.</para>
202     /// <para></para>
203     /// </param>
204     /// <param name="argumentName">
205     /// <para>The argument name.</para>
206     /// <para></para>
207     /// </param>

```

```

207     /// <param name="message">
208     /// <para>The message.</para>
209     /// <para></para>
210     /// </param>
211     [Conditional("DEBUG")]
212     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
213     ↪ ICollection<T> argument, string argumentName, string message) =>
214     ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
215
216     /// <summary>
217     /// <para>
218     /// Arguments the not empty using the specified root.
219     /// </para>
220     /// <para></para>
221     /// </summary>
222     /// <typeparam name="T">
223     /// <para>The .</para>
224     /// <para></para>
225     /// </typeparam>
226     /// <param name="root">
227     /// <para>The root.</para>
228     /// <para></para>
229     /// </param>
230     /// <param name="argument">
231     /// <para>The argument.</para>
232     /// <para></para>
233     /// </param>
234     /// <param name="argumentName">
235     /// <para>The argument name.</para>
236     /// <para></para>
237     /// </param>
238     [Conditional("DEBUG")]
239     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
240     ↪ ICollection<T> argument, string argumentName) =>
241     ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
242
243     /// <summary>
244     /// <para>
245     /// Arguments the not empty using the specified root.
246     /// </para>
247     /// <para></para>
248     /// </summary>
249     /// <typeparam name="T">
250     /// <para>The .</para>
251     /// <para></para>
252     /// </typeparam>
253     /// <param name="root">
254     /// <para>The root.</para>
255     /// <para></para>
256     /// </param>
257     /// <param name="argument">
258     /// <para>The argument.</para>
259     /// <para></para>
260     /// </param>
261     [Conditional("DEBUG")]
262     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
263     ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
264
265     /// <summary>
266     /// <para>
267     /// Arguments the not empty and not white space using the specified root.
268     /// </para>
269     /// <para></para>
270     /// </summary>
271     /// <param name="root">
272     /// <para>The root.</para>
273     /// <para></para>
274     /// </param>
275     /// <param name="argument">
276     /// <para>The argument.</para>
277     /// <para></para>
278     /// </param>
279     /// <param name="argumentName">
280     /// <para>The argument name.</para>
281     /// <para></para>
282     /// </param>
283     /// <param name="message">

```

```

279     /// <para>The message.</para>
280     /// <para></para>
281     /// </param>
282     [Conditional("DEBUG")]
283     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ⇒ root, string argument, string argumentName, string message) =>
        ⇒ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);

284
285     /// <summary>
286     /// <para>
287     /// Arguments the not empty and not white space using the specified root.
288     /// </para>
289     /// <para></para>
290     /// </summary>
291     /// <param name="root">
292     /// <para>The root.</para>
293     /// <para></para>
294     /// </param>
295     /// <param name="argument">
296     /// <para>The argument.</para>
297     /// <para></para>
298     /// </param>
299     /// <param name="argumentName">
300     /// <para>The argument name.</para>
301     /// <para></para>
302     /// </param>
303     [Conditional("DEBUG")]
304     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ⇒ root, string argument, string argumentName) =>
        ⇒ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);

305
306     /// <summary>
307     /// <para>
308     /// Arguments the not empty and not white space using the specified root.
309     /// </para>
310     /// <para></para>
311     /// </summary>
312     /// <param name="root">
313     /// <para>The root.</para>
314     /// <para></para>
315     /// </param>
316     /// <param name="argument">
317     /// <para>The argument.</para>
318     /// <para></para>
319     /// </param>
320     [Conditional("DEBUG")]
321     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ⇒ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
        ⇒ null, null);

322
323     #endregion
324 }
325 }

```

1.13 ./csharp/Platform.Collections/ICollectionExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections
8  {
9      /// <summary>
10     /// <para>Presents a set of methods for working with collections.</para>
11     /// <para>Представляет набор методов для работы с коллекциями.</para>
12     /// </summary>
13     public static class ICollectionExtensions
14     {
15         /// <summary>
16         /// <para>Checking collection for empty.</para>
17         /// <para>Проверяет коллекцию на пустоту.</para>
18         /// </summary>
19         /// <param name="collection">
20         /// <para>Method takes an elements collection of <see cref="ICollection<T>" />
21         ⇒ type.</para>
22         /// <para>Метода принимает коллекцию элементов <see cref="ICollection<T>" /> типа.</para>
23         /// </param>

```

```

23     /// <returns>
24     /// <para>Returns a <see cref="bool"/> type variable equal to False if the collection is
    → empty else returns true.</para>
25     /// <para>Возвращает переменную типа <see cref="bool"/> равной false если коллекция
    → пустая иначе возвращает true.</para>
26     /// </returns>
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
    → null || collection.Count == 0;
29
30     /// <summary>
31     /// <para>
32     /// Determines whether all equal to default.
33     /// </para>
34     /// <para></para>
35     /// </summary>
36     /// <typeparam name="T">
37     /// <para>The .</para>
38     /// <para></para>
39     /// </typeparam>
40     /// <param name="collection">
41     /// <para>The collection.</para>
42     /// <para></para>
43     /// </param>
44     /// <returns>
45     /// <para>The bool</para>
46     /// <para></para>
47     /// </returns>
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public static bool AllEqualToDefault<T>(this ICollection<T> collection)
50     {
51         var equalityComparer = EqualityComparer<T>.Default;
52         return collection.All(item => equalityComparer.Equals(item, default));
53     }
54 }
55 }

```

1.14 ./csharp/Platform.Collections/IDictionaryExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the dictionary extensions.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     public static class IDictionaryExtensions
16     {
17         /// <summary>
18         /// <para>
19         /// Gets the or default using the specified dictionary.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <typeparam name="TKey">
24         /// <para>The key.</para>
25         /// <para></para>
26         /// </typeparam>
27         /// <typeparam name="TValue">
28         /// <para>The value.</para>
29         /// <para></para>
30         /// </typeparam>
31         /// <param name="dictionary">
32         /// <para>The dictionary.</para>
33         /// <para></para>
34         /// </param>
35         /// <param name="key">
36         /// <para>The key.</para>
37         /// <para></para>
38         /// </param>
39         /// <returns>
40         /// <para>The value.</para>

```

```

41     /// <para></para>
42     /// </returns>
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     public static TValue GetOrCreateDefault<TKey, TValue>(this IDictionary<TKey, TValue>
    ↪ dictionary, TKey key)
45     {
46         dictionary.TryGetValue(key, out TValue value);
47         return value;
48     }
49
50     /// <summary>
51     /// <para>
52     /// Gets the or add using the specified dictionary.
53     /// </para>
54     /// <para></para>
55     /// </summary>
56     /// <typeparam name="TKey">
57     /// <para>The key.</para>
58     /// <para></para>
59     /// </typeparam>
60     /// <typeparam name="TValue">
61     /// <para>The value.</para>
62     /// <para></para>
63     /// </typeparam>
64     /// <param name="dictionary">
65     /// <para>The dictionary.</para>
66     /// <para></para>
67     /// </param>
68     /// <param name="key">
69     /// <para>The key.</para>
70     /// <para></para>
71     /// </param>
72     /// <param name="valueFactory">
73     /// <para>The value factory.</para>
74     /// <para></para>
75     /// </param>
76     /// <returns>
77     /// <para>The value.</para>
78     /// <para></para>
79     /// </returns>
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static TValue GetOrCreateAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
    ↪ TKey key, Func<TKey, TValue> valueFactory)
82     {
83         if (!dictionary.TryGetValue(key, out TValue value))
84         {
85             value = valueFactory(key);
86             dictionary.Add(key, value);
87             return value;
88         }
89         return value;
90     }
91 }
92 }

```

1.15 ./csharp/Platform.Collections/Lists/CharIListExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the char list extensions.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public static class CharIListExtensions
13     {
14         /// <summary>
15         /// <para>Generates a hash code for the entire list based on the values of its
    ↪ elements.</para>
16         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
17         /// </summary>
18         /// <param name="list"><para>The list to be hashed.</para><para>Список для
    ↪ хеширования.</para></param>
19         /// <returns>
20         /// <para>The hash code of the list.</para>

```



```

21     /// <para>Хэш-код списка.</para>
22     /// </returns>
23     /// <remarks>
24     /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    → a3eda37d3d4cd10/mscorlib/system/string.cs#L833
25     /// </remarks>
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public static int GenerateHashCode(this IList<char> list)
28     {
29         var hashSeed = 5381;
30         var hashAccumulator = hashSeed;
31         for (var i = 0; i < list.Count; i++)
32         {
33             hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
34         }
35         return hashAccumulator + (hashSeed * 1566083941);
36     }
37
38     /// <summary>
39     /// <para>Compares two lists for equality.</para>
40     /// <para>Сравнивает два списка на равенство.</para>
41     /// </summary>
42     /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>
43     /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
44     /// <returns>
45     /// <para>True, if the passed lists are equal to each other otherwise false.</para>
46     /// <para>True, если переданные списки равны друг другу, иначе false.</para>
47     /// </returns>
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public static bool EqualTo(this IList<char> left, IList<char> right) =>
    → left.EqualTo(right, ContentEqualTo);
50
51     /// <summary>
52     /// <para>Compares each element in the list for equality.</para>
53     /// <para>Сравнивает на равенство каждый элемент списка.</para>
54     /// </summary>
55     /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>
56     /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
57     /// <returns>
58     /// <para>If at least one element of one list is not equal to the corresponding element
    → from another list returns false, otherwise - true.</para>
59     /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
    → из другого списка возвращает false, иначе - true.</para>
60     /// </returns>
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public static bool ContentEqualTo(this IList<char> left, IList<char> right)
63     {
64         for (var i = left.Count - 1; i >= 0; --i)
65         {
66             if (left[i] != right[i])
67             {
68                 return false;
69             }
70         }
71         return true;
72     }
73 }
74 }

```

1.16 ./csharp/Platform.Collections/Lists/IListComparer.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Lists
5 {
6     /// <summary>
7     /// <para>
8     /// Represents the list comparer.
9     /// </para>
10    /// <para></para>
11    /// </summary>
12    /// <seealso cref="IComparer{IList{T}}">
13    public class IListComparer<T> : IComparer<IList<T>>
14    {

```

```

15     /// <summary>
16     /// <para>Compares two lists.</para>
17     /// <para>Сравнивает два списка.</para>
18     /// </summary>
19     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
20     /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>
21     /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
22     /// <returns>
23     /// <para>
24     ///     A signed integer that indicates the relative values of <paramref name="left" />
    → and <paramref name="right" /> lists' elements, as shown in the following table.
25     ///     <list type="table">
26     ///         <listheader>
27     ///             <term>Value</term>
28     ///             <description>Meaning</description>
29     ///         </listheader>
30     ///         <item>
31     ///             <term>Is less than zero</term>
32     ///             <description>First non equal element of <paramref name="left" /> list is
    → less than first not equal element of <paramref name="right" /> list.</description>
33     ///         </item>
34     ///         <item>
35     ///             <term>Zero</term>
36     ///             <description>All elements of <paramref name="left" /> list equals to all
    → elements of <paramref name="right" /> list.</description>
37     ///         </item>
38     ///         <item>
39     ///             <term>Is greater than zero</term>
40     ///             <description>First non equal element of <paramref name="left" /> list is
    → greater than first not equal element of <paramref name="right" /> list.</description>
41     ///         </item>
42     ///     </list>
43     /// </para>
44     /// <para>
45     ///     Целое число со знаком, которое указывает относительные значения элементов
    → списков <paramref name="left" /> и <paramref name="right" /> как показано в
    → следующей таблице.
46     ///     <list type="table">
47     ///         <listheader>
48     ///             <term>Значение</term>
49     ///             <description>Смысл</description>
50     ///         </listheader>
51     ///         <item>
52     ///             <term>Меньше нуля</term>
53     ///             <description>Первый не равный элемент <paramref name="left" /> списка
    → меньше первого неравного элемента <paramref name="right" /> списка.</description>
54     ///         </item>
55     ///         <item>
56     ///             <term>Ноль</term>
57     ///             <description>Все элементы <paramref name="left" /> списка равны всем
    → элементам <paramref name="right" /> списка.</description>
58     ///         </item>
59     ///         <item>
60     ///             <term>Больше нуля</term>
61     ///             <description>Первый не равный элемент <paramref name="left" /> списка
    → больше первого неравного элемента <paramref name="right" /> списка.</description>
62     ///         </item>
63     ///     </list>
64     /// </para>
65     /// </returns>
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
68 }
69 }

```

1.17 ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Lists
5 {
6     /// <summary>
7     /// <para>
8     ///     Represents the list equality comparer.

```

```

9    /// </para>
10   /// <para></para>
11   /// </summary>
12   /// <seealso cref="IEqualityComparer{IList{T}}"/>
13   public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
14   {
15       /// <summary>
16       /// <para>Compares two lists for equality.</para>
17       /// <para>Сравнивает два списка на равенство.</para>
18       /// </summary>
19       /// <param name="left"><para>The first compared list.</para><para>Первый список для
20       → сравнения.</para></param>
21       /// <param name="right"><para>The second compared list.</para><para>Второй список для
22       → сравнения.</para></param>
23       /// <returns>
24       /// <para>If the passed lists are equal to each other, true is returned, otherwise
25       → false.</para>
26       /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
27       → false.</para>
28       /// </returns>
29       [MethodImpl(MethodImplOptions.AggressiveInlining)]
30       public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
31
32       /// <summary>
33       /// <para>Generates a hash code for the entire list based on the values of its
34       → elements.</para>
35       /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
36       /// </summary>
37       /// <param name="list"><para>Hash list.</para><para>Список для
38       → хеширования.</para></param>
39       /// <returns>
40       /// <para>The hash code of the list.</para>
41       /// <para>Хэш-код списка.</para>
42       /// </returns>
43       [MethodImpl(MethodImplOptions.AggressiveInlining)]
44       public int GetHashCode(IList<T> list) => list.GenerateHashCode();
45   }
46 }

```

1.18 ./csharp/Platform.Collections/Lists/IListExtensions.cs

```

1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   namespace Platform.Collections.Lists
6   {
7       /// <summary>
8       /// <para>
9       /// Represents the list extensions.
10      /// </para>
11      /// <para></para>
12      /// </summary>
13      public static class IListExtensions
14      {
15          /// <summary>
16          /// <para>Gets the element from specified index if the list is not null and the index is
17          → within the list's boundaries, otherwise it returns default value of type T.</para>
18          /// <para>Получает элемент из указанного индекса, если список не является null и индекс
19          → находится в границах списка, в противном случае он возвращает значение по умолчанию
20          → типа T.</para>
21          /// </summary>
22          /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
23          → списка.</para></typeparam>
24          /// <param name="list"><para>The checked list.</para><para>Проверяемый
25          → список.</para></param>
26          /// <param name="index"><para>The index of element.</para><para>Индекс
27          → элемента.</para></param>
28          /// <returns>
29          /// <para>If the specified index is within list's boundaries, then - list[index],
30          → otherwise the default value.</para>
31          /// <para>Если указанный индекс находится в пределах границ списка, тогда - list[index],
32          → иначе же значение по умолчанию.</para>
33          /// </returns>
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
36          → list.Count > index ? list[index] : default;
37      }
38  }

```

```

29     /// <summary>
30     /// <para>Checks if a list is passed, checks its length, and if successful, copies the
    → value of list [index] into the element variable. Otherwise, the element variable has
    → a default value.</para>
31     /// <para>Проверяет, передан ли список, сверяет его длину и в случае успеха копирует
    → значение list[index] в переменную element. Иначе переменная element имеет значение
    → по умолчанию.</para>
32     /// </summary>
33     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
34     /// <param name="list"><para>The checked list.</para><para>Список для
    → проверки.</para></param>
35     /// <param name="index"><para>The index of element.</para><para>Индекс
    → элемента.</para></param>
36     /// <param name="element"><para>Variable for passing the index
    → value.</para><para>Переменная для передачи значения индекса.</para></param>
37     /// <returns>
38     /// <para>True on success, false otherwise.</para>
39     /// <para>True в случае успеха, иначе false.</para>
40     /// </returns>
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
43     {
44         if (list != null && list.Count > index)
45         {
46             element = list[index];
47             return true;
48         }
49         else
50         {
51             element = default;
52             return false;
53         }
54     }
55
56     /// <summary>
57     /// <para>Adds a value to the list.</para>
58     /// <para>Добавляет значение в список.</para>
59     /// </summary>
60     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
61     /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
62     /// <param name="element"><para>The item to add to the list.</para><para>Элемент который
    → нужно добавить в список.</para></param>
63     /// <returns>
64     /// <para>True value in any case.</para>
65     /// <para>Значение true в любом случае.</para>
66     /// </returns>
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
69     {
70         list.Add(element);
71         return true;
72     }
73
74     /// <summary>
75     /// <para>Adds the value with first index from other list to this list.</para>
76     /// <para>Добавляет в этот список значение с первым индексом из другого списка.</para>
77     /// </summary>
78     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
79     /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
80     /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
    → который нужно добавить в список</para></param>
81     /// <returns>
82     /// <para>True value in any case.</para>
83     /// <para>Значение true в любом случае.</para>
84     /// </returns>
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
87     {
88         list.AddFirst(elements);
89         return true;
90     }
91

```

```

92     /// <summary>
93     /// <para>Adds a value to the list at the first index.</para>
94     /// <para>Добавляет значение в список по первому индексу.</para>
95     /// </summary>
96     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
97     /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
98     /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
    → который нужно добавить в список</para></param>
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
    → list.Add(elements[0]);
101
102    /// <summary>
103    /// <para>Adds all elements from other list to this list and returns true.</para>
104    /// <para>Добавляет все элементы из другого списка в этот список и возвращает
    → true.</para>
105    /// </summary>
106    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
107    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
108    /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
109    /// <returns>
110    /// <para>True value in any case.</para>
111    /// <para>Значение true в любом случае.</para>
112    /// </returns>
113    [MethodImpl(MethodImplOptions.AggressiveInlining)]
114    public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
115    {
116        list.AddAll(elements);
117        return true;
118    }
119
120    /// <summary>
121    /// <para>Adds all elements from other list to this list.</para>
122    /// <para>Добавляет все элементы из другого списка в этот список.</para>
123    /// </summary>
124    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
125    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
126    /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
127    [MethodImpl(MethodImplOptions.AggressiveInlining)]
128    public static void AddAll<T>(this IList<T> list, IList<T> elements)
129    {
130        for (var i = 0; i < elements.Count; i++)
131        {
132            list.Add(elements[i]);
133        }
134    }
135
136    /// <summary>
137    /// <para>Adds values to the list skipping the first element.</para>
138    /// <para>Добавляет значения в список пропуская первый элемент.</para>
139    /// </summary>
140    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
141    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
142    /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
143    /// <returns>
144    /// <para>True value in any case.</para>
145    /// <para>Значение true в любом случае.</para>
146    /// </returns>
147    [MethodImpl(MethodImplOptions.AggressiveInlining)]
148    public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
149    {
150        list.AddSkipFirst(elements);
151        return true;
152    }
153
154    /// <summary>

```

```

155 /// <para>Adds values to the list skipping the first element.</para>
156 /// <para>Добавляет значения в список пропуская первый элемент.</para>
157 /// </summary>
158 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
159 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    ↳ нужно добавить значения.</para></param>
160 /// <param name="elements"><para>List of values to add.</para><para>Список значений
    ↳ которые необходимо добавить.</para></param>
161 [MethodImpl(MethodImplOptions.AggressiveInlining)]
162 public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
    ↳ list.AddSkipFirst(elements, 1);
163
164 /// <summary>
165 /// <para>Adds values to the list skipping a specified number of first elements.</para>
166 /// <para>Добавляет в список значения пропуская определенное количество первых
    ↳ элементов.</para>
167 /// </summary>
168 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
169 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    ↳ нужно добавить значения.</para></param>
170 /// <param name="elements"><para>List of values to add.</para><para>Список значений
    ↳ которые необходимо добавить.</para></param>
171 /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
    ↳ пропускаемых элементов.</para></param>
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
174 {
175     for (var i = skip; i < elements.Count; i++)
176     {
177         list.Add(elements[i]);
178     }
179 }
180
181 /// <summary>
182 /// <para>Reads the number of elements in the list.</para>
183 /// <para>Считывает число элементов списка.</para>
184 /// </summary>
185 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
186 /// <param name="list"><para>The checked list.</para><para>Список для
    ↳ проверки.</para></param>
187 /// <returns>
188 /// <para>The number of items contained in the list or 0.</para>
189 /// <para>Число элементов содержащихся в списке или же 0.</para>
190 /// </returns>
191 [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
193
194 /// <summary>
195 /// <para>Compares two lists for equality.</para>
196 /// <para>Сравнивает два списка на равенство.</para>
197 /// </summary>
198 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
199 /// <param name="left"><para>The first compared list.</para><para>Первый список для
    ↳ сравнения.</para></param>
200 /// <param name="right"><para>The second compared list.</para><para>Второй список для
    ↳ сравнения.</para></param>
201 /// <returns>
202 /// <para>If the passed lists are equal to each other, true is returned, otherwise
    ↳ false.</para>
203 /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
    ↳ false.</para>
204 /// </returns>
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
    ↳ right, ContentEqualTo);
207
208 /// <summary>
209 /// <para>Compares two lists for equality.</para>
210 /// <para>Сравнивает два списка на равенство.</para>
211 /// </summary>
212 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>

```

```

213 /// <param name="left"><para>The first compared list.</para><para>Первый список для
214   → проверки.</para></param>
215 /// <param name="right"><para>The second compared list.</para><para>Второй список для
216   → сравнения.</para></param>
217 /// <param name="contentEqualityComparer"><para>Function to test two lists for their
218   → content equality.</para><para>Функция для проверки двух списков на равенство их
219   → содержимого.</para></param>
220 /// <returns>
221 /// <para>If the passed lists are equal to each other, true is returned, otherwise
222   → false.</para>
223 /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
224   → false.</para>
225 /// </returns>
226 [MethodImpl(MethodImplOptions.AggressiveInlining)]
227 public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
228   → IList<T>, bool> contentEqualityComparer)
229 {
230     if (ReferenceEquals(left, right))
231     {
232         return true;
233     }
234     var leftCount = left.GetCountOrZero();
235     var rightCount = right.GetCountOrZero();
236     if (leftCount == 0 && rightCount == 0)
237     {
238         return true;
239     }
240     if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
241     {
242         return false;
243     }
244     return contentEqualityComparer(left, right);
245 }
246
247 /// <summary>
248 /// <para>Compares each element in the list for identity.</para>
249 /// <para>Сравнивает на равенство каждый элемент списка.</para>
250 /// </summary>
251 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
252   → списка.</para></typeparam>
253 /// <param name="left"><para>The first compared list.</para><para>Первый список для
254   → сравнения.</para></param>
255 /// <param name="right"><para>The second compared list.</para><para>Второй список для
256   → сравнения.</para></param>
257 /// <returns>
258 /// <para>If at least one element of one list is not equal to the corresponding element
259   → from another list returns false, otherwise - true.</para>
260 /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
261   → из другого списка возвращает false, иначе - true.</para>
262 /// </returns>
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
265 {
266     var equalityComparer = EqualityComparer<T>.Default;
267     for (var i = left.Count - 1; i >= 0; --i)
268     {
269         if (!equalityComparer.Equals(left[i], right[i]))
270         {
271             return false;
272         }
273     }
274     return true;
275 }
276
277 /// <summary>
278 /// <para>Creates an array by copying all elements from the list that satisfy the
279   → predicate. If no list is passed, null is returned.</para>
280 /// <para>Создаёт массив, копируя из списка все элементы которые удовлетворяют
281   → предикату. Если список не передан, возвращается null.</para>
282 /// </summary>
283 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
284   → списка.</para></typeparam>
285 /// <param name="list"><para>The list to copy from.</para><para>Список для копирования.</para></param>
286 /// <param name="predicate"><para>A function that determines whether an element should
287   → be copied.</para><para>Функция определяющая должен ли копироваться
288   → элемент.</para></param>
289 /// <returns>

```

```

273 /// <para>An array with copied elements from the list.</para>
274 /// <para>Массив с скопированными элементами из списка.</para>
275 /// </returns>
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
278 {
279     if (list == null)
280     {
281         return null;
282     }
283     var result = new List<T>(list.Count);
284     for (var i = 0; i < list.Count; i++)
285     {
286         if (predicate(list[i]))
287         {
288             result.Add(list[i]);
289         }
290     }
291     return result.ToArray();
292 }
293
294 /// <summary>
295 /// <para>Copies all the elements of the list into an array and returns it.</para>
296 /// <para>Копирует все элементы списка в массив и возвращает его.</para>
297 /// </summary>
298 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
299     ↪ списка.</para></typeparam>
300 /// <param name="list"><para>The list to copy from.</para><para>Список для
301     ↪ копирования.</para></param>
302 /// <returns>
303 /// <para>An array with all the elements of the passed list.</para>
304 /// <para>Массив со всеми элементами переданного списка.</para>
305 /// </returns>
306 [MethodImpl(MethodImplOptions.AggressiveInlining)]
307 public static T[] ToArray<T>(this IList<T> list)
308 {
309     var array = new T[list.Count];
310     list.CopyTo(array, 0);
311     return array;
312 }
313
314 /// <summary>
315 /// <para>Executes the passed action for each item in the list.</para>
316 /// <para>Выполняет переданное действие для каждого элемента в списке.</para>
317 /// </summary>
318 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
319     ↪ списка.</para></typeparam>
320 /// <param name="list"><para>The list of elements for which the action will be
321     ↪ executed.</para><para>Список элементов для которых будет выполняться
322     ↪ действие.</para></param>
323 /// <param name="action"><para>A function that will be called for each element of the
324     ↪ list.</para><para>Функция которая будет вызываться для каждого элемента
325     ↪ списка.</para></param>
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 public static void ForEach<T>(this IList<T> list, Action<T> action)
328 {
329     for (var i = 0; i < list.Count; i++)
330     {
331         action(list[i]);
332     }
333 }
334
335 /// <summary>
336 /// <para>Generates a hash code for the entire list based on the values of its
337     ↪ elements.</para>
338 /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
339 /// </summary>
340 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
341     ↪ списка.</para></typeparam>
342 /// <param name="list"><para>Hash list.</para><para>Список для
343     ↪ хеширования.</para></param>
344 /// <returns>
345 /// <para>The hash code of the list.</para>
346 /// <para>Хэш-код списка.</para>
347 /// </returns>
348 /// <remarks>

```



```

339 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
340 → -overridden-system-object-gethashcode
341 /// </remarks>
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 public static int GenerateHashCode<T>(this IList<T> list)
344 {
345     var hashAccumulator = 17;
346     for (var i = 0; i < list.Count; i++)
347     {
348         hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
349     }
350     return hashAccumulator;
351 }
352
353 /// <summary>
354 /// <para>Compares two lists.</para>
355 /// <para>Сравнивает два списка.</para>
356 /// </summary>
357 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
358 → списка.</para></typeparam>
359 /// <param name="left"><para>The first compared list.</para><para>Первый список для
360 → сравнения.</para></param>
361 /// <param name="right"><para>The second compared list.</para><para>Второй список для
362 → сравнения.</para></param>
363 /// <returns>
364 /// <para>
365 /// A signed integer that indicates the relative values of <paramref name="left" />
366 → and <paramref name="right" /> lists' elements, as shown in the following table.
367 /// <list type="table">
368 /// <listheader>
369 /// <term>Value</term>
370 /// <description>Meaning</description>
371 /// </listheader>
372 /// <item>
373 /// <term>Is less than zero</term>
374 /// <description>First non equal element of <paramref name="left" /> list is
375 → less than first not equal element of <paramref name="right" /> list.</description>
376 /// </item>
377 /// <item>
378 /// <term>Zero</term>
379 /// <description>All elements of <paramref name="left" /> list equals to all
380 → elements of <paramref name="right" /> list.</description>
381 /// </item>
382 /// <item>
383 /// <term>Is greater than zero</term>
384 /// <description>First non equal element of <paramref name="left" /> list is
385 → greater than first not equal element of <paramref name="right" /> list.</description>
386 /// </item>
387 /// </list>
388 /// </para>
389 /// <para>
390 /// Целое число со знаком, которое указывает относительные значения элементов
391 → списков <paramref name="left" /> и <paramref name="right" /> как показано в
392 → следующей таблице.
393 /// <list type="table">
394 /// <listheader>
395 /// <term>Значение</term>
396 /// <description>Смысл</description>
397 /// </listheader>
398 /// <item>
399 /// <term>Меньше нуля</term>
400 /// <description>Первый не равный элемент <paramref name="left" /> списка
401 → меньше первого неравного элемента <paramref name="right" /> списка.</description>
402 /// </item>
403 /// <item>
404 /// <term>Ноль</term>
405 /// <description>Все элементы <paramref name="left" /> списка равны всем
406 → элементам <paramref name="right" /> списка.</description>
407 /// </item>
408 /// <item>
409 /// <term>Больше нуля</term>
410 /// <description>Первый не равный элемент <paramref name="left" /> списка
411 → больше первого неравного элемента <paramref name="right" /> списка.</description>
412 /// </item>
413 /// </list>
414 /// </para>
415 /// </returns>

```

```

403 [MethodImpl(MethodImplOptions.AggressiveInlining)]
404 public static int CompareTo<T>(this IList<T> left, IList<T> right)
405 {
406     var comparer = Comparer<T>.Default;
407     var leftCount = left.GetCountOrZero();
408     var rightCount = right.GetCountOrZero();
409     var intermediateResult = leftCount.CompareTo(rightCount);
410     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
411     {
412         intermediateResult = comparer.Compare(left[i], right[i]);
413     }
414     return intermediateResult;
415 }
416
417 /// <summary>
418 /// <para>Skips one element in the list and builds an array from the remaining
419   → elements.</para>
420 /// <para>Пропускает один элемент списка и составляет из оставшихся элементов
421   → массив.</para>
422 /// </summary>
423 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
424   → списка.</para></typeparam>
425 /// <param name="list"><para>The list to copy from.</para><para>Список для
426   → копирования.</para></param>
427 /// <returns>
428 /// <para>If the list is empty, returns an empty array, otherwise - an array with a
429   → missing first element.</para>
430 /// <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
431   → первым элементом.</para>
432 /// </returns>
433 [MethodImpl(MethodImplOptions.AggressiveInlining)]
434 public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
435
436 /// <summary>
437 /// <para>Skips the specified number of elements in the list and builds an array from
438   → the remaining elements.</para>
439 /// <para>Пропускает указанное количество элементов списка и составляет из оставшихся
440   → элементов массив.</para>
441 /// </summary>
442 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
443   → списка.</para></typeparam>
444 /// <param name="list"><para>The list to copy from.</para><para>Список для
445   → копирования.</para></param>
446 /// <param name="skip"><para>The number of items to skip.</para><para>Количество
447   → пропускаемых элементов.</para></param>
448 /// <returns>
449 /// <para>If the list is empty, or the number of skipped elements is greater than the
450   → list, returns an empty array, otherwise - an array with the specified number of
451   → missing elements.</para>
452 /// <para>Если список пуст, или количество пропускаемых элементов больше списка -
453   → возвращает пустой массив, иначе - массив с указанным количеством пропущенных
454   → элементов.</para>
455 /// </returns>
456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
457 public static T[] SkipFirst<T>(this IList<T> list, int skip)
458 {
459     if (list.IsNullOrEmpty() || list.Count <= skip)
460     {
461         return Array.Empty<T>();
462     }
463     var result = new T[list.Count - skip];
464     for (int r = skip, w = 0; r < list.Count; r++, w++)
465     {
466         result[w] = list[r];
467     }
468     return result;
469 }
470
471 /// <summary>
472 /// <para>Shifts all the elements of the list by one position to the right.</para>
473 /// <para>Сдвигает вправо все элементы списка на одну позицию.</para>
474 /// </summary>
475 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
476   → списка.</para></typeparam>
477 /// <param name="list"><para>The list to copy from.</para><para>Список для
478   → копирования.</para></param>
479 /// <returns>

```

```

463     /// <para>Array with a shift of elements by one position.</para>
464     /// <para>Массив со сдвигом элементов на одну позицию.</para>
465     /// </returns>
466     [MethodImpl(MethodImplOptions.AggressiveInlining)]
467     public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
468
469     /// <summary>
470     /// <para>Shifts all elements of the list to the right by the specified number of
471     ///     elements.</para>
472     /// <para>Сдвигает вправо все элементы списка на указанное количество элементов.</para>
473     /// </summary>
474     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
475     ///     списка.</para></typeparam>
476     /// <param name="list"><para>The list to copy from.</para><para>Список для
477     ///     копирования.</para></param>
478     /// <param name="shift"><para>The number of items to shift.</para><para>Количество
479     ///     сдвигаемых элементов.</para></param>
480     /// <returns>
481     /// <para>If the value of the shift variable is less than zero - an <see
482     ///     cref="NotImplementedException"/> exception is thrown, but if the value of the shift
483     ///     variable is 0 - an exact copy of the array is returned. Otherwise, an array is
484     ///     returned with the shift of the elements.</para>
485     /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
486     ///     cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
487     ///     возвращается точная копия массива. Иначе возвращается массив со сдвигом
488     ///     элементов.</para>
489     /// </returns>
490     [MethodImpl(MethodImplOptions.AggressiveInlining)]
491     public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
492     {
493         if (shift < 0)
494         {
495             throw new NotImplementedException();
496         }
497         if (shift == 0)
498         {
499             return list.ToArray();
500         }
501         else
502         {
503             var result = new T[list.Count + shift];
504             for (int r = 0, w = shift; r < list.Count; r++, w++)
505             {
506                 result[w] = list[r];
507             }
508             return result;
509         }
510     }
511 }
512

```

1.19 ./csharp/Platform.Collections/Lists/ListFiller.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      /// <summary>
7      /// <para>
8      ///     Represents the list filler.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public class ListFiller<TElement, TReturnConstant>
13     {
14         /// <summary>
15         /// <para>
16         ///     The list.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         protected readonly List<TElement> _list;
21         /// <summary>
22         /// <para>
23         ///     The return constant.
24         /// </para>
25         /// <para></para>
26         /// </summary>

```

```

27     protected readonly TReturnConstant _returnConstant;
28
29     /// <summary>
30     /// <para>Initializes a new instance of the ListFiller class.</para>
31     /// <para>Инициализирует новый экземпляр класса ListFiller.</para>
32     /// </summary>
33     /// <param name="list"><para>The list to be filled.</para><para>Список который будет
    → заполняться.</para></param>
34     /// <param name="returnConstant"><para>The value for the constant returned by
    → corresponding methods.</para><para>Значение для константы возвращаемой
    → соответствующими методами.</para></param>
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public ListFiller(List<TElement> list, TReturnConstant returnConstant)
37     {
38         _list = list;
39         _returnConstant = returnConstant;
40     }
41
42     /// <summary>
43     /// <para>
44     /// Initializes a new <see cref="ListFiller"/> instance.
45     /// </para>
46     /// <para></para>
47     /// </summary>
48     /// <param name="list">
49     /// <para>A list.</para>
50     /// <para></para>
51     /// </param>
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public ListFiller(List<TElement> list) : this(list, default) { }
54
55     /// <summary>
56     /// <para>Adds an item to the end of the list.</para>
57     /// <para>Добавляет элемент в конец списка.</para>
58     /// </summary>
59     /// <param name="element"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     public void Add(TElement element) => _list.Add(element);
62
63     /// <summary>
64     /// <para>Adds an item to the end of the list and return true.</para>
65     /// <para>Добавляет элемент в конец списка и возвращает true.</para>
66     /// </summary>
67     /// <param name="element"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
68     /// <returns>
69     /// <para>True value in any case.</para>
70     /// <para>Значение true в любом случае.</para>
71     /// </returns>
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
74
75     /// <summary>
76     /// <para>Adds a value to the list at the first index and return true.</para>
77     /// <para>Добавляет значение в список по первому индексу и возвращает true.</para>
78     /// </summary>
79     /// <param name="elements"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
80     /// <returns>
81     /// <para>True value in any case.</para>
82     /// <para>Значение true в любом случае.</para>
83     /// </returns>
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
    → _list.AddFirstAndReturnTrue(elements);
86
87     /// <summary>
88     /// <para>Adds all elements from other list to this list and returns true.</para>
89     /// <para>Добавляет все элементы из другого списка в этот список и возвращает
    → true.</para>
90     /// </summary>
91     /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
92     /// <returns>
93     /// <para>True value in any case.</para>
94     /// <para>Значение true в любом случае.</para>

```

```

95     /// </returns>
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public bool AddAllAndReturnTrue(IList<TElement> elements) =>
98         ↪ _list.AddAllAndReturnTrue(elements);
99
100    /// <summary>
101    /// <para>Adds values to the list skipping the first element.</para>
102    /// <para>Добавляет значения в список пропуская первый элемент.</para>
103    /// </summary>
104    /// <param name="elements"><para>The list of values to add.</para><para>Список значений
105    ↪ которые необходимо добавить.</para></param>
106    /// <returns>
107    /// <para>True value in any case.</para>
108    /// <para>Значение true в любом случае.</para>
109    /// </returns>
110    [MethodImpl(MethodImplOptions.AggressiveInlining)]
111    public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
112        ↪ _list.AddSkipFirstAndReturnTrue(elements);
113
114    /// <summary>
115    /// <para>Adds an item to the end of the list and return constant.</para>
116    /// <para>Добавляет элемент в конец списка и возвращает константу.</para>
117    /// </summary>
118    /// <param name="element"><para>Element to add.</para><para>Добавляемый
119    ↪ элемент.</para></param>
120    /// <returns>
121    /// <para>Constant value in any case.</para>
122    /// <para>Значение константы в любом случае.</para>
123    /// </returns>
124    [MethodImpl(MethodImplOptions.AggressiveInlining)]
125    public TReturnConstant AddAndReturnConstant(TElement element)
126    {
127        ↪ _list.Add(element);
128        ↪ return _returnConstant;
129    }
130
131    /// <summary>
132    /// <para>Adds a value to the list at the first index and return constant.</para>
133    /// <para>Добавляет значение в список по первому индексу и возвращает константу.</para>
134    /// </summary>
135    /// <param name="element"><para>Element to add.</para><para>Добавляемый
136    ↪ элемент.</para></param>
137    /// <returns>
138    /// <para>Constant value in any case.</para>
139    /// <para>Значение константы в любом случае.</para>
140    /// </returns>
141    [MethodImpl(MethodImplOptions.AggressiveInlining)]
142    public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
143    {
144        ↪ _list.AddFirst(elements);
145        ↪ return _returnConstant;
146    }
147
148    /// <summary>
149    /// <para>Adds all elements from other list to this list and returns constant.</para>
150    /// <para>Добавляет все элементы из другого списка в этот список и возвращает
151    ↪ константу.</para>
152    /// </summary>
153    /// <param name="elements"><para>List of values to add.</para><para>Список значений
154    ↪ которые необходимо добавить.</para></param>
155    /// <returns>
156    /// <para>Constant value in any case.</para>
157    /// <para>Значение константы в любом случае.</para>
158    /// </returns>
159    [MethodImpl(MethodImplOptions.AggressiveInlining)]
160    public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
161    {
162        ↪ _list.AddAll(elements);
163        ↪ return _returnConstant;
164    }
165
166    /// <summary>
167    /// <para>Adds values to the list skipping the first element and return constant
168    ↪ value.</para>
169    /// <para>Добавляет значения в список пропуская первый элемент и возвращает значение
170    ↪ константы.</para>
171    /// </summary>

```

```

163     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    ↪     которые необходимо добавить.</para></param>
164     /// <returns>
165     /// <para>constant value in any case.</para>
166     /// <para>Значение константы в любом случае.</para>
167     /// </returns>
168     [MethodImpl(MethodImplOptions.AggressiveInlining)]
169     public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
170     {
171         _list.AddSkipFirst(elements);
172         return _returnConstant;
173     }
174 }
175 }

```

1.20 ./csharp/Platform.Collections/Segments/CharSegment.cs

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the char segment.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="Segment{char}"/>
18     public class CharSegment : Segment<char>
19     {
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="CharSegment"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="@base">
27         /// <para>A base.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="offset">
31         /// <para>A offset.</para>
32         /// <para></para>
33         /// </param>
34         /// <param name="length">
35         /// <para>A length.</para>
36         /// <para></para>
37         /// </param>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
    ↪     length) { }
40
41         /// <summary>
42         /// <para>
43         /// Gets the hash code.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <returns>
48         /// <para>The int</para>
49         /// <para></para>
50         /// </returns>
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public override int GetHashCode()
53         {
54             // Base can be not an array, but still IList<char>
55             if (Base is char[] baseArray)
56             {
57                 return baseArray.GenerateHashCode(Offset, Length);
58             }
59             else
60             {
61                 return this.GenerateHashCode();

```

```

62     }
63 }
64
65 /// <summary>
66 /// <para>
67 /// Determines whether this instance equals.
68 /// </para>
69 /// <para></para>
70 /// </summary>
71 /// <param name="other">
72 /// <para>The other.</para>
73 /// <para></para>
74 /// </param>
75 /// <returns>
76 /// <para>The bool</para>
77 /// <para></para>
78 /// </returns>
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public override bool Equals(Segment<char> other)
81 {
82     bool contentEqualityComparer(IList<char> left, IList<char> right)
83     {
84         // Base can be not an array, but still IList<char>
85         if (Base is char[] baseArray && other.Base is char[] otherArray)
86         {
87             return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
88         }
89         else
90         {
91             return left.ContentEqualTo(right);
92         }
93     }
94     return this.EqualTo(other, contentEqualityComparer);
95 }
96
97 /// <summary>
98 /// <para>
99 /// Determines whether this instance equals.
100 /// </para>
101 /// <para></para>
102 /// </summary>
103 /// <param name="obj">
104 /// <para>The obj.</para>
105 /// <para></para>
106 /// </param>
107 /// <returns>
108 /// <para>The bool</para>
109 /// <para></para>
110 /// </returns>
111 public override bool Equals(object obj) => obj is Segment<char> charSegment ?
    ↳ Equals(charSegment) : false;
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 public static implicit operator string(CharSegment segment)
115 {
116     if (!(segment.Base is char[] array))
117     {
118         array = segment.Base.ToArray();
119     }
120     return new string(array, segment.Offset, segment.Length);
121 }
122
123 /// <summary>
124 /// <para>
125 /// Returns the string.
126 /// </para>
127 /// <para></para>
128 /// </summary>
129 /// <returns>
130 /// <para>The string</para>
131 /// <para></para>
132 /// </returns>
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 public override string ToString() => this;
135 }
136 }

```

1.21 ./csharp/Platform.Collections/Segments/Segment.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Runtime.CompilerServices;
5 using Platform.Collections.Arrays;
6 using Platform.Collections.Lists;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Segments
11 {
12     /// <summary>
13     /// <para>Represents the segment of an <see cref="IList"/>.</para>
14     /// <para>Представляет сегмент <see cref="IList"/>.</para>
15     /// </summary>
16     /// <typeparam name="T"><para>The segment elements type.</para><para>Тип элементов
17     ↪ сегмента.</para></typeparam>
18     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
19     {
20         /// <summary>
21         /// <para>Gets the original list (this segment is a part of it).</para>
22         /// <para>Возвращает исходный список (частью которого является этот сегмент).</para>
23         /// </summary>
24         public IList<T> Base
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29         /// <summary>
30         /// <para>Gets the offset relative to the source list (the index at which this segment
31         ↪ starts).</para>
32         /// <para>Возвращает смещение относительно исходного списка (индекс с которого
33         ↪ начинается этот сегмент).</para>
34         /// </summary>
35         public int Offset
36         {
37             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38             get;
39         }
40         /// <summary>
41         /// <para>Gets the length of a segment.</para>
42         /// <para>Возвращает длину сегмента.</para>
43         /// </summary>
44         public int Length
45         {
46             [MethodImpl(MethodImplOptions.AggressiveInlining)]
47             get;
48         }
49         /// <summary>
50         /// <para>Initializes a new instance of the <see cref="Segment"/> class, using the
51         ↪ <paramref name="base"/> list, <paramref name="offset"/> of the segment and its
52         ↪ <paramref name="length" />.</para>
53         /// <para>Инициализирует новый экземпляр класса <see cref="Segment"/>, используя список
54         ↪ <paramref name="base"/>, <paramref name="offset"/> сегмента и его <paramref
55         ↪ name="length"/>.</para>
56         /// </summary>
57         /// <param name="base"><para>The reference to the original list containing the elements
58         ↪ of this segment.</para><para>Ссылка на исходный список в котором находятся элементы
59         ↪ этого сегмента.</para></param>
60         /// <param name="offset"><para>The offset relative to the <paramref name="base"/> list
61         ↪ from which the segment starts.</para><para>Смещение относительно списка <paramref
62         ↪ name="base"/>, с которого начинается сегмент.</para></param>
63         /// <param name="length"><para>The segment's length.</para><para>Длина
64         ↪ сегмента.</para></param>
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public Segment(IList<T> @base, int offset, int length)
67         {
68             Base = @base;
69             Offset = offset;
70             Length = length;
71         }
72         /// <summary>
73         /// <para>Gets the hash code of the current <see cref="Segment"/> instance.</para>
74         /// <para>Возвращает хэш-код текущего экземпляра <see cref="Segment"/>.</para>
75         /// </summary>

```



```

66     /// <returns></returns>
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public override int GetHashCode() => this.GenerateHashCode();
69
70     /// <summary>
71     /// <para>Returns a value indicating whether the current <see cref="Segment"/> is equal
72     → to another <see cref="Segment" />.</para>
73     /// <para>Возвращает значение определяющее, равен ли текущий <see cref="Segment"/>
74     → другому <see cref="Segment"/>.</para>
75     /// </summary>
76     /// <param name="other"><para>An <see cref="Segment"/> object to compare with the
77     → current <see cref="Segment"/>.</para><para>Объект <see cref="Segment"/> для
78     → сравнения с текущим <see cref="Segment"/>.</para></param>
79     /// <returns>
80     /// <para><see langword="true"/> if the current <see cref="Segment"/> is equal to the
81     → <paramref name="other"/> parameter; otherwise, <see langword="false"/>.</para>
82     /// <para><see langword="true"/>, если текущий <see cref="Segment"/> равен параметру
83     → <paramref name="other"/>, в противном случае - <see langword="false"/>.</para>
84     /// </returns>
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
87
88     /// <summary>
89     /// <para>
90     /// Determines whether this instance equals.
91     /// </para>
92     /// <para></para>
93     /// </summary>
94     /// <param name="obj">
95     /// <para>The obj.</para>
96     /// <para></para>
97     /// </param>
98     /// <returns>
99     /// <para>The bool</para>
100    /// <para></para>
101    /// </returns>
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
104    → false;
105
106    #region IList
107
108    /// <summary>
109    /// <para>
110    /// The value.
111    /// </para>
112    /// <para></para>
113    /// </summary>
114    public T this[int i]
115    {
116        [MethodImpl(MethodImplOptions.AggressiveInlining)]
117        get => Base[Offset + i];
118        [MethodImpl(MethodImplOptions.AggressiveInlining)]
119        set => Base[Offset + i] = value;
120    }
121
122    /// <summary>
123    /// <para>Gets the number of elements contained in the <see cref="Segment"/>.</para>
124    /// <para>Возвращает число элементов, содержащихся в <see cref="Segment"/>.</para>
125    /// </summary>
126    /// <value>
127    /// <para>The number of elements contained in the <see cref="Segment"/>.</para>
128    /// <para>Число элементов, содержащихся в <see cref="Segment"/>.</para>
129    /// </value>
130    public int Count
131    {
132        [MethodImpl(MethodImplOptions.AggressiveInlining)]
133        get => Length;
134    }
135
136    /// <summary>
137    /// <para>Gets a value indicating whether the <see cref="Segment"/> is read-only.</para>
138    /// <para>Возвращает значение, указывающее, является ли <see cref="Segment"/> доступным
139    → только для чтения.</para>
140    /// </summary>
141    /// <value>
142    /// <para><see langword="true"/> if the <see cref="Segment"/> is read-only; otherwise,
143    → <see langword="false"/>.</para>

```

```

135 /// <para>Значение <see langword="true"/>, если <see cref="Segment"/> доступен только
136   → для чтения, в противном случае - значение <see langword="false"/>.</para>
137 /// </value>
138 /// <remarks>
139 /// <para>Any <see cref="Segment"/> is read-only.</para>
140 /// <para>Любой <see cref="Segment"/> доступен только для чтения.</para>
141 /// </remarks>
142 public bool IsReadOnly
143 {
144     [MethodImpl(MethodImplOptions.AggressiveInlining)]
145     get => true;
146 }
147
148 /// <summary>
149 /// <para>Determines the index of a specific item in the <see cref="Segment"/>.</para>
150 /// <para>Определяет индекс конкретного элемента в <see cref="Segment"/>.</para>
151 /// </summary>
152 /// <param name="item"><para>The object to locate in the <see
153   → cref="Segment"/>.</para><para>Элемент для поиска в <see
154   → cref="Segment"/>.</para></param>
155 /// <returns>
156 /// <para>The index of <paramref name="item"/> if found in the segment; otherwise,
157   → -1.</para>
158 /// <para>Индекс <paramref name="item"/>, если он найден в сегменте; в противном случае
159   → - значение -1.</para>
160 /// </returns>
161 [MethodImpl(MethodImplOptions.AggressiveInlining)]
162 public int IndexOf(T item)
163 {
164     var index = Base.IndexOf(item);
165     if (index >= Offset)
166     {
167         var actualIndex = index - Offset;
168         if (actualIndex < Length)
169         {
170             return actualIndex;
171         }
172     }
173     return -1;
174 }
175
176 /// <summary>
177 /// <para>Inserts an item to the <see cref="Segment"/> at the specified index.</para>
178 /// <para>Вставляет элемент в <see cref="Segment"/> по указанному индексу.</para>
179 /// </summary>
180 /// <param name="index"><para>The zero-based index at which <paramref name="item"/>
181   → should be inserted.</para><para>Отсчитываемый от нуля индекс, по которому следует
182   → вставить элемент <paramref name="item"/>.</para></param>
183 /// <param name="item"><para>The element to insert into the <see
184   → cref="Segment"/>.</para><para>Элемент, вставляемый в <see
185   → cref="Segment"/>.</para></param>
186 /// <exception cref="NotSupportedException">
187 /// <para>The <see cref="Segment"/> is read-only.</para>
188 /// <para><see cref="Segment"/> доступен только для чтения.</para>
189 /// </exception>
190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 public void Insert(int index, T item) => throw new NotSupportedException();
192
193 /// <summary>
194 /// <para>Removes the <see cref="Segment"/> item at the specified index.</para>
195 /// <para>Удаляет элемент <see cref="Segment"/> по указанному индексу.</para>
196 /// </summary>
197 /// <param name="index"><para>The zero-based index of the item to
198   → remove.</para><para>Отсчитываемый от нуля индекс элемента для
199   → удаления.</para></param>
200 /// <exception cref="NotSupportedException">
201 /// <para>The <see cref="Segment"/> is read-only.</para>
202 /// <para><see cref="Segment"/> доступен только для чтения.</para>
203 /// </exception>
204 [MethodImpl(MethodImplOptions.AggressiveInlining)]
205 public void RemoveAt(int index) => throw new NotSupportedException();
206
207 /// <summary>
208 /// <para>Adds an item to the <see cref="Segment"/>.</para>
209 /// <para>Добавляет элемент в <see cref="Segment"/>.</para>
210 /// </summary>

```

```

200  /// <param name="item"><para>The element to add to the <see
    → cref="Segment"/>.</para><para>Элемент, добавляемый в <see
    → cref="Segment"/>.</para></param>
201  /// <exception cref="NotSupportedException">
202  /// <para>The <see cref="Segment"/> is read-only.</para>
203  /// <para><see cref="Segment"/> доступен только для чтения.</para>
204  /// </exception>
205  [MethodImpl(MethodImplOptions.AggressiveInlining)]
206  public void Add(T item) => throw new NotSupportedException();
207
208  /// <summary>
209  /// <para>Removes all items from the <see cref="Segment"/>.</para>
210  /// <para>Удаляет все элементы из <see cref="Segment"/>.</para>
211  /// </summary>
212  /// <exception cref="NotSupportedException">
213  /// <para>The <see cref="Segment"/> is read-only.</para>
214  /// <para><see cref="Segment"/> доступен только для чтения.</para>
215  /// </exception>
216  [MethodImpl(MethodImplOptions.AggressiveInlining)]
217  public void Clear() => throw new NotSupportedException();
218
219  /// <summary>
220  /// <para>Determines whether the <see cref="Segment"/> contains a specific value.</para>
221  /// <para>Определяет, содержит ли <see cref="Segment"/> определенное значение.</para>
222  /// </summary>
223  /// <param name="item"><para>The value to locate in the <see
    → cref="Segment"/>.</para><para>Значение, которое нужно найти в <see
    → cref="Segment"/>.</para></param>
224  /// <returns>
225  /// <para><see langword="true"/> if the value is found in the <see cref="Segment"/>;
    → otherwise, <see langword="false"/>.</para>
226  /// <para>Значение <see langword="true"/>, если значение находится в <see
    → cref="Segment"/>; в противном случае - <see langword="false"/>.</para>
227  /// </returns>
228  [MethodImpl(MethodImplOptions.AggressiveInlining)]
229  public bool Contains(T item) => IndexOf(item) >= 0;
230
231  /// <summary>
232  /// <para>Copies the elements of the <see cref="Segment"/> into an array, starting at a
    → specific array index.</para>
233  /// <para>Копирует элементы <see cref="Segment"/> в массив, начиная с определенного
    → индекса массива.</para>
234  /// </summary>
235  /// <param name="array"><para>A one-dimensional array that is the destination of the
    → elements copied from <see cref="Segment"/></para><para>Одномерный массив, который
    → является местом назначения элементов, скопированных из <see
    → cref="Segment"/>.</para></param>
236  /// <param name="arrayIndex"><para>The zero-based index in <paramref name="array"/> at
    → which copying begins.</para><para>Отсчитываемый от нуля индекс в массиве <paramref
    → name="array"/>, с которого начинается копирование.</para></param>
237  [MethodImpl(MethodImplOptions.AggressiveInlining)]
238  public void CopyTo(T[] array, int arrayIndex)
239  {
240      for (var i = 0; i < Length; i++)
241      {
242          array.Add(ref arrayIndex, this[i]);
243      }
244  }
245
246  /// <summary>
247  /// <para>Removes the first occurrence of a specific value from the <see
    → cref="Segment"/>.</para>
248  /// <para>Удаляет первое вхождение указанного значения из <see cref="Segment"/>.</para>
249  /// </summary>
250  /// <param name="item"><para>The value to remove from the <see
    → cref="Segment"/>.</para><para>Значение, которые нужно удалить из <see
    → cref="Segment"/>.</para></param>
251  /// <returns></returns>
252  /// <exception cref="NotSupportedException">
253  /// <para>The <see cref="Segment"/> is read-only.</para>
254  /// <para><see cref="Segment"/> доступен только для чтения.</para>
255  /// </exception>
256  [MethodImpl(MethodImplOptions.AggressiveInlining)]
257  public bool Remove(T item) => throw new NotSupportedException();
258
259  /// <summary>
260  /// <para>Gets an enumerator that iterates through a <see cref="Segment"/>.</para>

```

```

261     /// <para>Возвращает перечислитель, который осуществляет итерацию по <see
    ↪ cref="Segment"/>.</para>
262     /// </summary>
263     /// <returns>
264     /// <para>An <see cref="T:System.Collections.IEnumerator"/> object that can be used to
    ↪ iterate through the the <see cref="Segment"/>.</para>
265     /// <para>Объект <see cref="T:System.Collections.IEnumerator"/>, который можно
    ↪ использовать для перебора <see cref="Segment"/>.</para>
266     /// </returns>
267     [MethodImpl(MethodImplOptions.AggressiveInlining)]
268     public IEnumerator<T> GetEnumerator()
269     {
270         for (var i = 0; i < Length; i++)
271         {
272             yield return this[i];
273         }
274     }
275
276     /// <summary>
277     /// <para>Gets an enumerator that iterates through a <see cref="Segment"/>.</para>
278     /// <para>Возвращает перечислитель, который осуществляет итерацию по <see
    ↪ cref="Segment"/>.</para>
279     /// </summary>
280     /// <returns>
281     /// <para>An <see cref="T:System.Collections.IEnumerator"/> object that can be used to
    ↪ iterate through the collection.</para>
282     /// <para>Объект <see cref="T:System.Collections.IEnumerator"/>, который можно
    ↪ использовать для перебора <see cref="Segment"/>.</para>
283     /// </returns>
284     [MethodImpl(MethodImplOptions.AggressiveInlining)]
285     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
286
287     #endregion
288 }
289 }

```

1.22 ./csharp/Platform.Collections.Segments.Walkers/AllSegmentsWalkerBase.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the all segments walker base.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public abstract class AllSegmentsWalkerBase
12     {
13         /// <summary>
14         /// <para>
15         /// The default minimum string segment length.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         public static readonly int DefaultMinimumStringSegmentLength = 2;
20     }
21 }

```

1.23 ./csharp/Platform.Collections.Segments.Walkers/AllSegmentsWalkerBase<T, TSegment>.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the all segments walker base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="AllSegmentsWalkerBase"/>
15     public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
16     where TSegment : Segment<T>
17     {
18         private readonly int _minimumStringSegmentLength;

```

```

19
20     /// <summary>
21     /// <para>
22     /// Initializes a new <see cref="AllSegmentsWalkerBase"/> instance.
23     /// </para>
24     /// <para></para>
25     /// </summary>
26     /// <param name="minimumStringSegmentLength">
27     /// <para>A minimum string segment length.</para>
28     /// <para></para>
29     /// </param>
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
32     ↪     _minimumStringSegmentLength = minimumStringSegmentLength;
33
34     /// <summary>
35     /// <para>
36     /// Initializes a new <see cref="AllSegmentsWalkerBase"/> instance.
37     /// </para>
38     /// <para></para>
39     /// </summary>
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
42
43     /// <summary>
44     /// <para>
45     /// Walks the all using the specified elements.
46     /// </para>
47     /// <para></para>
48     /// </summary>
49     /// <param name="elements">
50     /// <para>The elements.</para>
51     /// <para></para>
52     /// </param>
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public virtual void WalkAll(ICollection<T> elements)
55     {
56         for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
57             ↪ offset <= maxOffset; offset++)
58         {
59             for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
60                 ↪ offset; length <= maxLength; length++)
61             {
62                 Iteration(CreateSegment(elements, offset, length));
63             }
64         }
65     }
66
67     /// <summary>
68     /// <para>
69     /// Creates the segment using the specified elements.
70     /// </para>
71     /// <para></para>
72     /// </summary>
73     /// <param name="elements">
74     /// <para>The elements.</para>
75     /// <para></para>
76     /// </param>
77     /// <param name="offset">
78     /// <para>The offset.</para>
79     /// <para></para>
80     /// </param>
81     /// <param name="length">
82     /// <para>The length.</para>
83     /// <para></para>
84     /// </param>
85     /// <returns>
86     /// <para>The segment</para>
87     /// <para></para>
88     /// </returns>
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);
91
92     /// <summary>
93     /// <para>
94     /// Iterations the segment.
95     /// </para>
96     /// <para></para>
97     /// </summary>
98     /// <param name="segment">
99     /// <para>The segment to iterate.</para>
100    /// <para></para>
101    /// </param>

```

```

94     /// </summary>
95     /// <param name="segment">
96     /// <para>The segment.</para>
97     /// <para></para>
98     /// </param>
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100     protected abstract void Iteration(TSegment segment);
101 }
102 }

```

1.24 ./csharp/Platform.Collections.Segments/Walkers/AllSegmentsWalkerBase[T].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the all segments walker base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="AllSegmentsWalkerBase{T, Segment{T}}"/>
15     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
16     {
17         /// <summary>
18         /// <para>
19         /// Creates the segment using the specified elements.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="elements">
24         /// <para>The elements.</para>
25         /// <para></para>
26         /// </param>
27         /// <param name="offset">
28         /// <para>The offset.</para>
29         /// <para></para>
30         /// </param>
31         /// <param name="length">
32         /// <para>The length.</para>
33         /// <para></para>
34         /// </param>
35         /// <returns>
36         /// <para>A segment of t</para>
37         /// <para></para>
38         /// </returns>
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected override Segment<T> CreateSegment(ICollection<T> elements, int offset, int length)
41         {
42             => new Segment<T>(elements, offset, length);
43         }
44     }
45 }

```

1.25 ./csharp/Platform.Collections.Segments/Walkers/AllSegmentsWalkerExtensions.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Segments.Walkers
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the all segments walker extensions.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     public static class AllSegmentsWalkerExtensions
14     {
15         /// <summary>
16         /// <para>
17         /// Walks the all using the specified walker.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <param name="walker">
22         /// <para>The walker.</para>

```

```

23     /// <para></para>
24     /// </param>
25     /// <param name="@string">
26     /// <para>The string.</para>
27     /// <para></para>
28     /// </param>
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
31         ↪ walker.WalkAll(@string.ToCharArray());
32
33     /// <summary>
34     /// <para>
35     /// Walks the all using the specified walker.
36     /// </para>
37     /// </summary>
38     /// <typeparam name="TSegment">
39     /// <para>The segment.</para>
40     /// <para></para>
41     /// </typeparam>
42     /// <param name="walker">
43     /// <para>The walker.</para>
44     /// <para></para>
45     /// </param>
46     /// <param name="@string">
47     /// <para>The string.</para>
48     /// <para></para>
49     /// </param>
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
52         ↪ string @string) where TSegment : Segment<char> =>
53         ↪ walker.WalkAll(@string.ToCharArray());
54 }
55 }

```

1.26 ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, TSegment]

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Segments.Walkers
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the dictionary based duplicate segments walker base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="DuplicateSegmentsWalkerBase{T, TSegment}"/>
16     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
17         ↪ DuplicateSegmentsWalkerBase<T, TSegment>
18         where TSegment : Segment<T>
19     {
20         /// <summary>
21         /// <para>
22         /// The default reset dictionary on each walk.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         public static readonly bool DefaultResetDictionaryOnEachWalk;
27         private readonly bool _resetDictionaryOnEachWalk;
28         /// <summary>
29         /// <para>
30         /// The dictionary.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         protected IDictionary<TSegment, long> Dictionary;
35
36         /// <summary>
37         /// <para>
38         /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
39         /// </para>
40         /// <para></para>
41         /// </summary>
42         /// <param name="dictionary">
43         /// <para>A dictionary.</para>

```

```

43     /// <para></para>
44     /// </param>
45     /// <param name="minimumStringSegmentLength">
46     /// <para>A minimum string segment length.</para>
47     /// <para></para>
48     /// </param>
49     /// <param name="resetDictionaryOnEachWalk">
50     /// <para>A reset dictionary on each walk.</para>
51     /// <para></para>
52     /// </param>
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
55     ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
56     : base(minimumStringSegmentLength)
57     {
58         Dictionary = dictionary;
59         _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
60     }
61     /// <summary>
62     /// <para>
63     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
64     /// </para>
65     /// <para></para>
66     /// </summary>
67     /// <param name="dictionary">
68     /// <para>A dictionary.</para>
69     /// <para></para>
70     /// </param>
71     /// <param name="minimumStringSegmentLength">
72     /// <para>A minimum string segment length.</para>
73     /// <para></para>
74     /// </param>
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
77     ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
78     ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
79     /// <summary>
80     /// <para>
81     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
82     /// </para>
83     /// <para></para>
84     /// </summary>
85     /// <param name="dictionary">
86     /// <para>A dictionary.</para>
87     /// <para></para>
88     /// </param>
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
91     ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
92     ↪ DefaultResetDictionaryOnEachWalk) { }
93     /// <summary>
94     /// <para>
95     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
96     /// </para>
97     /// <para></para>
98     /// </summary>
99     /// <param name="minimumStringSegmentLength">
100    /// <para>A minimum string segment length.</para>
101    /// <para></para>
102    /// </param>
103    /// <param name="resetDictionaryOnEachWalk">
104    /// <para>A reset dictionary on each walk.</para>
105    /// <para></para>
106    /// </param>
107    [MethodImpl(MethodImplOptions.AggressiveInlining)]
108    protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
109    ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
110    ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
111    { }
112    /// <summary>
113    /// <para>
114    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
115    /// </para>

```



```

112     /// <para></para>
113     /// </summary>
114     /// <param name="minimumStringSegmentLength">
115     /// <para>A minimum string segment length.</para>
116     /// <para></para>
117     /// </param>
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
120         ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
121
122     /// <summary>
123     /// <para>
124     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
125     /// </para>
126     /// <para></para>
127     /// </summary>
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
130         ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
131
132     /// <summary>
133     /// <para>
134     /// Walks the all using the specified elements.
135     /// </para>
136     /// <para></para>
137     /// </summary>
138     /// <param name="elements">
139     /// <para>The elements.</para>
140     /// <para></para>
141     /// </param>
142     [MethodImpl(MethodImplOptions.AggressiveInlining)]
143     public override void WalkAll(ICollection<T> elements)
144     {
145         if (_resetDictionaryOnEachWalk)
146         {
147             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
148             Dictionary = new Dictionary<TSegment, long>((int)capacity);
149         }
150         base.WalkAll(elements);
151     }
152
153     /// <summary>
154     /// <para>
155     /// Gets the segment frequency using the specified segment.
156     /// </para>
157     /// <para></para>
158     /// </summary>
159     /// <param name="segment">
160     /// <para>The segment.</para>
161     /// <para></para>
162     /// </param>
163     /// <returns>
164     /// <para>The long</para>
165     /// <para></para>
166     /// </returns>
167     [MethodImpl(MethodImplOptions.AggressiveInlining)]
168     protected override long GetSegmentFrequency(TSegment segment) =>
169         ↪ Dictionary.GetOrDefault(segment);
170
171     /// <summary>
172     /// <para>
173     /// Sets the segment frequency using the specified segment.
174     /// </para>
175     /// <para></para>
176     /// </summary>
177     /// <param name="segment">
178     /// <para>The segment.</para>
179     /// <para></para>
180     /// </param>
181     /// <param name="frequency">
182     /// <para>The frequency.</para>
183     /// <para></para>
184     /// </param>
185     [MethodImpl(MethodImplOptions.AggressiveInlining)]
186     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
187         ↪ Dictionary[segment] = frequency;
188 }

```

1.27 ./csharp/Platform.Collections.Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the dictionary based duplicate segments walker base.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{T, Segment{T}}"/>
15     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
16     DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
17     {
18         /// <summary>
19         /// <para>
20         /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         /// <param name="dictionary">
25         /// <para>A dictionary.</para>
26         /// <para></para>
27         /// </param>
28         /// <param name="minimumStringSegmentLength">
29         /// <para>A minimum string segment length.</para>
30         /// <para></para>
31         /// </param>
32         /// <param name="resetDictionaryOnEachWalk">
33         /// <para>A reset dictionary on each walk.</para>
34         /// <para></para>
35         /// </param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
38         dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
39         base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
40
41         /// <summary>
42         /// <para>
43         /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="dictionary">
48         /// <para>A dictionary.</para>
49         /// <para></para>
50         /// </param>
51         /// <param name="minimumStringSegmentLength">
52         /// <para>A minimum string segment length.</para>
53         /// <para></para>
54         /// </param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
57         dictionary, int minimumStringSegmentLength) : base(dictionary,
58         minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
59
60         /// <summary>
61         /// <para>
62         /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
63         /// </para>
64         /// <para></para>
65         /// </summary>
66         /// <param name="dictionary">
67         /// <para>A dictionary.</para>
68         /// <para></para>
69         /// </param>
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
72         dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
73         DefaultResetDictionaryOnEachWalk) { }
74
75         /// <summary>
76         /// <para>
77         /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.

```

```

71     /// </para>
72     /// <para></para>
73     /// </summary>
74     /// <param name="minimumStringSegmentLength">
75     /// <para>A minimum string segment length.</para>
76     /// <para></para>
77     /// </param>
78     /// <param name="resetDictionaryOnEachWalk">
79     /// <para>A reset dictionary on each walk.</para>
80     /// <para></para>
81     /// </param>
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
84         ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
85         ↪ resetDictionaryOnEachWalk) { }
86
87     /// <summary>
88     /// <para>
89     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
90     /// </para>
91     /// <para></para>
92     /// </summary>
93     /// <param name="minimumStringSegmentLength">
94     /// <para>A minimum string segment length.</para>
95     /// <para></para>
96     /// </param>
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
99     ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
100
101     /// <summary>
102     /// <para>
103     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
104     /// </para>
105     /// <para></para>
106     /// </summary>
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
109     ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
110 }
111 }

```

1.28 ./csharp/Platform.Collections.Segments.Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Segments.Walkers
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the duplicate segments walker base.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="AllSegmentsWalkerBase{T, TSegment}"/>
14     public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
15     ↪ TSegment>
16     where TSegment : Segment<T>
17     {
18         /// <summary>
19         /// <para>
20         /// Initializes a new <see cref="DuplicateSegmentsWalkerBase"/> instance.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         /// <param name="minimumStringSegmentLength">
25         /// <para>A minimum string segment length.</para>
26         /// <para></para>
27         /// </param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
30         ↪ base(minimumStringSegmentLength) { }
31
32         /// <summary>
33         /// <para>
34         /// Initializes a new <see cref="DuplicateSegmentsWalkerBase"/> instance.
35         /// </para>

```

```

34     /// <para></para>
35     /// </summary>
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
38
39     /// <summary>
40     /// <para>
41     /// Iterations the segment.
42     /// </para>
43     /// <para></para>
44     /// </summary>
45     /// <param name="segment">
46     /// <para>The segment.</para>
47     /// <para></para>
48     /// </param>
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected override void Iteration(TSegment segment)
51     {
52         var frequency = GetSegmentFrequency(segment);
53         if (frequency == 1)
54         {
55             OnDuplicateFound(segment);
56         }
57         SetSegmentFrequency(segment, frequency + 1);
58     }
59
60     /// <summary>
61     /// <para>
62     /// Ons the duplicate found using the specified segment.
63     /// </para>
64     /// <para></para>
65     /// </summary>
66     /// <param name="segment">
67     /// <para>The segment.</para>
68     /// <para></para>
69     /// </param>
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     protected abstract void OnDuplicateFound(TSegment segment);
72
73     /// <summary>
74     /// <para>
75     /// Gets the segment frequency using the specified segment.
76     /// </para>
77     /// <para></para>
78     /// </summary>
79     /// <param name="segment">
80     /// <para>The segment.</para>
81     /// <para></para>
82     /// </param>
83     /// <returns>
84     /// <para>The long</para>
85     /// <para></para>
86     /// </returns>
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     protected abstract long GetSegmentFrequency(TSegment segment);
89
90     /// <summary>
91     /// <para>
92     /// Sets the segment frequency using the specified segment.
93     /// </para>
94     /// <para></para>
95     /// </summary>
96     /// <param name="segment">
97     /// <para>The segment.</para>
98     /// <para></para>
99     /// </param>
100    /// <param name="frequency">
101    /// <para>The frequency.</para>
102    /// <para></para>
103    /// </param>
104    [MethodImpl(MethodImplOptions.AggressiveInlining)]
105    protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
106 }
107 }

```

1.29 ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2

```

```

3 namespace Platform.Collections.Segments.Walkers
4 {
5     /// <summary>
6     /// <para>
7     /// Represents the duplicate segments walker base.
8     /// </para>
9     /// <para></para>
10    /// </summary>
11    /// <seealso cref="DuplicateSegmentsWalkerBase{T, Segment{T}}"/>
12    public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
    ↪ Segment<T>>
13    {
14    }
15 }

```

1.30 ./csharp/Platform.Collections.Sets/ISetExtensions.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Sets
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the set extensions.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public static class ISetExtensions
15    {
16        /// <summary>
17        /// <para>
18        /// Adds the and return void using the specified set.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        /// <typeparam name="T">
23        /// <para>The .</para>
24        /// <para></para>
25        /// </typeparam>
26        /// <param name="set">
27        /// <para>The set.</para>
28        /// <para></para>
29        /// </param>
30        /// <param name="element">
31        /// <para>The element.</para>
32        /// <para></para>
33        /// </param>
34        [MethodImpl(MethodImplOptions.AggressiveInlining)]
35        public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
36
37        /// <summary>
38        /// <para>
39        /// Removes the and return void using the specified set.
40        /// </para>
41        /// <para></para>
42        /// </summary>
43        /// <typeparam name="T">
44        /// <para>The .</para>
45        /// <para></para>
46        /// </typeparam>
47        /// <param name="set">
48        /// <para>The set.</para>
49        /// <para></para>
50        /// </param>
51        /// <param name="element">
52        /// <para>The element.</para>
53        /// <para></para>
54        /// </param>
55        [MethodImpl(MethodImplOptions.AggressiveInlining)]
56        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
    ↪ set.Remove(element);
57
58        /// <summary>
59        /// <para>
60        /// Determines whether add and return true.
61        /// </para>

```

```

62     /// <para></para>
63     /// </summary>
64     /// <typeparam name="T">
65     /// <para>The .</para>
66     /// <para></para>
67     /// </typeparam>
68     /// <param name="set">
69     /// <para>The set.</para>
70     /// <para></para>
71     /// </param>
72     /// <param name="element">
73     /// <para>The element.</para>
74     /// <para></para>
75     /// </param>
76     /// <returns>
77     /// <para>The bool</para>
78     /// <para></para>
79     /// </returns>
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
82     {
83         set.Add(element);
84         return true;
85     }
86
87     /// <summary>
88     /// <para>
89     /// Determines whether add first and return true.
90     /// </para>
91     /// <para></para>
92     /// </summary>
93     /// <typeparam name="T">
94     /// <para>The .</para>
95     /// <para></para>
96     /// </typeparam>
97     /// <param name="set">
98     /// <para>The set.</para>
99     /// <para></para>
100    /// </param>
101    /// <param name="elements">
102    /// <para>The elements.</para>
103    /// <para></para>
104    /// </param>
105    /// <returns>
106    /// <para>The bool</para>
107    /// <para></para>
108    /// </returns>
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
111    {
112        AddFirst(set, elements);
113        return true;
114    }
115
116    /// <summary>
117    /// <para>
118    /// Adds the first using the specified set.
119    /// </para>
120    /// <para></para>
121    /// </summary>
122    /// <typeparam name="T">
123    /// <para>The .</para>
124    /// <para></para>
125    /// </typeparam>
126    /// <param name="set">
127    /// <para>The set.</para>
128    /// <para></para>
129    /// </param>
130    /// <param name="elements">
131    /// <para>The elements.</para>
132    /// <para></para>
133    /// </param>
134    [MethodImpl(MethodImplOptions.AggressiveInlining)]
135    public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
136        ↪ set.Add(elements[0]);
137
138    /// <summary>
139    /// <para>

```

```

139     /// Determines whether add all and return true.
140     /// </para>
141     /// <para></para>
142     /// </summary>
143     /// <typeparam name="T">
144     /// <para>The .</para>
145     /// <para></para>
146     /// </typeparam>
147     /// <param name="set">
148     /// <para>The set.</para>
149     /// <para></para>
150     /// </param>
151     /// <param name="elements">
152     /// <para>The elements.</para>
153     /// <para></para>
154     /// </param>
155     /// <returns>
156     /// <para>The bool</para>
157     /// <para></para>
158     /// </returns>
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
161     {
162         set.AddAll(elements);
163         return true;
164     }
165
166     /// <summary>
167     /// <para>
168     /// Adds the all using the specified set.
169     /// </para>
170     /// <para></para>
171     /// </summary>
172     /// <typeparam name="T">
173     /// <para>The .</para>
174     /// <para></para>
175     /// </typeparam>
176     /// <param name="set">
177     /// <para>The set.</para>
178     /// <para></para>
179     /// </param>
180     /// <param name="elements">
181     /// <para>The elements.</para>
182     /// <para></para>
183     /// </param>
184     [MethodImpl(MethodImplOptions.AggressiveInlining)]
185     public static void AddAll<T>(this ISet<T> set, IList<T> elements)
186     {
187         for (var i = 0; i < elements.Count; i++)
188         {
189             set.Add(elements[i]);
190         }
191     }
192
193     /// <summary>
194     /// <para>
195     /// Determines whether add skip first and return true.
196     /// </para>
197     /// <para></para>
198     /// </summary>
199     /// <typeparam name="T">
200     /// <para>The .</para>
201     /// <para></para>
202     /// </typeparam>
203     /// <param name="set">
204     /// <para>The set.</para>
205     /// <para></para>
206     /// </param>
207     /// <param name="elements">
208     /// <para>The elements.</para>
209     /// <para></para>
210     /// </param>
211     /// <returns>
212     /// <para>The bool</para>
213     /// <para></para>
214     /// </returns>
215     [MethodImpl(MethodImplOptions.AggressiveInlining)]
216     public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)

```

```

217 {
218     set.AddSkipFirst(elements);
219     return true;
220 }
221
222 /// <summary>
223 /// <para>
224 /// Adds the skip first using the specified set.
225 /// </para>
226 /// <para></para>
227 /// </summary>
228 /// <typeparam name="T">
229 /// <para>The .</para>
230 /// <para></para>
231 /// </typeparam>
232 /// <param name="set">
233 /// <para>The set.</para>
234 /// <para></para>
235 /// </param>
236 /// <param name="elements">
237 /// <para>The elements.</para>
238 /// <para></para>
239 /// </param>
240 [MethodImpl(MethodImplOptions.AggressiveInlining)]
241 public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
    ↪ set.AddSkipFirst(elements, 1);
242
243 /// <summary>
244 /// <para>
245 /// Adds the skip first using the specified set.
246 /// </para>
247 /// <para></para>
248 /// </summary>
249 /// <typeparam name="T">
250 /// <para>The .</para>
251 /// <para></para>
252 /// </typeparam>
253 /// <param name="set">
254 /// <para>The set.</para>
255 /// <para></para>
256 /// </param>
257 /// <param name="elements">
258 /// <para>The elements.</para>
259 /// <para></para>
260 /// </param>
261 /// <param name="skip">
262 /// <para>The skip.</para>
263 /// <para></para>
264 /// </param>
265 [MethodImpl(MethodImplOptions.AggressiveInlining)]
266 public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
267 {
268     for (var i = skip; i < elements.Count; i++)
269     {
270         set.Add(elements[i]);
271     }
272 }
273
274 /// <summary>
275 /// <para>
276 /// Determines whether do not contains.
277 /// </para>
278 /// <para></para>
279 /// </summary>
280 /// <typeparam name="T">
281 /// <para>The .</para>
282 /// <para></para>
283 /// </typeparam>
284 /// <param name="set">
285 /// <para>The set.</para>
286 /// <para></para>
287 /// </param>
288 /// <param name="element">
289 /// <para>The element.</para>
290 /// <para></para>
291 /// </param>
292 /// <returns>
293 /// <para>The bool</para>

```



```

294     /// <para></para>
295     /// </returns>
296     [MethodImpl(MethodImplOptions.AggressiveInlining)]
297     public static bool DoNotContains<T>(this ISet<T> set, T element) =>
        ↪ !set.Contains(element);
298 }
299 }

```

1.31 ./csharp/Platform.Collections/Sets/SetFiller.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Sets
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the set filler.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public class SetFiller<TElement, TReturnConstant>
15     {
16         /// <summary>
17         /// <para>
18         /// The set.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         protected readonly ISet<TElement> _set;
23         /// <summary>
24         /// <para>
25         /// The return constant.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         protected readonly TReturnConstant _returnConstant;
30
31         /// <summary>
32         /// <para>
33         /// Initializes a new <see cref="SetFiller"/> instance.
34         /// </para>
35         /// <para></para>
36         /// </summary>
37         /// <param name="set">
38         /// <para>A set.</para>
39         /// <para></para>
40         /// </param>
41         /// <param name="returnConstant">
42         /// <para>A return constant.</para>
43         /// <para></para>
44         /// </param>
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
47         {
48             _set = set;
49             _returnConstant = returnConstant;
50         }
51
52         /// <summary>
53         /// <para>
54         /// Initializes a new <see cref="SetFiller"/> instance.
55         /// </para>
56         /// <para></para>
57         /// </summary>
58         /// <param name="set">
59         /// <para>A set.</para>
60         /// <para></para>
61         /// </param>
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public SetFiller(ISet<TElement> set) : this(set, default) { }
64
65         /// <summary>
66         /// <para>
67         /// Adds the element.
68         /// </para>
69         /// <para></para>
70         /// </summary>

```

```

71     /// <param name="element">
72     /// <para>The element.</para>
73     /// </para>
74     /// </param>
75     [MethodImpl(MethodImplOptions.AggressiveInlining)]
76     public void Add(TElement element) => _set.Add(element);
77
78     /// <summary>
79     /// <para>
80     /// Determines whether this instance add and return true.
81     /// </para>
82     /// </para>
83     /// </summary>
84     /// <param name="element">
85     /// <para>The element.</para>
86     /// </para>
87     /// </param>
88     /// <returns>
89     /// <para>The bool</para>
90     /// </para>
91     /// </returns>
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);
94
95     /// <summary>
96     /// <para>
97     /// Determines whether this instance add first and return true.
98     /// </para>
99     /// </para>
100    /// </summary>
101    /// <param name="elements">
102    /// <para>The elements.</para>
103    /// </para>
104    /// </param>
105    /// <returns>
106    /// <para>The bool</para>
107    /// </para>
108    /// </returns>
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
111        ↪ _set.AddFirstAndReturnTrue(elements);
112
113    /// <summary>
114    /// <para>
115    /// Determines whether this instance add all and return true.
116    /// </para>
117    /// </para>
118    /// </summary>
119    /// <param name="elements">
120    /// <para>The elements.</para>
121    /// </para>
122    /// </param>
123    /// <returns>
124    /// <para>The bool</para>
125    /// </para>
126    /// </returns>
127    [MethodImpl(MethodImplOptions.AggressiveInlining)]
128    public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
129        ↪ _set.AddAllAndReturnTrue(elements);
130
131    /// <summary>
132    /// <para>
133    /// Determines whether this instance add skip first and return true.
134    /// </para>
135    /// </para>
136    /// </summary>
137    /// <param name="elements">
138    /// <para>The elements.</para>
139    /// </para>
140    /// </param>
141    /// <returns>
142    /// <para>The bool</para>
143    /// </para>
144    /// </returns>
145    [MethodImpl(MethodImplOptions.AggressiveInlining)]
146    public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
147        ↪ _set.AddSkipFirstAndReturnTrue(elements);

```

```

146    /// <summary>
147    /// <para>
148    /// Adds the and return constant using the specified element.
149    /// </para>
150    /// <para></para>
151    /// </summary>
152    /// <param name="element">
153    /// <para>The element.</para>
154    /// <para></para>
155    /// </param>
156    /// <returns>
157    /// <para>The return constant.</para>
158    /// <para></para>
159    /// </returns>
160    [MethodImpl(MethodImplOptions.AggressiveInlining)]
161    public TReturnConstant AddAndReturnConstant(TElement element)
162    {
163        _set.Add(element);
164        return _returnConstant;
165    }
166
167    /// <summary>
168    /// <para>
169    /// Adds the first and return constant using the specified elements.
170    /// </para>
171    /// <para></para>
172    /// </summary>
173    /// <param name="elements">
174    /// <para>The elements.</para>
175    /// <para></para>
176    /// </param>
177    /// <returns>
178    /// <para>The return constant.</para>
179    /// <para></para>
180    /// </returns>
181    [MethodImpl(MethodImplOptions.AggressiveInlining)]
182    public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
183    {
184        _set.AddFirst(elements);
185        return _returnConstant;
186    }
187
188    /// <summary>
189    /// <para>
190    /// Adds the all and return constant using the specified elements.
191    /// </para>
192    /// <para></para>
193    /// </summary>
194    /// <param name="elements">
195    /// <para>The elements.</para>
196    /// <para></para>
197    /// </param>
198    /// <returns>
199    /// <para>The return constant.</para>
200    /// <para></para>
201    /// </returns>
202    [MethodImpl(MethodImplOptions.AggressiveInlining)]
203    public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
204    {
205        _set.AddAll(elements);
206        return _returnConstant;
207    }
208
209    /// <summary>
210    /// <para>
211    /// Adds the skip first and return constant using the specified elements.
212    /// </para>
213    /// <para></para>
214    /// </summary>
215    /// <param name="elements">
216    /// <para>The elements.</para>
217    /// <para></para>
218    /// </param>
219    /// <returns>
220    /// <para>The return constant.</para>
221    /// <para></para>
222    /// </returns>
223    [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

224         public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements)
225         {
226             _set.AddSkipFirst(elements);
227             return _returnConstant;
228         }
229     }
230 }

```

1.32 ./csharp/Platform.Collections/Stacks/DefaultStack.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Stacks
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the default stack.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="Stack{TElement}"/>
15     /// <seealso cref="IStack{TElement}"/>
16     public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
17     {
18         /// <summary>
19         /// <para>
20         /// Gets the is empty value.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public bool IsEmpty
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get => Count <= 0;
28         }
29     }
30 }

```

1.33 ./csharp/Platform.Collections/Stacks/IStack.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Stacks
6  {
7      /// <summary>
8      /// <para>
9      /// Defines the stack.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     public interface IStack<TElement>
14     {
15         /// <summary>
16         /// <para>
17         /// Gets the is empty value.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         bool IsEmpty
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get;
25         }
26
27         /// <summary>
28         /// <para>
29         /// Pushes the element.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         /// <param name="element">
34         /// <para>The element.</para>
35         /// <para></para>
36         /// </param>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         void Push(TElement element);

```

```

39
40     /// <summary>
41     /// <para>
42     /// Pops this instance.
43     /// </para>
44     /// <para></para>
45     /// </summary>
46     /// <returns>
47     /// <para>The element</para>
48     /// <para></para>
49     /// </returns>
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     TElement Pop();
52
53     /// <summary>
54     /// <para>
55     /// Peeks this instance.
56     /// </para>
57     /// <para></para>
58     /// </summary>
59     /// <returns>
60     /// <para>The element</para>
61     /// <para></para>
62     /// </returns>
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     TElement Peek();
65 }
66 }

```

1.34 ./csharp/Platform.Collections/Stacks/IStackExtensions.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Stacks
6  {
7      /// <summary>
8      /// <para>
9      /// Represents the stack extensions.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     public static class IStackExtensions
14     {
15         /// <summary>
16         /// <para>
17         /// Clears the stack.
18         /// </para>
19         /// <para></para>
20         /// </summary>
21         /// <typeparam name="T">
22         /// <para>The .</para>
23         /// <para></para>
24         /// </typeparam>
25         /// <param name="stack">
26         /// <para>The stack.</para>
27         /// <para></para>
28         /// </param>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public static void Clear<T>(this IStack<T> stack)
31         {
32             while (!stack.IsEmpty)
33             {
34                 _ = stack.Pop();
35             }
36         }
37
38         /// <summary>
39         /// <para>
40         /// Pops the or default using the specified stack.
41         /// </para>
42         /// <para></para>
43         /// </summary>
44         /// <typeparam name="T">
45         /// <para>The .</para>
46         /// <para></para>
47         /// </typeparam>
48         /// <param name="stack">

```

```

49     /// <para>The stack.</para>
50     /// <para></para>
51     /// </param>
52     /// <returns>
53     /// <para>The</para>
54     /// <para></para>
55     /// </returns>
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪ stack.Pop();
58
59     /// <summary>
60     /// <para>
61     /// Peeks the or default using the specified stack.
62     /// </para>
63     /// <para></para>
64     /// </summary>
65     /// <typeparam name="T">
66     /// <para>The .</para>
67     /// <para></para>
68     /// </typeparam>
69     /// <param name="stack">
70     /// <para>The stack.</para>
71     /// <para></para>
72     /// </param>
73     /// <returns>
74     /// <para>The</para>
75     /// <para></para>
76     /// </returns>
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪ stack.Peek();
79 }
80 }

```

1.35 ./csharp/Platform.Collections/Stacks/IStackFactory.cs

```

1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Stacks
6  {
7      /// <summary>
8      /// <para>
9      /// Defines the stack factory.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="IFactory{IStack{TElement}}"/>
14     public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
15     {
16     }
17 }

```

1.36 ./csharp/Platform.Collections/Stacks/StackExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Stacks
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the stack extensions.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class StackExtensions
15     {
16         /// <summary>
17         /// <para>
18         /// Pops the or default using the specified stack.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <typeparam name="T">
23         /// <para>The .</para>

```

```

24     /// <para></para>
25     /// </typeparam>
26     /// <param name="stack">
27     /// <para>The stack.</para>
28     /// <para></para>
29     /// </param>
30     /// <returns>
31     /// <para>The</para>
32     /// <para></para>
33     /// </returns>
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
        ↪ default;
36
37     /// <summary>
38     /// <para>
39     /// Peeks the or default using the specified stack.
40     /// </para>
41     /// <para></para>
42     /// </summary>
43     /// <typeparam name="T">
44     /// <para>The .</para>
45     /// <para></para>
46     /// </typeparam>
47     /// <param name="stack">
48     /// <para>The stack.</para>
49     /// <para></para>
50     /// </param>
51     /// <returns>
52     /// <para>The</para>
53     /// <para></para>
54     /// </returns>
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
        ↪ : default;
57 }
58 }

```

1.37 ./csharp/Platform.Collections/StringExtensions.cs

```

1  using System;
2  using System.Globalization;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the string extensions.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     public static class StringExtensions
16     {
17         /// <summary>
18         /// <para>
19         /// Capitalizes the first letter using the specified string.
20         /// </para>
21         /// <para></para>
22         /// </summary>
23         /// <param name="@string">
24         /// <para>The string.</para>
25         /// <para></para>
26         /// </param>
27         /// <returns>
28         /// <para>The string.</para>
29         /// <para></para>
30         /// </returns>
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public static string CapitalizeFirstLetter(this string @string)
33         {
34             if (string.IsNullOrEmpty(@string))
35             {
36                 return @string;
37             }
38             var chars = @string.ToCharArray();
39             for (var i = 0; i < chars.Length; i++)

```

```

40     {
41         var category = char.GetUnicodeCategory(chars[i]);
42         if (category == UnicodeCategory.UppercaseLetter)
43         {
44             return @string;
45         }
46         if (category == UnicodeCategory.LowercaseLetter)
47         {
48             chars[i] = char.ToUpper(chars[i]);
49             return new string(chars);
50         }
51     }
52     return @string;
53 }
54
55 /// <summary>
56 /// <para>
57 /// Truncates the string.
58 /// </para>
59 /// <para></para>
60 /// </summary>
61 /// <param name="@string">
62 /// <para>The string.</para>
63 /// <para></para>
64 /// </param>
65 /// <param name="maxLength">
66 /// <para>The max length.</para>
67 /// <para></para>
68 /// </param>
69 /// <returns>
70 /// <para>The string</para>
71 /// <para></para>
72 /// </returns>
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public static string Truncate(this string @string, int maxLength) =>
75     ↪ string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
76     ↪ Math.Min(@string.Length, maxLength));
77
78 /// <summary>
79 /// <para>
80 /// Trims the single using the specified string.
81 /// </para>
82 /// <para></para>
83 /// </summary>
84 /// <param name="@string">
85 /// <para>The string.</para>
86 /// <para></para>
87 /// </param>
88 /// <param name="charToTrim">
89 /// <para>The char to trim.</para>
90 /// <para></para>
91 /// </param>
92 /// <returns>
93 /// <para>The string</para>
94 /// <para></para>
95 /// </returns>
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public static string TrimSingle(this string @string, char charToTrim)
98 {
99     if (!string.IsNullOrEmpty(@string))
100     {
101         if (@string.Length == 1)
102         {
103             if (@string[0] == charToTrim)
104             {
105                 return "";
106             }
107             else
108             {
109                 return @string;
110             }
111         }
112         else
113         {
114             var left = 0;
115             var right = @string.Length - 1;
116             if (@string[left] == charToTrim)
117             {

```



```

116         left++;
117     }
118     if (@string[right] == charToTrim)
119     {
120         right--;
121     }
122     return @string.Substring(left, right - left + 1);
123 }
124 }
125 else
126 {
127     return @string;
128 }
129 }
130 }
131 }

```

1.38 ./csharp/Platform.Collections/Trees/Node.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 // ReSharper disable ForCanBeConvertedToForeach
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Trees
8 {
9     /// <summary>
10    /// <para>
11    /// Represents the node.
12    /// </para>
13    /// <para></para>
14    /// </summary>
15    public class Node
16    {
17        private Dictionary<object, Node> _childNodes;
18
19        /// <summary>
20        /// <para>
21        /// Gets or sets the value value.
22        /// </para>
23        /// <para></para>
24        /// </summary>
25        public object Value
26        {
27            [MethodImpl(MethodImplOptions.AggressiveInlining)]
28            get;
29            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30            set;
31        }
32
33        /// <summary>
34        /// <para>
35        /// Gets the child nodes value.
36        /// </para>
37        /// <para></para>
38        /// </summary>
39        public Dictionary<object, Node> ChildNodes
40        {
41            [MethodImpl(MethodImplOptions.AggressiveInlining)]
42            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
43        }
44
45        /// <summary>
46        /// <para>
47        /// The key.
48        /// </para>
49        /// <para></para>
50        /// </summary>
51        public Node this[object key]
52        {
53            [MethodImpl(MethodImplOptions.AggressiveInlining)]
54            get => GetChild(key) ?? AddChild(key);
55            [MethodImpl(MethodImplOptions.AggressiveInlining)]
56            set => SetChildValue(value, key);
57        }
58
59        /// <summary>
60        /// <para>
61        /// Initializes a new <see cref="Node"/> instance.

```

```

62     /// </para>
63     /// <para></para>
64     /// </summary>
65     /// <param name="value">
66     /// <para>A value.</para>
67     /// <para></para>
68     /// </param>
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     public Node(object value) => Value = value;
71
72     /// <summary>
73     /// <para>
74     /// Initializes a new <see cref="Node"/> instance.
75     /// </para>
76     /// <para></para>
77     /// </summary>
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     public Node() : this(null) { }
80
81     /// <summary>
82     /// <para>
83     /// Determines whether this instance contains child.
84     /// </para>
85     /// <para></para>
86     /// </summary>
87     /// <param name="keys">
88     /// <para>The keys.</para>
89     /// <para></para>
90     /// </param>
91     /// <returns>
92     /// <para>The bool</para>
93     /// <para></para>
94     /// </returns>
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
97
98     /// <summary>
99     /// <para>
100    /// Gets the child using the specified keys.
101    /// </para>
102    /// <para></para>
103    /// </summary>
104    /// <param name="keys">
105    /// <para>The keys.</para>
106    /// <para></para>
107    /// </param>
108    /// <returns>
109    /// <para>The node.</para>
110    /// <para></para>
111    /// </returns>
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    public Node GetChild(params object[] keys)
114    {
115        var node = this;
116        for (var i = 0; i < keys.Length; i++)
117        {
118            node.ChildNodes.TryGetValue(keys[i], out node);
119            if (node == null)
120            {
121                return null;
122            }
123        }
124        return node;
125    }
126
127    /// <summary>
128    /// <para>
129    /// Gets the child value using the specified keys.
130    /// </para>
131    /// <para></para>
132    /// </summary>
133    /// <param name="keys">
134    /// <para>The keys.</para>
135    /// <para></para>
136    /// </param>
137    /// <returns>
138    /// <para>The object</para>
139    /// <para></para>

```

```

140     /// </returns>
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
143
144     /// <summary>
145     /// <para>
146     /// Adds the child using the specified key.
147     /// </para>
148     /// <para></para>
149     /// </summary>
150     /// <param name="key">
151     /// <para>The key.</para>
152     /// <para></para>
153     /// </param>
154     /// <returns>
155     /// <para>The node</para>
156     /// <para></para>
157     /// </returns>
158     [MethodImpl(MethodImplOptions.AggressiveInlining)]
159     public Node AddChild(object key) => AddChild(key, new Node(null));
160
161     /// <summary>
162     /// <para>
163     /// Adds the child using the specified key.
164     /// </para>
165     /// <para></para>
166     /// </summary>
167     /// <param name="key">
168     /// <para>The key.</para>
169     /// <para></para>
170     /// </param>
171     /// <param name="value">
172     /// <para>The value.</para>
173     /// <para></para>
174     /// </param>
175     /// <returns>
176     /// <para>The node</para>
177     /// <para></para>
178     /// </returns>
179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
180     public Node AddChild(object key, object value) => AddChild(key, new Node(value));
181
182     /// <summary>
183     /// <para>
184     /// Adds the child using the specified key.
185     /// </para>
186     /// <para></para>
187     /// </summary>
188     /// <param name="key">
189     /// <para>The key.</para>
190     /// <para></para>
191     /// </param>
192     /// <param name="child">
193     /// <para>The child.</para>
194     /// <para></para>
195     /// </param>
196     /// <returns>
197     /// <para>The child.</para>
198     /// <para></para>
199     /// </returns>
200     [MethodImpl(MethodImplOptions.AggressiveInlining)]
201     public Node AddChild(object key, Node child)
202     {
203         ChildNodes.Add(key, child);
204         return child;
205     }
206
207     /// <summary>
208     /// <para>
209     /// Sets the child using the specified keys.
210     /// </para>
211     /// <para></para>
212     /// </summary>
213     /// <param name="keys">
214     /// <para>The keys.</para>
215     /// <para></para>
216     /// </param>
217     /// </returns>

```

```

218 /// <para>The node</para>
219 /// <para></para>
220 /// </returns>
221 [MethodImpl(MethodImplOptions.AggressiveInlining)]
222 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
223
224 /// <summary>
225 /// <para>
226 /// Sets the child using the specified key.
227 /// </para>
228 /// <para></para>
229 /// </summary>
230 /// <param name="key">
231 /// <para>The key.</para>
232 /// <para></para>
233 /// </param>
234 /// <returns>
235 /// <para>The node</para>
236 /// <para></para>
237 /// </returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public Node SetChild(object key) => SetChildValue(null, key);
240
241 /// <summary>
242 /// <para>
243 /// Sets the child value using the specified value.
244 /// </para>
245 /// <para></para>
246 /// </summary>
247 /// <param name="value">
248 /// <para>The value.</para>
249 /// <para></para>
250 /// </param>
251 /// <param name="keys">
252 /// <para>The keys.</para>
253 /// <para></para>
254 /// </param>
255 /// <returns>
256 /// <para>The node.</para>
257 /// <para></para>
258 /// </returns>
259 [MethodImpl(MethodImplOptions.AggressiveInlining)]
260 public Node SetChildValue(object value, params object[] keys)
261 {
262     var node = this;
263     for (var i = 0; i < keys.Length; i++)
264     {
265         node = SetChildValue(value, keys[i]);
266     }
267     node.Value = value;
268     return node;
269 }
270
271 /// <summary>
272 /// <para>
273 /// Sets the child value using the specified value.
274 /// </para>
275 /// <para></para>
276 /// </summary>
277 /// <param name="value">
278 /// <para>The value.</para>
279 /// <para></para>
280 /// </param>
281 /// <param name="key">
282 /// <para>The key.</para>
283 /// <para></para>
284 /// </param>
285 /// <returns>
286 /// <para>The child.</para>
287 /// <para></para>
288 /// </returns>
289 [MethodImpl(MethodImplOptions.AggressiveInlining)]
290 public Node SetChildValue(object value, object key)
291 {
292     if (!ChildNodes.TryGetValue(key, out Node child))
293     {
294         child = AddChild(key, value);
295     }

```

```

296         child.Value = value;
297         return child;
298     }
299 }
300 }

```

1.39 ./csharp/Platform.Collections.Tests/ArrayTests.cs

```

1  using Xunit;
2  using Platform.Collections.Arrays;
3
4  namespace Platform.Collections.Tests
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the array tests.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public class ArrayTests
13     {
14         /// <summary>
15         /// <para>
16         /// Tests that get element test.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         [Fact]
21         public void GetElementTest()
22         {
23             var nullArray = (int[])null;
24             Assert.Equal(0, nullArray.GetElementOrDefault(1));
25             Assert.False(nullArray.TryGetElement(1, out int element));
26             Assert.Equal(0, element);
27             var array = new int[] { 1, 2, 3 };
28             Assert.Equal(3, array.GetElementOrDefault(2));
29             Assert.True(array.TryGetElement(2, out element));
30             Assert.Equal(3, element);
31             Assert.Equal(0, array.GetElementOrDefault(10));
32             Assert.False(array.TryGetElement(10, out element));
33             Assert.Equal(0, element);
34         }
35     }
36 }

```

1.40 ./csharp/Platform.Collections.Tests/BitStringTests.cs

```

1  using System;
2  using System.Collections;
3  using Xunit;
4  using Platform.Random;
5
6  namespace Platform.Collections.Tests
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the bit string tests.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class BitStringTests
15     {
16         /// <summary>
17         /// <para>
18         /// Tests that bit get set test.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         [Fact]
23         public static void BitGetSetTest()
24         {
25             const int n = 250;
26             var bitArray = new BitArray(n);
27             var bitString = new BitString(n);
28             for (var i = 0; i < n; i++)
29             {
30                 var value = RandomHelpers.Default.NextBoolean();
31                 bitArray.Set(i, value);
32                 bitString.Set(i, value);
33                 Assert.Equal(value, bitArray.Get(i));

```

```

34         Assert.Equal(value, bitString.Get(i));
35     }
36 }
37
38 /// <summary>
39 /// <para>
40 /// Tests that bit vector not test.
41 /// </para>
42 /// <para></para>
43 /// </summary>
44 [Fact]
45 public static void BitVectorNotTest()
46 {
47     TestToOperationsWithSameMeaning((x, y, w, v) =>
48     {
49         x.VectorNot();
50         w.Not();
51     });
52 }
53
54 /// <summary>
55 /// <para>
56 /// Tests that bit parallel not test.
57 /// </para>
58 /// <para></para>
59 /// </summary>
60 [Fact]
61 public static void BitParallelNotTest()
62 {
63     TestToOperationsWithSameMeaning((x, y, w, v) =>
64     {
65         x.ParallelNot();
66         w.Not();
67     });
68 }
69
70 /// <summary>
71 /// <para>
72 /// Tests that bit parallel vector not test.
73 /// </para>
74 /// <para></para>
75 /// </summary>
76 [Fact]
77 public static void BitParallelVectorNotTest()
78 {
79     TestToOperationsWithSameMeaning((x, y, w, v) =>
80     {
81         x.ParallelVectorNot();
82         w.Not();
83     });
84 }
85
86 /// <summary>
87 /// <para>
88 /// Tests that bit vector and test.
89 /// </para>
90 /// <para></para>
91 /// </summary>
92 [Fact]
93 public static void BitVectorAndTest()
94 {
95     TestToOperationsWithSameMeaning((x, y, w, v) =>
96     {
97         x.VectorAnd(y);
98         w.And(v);
99     });
100 }
101
102 /// <summary>
103 /// <para>
104 /// Tests that bit parallel and test.
105 /// </para>
106 /// <para></para>
107 /// </summary>
108 [Fact]
109 public static void BitParallelAndTest()
110 {
111     TestToOperationsWithSameMeaning((x, y, w, v) =>

```

```

112     {
113         x.ParallelAnd(y);
114         w.And(v);
115     });
116 }
117
118 /// <summary>
119 /// <para>
120 /// Tests that bit parallel vector and test.
121 /// </para>
122 /// <para></para>
123 /// </summary>
124 [Fact]
125 public static void BitParallelVectorAndTest()
126 {
127     TestToOperationsWithSameMeaning((x, y, w, v) =>
128     {
129         x.ParallelVectorAnd(y);
130         w.And(v);
131     });
132 }
133
134 /// <summary>
135 /// <para>
136 /// Tests that bit vector or test.
137 /// </para>
138 /// <para></para>
139 /// </summary>
140 [Fact]
141 public static void BitVectorOrTest()
142 {
143     TestToOperationsWithSameMeaning((x, y, w, v) =>
144     {
145         x.VectorOr(y);
146         w.Or(v);
147     });
148 }
149
150 /// <summary>
151 /// <para>
152 /// Tests that bit parallel or test.
153 /// </para>
154 /// <para></para>
155 /// </summary>
156 [Fact]
157 public static void BitParallelOrTest()
158 {
159     TestToOperationsWithSameMeaning((x, y, w, v) =>
160     {
161         x.ParallelOr(y);
162         w.Or(v);
163     });
164 }
165
166 /// <summary>
167 /// <para>
168 /// Tests that bit parallel vector or test.
169 /// </para>
170 /// <para></para>
171 /// </summary>
172 [Fact]
173 public static void BitParallelVectorOrTest()
174 {
175     TestToOperationsWithSameMeaning((x, y, w, v) =>
176     {
177         x.ParallelVectorOr(y);
178         w.Or(v);
179     });
180 }
181
182 /// <summary>
183 /// <para>
184 /// Tests that bit vector xor test.
185 /// </para>
186 /// <para></para>
187 /// </summary>
188 [Fact]
189 public static void BitVectorXorTest()

```

```

190     {
191         TestToOperationsWithSameMeaning((x, y, w, v) =>
192         {
193             x.VectorXor(y);
194             w.Xor(v);
195         });
196     }
197
198     /// <summary>
199     /// <para>
200     /// Tests that bit parallel xor test.
201     /// </para>
202     /// <para></para>
203     /// </summary>
204     [Fact]
205     public static void BitParallelXorTest()
206     {
207         TestToOperationsWithSameMeaning((x, y, w, v) =>
208         {
209             x.ParallelXor(y);
210             w.Xor(v);
211         });
212     }
213
214     /// <summary>
215     /// <para>
216     /// Tests that bit parallel vector xor test.
217     /// </para>
218     /// <para></para>
219     /// </summary>
220     [Fact]
221     public static void BitParallelVectorXorTest()
222     {
223         TestToOperationsWithSameMeaning((x, y, w, v) =>
224         {
225             x.ParallelVectorXor(y);
226             w.Xor(v);
227         });
228     }
229     private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
230     ↪ BitString, BitString> test)
231     {
232         const int n = 5654;
233         var x = new BitString(n);
234         var y = new BitString(n);
235         while (x.Equals(y))
236         {
237             x.SetRandomBits();
238             y.SetRandomBits();
239         }
240         var w = new BitString(x);
241         var v = new BitString(y);
242         Assert.False(x.Equals(y));
243         Assert.False(w.Equals(v));
244         Assert.True(x.Equals(w));
245         Assert.True(y.Equals(v));
246         test(x, y, w, v);
247         Assert.True(x.Equals(w));
248     }
249 }

```

1.41 ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs

```

1  using Xunit;
2  using Platform.Collections.Segments;
3
4  namespace Platform.Collections.Tests
5  {
6      /// <summary>
7      /// <para>
8      /// Represents the chars segment tests.
9      /// </para>
10     /// <para></para>
11     /// </summary>
12     public static class CharsSegmentTests
13     {
14         /// <summary>
15         /// <para>

```



```

16     /// Tests that get hash code equals test.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     [Fact]
21     public static void GetHashCodeEqualsTest()
22     {
23         const string testString = "test test";
24         var testArray = testString.ToCharArray();
25         var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
26         var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
27         Assert.Equal(firstHashCode, secondHashCode);
28     }
29
30     /// <summary>
31     /// <para>
32     /// Tests that equals test.
33     /// </para>
34     /// <para></para>
35     /// </summary>
36     [Fact]
37     public static void EqualsTest()
38     {
39         const string testString = "test test";
40         var testArray = testString.ToCharArray();
41         var first = new CharSegment(testArray, 0, 4);
42         var second = new CharSegment(testArray, 5, 4);
43         Assert.True(first.Equals(second));
44     }
45 }
46 }

```

1.42 ./csharp/Platform.Collections.Tests/ListTests.cs

```

1 using System.Collections.Generic;
2 using Xunit;
3 using Platform.Collections.Lists;
4
5
6 namespace Platform.Collections.Tests
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the list tests.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public class ListTests
15    {
16        /// <summary>
17        /// <para>
18        /// Tests that get element test.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        [Fact]
23        public void GetElementTest()
24        {
25            var nullList = (IList<int>)null;
26            Assert.Equal(0, nullList.GetElementOrDefault(1));
27            Assert.False(nullList.TryGetElement(1, out int element));
28            Assert.Equal(0, element);
29            var list = new List<int>() { 1, 2, 3 };
30            Assert.Equal(3, list.GetElementOrDefault(2));
31            Assert.True(list.TryGetElement(2, out element));
32            Assert.Equal(3, element);
33            Assert.Equal(0, list.GetElementOrDefault(10));
34            Assert.False(list.TryGetElement(10, out element));
35            Assert.Equal(0, element);
36        }
37    }
38 }

```

1.43 ./csharp/Platform.Collections.Tests/StringTests.cs

```

1 using Xunit;
2
3 namespace Platform.Collections.Tests
4 {
5     /// <summary>

```

```

6     /// <para>
7     /// Represents the string tests.
8     /// </para>
9     /// <para></para>
10    /// </summary>
11    public static class StringTests
12    {
13        /// <summary>
14        /// <para>
15        /// Tests that capitalize first letter test.
16        /// </para>
17        /// <para></para>
18        /// </summary>
19        [Fact]
20        public static void CapitalizeFirstLetterTest()
21        {
22            Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
23            Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
24            Assert.Equal(" Hello", " hello".CapitalizeFirstLetter());
25        }
26
27        /// <summary>
28        /// <para>
29        /// Tests that trim single test.
30        /// </para>
31        /// <para></para>
32        /// </summary>
33        [Fact]
34        public static void TrimSingleTest()
35        {
36            Assert.Equal("", "".TrimSingle('\'));
37            Assert.Equal("", "''.TrimSingle('\'));
38            Assert.Equal("hello", "'hello'.TrimSingle('\'));
39            Assert.Equal("hello", "hello".TrimSingle('\'));
40            Assert.Equal("hello", "'hello'.TrimSingle('\'));
41        }
42    }
43 }

```

1.44 ./csharp/Platform.Collections.Tests/WalkersTests.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using Platform.Collections.Segments;
5  using Platform.Collections.Segments.Walkers;
6  using Platform.Collections.Trees;
7  using Xunit;
8  using Xunit.Abstractions;
9
10
11 namespace Platform.Collections.Tests
12 {
13     /// <summary>
14     /// <para>
15     /// Represents the all repeating substrings in string.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     public class AllRepeatingSubstringsInString
20     {
21         private static readonly string elfen_lied = @"Nacht im Dorf der Wächter rief: Elfe! Ein
                ↳ ganz kleines Elfchen im Walde schlief wohl um die Elfe! Und meint, es rief ihm aus
                ↳ dem Tal bei seinem Namen die Nachtigall, oder Silpelit hätt' ihm gerufen.
Reibt sich der Elf' die Augen aus, begibt sich vor sein Schneckenhaus und ist als wie ein
                ↳ trunken Mann, sein Schläflein war nicht voll getan, und humpelt also tippe tapp durch's
                ↳ Haselholz in's Tal hinab, schlupft an der Mauer hin so dicht, da sitzt der Glühwurm Licht an
                ↳ Licht.
23 Was sind das helle Fensterlein? Da drin wird eine Hochzeit sein: die Kleinen sitzen bei'm Mahle,
                ↳ und treiben's in dem Saale. Da guck' ich wohl ein wenig 'nein!""
24 Pfui, stößt den Kopf an harten Stein! Elfe, gelt, du hast genug? Gukuk!";
25         private static readonly string _exampleText =
26             @"([english version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
                ↳ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
                ↳ где есть место для нового начала? Разве пустота это не характеристика пространства?
                ↳ Пространство это то, что можно чем-то наполнить?
28 [!чёрное пространство, белое
                ↳ пространство] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
                ↳ ""чёрное пространство, белое пространство"") (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)

```

29 Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
→ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

30 [![чёрное пространство, чёрная
→ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
→ "чёрное пространство, чёрная
→ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)

31 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
→ так? Инверсия? Отражение? Сумма?

32 [![белая точка, чёрная
→ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png "белая
→ точка, чёрная
→ точка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

33 А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
→ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
→ Гранью? Разделителем? Единицей?

34 [![две белые точки, чёрная вертикальная
→ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png "две
→ белые точки, чёрная вертикальная
→ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

35 Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
→ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
→ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
→ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
→ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
→ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

36 [![белая вертикальная линия, чёрный
→ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png "белая
→ вертикальная линия, чёрный
→ круг")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

37 Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
→ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
→ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
→ элементарная единица смысла?

38 [![белый круг, чёрная горизонтальная
→ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png "белый
→ круг, чёрная горизонтальная
→ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

39 Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла "соединить,
→ связать", есть ещё и смысл направления "от начала к концу"? От предка к потомку? От
→ родителя к ребёнку? От общего к частному?

40 [![белая горизонтальная линия, чёрная горизонтальная
→ стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png
→ "белая горизонтальная линия, чёрная горизонтальная
→ стрелка")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

41 Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она
→ может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть
→ граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два
→ объекта, как бы это выглядело?

42 [![белая связь, чёрная направленная
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png "белая
→ связь, чёрная направленная
→ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)

43 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли
→ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
→ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
→ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
→ его конечном состоянии, если конечно конец определён направлением?

44 [![белая обычная и направленная связи, чёрная типизированная
→ связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png "белая
→ обычная и направленная связи, чёрная типизированная
→ связь")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)

45 А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри?
→ Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать
→ сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?

46 [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная
→ связь с рекурсивной внутренней
→ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png
→ "белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная
→ типизированная связь с рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)

47 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
→ рекурсии или фрактала?

48 [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
→ типизированная связь с двойной рекурсивной внутренней
→ структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png
→ "белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная
→ типизированная связь с двойной рекурсивной внутренней структурой")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)

```

49 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
   ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
50 [[белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
   ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
   ↳ /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
   ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
   ↳ типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
   ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
51
52 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
   ↳ ation-500.gif
   ↳ "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
   ↳ -animation-500.gif)];
53     private static readonly string _examTpleText = @"Lorem ipsum dolor sit amet, consectetur
   ↳ adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
   ↳ Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip
   ↳ ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit
   ↳ esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
   ↳ proident, sunt in culpa qui officia deserunt mollit anim id est laborum.";
54
55     /// <summary>
56     /// <para>
57     /// Tests that console tests.
58     /// </para>
59     /// <para></para>
60     /// </summary>
61     [Fact]
62     public void ConsoleTests()
63     {
64         string text = elfen_lied;
65
66         var iterationsCounter = new IterationsCounter();
67         iterationsCounter.WalkAll(text);
68         var result = iterationsCounter.IterationsCount;
69         Console.WriteLine($"{TextLength: {text.Length}. Iterations: {result}.");
70
71         {
72             var start = new Stopwatch();
73             start.Start();
74
75             var walker = new Walker4();
76             walker.WalkAll(text);
77
78             //foreach (var (key, value) in walker.PublicDictionary)
79             //{
80                 Console.WriteLine($"{key} {value}");
81             //}
82
83             start.Stop();
84             Console.WriteLine($"{start.ElapsedMilliseconds}ms");
85         }
86
87         {
88             var start = new Stopwatch();
89             start.Start();
90
91             var walker = new Walker2();
92             walker.WalkAll(text);
93
94             //foreach (var (key, value) in walker._cache)
95             //{
96                 Console.WriteLine($"{key} {value}");
97             //}
98
99             start.Stop();
100            Console.WriteLine($"{start.ElapsedMilliseconds}ms");
101        }
102
103        {
104            var start = new Stopwatch();
105            start.Start();
106
107            var walker = new Walker1();
108            walker.WalkAll(text);
109
110            start.Stop();
111            Console.WriteLine($"{start.ElapsedMilliseconds}ms");
112        }
113    }

```

```

114     }
115 }
116
117 /// <summary>
118 /// <para>
119 /// Represents the console printed duplicate walker base.
120 /// </para>
121 /// <para></para>
122 /// </summary>
123 /// <seealso cref="DuplicateSegmentsWalkerBase{char, CharSegment}"/>
124 public abstract class ConsolePrintedDuplicateWalkerBase : DuplicateSegmentsWalkerBase<char,
    ↪ CharSegment>
125 {
126     //protected override void OnDuplicateFound(CharSegment segment) =>
    ↪ Console.WriteLine(segment);
127
128     /// <summary>
129     /// <para>
130     /// Creates the segment using the specified elements.
131     /// </para>
132     /// <para></para>
133     /// </summary>
134     /// <param name="elements">
135     /// <para>The elements.</para>
136     /// <para></para>
137     /// </param>
138     /// <param name="offset">
139     /// <para>The offset.</para>
140     /// <para></para>
141     /// </param>
142     /// <param name="length">
143     /// <para>The length.</para>
144     /// <para></para>
145     /// </param>
146     /// <returns>
147     /// <para>The char segment</para>
148     /// <para></para>
149     /// </returns>
150     protected override CharSegment CreateSegment(IList<char> elements, int offset, int
    ↪ length) => new CharSegment(elements, offset, length);
151 }
152
153 /// <summary>
154 /// <para>
155 /// Represents the walker.
156 /// </para>
157 /// <para></para>
158 /// </summary>
159 /// <seealso cref="ConsolePrintedDuplicateWalkerBase"/>
160 public class Walker1 : ConsolePrintedDuplicateWalkerBase
161 {
162     private Node _rootNode;
163     private Node _currentNode;
164
165     /// <summary>
166     /// <para>
167     /// Walks the all using the specified elements.
168     /// </para>
169     /// <para></para>
170     /// </summary>
171     /// <param name="elements">
172     /// <para>The elements.</para>
173     /// <para></para>
174     /// </param>
175     public override void WalkAll(IList<char> elements)
176     {
177         _rootNode = new Node();
178
179         base.WalkAll(elements);
180
181         Console.WriteLine(_rootNode.Value);
182     }
183
184     /// <summary>
185     /// <para>
186     /// Ons the duplicate found using the specified segment.
187     /// </para>
188     /// <para></para>

```

```

189     /// </summary>
190     /// <param name="segment">
191     /// <para>The segment.</para>
192     /// <para></para>
193     /// </param>
194     protected override void OnDuplicateFound(CharSegment segment)
195     {
196
197     }
198
199     /// <summary>
200     /// <para>
201     /// Gets the segment frequency using the specified segment.
202     /// </para>
203     /// <para></para>
204     /// </summary>
205     /// <param name="segment">
206     /// <para>The segment.</para>
207     /// <para></para>
208     /// </param>
209     /// <returns>
210     /// <para>The long</para>
211     /// <para></para>
212     /// </returns>
213     protected override long GetSegmentFrequency(CharSegment segment)
214     {
215         for (int i = 0; i < segment.Length; i++)
216         {
217             var element = segment[i];
218
219             _currentNode = _currentNode[element];
220         }
221
222         if (_currentNode.Value is int)
223         {
224             return (int)_currentNode.Value;
225         }
226         else
227         {
228             return 0;
229         }
230     }
231
232     /// <summary>
233     /// <para>
234     /// Sets the segment frequency using the specified segment.
235     /// </para>
236     /// <para></para>
237     /// </summary>
238     /// <param name="segment">
239     /// <para>The segment.</para>
240     /// <para></para>
241     /// </param>
242     /// <param name="frequency">
243     /// <para>The frequency.</para>
244     /// <para></para>
245     /// </param>
246     protected override void SetSegmentFrequency(CharSegment segment, long frequency) =>
247     ↪ _currentNode.Value = frequency;
248
249     /// <summary>
250     /// <para>
251     /// Iterations the segment.
252     /// </para>
253     /// <para></para>
254     /// </summary>
255     /// <param name="segment">
256     /// <para>The segment.</para>
257     /// <para></para>
258     /// </param>
259     protected override void Iteration(CharSegment segment)
260     {
261         _currentNode = _rootNode;
262
263         base.Iteration(segment);
264     }
265
266     // Too much memory, but fast

```

```

267 /// <summary>
268 /// <para>
269 /// Represents the walker.
270 /// </para>
271 /// <para></para>
272 /// </summary>
273 /// <seealso cref="ConsolePrintedDuplicateWalkerBase"/>
274 public class Walker2 : ConsolePrintedDuplicateWalkerBase
275 {
276     /// <summary>
277     /// <para>
278     /// The cache.
279     /// </para>
280     /// <para></para>
281     /// </summary>
282     public Dictionary<string, long> _cache;
283     private string _currentKey;
284     private int _totalDuplicates;
285
286     /// <summary>
287     /// <para>
288     /// Walks the all using the specified elements.
289     /// </para>
290     /// <para></para>
291     /// </summary>
292     /// <param name="elements">
293     /// <para>The elements.</para>
294     /// <para></para>
295     /// </param>
296     public override void WalkAll(ICollection<char> elements)
297     {
298         _cache = new Dictionary<string, long>();
299
300         base.WalkAll(elements);
301
302         Console.WriteLine($"Unique string segments: {_cache.Count}. Total duplicates:
303         ↳ {_totalDuplicates}");
304     }
305
306     /// <summary>
307     /// <para>
308     /// Ons the duplicate found using the specified segment.
309     /// </para>
310     /// <para></para>
311     /// </summary>
312     /// <param name="segment">
313     /// <para>The segment.</para>
314     /// <para></para>
315     /// </param>
316     protected override void OnDuplicateFound(CharSegment segment)
317     {
318         _totalDuplicates++;
319     }
320
321     /// <summary>
322     /// <para>
323     /// Gets the segment frequency using the specified segment.
324     /// </para>
325     /// <para></para>
326     /// </summary>
327     /// <param name="segment">
328     /// <para>The segment.</para>
329     /// <para></para>
330     /// </param>
331     /// <returns>
332     /// <para>The long</para>
333     /// <para></para>
334     /// </returns>
335     protected override long GetSegmentFrequency(CharSegment segment) =>
336     ↳ _cache.GetOrDefault(_currentKey);
337
338     /// <summary>
339     /// <para>
340     /// Sets the segment frequency using the specified segment.
341     /// </para>
342     /// <para></para>
343     /// </summary>
344     /// <param name="segment">

```

```

343     /// <para>The segment.</para>
344     /// <para></para>
345     /// </param>
346     /// <param name="frequency">
347     /// <para>The frequency.</para>
348     /// <para></para>
349     /// </param>
350     protected override void SetSegmentFrequency(CharSegment segment, long frequency) =>
351         ↪ _cache[_currentKey] = frequency;
352
353     /// <summary>
354     /// <para>
355     /// Iterations the segment.
356     /// </para>
357     /// </summary>
358     /// <param name="segment">
359     /// <para>The segment.</para>
360     /// <para></para>
361     /// </param>
362     protected override void Iteration(CharSegment segment)
363     {
364         _currentKey = segment;
365
366         base.Iteration(segment);
367     }
368 }
369
370     /// <summary>
371     /// <para>
372     /// Represents the walker.
373     /// </para>
374     /// <para></para>
375     /// </summary>
376     /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase<char, CharSegment>" />
377     public class Walker4 : DictionaryBasedDuplicateSegmentsWalkerBase<char, CharSegment>
378     {
379         /// <summary>
380         /// <para>
381         /// Gets the public dictionary value.
382         /// </para>
383         /// <para></para>
384         /// </summary>
385         public IDictionary<CharSegment, long> PublicDictionary
386         {
387             get
388             {
389                 return Dictionary;
390             }
391         }
392
393         /// <summary>
394         /// <para>
395         /// Initializes a new <see cref="Walker4" /> instance.
396         /// </para>
397         /// <para></para>
398         /// </summary>
399         public Walker4()
400             : base(DefaultMinimumStringSegmentLength, resetDictionaryOnEachWalk: true)
401         {
402         }
403         private int _totalDuplicates;
404
405         /// <summary>
406         /// <para>
407         /// Walks the all using the specified elements.
408         /// </para>
409         /// <para></para>
410         /// </summary>
411         /// <param name="elements">
412         /// <para>The elements.</para>
413         /// <para></para>
414         /// </param>
415         public override void WalkAll(ICollection<char> elements)
416         {
417             _totalDuplicates = 0;
418
419             base.WalkAll(elements);

```



```

420         Console.WriteLine($"Unique string segments: {Dictionary.Count}. Total duplicates:
         ↪     {_totalDuplicates}.");
421     }
422
423     /// <summary>
424     /// <para>
425     /// Creates the segment using the specified elements.
426     /// </para>
427     /// <para></para>
428     /// </summary>
429     /// <param name="elements">
430     /// <para>The elements.</para>
431     /// <para></para>
432     /// </param>
433     /// <param name="offset">
434     /// <para>The offset.</para>
435     /// <para></para>
436     /// </param>
437     /// <param name="length">
438     /// <para>The length.</para>
439     /// <para></para>
440     /// </param>
441     /// <returns>
442     /// <para>The char segment</para>
443     /// <para></para>
444     /// </returns>
445     protected override CharSegment CreateSegment(IList<char> elements, int offset, int
         ↪     length) => new CharSegment(elements, offset, length);
446
447     /// <summary>
448     /// <para>
449     /// Ons the duplicate found using the specified segment.
450     /// </para>
451     /// <para></para>
452     /// </summary>
453     /// <param name="segment">
454     /// <para>The segment.</para>
455     /// <para></para>
456     /// </param>
457     protected override void OnDuplicateFound(CharSegment segment)
458     {
459         _totalDuplicates++;
460     }
461 }
462
463     /// <summary>
464     /// <para>
465     /// Represents the iterations counter.
466     /// </para>
467     /// <para></para>
468     /// </summary>
469     /// <seealso cref="AllSegmentsWalkerBase{char}"/>
470     public class IterationsCounter : AllSegmentsWalkerBase<char>
471     {
472         /// <summary>
473         /// <para>
474         /// The iterations count.
475         /// </para>
476         /// <para></para>
477         /// </summary>
478         public long IterationsCount;
479
480         /// <summary>
481         /// <para>
482         /// Iterations the segment.
483         /// </para>
484         /// <para></para>
485         /// </summary>
486         /// <param name="segment">
487         /// <para>The segment.</para>
488         /// <para></para>
489         /// </param>
490         protected override void Iteration(Segment<char> segment) => IterationsCount++;
491     }
492 }

```

Index

- ./csharp/Platform.Collections.Tests/ArrayTests.cs, 93
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 93
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 96
- ./csharp/Platform.Collections.Tests/ListTests.cs, 97
- ./csharp/Platform.Collections.Tests/StringTests.cs, 97
- ./csharp/Platform.Collections.Tests/WalkersTests.cs, 98
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayPool.cs, 4
- ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs, 5
- ./csharp/Platform.Collections/Arrays/ArrayString.cs, 7
- ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs, 7
- ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 9
- ./csharp/Platform.Collections/BitString.cs, 15
- ./csharp/Platform.Collections/BitStringExtensions.cs, 40
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 40
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 41
- ./csharp/Platform.Collections/EnsureExtensions.cs, 42
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 46
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 47
- ./csharp/Platform.Collections/Lists/CharIListExtensions.cs, 48
- ./csharp/Platform.Collections/Lists/IListComparer.cs, 49
- ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 50
- ./csharp/Platform.Collections/Lists/IListExtensions.cs, 51
- ./csharp/Platform.Collections/Lists/ListFiller.cs, 59
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 62
- ./csharp/Platform.Collections/Segments/Segment.cs, 63
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 68
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 68
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 70
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 70
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 71
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 74
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 75
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 76
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 77
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 81
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 84
- ./csharp/Platform.Collections/Stacks/IStack.cs, 84
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 85
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 86
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 86
- ./csharp/Platform.Collections/StringExtensions.cs, 87
- ./csharp/Platform.Collections/Trees/Node.cs, 89