

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      /// <para>Представляет заполнитель массива <see cref="T:TElement[]" /> с дополнительными
9      /// </summary>
10     /// <typeparam name="TElement"><para>The elements' type.</para><para>Тип элементов
11     /// <typeparam name="TReturnConstant"><para>The return constant's type.</para><para>Тип
12     public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
13     {
14         /// <summary>
15         /// <para>
16         /// The return constant.
17         /// </para>
18         /// <para></para>
19         /// </summary>
20         protected readonly TReturnConstant _returnConstant;
21
22         /// <summary>
23         /// <para>Initializes a new instance of the <see cref="ArrayFiller" /> class using the
24         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller" />, используя
25         /// <para>указанный массив, смещение с которого начнётся заполнение и константу возвращаемую
26         /// <para>при заполнении элементов.</para>
27         /// </summary>
28         /// <param name="array"><para>The array to fill.</para><para>Массив для
29         /// <param name="offset"><para>The offset from which to start the array
30         /// <param name="returnConstant"><para>The constant's value.</para><para>Значение
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
33             base(array, offset) => _returnConstant = returnConstant;
34
35         /// <summary>
36         /// <para>Initializes a new instance of the <see cref="ArrayFiller" /> class using the
37         /// <para>will start from the beginning of the array.</para>
38         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller" />, используя
39         /// <para>указанный массив и константу возвращаемую при заполнении элементов. Заполнение
40         /// <para>начнётся с начала массива.</para>
41         /// </summary>
42         /// <param name="array"><para>The array to fill.</para><para>Массив для
43         /// <param name="returnConstant"><para>The constant's value.</para><para>Значение
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
46             returnConstant) { }
47
48         /// <summary>
49         /// <para>Adds an item into the array and returns the constant.</para>
50         /// <para>Добавляет элемент в массив и возвращает константу.</para>
51         /// </summary>
52         /// <param name="element"><para>The element to add.</para><para>Добавляемый
53         /// <returns>
54         /// <para>The constant's value.</para>
55         /// <para>Значение константы.</para>
56         /// </returns>
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         public TReturnConstant AddAndReturnConstant(TElement element) =>
59             _array.AddAndReturnConstant(ref _position, element, _returnConstant);
60
61         /// <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 /// <summary>
30 /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
31   ↳ specified array as the array to fill and the offset from which to start
32   ↳ filling.</para>
33 /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
34   ↳ указанный массив в качестве заполняемого и смещение с которого начнётся
35   ↳ заполнение.</para>
36 /// </summary>
37 /// <param name="array"><para>The array to fill.</para><para>Массив для
38   ↳ заполнения.</para></param>
39 /// <param name="offset"><para>The offset from which to start filling the
40   ↳ array.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 public ArrayFiller(TElement[] array, long offset)
43 {
44     _array = array;
45     _position = offset;
46 }
47
48 /// <summary>
49 /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
50   ↳ specified array. Filling will start from the beginning of the array.</para>
51 /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
52   ↳ указанный массив. Заполнение начнётся с начала массива.</para>
53 /// </summary>
54 /// <param name="array"><para>The array to fill.</para><para>Массив для
55   ↳ заполнения.</para></param>
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 public ArrayFiller(TElement[] array) : this(array, 0) { }
58
59 /// <summary>
60 /// <para>Adds an item into the array.</para>
61 /// <para>Добавляет элемент в массив.</para>
62 /// </summary>
63 /// <param name="element"><para>The element to add.</para><para>Добавляемый
64   ↳ элемент.</para></param>
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public void Add(TElement element) => _array[_position++] = element;
67
68 /// <summary>
69 /// <para>Adds an item into the array and returns <see langword="true"/>.</para>
70 /// <para>Добавляет элемент в массив и возвращает <see langword="true"/>.</para>
71 /// </summary>
72 /// <param name="element"><para>The element to add.</para><para>Добавляемый
73   ↳ элемент.</para></param>
74 /// <returns>
75 /// <para>The <see langword="true"/> value.</para>
76 /// <para>Значение <see langword="true"/>.</para>
77 /// </returns>
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
80   ↳ _position, element, true);
81
82 /// <summary>
83 /// <para>Adds the first element from the specified list to the array to fill and
84   ↳ returns <see langword="true"/>.</para>
85 /// <para>Добавляет первый элемент из указанного списка в заполняемый массив и
86   ↳ возвращает <see langword="true"/>.</para>
87 /// </summary>
88 /// <param name="elements"><para>The list from which the first item will be
89   ↳ added.</para><para>Список из которого будет добавлен первый элемент.</para></param>
90 /// <returns>
91 /// <para>The <see langword="true"/> value.</para>
92 /// <para>Значение <see langword="true"/>.</para>
93 /// </returns>
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
96   ↳ _array.AddFirstAndReturnConstant(ref _position, elements, true);
97
98 /// <summary>
99 /// <para>Adds all elements from the specified list to the array to fill and returns
100   ↳ <see langword="true"/>.</para>
101 /// <para>Добавляет все элементы из указанного списка в заполняемый массив и возвращает
102   ↳ <see langword="true"/>.</para>
103 /// </summary>
104

```

```

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></param>
    ↪    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></typeparam>
    ↪    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
36         /// <summary>
37         /// <para>
38         /// The max arrays per size.
39         /// </para>
40         /// <para></para>
41         /// </summary>
42         private readonly int _maxArraysPerSize;
43         /// <summary>
44         /// <para>
45         /// The default sizes amount.
46         /// </para>
47         /// <para></para>
48         /// </summary>
49         private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
    ↪    Stack<T[]>>(ArrayPool.DefaultSizesAmount);
50
51         /// <summary>
52         /// <para>Initializes a new instance of the ArrayPool class using the specified maximum
    ↪    number of arrays per size.</para>
53         /// <para>Инициализирует новый экземпляр класса ArrayPool, используя указанное
    ↪    максимальное количество массивов на каждый размер.</para>
54         /// </summary>
55         /// <param name="maxArraysPerSize"><para>The maximum number of arrays in the pool per
    ↪    size.</para></param>
    ↪    Максимальное количество массивов в пуле на каждый
    ↪    размер.</para></param>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
58
59         /// <summary>
60         /// <para>Initializes a new instance of the ArrayPool class using the default maximum
    ↪    number of arrays per size.</para>

```

```

61    /// <para>Инициализирует новый экземпляр класса ArrayPool, используя максимальное
62    ↪ количество массивов на каждый размер по умолчанию.</para>
63    /// </summary>
64    [MethodImpl(MethodImplOptions.AggressiveInlining)]
65    public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
66
67    /// <summary>
68    /// <para>Retrieves an array from the pool, which will automatically return to the pool
69    ↪ when the container is disposed.</para>
70    /// <para>Извлекает из пула массив, который автоматически вернётся в пул при
71    ↪ высвобождении контейнера.</para>
72    /// </summary>
73    /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
74    ↪ массива.</para></param>
75    /// <returns>
76    /// <para>The disposable container containing either a new array or an array from the
77    ↪ pool.</para>
78    /// <para>Высвобождаемый контейнер содержащий либо новый массив, либо массив из
79    ↪ пула.</para>
80    /// </returns>
81    [MethodImpl(MethodImplOptions.AggressiveInlining)]
82    public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
83
84    /// <summary>
85    /// <para>Replaces the array with another array from the pool with the specified
86    ↪ size.</para>
87    /// <para>Заменяет массив на другой массив из пула с указанным размером.</para>
88    /// </summary>
89    /// <param name="source"><para>The source array.</para><para>Исходный
90    ↪ массив.</para></param>
91    /// <param name="size"><para>A new array size.</para><para>Новый размер
92    ↪ массива.</para></param>
93    /// <returns>
94    /// <para>An array with a new size.</para>
95    /// <para>Массив с новым размером.</para>
96    /// </returns>
97    [MethodImpl(MethodImplOptions.AggressiveInlining)]
98    public Disposable<T[]> Resize(Disposable<T[]> source, long size)
99    {
100        var destination = AllocateDisposable(size);
101        T[] sourceArray = source;
102        if (!sourceArray.IsNullOrEmpty())
103        {
104            T[] destinationArray = destination;
105            Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
106            ↪ sourceArray.LongLength);
107            source.Dispose();
108        }
109        return destination;
110    }
111
112    /// <summary>
113    /// <para>Clears the pool.</para>
114    /// <para>Очищает пул.</para>
115    /// </summary>
116    [MethodImpl(MethodImplOptions.AggressiveInlining)]
117    public virtual void Clear() => _pool.Clear();
118
119    /// <summary>
120    /// <para>Retrieves an array with the specified size from the pool.</para>
121    /// <para>Извлекает из пула массив с указанным размером.</para>
122    /// </summary>
123    /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
124    ↪ массива.</para></param>
125    /// <returns>
126    /// <para>An array from the pool or a new array.</para>
127    /// <para>Массив из пула или новый массив.</para>
128    /// </returns>
129    [MethodImpl(MethodImplOptions.AggressiveInlining)]
130    public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
131    ↪ _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
132
133    /// <summary>
134    /// <para>Frees the array to the pool for later reuse.</para>
135    /// <para>Освобождает массив в пул для последующего повторного использования.</para>
136    /// </summary>

```

```

125     /// <param name="array"><para>The array to be freed into the pool.</para></param>
    ↪    который нужно освободить в пул.</para></param>
126     [MethodImpl(MethodImplOptions.AggressiveInlining)]
127     public virtual void Free(T[] array)
128     {
129         if (array.IsNullOrEmpty())
130         {
131             return;
132         }
133         var stack = _pool.GetOrAdd(array.LongLength, size => new
    ↪    Stack<T[]>(_maxArraysPerSize));
134         if (stack.Count == _maxArraysPerSize) // Stack is full
135         {
136             return;
137         }
138         stack.Push(array);
139     }
140 }
141 }

```

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\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
35         →
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             }
50             return hashAccumulator + (hashSeed * 1566083941);
51         }
52
53         /// <summary>
54         /// <para>Checks if all elements of two lists are equal.</para>
55         /// <para>Проверяет равны ли все элементы двух списков.</para>
56         /// </summary>
57         /// <param name="left"><para>The first compared array.</para><para>Первый массив для
58         → сравнения.</para></param>
59         /// <param name="leftOffset"><para>The offset from which reading of the specified number
60         → of elements in the first array starts.</para><para>Смещение, с которого начинается
61         → чтение элементов в первом массиве.</para></param>
62         /// <param name="length"><para>The number of checked elements.</para><para>Количество
63         → проверяемых элементов.</para></param>
64         /// <param name="right"><para>The second compared array.</para><para>Второй массив для
65         → сравнения.</para></param>
66         /// <param name="rightOffset"><para>The offset from which reading of the specified
67         → number of elements in the second array starts.</para><para>Смещение, с которого
68         → начинается чтение элементов в втором массиве.</para></param>
69         /// <returns>
70         /// <para><see langword="true"/> if the segments of the passed arrays are equal to each
71         → other otherwise <see langword="false"/>.</para>

```



```

53  /// <para><see langword="true"/>, если сегменты переданных массивов равны друг другу,
54  ↪ иначе же <see langword="false"/>.</para>
55  /// </returns>
56  /// <remarks>
57  /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
58  ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
59  /// </remarks>
60  [MethodImpl(MethodImplOptions.AggressiveInlining)]
61  public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
62  ↪ right, int rightOffset)
63  {
64      fixed (char* leftPointer = &left[leftOffset])
65      {
66          fixed (char* rightPointer = &right[rightOffset])
67          {
68              char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
69              if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
70              ↪ rightPointerCopy, ref length))
71              {
72                  return false;
73              }
74              CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
75              ↪ ref length);
76              return length <= 0;
77          }
78      }
79  }
80
81  /// <summary>
82  /// <para>
83  /// Determines whether check arrays main part for equality.
84  /// </para>
85  /// </summary>
86  /// <param name="left">
87  /// <para>The left.</para>
88  /// </param>
89  /// <param name="right">
90  /// <para>The right.</para>
91  /// </param>
92  /// <param name="length">
93  /// <para>The length.</para>
94  /// </param>
95  /// <returns>
96  /// <para>The bool</para>
97  /// </returns>
98  [MethodImpl(MethodImplOptions.AggressiveInlining)]
99  private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
100  ↪ int length)
101  {
102      while (length >= 10)
103      {
104          if ((* (int*)left != *(int*)right)
105              || (*(int*)(left + 2) != *(int*)(right + 2))
106              || (*(int*)(left + 4) != *(int*)(right + 4))
107              || (*(int*)(left + 6) != *(int*)(right + 6))
108              || (*(int*)(left + 8) != *(int*)(right + 8)))
109          {
110              return false;
111          }
112          left += 10;
113          right += 10;
114          length -= 10;
115      }
116      return true;
117  }
118
119  /// <summary>
120  /// <para>
121  /// Checks the arrays remainder for equality using the specified left.
122  /// </para>
123  /// </summary>
124  /// <param name="left">

```

```

125     /// <para>The left.</para>
126     /// <para></para>
127     /// </param>
128     /// <param name="right">
129     /// <para>The right.</para>
130     /// <para></para>
131     /// </param>
132     /// <param name="length">
133     /// <para>The length.</para>
134     /// <para></para>
135     /// </param>
136     [MethodImpl(MethodImplOptions.AggressiveInlining)]
137     private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
    ↪ int length)
138     {
139         // This depends on the fact that the String objects are
140         // always zero terminated and that the terminating zero is not included
141         // in the length. For odd string sizes, the last compare will include
142         // the zero terminator.
143         while (length > 0)
144         {
145             if (*(int*)left != *(int*)right)
146             {
147                 break;
148             }
149             left += 2;
150             right += 2;
151             length -= 2;
152         }
153     }
154 }
155 }

```

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
15         ↪ variable type int, and if the array length is greater than the index - return
16         ↪ array[index], otherwise - default value.</para>
17         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
18         ↪ помощью переменной index, и если длина массива больше индекса - возвращает
19         ↪ array[index], иначе - значение по умолчанию.</para>
20         /// </summary>
21         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
22         ↪ массива.</para></typeparam>
23         /// <param name="array"><para>Array that will participate in
24         ↪ verification.</para><para>Массив который будет участвовать в
25         ↪ проверке.</para></param>
26         /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
27         ↪ сравнения.</para></param>
28         /// <returns><para>Array element or default value.</para><para>Элемент массива или же
29         ↪ значение по умолчанию.</para></returns>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
32         ↪ array.Length > index ? array[index] : default;
33
34         /// <summary>
35         /// <para>Checks whether the array exists, if so, checks the array length using the
36         ↪ index variable type long, and if the array length is greater than the index - return
37         ↪ array[index], otherwise - default value.</para>
38         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
39         ↪ помощью переменной index, и если длина массива больше индекса - возвращает
40         ↪ array[index], иначе - значение по умолчанию.</para>
41         /// </summary>
42         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
43         ↪ 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
88   → массива.</para></typeparam>
89 /// <param name="array"><para>The array to copy.</para><para>Массив который необходимо
90   → скопировать.</para></param>
91 /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public static T[] Clone<T>(this T[] array)
94 {
95     var copy = new T[array.LongLength];
96     Array.Copy(array, 0L, copy, 0L, array.LongLength);
97     return copy;
98 }
99
100 /// <summary>
101 /// <para>Shifts all the elements of the array by one position to the right.</para>
102 /// <para>Сдвигает вправо все элементы массива на одну позицию.</para>
103 /// </summary>
104 /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
105   → массива.</para></typeparam>
106 /// <param name="array"><para>The array to copy from.</para><para>Массив для
107   → копирования.</para></param>
108 /// <returns>
109 /// <para>Array with a shift of elements by one position.</para>
110 /// <para>Массив со сдвигом элементов на одну позицию.</para>
111 /// </returns>
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
114
115 /// <summary>
116 /// <para>Shifts all elements of the array to the right by the specified number of
117   → elements.</para>
118 /// <para>Сдвигает вправо все элементы массива на указанное количество элементов.</para>
119 /// </summary>
120 /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
121   → массива.</para></typeparam>
122 /// <param name="array"><para>The array to copy from.</para><para>Массив для
123   → копирования.</para></param>
124 /// <param name="shift"><para>The number of items to shift.</para><para>Количество
125   → сдвигаемых элементов.</para></param>
126 /// <returns>
127 /// <para>If the value of the shift variable is less than zero - an <see
128   → cref="NotImplementedException"/> exception is thrown, but if the value of the shift
129   → variable is 0 - an exact copy of the array is returned. Otherwise, an array is
130   → returned with the shift of the elements.</para>
131 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
132   → cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
133   → возвращается точная копия массива. Иначе возвращается массив со сдвигом
134   → элементов.</para>
135 /// </returns>
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public static IList<T> ShiftRight<T>(this T[] array, long shift)
138 {
139     if (shift < 0)
140     {
141         throw new NotImplementedException();
142     }
143     if (shift == 0)
144     {
145         return array.Clone<T>();
146     }
147     else
148     {

```

```

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
143   ↳ position value by one.</para>
144 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
145   ↳ значение position на единицу.</para>
146 /// </summary>
147 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
148   ↳ массива.</para></typeparam>
149 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
150   ↳ который необходимо добавить элемент.</para></param>
151 /// <param name="position"><para>A reference to the position of type int where the
152   ↳ element will be added.</para><para>Ссылка на позицию типа int, в которую будет
153   ↳ добавлен элемент.</para></param>
154 /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
155   ↳ который нужно добавить в массив.</para></param>
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 public static void Add<T>(this T[] array, ref int position, T element) =>
158   ↳ array[position++] = element;
159
160 /// <summary>
161 /// <para>Adding in array the passed element at the specified position and increments
162   ↳ position value by one.</para>
163 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
164   ↳ значение position на единицу.</para>
165 /// </summary>
166 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
167   ↳ массива.</para></typeparam>
168 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
169   ↳ который необходимо добавить элемент.</para></param>
170 /// <param name="position"><para>A reference to the position of type long where the
171   ↳ element will be added.</para><para>Ссылка на позицию типа long, в которую будет
172   ↳ добавлен элемент.</para></param>
173 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
174   ↳ который необходимо добавить в массив.</para></param>
175 [MethodImpl(MethodImplOptions.AggressiveInlining)]
176 public static void Add<T>(this T[] array, ref long position, T element) =>
177   ↳ array[position++] = element;
178
179 /// <summary>
180 /// <para>Adding in array the passed element, at the specified position, increments
181   ↳ position value by one and returns the value of the passed constant.</para>
182 /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
183   ↳ значение position на единицу и возвращает значение переданной константы.</para>
184 /// </summary>
185 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
186   ↳ массива.</para></typeparam>
187 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
188   ↳ возвращаемой константы.</para></typeparam>
189 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
190   ↳ который необходимо добавить элемент.</para></param>
191 /// <param name="position"><para>Reference to the position to which the element will be
192   ↳ added.</para><para>Ссылка на позицию, в которую будет добавлен
193   ↳ элемент.</para></param>
194 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
195   ↳ который необходимо добавить в массив.</para></param>
196 /// <param name="returnConstant"><para>The constant value that will be
197   ↳ returned.</para><para>Значение константы, которое будет возвращено.</para></param>
198 /// <returns>
199 /// <para>The constant value passed as an argument.</para>
200 /// <para>Значение константы, переданное в качестве аргумента.</para>
201 /// </returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
204   ↳ TElement[] array, ref long position, TElement element, TReturnConstant
205   ↳ returnConstant)
206 {
207     array.Add(ref position, element);
208     return returnConstant;
209 }

```

```

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
277     → and increments position value by one.</para>
278     /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию и
279     → увеличивает значение position на единицу.</para>
280     /// </summary>
281     /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
282     → массива.</para></typeparam>
283     /// <param name="array"><para>The array to add items to.</para><para>Массив в который
284     → необходимо добавить элементы.</para></param>
285     /// <param name="position"><para>Reference to the position from which to start adding
286     → elements.</para><para>Ссылка на позицию, с которой начинается добавление
287     → элементов.</para></param>
288     /// <param name="elements"><para>List, whose elements will be added to the
289     → array.</para><para>Список, элементы которого будут добавлены в
290     → массив.</para></param>
291     [MethodImpl(MethodImplOptions.AggressiveInlining)]
292     public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
293     → => array.AddSkipFirst(ref position, elements, 1);
294
295     /// <summary>
296     /// <para>Adding in array all but the first element, skipping a specified number of
297     → positions and increments position value by one.</para>
298     /// <para>Добавляет в массив все элементы коллекции, кроме первого, пропуская
299     → определенное количество позиций и увеличивает значение position на единицу.</para>
300     /// </summary>
301     /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
302     → массива.</para></typeparam>
303     /// <param name="array"><para>The array to add items to.</para><para>Массив в который
304     → необходимо добавить элементы.</para></param>
305     /// <param name="position"><para>Reference to the position from which to start adding
306     → elements.</para><para>Ссылка на позицию, с которой начинается добавление
307     → элементов.</para></param>
308     /// <param name="elements"><para>List, whose elements will be added to the
309     → array.</para><para>Список, элементы которого будут добавлены в
310     → массив.</para></param>
311     /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
312     → пропускаемых элементов.</para></param>
313     [MethodImpl(MethodImplOptions.AggressiveInlining)]
314     public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
315     → int skip)
316     {
317         for (var i = skip; i < elements.Count; i++)
318         {
319             array.Add(ref position, elements[i]);
320         }
321     }
322 }
323
324 }

```

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         /// <summary>

```



```

25     /// <para>
26     /// The bits set in 16 bits.
27     /// </para>
28     /// <para></para>
29     /// </summary>
30     private static readonly byte[] [] _bitsSetIn16Bits;
31     /// <summary>
32     /// <para>
33     /// The array.
34     /// </para>
35     /// <para></para>
36     /// </summary>
37     private long[] _array;
38     /// <summary>
39     /// <para>
40     /// The length.
41     /// </para>
42     /// <para></para>
43     /// </summary>
44     private long _length;
45     /// <summary>
46     /// <para>
47     /// The min positive word.
48     /// </para>
49     /// <para></para>
50     /// </summary>
51     private long _minPositiveWord;
52     /// <summary>
53     /// <para>
54     /// The max positive word.
55     /// </para>
56     /// <para></para>
57     /// </summary>
58     private long _maxPositiveWord;
59
60     /// <summary>
61     /// <para>
62     /// The value.
63     /// </para>
64     /// <para></para>
65     /// </summary>
66     public bool this[long index]
67     {
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         get => Get(index);
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         set => Set(index, value);
72     }
73
74     /// <summary>
75     /// <para>
76     /// Gets or sets the length value.
77     /// </para>
78     /// <para></para>
79     /// </summary>
80     public long Length
81     {
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         get => _length;
84         [MethodImpl(MethodImplOptions.AggressiveInlining)]
85         set
86         {
87             if (_length == value)
88             {
89                 return;
90             }
91             Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
92             // Currently we never shrink the array
93             if (value > _length)
94             {
95                 var words = GetWordsCountFromIndex(value);
96                 var oldWords = GetWordsCountFromIndex(_length);
97                 if (words > _array.LongLength)
98                 {
99                     var copy = new long[words];
100                     Array.Copy(_array, copy, _array.LongLength);
101                     _array = copy;
102                 }
103                 else

```

```

104         {
105             // What is going on here?
106             Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
107         }
108         // What is going on here?
109         var mask = (int)(_length % 64);
110         if (mask > 0)
111         {
112             _array[oldWords - 1] &= (1L << mask) - 1;
113         }
114     }
115     else
116     {
117         // Looks like minimum and maximum positive words are not updated
118         throw new NotImplementedException();
119     }
120     _length = value;
121 }
122
123
124 #region Constructors
125
126 /// <summary>
127 /// <para>
128 /// Initializes a new <see cref="BitString"/> instance.
129 /// </para>
130 /// <para></para>
131 /// </summary>
132 [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 static BitString()
134 {
135     _bitsSetIn16Bits = new byte[65536][];
136     int i, c, k;
137     byte bitIndex;
138     for (i = 0; i < 65536; i++)
139     {
140         // Calculating size of array (number of positive bits)
141         for (c = 0, k = 1; k <= 65536; k <= 1)
142         {
143             if ((i & k) == k)
144             {
145                 c++;
146             }
147         }
148         var array = new byte[c];
149         // Adding positive bits indices into array
150         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
151         {
152             if ((i & k) == k)
153             {
154                 array[c++] = bitIndex;
155             }
156             bitIndex++;
157         }
158         _bitsSetIn16Bits[i] = array;
159     }
160 }
161
162 /// <summary>
163 /// <para>
164 /// Initializes a new <see cref="BitString"/> instance.
165 /// </para>
166 /// <para></para>
167 /// </summary>
168 /// <param name="other">
169 /// <para>A other.</para>
170 /// <para></para>
171 /// </param>
172 [MethodImpl(MethodImplOptions.AggressiveInlining)]
173 public BitString(BitString other)
174 {
175     Ensure.Always.ArgumentNotNull(other, nameof(other));
176     _length = other._length;
177     _array = new long[GetWordsCountFromIndex(_length)];
178     _minPositiveWord = other._minPositiveWord;
179     _maxPositiveWord = other._maxPositiveWord;
180     Array.Copy(other._array, _array, _array.LongLength);
181 }
182

```

```

183     /// <summary>
184     /// <para>
185     /// Initializes a new <see cref="BitString"/> instance.
186     /// </para>
187     /// <para></para>
188     /// </summary>
189     /// <param name="length">
190     /// <para>A length.</para>
191     /// <para></para>
192     /// </param>
193     [MethodImpl(MethodImplOptions.AggressiveInlining)]
194     public BitString(long length)
195     {
196         Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
197         _length = length;
198         _array = new long[GetWordsCountFromIndex(_length)];
199         MarkBordersAsAllBitsReset();
200     }
201
202     /// <summary>
203     /// <para>
204     /// Initializes a new <see cref="BitString"/> instance.
205     /// </para>
206     /// <para></para>
207     /// </summary>
208     /// <param name="length">
209     /// <para>A length.</para>
210     /// <para></para>
211     /// </param>
212     /// <param name="defaultValue">
213     /// <para>A default value.</para>
214     /// <para></para>
215     /// </param>
216     [MethodImpl(MethodImplOptions.AggressiveInlining)]
217     public BitString(long length, bool defaultValue)
218         : this(length)
219     {
220         if (defaultValue)
221         {
222             SetAll();
223         }
224     }
225
226     #endregion
227
228     /// <summary>
229     /// <para>
230     /// Nots this instance.
231     /// </para>
232     /// <para></para>
233     /// </summary>
234     /// <returns>
235     /// <para>The bit string</para>
236     /// <para></para>
237     /// </returns>
238     [MethodImpl(MethodImplOptions.AggressiveInlining)]
239     public BitString Not()
240     {
241         for (var i = 0L; i < _array.LongLength; i++)
242         {
243             _array[i] = ~_array[i];
244             RefreshBordersByWord(i);
245         }
246         return this;
247     }
248
249     /// <summary>
250     /// <para>
251     /// Parallels the not.
252     /// </para>
253     /// <para></para>
254     /// </summary>
255     /// <returns>
256     /// <para>The bit string</para>
257     /// <para></para>
258     /// </returns>
259     [MethodImpl(MethodImplOptions.AggressiveInlining)]
260     public BitString ParallelNot()

```

```

261 {
262     var threads = Environment.ProcessorCount / 2;
263     if (threads <= 1)
264     {
265         return Not();
266     }
267     var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
268     ↪ threads);
269     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
270     ↪ MaxDegreeOfParallelism = threads }, range =>
271     {
272         var maximum = range.Item2;
273         for (var i = range.Item1; i < maximum; i++)
274         {
275             _array[i] = ~_array[i];
276         }
277     });
278     MarkBordersAsAllBitsSet();
279     TryShrinkBorders();
280     return this;
281 }
282
283 /// <summary>
284 /// <para>
285 /// Vectors the not.
286 /// </para>
287 /// <para></para>
288 /// </summary>
289 /// <returns>
290 /// <para>The bit string</para>
291 /// <para></para>
292 /// </returns>
293 [MethodImpl(MethodImplOptions.AggressiveInlining)]
294 public BitString VectorNot()
295 {
296     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
297     {
298         return Not();
299     }
300     var step = Vector<long>.Count;
301     if (_array.Length < step)
302     {
303         return Not();
304     }
305     VectorNotLoop(_array, step, 0, _array.Length);
306     MarkBordersAsAllBitsSet();
307     TryShrinkBorders();
308     return this;
309 }
310
311 /// <summary>
312 /// <para>
313 /// Parallels the vector not.
314 /// </para>
315 /// <para></para>
316 /// </summary>
317 /// <returns>
318 /// <para>The bit string</para>
319 /// <para></para>
320 /// </returns>
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 public BitString ParallelVectorNot()
323 {
324     var threads = Environment.ProcessorCount / 2;
325     if (threads <= 1)
326     {
327         return VectorNot();
328     }
329     if (!Vector.IsHardwareAccelerated)
330     {
331         return ParallelNot();
332     }
333     var step = Vector<long>.Count;
334     if (_array.Length < (step * threads))
335     {
336         return VectorNot();
337     }
338     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);

```

```

337         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
338             ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
339             ↪ range.Item1, range.Item2));
340         MarkBordersAsAllBitsSet();
341         TryShrinkBorders();
342         return this;
343     }
344
345     /// <summary>
346     /// <para>
347     /// Vectors the not loop using the specified array.
348     /// </para>
349     /// </summary>
350     /// <param name="array">
351     /// <para>The array.</para>
352     /// </param>
353     /// <param name="step">
354     /// <para>The step.</para>
355     /// </param>
356     /// <param name="start">
357     /// <para>The start.</para>
358     /// </param>
359     /// <param name="maximum">
360     /// <para>The maximum.</para>
361     /// </param>
362     [MethodImpl(MethodImplOptions.AggressiveInlining)]
363     static private void VectorNotLoop(long[] array, int step, int start, int maximum)
364     {
365         var i = start;
366         var range = maximum - start - 1;
367         var stop = range - (range % step);
368         for (; i < stop; i += step)
369         {
370             (~new Vector<long>(array, i)).CopyTo(array, i);
371         }
372         for (; i < maximum; i++)
373         {
374             array[i] = ~array[i];
375         }
376     }
377
378     /// <summary>
379     /// <para>
380     /// Ands the other.
381     /// </para>
382     /// </summary>
383     /// <param name="other">
384     /// <para>The other.</para>
385     /// </param>
386     /// <returns>
387     /// <para>The bit string</para>
388     /// </returns>
389     [MethodImpl(MethodImplOptions.AggressiveInlining)]
390     public BitString And(BitString other)
391     {
392         EnsureBitStringHasTheSameSize(other, nameof(other));
393         GetCommonOuterBorders(this, other, out long from, out long to);
394         var otherArray = other._array;
395         for (var i = from; i <= to; i++)
396         {
397             _array[i] &= otherArray[i];
398             RefreshBordersByWord(i);
399         }
400         return this;
401     }
402
403     /// <summary>
404     /// <para>
405     /// Parallels the and using the specified other.
406     /// </para>

```

```

413 /// <para></para>
414 /// </summary>
415 /// <param name="other">
416 /// <para>The other.</para>
417 /// <para></para>
418 /// </param>
419 /// <returns>
420 /// <para>The bit string</para>
421 /// <para></para>
422 /// </returns>
423 [MethodImpl(MethodImplOptions.AggressiveInlining)]
424 public BitString ParallelAnd(BitString other)
425 {
426     var threads = Environment.ProcessorCount / 2;
427     if (threads <= 1)
428     {
429         return And(other);
430     }
431     EnsureBitStringHasTheSameSize(other, nameof(other));
432     GetCommonOuterBorders(this, other, out long from, out long to);
433     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
434     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
435         ↳ MaxDegreeOfParallelism = threads }, range =>
436     {
437         var maximum = range.Item2;
438         for (var i = range.Item1; i < maximum; i++)
439         {
440             _array[i] &= other._array[i];
441         }
442     });
443     MarkBordersAsAllBitsSet();
444     TryShrinkBorders();
445     return this;
446 }
447
448 /// <summary>
449 /// <para>
450 /// Vectors the and using the specified other.
451 /// </para>
452 /// <para></para>
453 /// </summary>
454 /// <param name="other">
455 /// <para>The other.</para>
456 /// <para></para>
457 /// </param>
458 /// <returns>
459 /// <para>The bit string</para>
460 /// <para></para>
461 /// </returns>
462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
463 public BitString VectorAnd(BitString other)
464 {
465     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
466     {
467         return And(other);
468     }
469     var step = Vector<long>.Count;
470     if (_array.Length < step)
471     {
472         return And(other);
473     }
474     EnsureBitStringHasTheSameSize(other, nameof(other));
475     GetCommonOuterBorders(this, other, out int from, out int to);
476     VectorAndLoop(_array, other._array, step, from, to + 1);
477     MarkBordersAsAllBitsSet();
478     TryShrinkBorders();
479     return this;
480 }
481
482 /// <summary>
483 /// <para>
484 /// Parallels the vector and using the specified other.
485 /// </para>
486 /// <para></para>
487 /// </summary>
488 /// <param name="other">
489 /// <para>The other.</para>
490 /// <para></para>

```

```

490 /// </param>
491 /// <returns>
492 /// <para>The bit string</para>
493 /// <para></para>
494 /// </returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public BitString ParallelVectorAnd(BitString other)
497 {
498     var threads = Environment.ProcessorCount / 2;
499     if (threads <= 1)
500     {
501         return VectorAnd(other);
502     }
503     if (!Vector.IsHardwareAccelerated)
504     {
505         return ParallelAnd(other);
506     }
507     var step = Vector<long>.Count;
508     if (_array.Length < (step * threads))
509     {
510         return VectorAnd(other);
511     }
512     EnsureBitStringHasTheSameSize(other, nameof(other));
513     GetCommonOuterBorders(this, other, out int from, out int to);
514     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
515     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
516         ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
517         ↪ step, range.Item1, range.Item2));
518     MarkBordersAsAllBitsSet();
519     TryShrinkBorders();
520     return this;
521 }
522
523 /// <summary>
524 /// <para>
525 /// Vectors the and loop using the specified array.
526 /// </para>
527 /// <para></para>
528 /// </summary>
529 /// <param name="array">
530 /// <para>The array.</para>
531 /// <para></para>
532 /// </param>
533 /// <param name="otherArray">
534 /// <para>The other array.</para>
535 /// <para></para>
536 /// </param>
537 /// <param name="step">
538 /// <para>The step.</para>
539 /// <para></para>
540 /// </param>
541 /// <param name="start">
542 /// <para>The start.</para>
543 /// <para></para>
544 /// </param>
545 /// <param name="maximum">
546 /// <para>The maximum.</para>
547 /// <para></para>
548 /// </param>
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
551 ↪ int maximum)
552 {
553     var i = start;
554     var range = maximum - start - 1;
555     var stop = range - (range % step);
556     for (; i < stop; i += step)
557     {
558         (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
559     }
560     for (; i < maximum; i++)
561     {
562         array[i] &= otherArray[i];
563     }
564 }
565
566 /// <summary>
567 /// <para>

```

```

565     /// Ors the other.
566     /// </para>
567     /// <para></para>
568     /// </summary>
569     /// <param name="other">
570     /// <para>The other.</para>
571     /// <para></para>
572     /// </param>
573     /// <returns>
574     /// <para>The bit string</para>
575     /// <para></para>
576     /// </returns>
577     [MethodImpl(MethodImplOptions.AggressiveInlining)]
578     public BitString Or(BitString other)
579     {
580         EnsureBitStringHasTheSameSize(other, nameof(other));
581         GetCommonOuterBorders(this, other, out long from, out long to);
582         for (var i = from; i <= to; i++)
583         {
584             _array[i] |= other._array[i];
585             RefreshBordersByWord(i);
586         }
587         return this;
588     }
589
590     /// <summary>
591     /// <para>
592     /// Parallels the or using the specified other.
593     /// </para>
594     /// <para></para>
595     /// </summary>
596     /// <param name="other">
597     /// <para>The other.</para>
598     /// <para></para>
599     /// </param>
600     /// <returns>
601     /// <para>The bit string</para>
602     /// <para></para>
603     /// </returns>
604     [MethodImpl(MethodImplOptions.AggressiveInlining)]
605     public BitString ParallelOr(BitString other)
606     {
607         var threads = Environment.ProcessorCount / 2;
608         if (threads <= 1)
609         {
610             return Or(other);
611         }
612         EnsureBitStringHasTheSameSize(other, nameof(other));
613         GetCommonOuterBorders(this, other, out long from, out long to);
614         var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
615         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
616             ↪ MaxDegreeOfParallelism = threads }, range =>
617         {
618             var maximum = range.Item2;
619             for (var i = range.Item1; i < maximum; i++)
620             {
621                 _array[i] |= other._array[i];
622             }
623         });
624         MarkBordersAsAllBitsSet();
625         TryShrinkBorders();
626         return this;
627     }
628
629     /// <summary>
630     /// <para>
631     /// Vectors the or using the specified other.
632     /// </para>
633     /// <para></para>
634     /// </summary>
635     /// <param name="other">
636     /// <para>The other.</para>
637     /// <para></para>
638     /// </param>
639     /// <returns>
640     /// <para>The bit string</para>
641     /// <para></para>
642     /// </returns>

```



```

642 [MethodImpl(MethodImplOptions.AggressiveInlining)]
643 public BitString VectorOr(BitString other)
644 {
645     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
646     {
647         return Or(other);
648     }
649     var step = Vector<long>.Count;
650     if (_array.Length < step)
651     {
652         return Or(other);
653     }
654     EnsureBitStringHasTheSameSize(other, nameof(other));
655     GetCommonOuterBorders(this, other, out int from, out int to);
656     VectorOrLoop(_array, other._array, step, from, to + 1);
657     MarkBordersAsAllBitsSet();
658     TryShrinkBorders();
659     return this;
660 }
661
662 /// <summary>
663 /// <para>
664 /// Parallels the vector or using the specified other.
665 /// </para>
666 /// <para></para>
667 /// </summary>
668 /// <param name="other">
669 /// <para>The other.</para>
670 /// <para></para>
671 /// </param>
672 /// <returns>
673 /// <para>The bit string</para>
674 /// <para></para>
675 /// </returns>
676 [MethodImpl(MethodImplOptions.AggressiveInlining)]
677 public BitString ParallelVectorOr(BitString other)
678 {
679     var threads = Environment.ProcessorCount / 2;
680     if (threads <= 1)
681     {
682         return VectorOr(other);
683     }
684     if (!Vector.IsHardwareAccelerated)
685     {
686         return ParallelOr(other);
687     }
688     var step = Vector<long>.Count;
689     if (_array.Length < (step * threads))
690     {
691         return VectorOr(other);
692     }
693     EnsureBitStringHasTheSameSize(other, nameof(other));
694     GetCommonOuterBorders(this, other, out int from, out int to);
695     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
696     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
697         ↪ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
698         ↪ step, range.Item1, range.Item2));
699     MarkBordersAsAllBitsSet();
700     TryShrinkBorders();
701     return this;
702 }
703
704 /// <summary>
705 /// <para>
706 /// Vectors the or loop using the specified array.
707 /// </para>
708 /// <para></para>
709 /// </summary>
710 /// <param name="array">
711 /// <para>The array.</para>
712 /// <para></para>
713 /// </param>
714 /// <param name="otherArray">
715 /// <para>The other array.</para>
716 /// <para></para>
717 /// </param>
718 /// <param name="step">
719 /// <para>The step.</para>

```

```

718     /// <para></para>
719     /// </param>
720     /// <param name="start">
721     /// <para>The start.</para>
722     /// <para></para>
723     /// </param>
724     /// <param name="maximum">
725     /// <para>The maximum.</para>
726     /// <para></para>
727     /// </param>
728     [MethodImpl(MethodImplOptions.AggressiveInlining)]
729     static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
730     ↪ int maximum)
731     {
732         var i = start;
733         var range = maximum - start - 1;
734         var stop = range - (range % step);
735         for (; i < stop; i += step)
736         {
737             (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
738         }
739         for (; i < maximum; i++)
740         {
741             array[i] |= otherArray[i];
742         }
743     }
744     /// <summary>
745     /// <para>
746     /// Xors the other.
747     /// </para>
748     /// <para></para>
749     /// </summary>
750     /// <param name="other">
751     /// <para>The other.</para>
752     /// <para></para>
753     /// </param>
754     /// <returns>
755     /// <para>The bit string</para>
756     /// <para></para>
757     /// </returns>
758     [MethodImpl(MethodImplOptions.AggressiveInlining)]
759     public BitString Xor(BitString other)
760     {
761         EnsureBitStringHasTheSameSize(other, nameof(other));
762         GetCommonOuterBorders(this, other, out long from, out long to);
763         for (var i = from; i <= to; i++)
764         {
765             _array[i] ^= other._array[i];
766             RefreshBordersByWord(i);
767         }
768         return this;
769     }
770     /// <summary>
771     /// <para>
772     /// Parallels the xor using the specified other.
773     /// </para>
774     /// <para></para>
775     /// </summary>
776     /// <param name="other">
777     /// <para>The other.</para>
778     /// <para></para>
779     /// </param>
780     /// <returns>
781     /// <para>The bit string</para>
782     /// <para></para>
783     /// </returns>
784     [MethodImpl(MethodImplOptions.AggressiveInlining)]
785     public BitString ParallelXor(BitString other)
786     {
787         var threads = Environment.ProcessorCount / 2;
788         if (threads <= 1)
789         {
790             return Xor(other);
791         }
792         EnsureBitStringHasTheSameSize(other, nameof(other));
793         GetCommonOuterBorders(this, other, out long from, out long to);

```

```

795     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
796     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
797         ↪ MaxDegreeOfParallelism = threads }, range =>
798     {
799         var maximum = range.Item2;
800         for (var i = range.Item1; i < maximum; i++)
801         {
802             _array[i] ^= other._array[i];
803         }
804     });
805     MarkBordersAsAllBitsSet();
806     TryShrinkBorders();
807     return this;
808 }
809
810 /// <summary>
811 /// <para>
812 /// Vectors the xor using the specified other.
813 /// </para>
814 /// <para></para>
815 /// </summary>
816 /// <param name="other">
817 /// <para>The other.</para>
818 /// <para></para>
819 /// </param>
820 /// <returns>
821 /// <para>The bit string</para>
822 /// <para></para>
823 /// </returns>
824 [MethodImpl(MethodImplOptions.AggressiveInlining)]
825 public BitString VectorXor(BitString other)
826 {
827     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
828     {
829         return Xor(other);
830     }
831     var step = Vector<long>.Count;
832     if (_array.Length < step)
833     {
834         return Xor(other);
835     }
836     EnsureBitStringHasTheSameSize(other, nameof(other));
837     GetCommonOuterBorders(this, other, out int from, out int to);
838     VectorXorLoop(_array, other._array, step, from, to + 1);
839     MarkBordersAsAllBitsSet();
840     TryShrinkBorders();
841     return this;
842 }
843
844 /// <summary>
845 /// <para>
846 /// Parallels the vector xor using the specified other.
847 /// </para>
848 /// <para></para>
849 /// </summary>
850 /// <param name="other">
851 /// <para>The other.</para>
852 /// <para></para>
853 /// </param>
854 /// <returns>
855 /// <para>The bit string</para>
856 /// <para></para>
857 /// </returns>
858 [MethodImpl(MethodImplOptions.AggressiveInlining)]
859 public BitString ParallelVectorXor(BitString other)
860 {
861     var threads = Environment.ProcessorCount / 2;
862     if (threads <= 1)
863     {
864         return VectorXor(other);
865     }
866     if (!Vector.IsHardwareAccelerated)
867     {
868         return ParallelXor(other);
869     }
870     var step = Vector<long>.Count;
871     if (_array.Length < (step * threads))
872     {

```

```

872         return VectorXor(other);
873     }
874     EnsureBitStringHasTheSameSize(other, nameof(other));
875     GetCommonOuterBorders(this, other, out int from, out int to);
876     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
877     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
        ↪ step, range.Item1, range.Item2));
878     MarkBordersAsAllBitsSet();
879     TryShrinkBorders();
880     return this;
881 }
882
883 /// <summary>
884 /// <para>
885 /// Vectors the xor loop using the specified array.
886 /// </para>
887 /// <para></para>
888 /// </summary>
889 /// <param name="array">
890 /// <para>The array.</para>
891 /// <para></para>
892 /// </param>
893 /// <param name="otherArray">
894 /// <para>The other array.</para>
895 /// <para></para>
896 /// </param>
897 /// <param name="step">
898 /// <para>The step.</para>
899 /// <para></para>
900 /// </param>
901 /// <param name="start">
902 /// <para>The start.</para>
903 /// <para></para>
904 /// </param>
905 /// <param name="maximum">
906 /// <para>The maximum.</para>
907 /// <para></para>
908 /// </param>
909 [MethodImpl(MethodImplOptions.AggressiveInlining)]
910 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
    ↪ int maximum)
911 {
912     var i = start;
913     var range = maximum - start - 1;
914     var stop = range - (range % step);
915     for (; i < stop; i += step)
916     {
917         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
918     }
919     for (; i < maximum; i++)
920     {
921         array[i] ^= otherArray[i];
922     }
923 }
924
925 /// <summary>
926 /// <para>
927 /// Refreshes the borders by word using the specified word index.
928 /// </para>
929 /// <para></para>
930 /// </summary>
931 /// <param name="wordIndex">
932 /// <para>The word index.</para>
933 /// <para></para>
934 /// </param>
935 [MethodImpl(MethodImplOptions.AggressiveInlining)]
936 private void RefreshBordersByWord(long wordIndex)
937 {
938     if (_array[wordIndex] == 0)
939     {
940         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
941         {
942             _minPositiveWord++;
943         }
944         if (wordIndex == _maxPositiveWord && wordIndex != 0)
945         {
946             _maxPositiveWord--;

```

```

947     }
948 }
949 else
950 {
951     if (wordIndex < _minPositiveWord)
952     {
953         _minPositiveWord = wordIndex;
954     }
955     if (wordIndex > _maxPositiveWord)
956     {
957         _maxPositiveWord = wordIndex;
958     }
959 }
960 }
961
962 /// <summary>
963 /// <para>
964 /// Determines whether this instance try shrink borders.
965 /// </para>
966 /// <para></para>
967 /// </summary>
968 /// <returns>
969 /// <para>The borders updated.</para>
970 /// <para></para>
971 /// </returns>
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 public bool TryShrinkBorders()
974 {
975     GetBorders(out long from, out long to);
976     while (from <= to && _array[from] == 0)
977     {
978         from++;
979     }
980     if (from > to)
981     {
982         MarkBordersAsAllBitsReset();
983         return true;
984     }
985     while (to >= from && _array[to] == 0)
986     {
987         to--;
988     }
989     if (to < from)
990     {
991         MarkBordersAsAllBitsReset();
992         return true;
993     }
994     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
995     if (bordersUpdated)
996     {
997         SetBorders(from, to);
998     }
999     return bordersUpdated;
1000 }
1001
1002 /// <summary>
1003 /// <para>
1004 /// Determines whether this instance get.
1005 /// </para>
1006 /// <para></para>
1007 /// </summary>
1008 /// <param name="index">
1009 /// <para>The index.</para>
1010 /// <para></para>
1011 /// </param>
1012 /// <returns>
1013 /// <para>The bool</para>
1014 /// <para></para>
1015 /// </returns>
1016 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1017 public bool Get(long index)
1018 {
1019     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
1020     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
1021 }
1022
1023 /// <summary>
1024 /// <para>
1025 /// Sets the index.

```

```

1026    /// </para>
1027    /// <para></para>
1028    /// </summary>
1029    /// <param name="index">
1030    /// <para>The index.</para>
1031    /// <para></para>
1032    /// </param>
1033    /// <param name="value">
1034    /// <para>The value.</para>
1035    /// <para></para>
1036    /// </param>
1037    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1038    public void Set(long index, bool value)
1039    {
1040        if (value)
1041        {
1042            Set(index);
1043        }
1044        else
1045        {
1046            Reset(index);
1047        }
1048    }
1049
1050    /// <summary>
1051    /// <para>
1052    /// Sets the index.
1053    /// </para>
1054    /// <para></para>
1055    /// </summary>
1056    /// <param name="index">
1057    /// <para>The index.</para>
1058    /// <para></para>
1059    /// </param>
1060    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1061    public void Set(long index)
1062    {
1063        Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
1064        var wordIndex = GetWordIndexFromIndex(index);
1065        var mask = GetBitMaskFromIndex(index);
1066        _array[wordIndex] |= mask;
1067        RefreshBordersByWord(wordIndex);
1068    }
1069
1070    /// <summary>
1071    /// <para>
1072    /// Resets the index.
1073    /// </para>
1074    /// <para></para>
1075    /// </summary>
1076    /// <param name="index">
1077    /// <para>The index.</para>
1078    /// <para></para>
1079    /// </param>
1080    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081    public void Reset(long index)
1082    {
1083        Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
1084        var wordIndex = GetWordIndexFromIndex(index);
1085        var mask = GetBitMaskFromIndex(index);
1086        _array[wordIndex] &= ~mask;
1087        RefreshBordersByWord(wordIndex);
1088    }
1089
1090    /// <summary>
1091    /// <para>
1092    /// Determines whether this instance add.
1093    /// </para>
1094    /// <para></para>
1095    /// </summary>
1096    /// <param name="index">
1097    /// <para>The index.</para>
1098    /// <para></para>
1099    /// </param>
1100    /// <returns>
1101    /// <para>The bool</para>
1102    /// <para></para>
1103    /// </returns>

```

```

1104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1105 public bool Add(long index)
1106 {
1107     var wordIndex = GetWordIndexFromIndex(index);
1108     var mask = GetBitMaskFromIndex(index);
1109     if ((_array[wordIndex] & mask) == 0)
1110     {
1111         _array[wordIndex] |= mask;
1112         RefreshBordersByWord(wordIndex);
1113         return true;
1114     }
1115     else
1116     {
1117         return false;
1118     }
1119 }
1120
1121 /// <summary>
1122 /// <para>
1123 /// Sets the all using the specified value.
1124 /// </para>
1125 /// <para></para>
1126 /// </summary>
1127 /// <param name="value">
1128 /// <para>The value.</para>
1129 /// <para></para>
1130 /// </param>
1131 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1132 public void SetAll(bool value)
1133 {
1134     if (value)
1135     {
1136         SetAll();
1137     }
1138     else
1139     {
1140         ResetAll();
1141     }
1142 }
1143
1144 /// <summary>
1145 /// <para>
1146 /// Sets the all.
1147 /// </para>
1148 /// <para></para>
1149 /// </summary>
1150 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1151 public void SetAll()
1152 {
1153     const long fillValue = unchecked((long)0xffffffffffffffff);
1154     var words = GetWordsCountFromIndex(_length);
1155     for (var i = 0; i < words; i++)
1156     {
1157         _array[i] = fillValue;
1158     }
1159     MarkBordersAsAllBitsSet();
1160 }
1161
1162 /// <summary>
1163 /// <para>
1164 /// Resets the all.
1165 /// </para>
1166 /// <para></para>
1167 /// </summary>
1168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1169 public void ResetAll()
1170 {
1171     const long fillValue = 0;
1172     GetBorders(out long from, out long to);
1173     for (var i = from; i <= to; i++)
1174     {
1175         _array[i] = fillValue;
1176     }
1177     MarkBordersAsAllBitsReset();
1178 }
1179
1180 /// <summary>
1181 /// <para>

```

```

1182     /// Gets the set indices.
1183     /// </para>
1184     /// <para></para>
1185     /// </summary>
1186     /// <returns>
1187     /// <para>The result.</para>
1188     /// <para></para>
1189     /// </returns>
1190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1191     public List<long> GetSetIndices()
1192     {
1193         var result = new List<long>();
1194         GetBorders(out long from, out long to);
1195         for (var i = from; i <= to; i++)
1196         {
1197             var word = _array[i];
1198             if (word != 0)
1199             {
1200                 AppendAllSetBitIndices(result, i, word);
1201             }
1202         }
1203         return result;
1204     }
1205
1206     /// <summary>
1207     /// <para>
1208     /// Gets the set u int 64 indices.
1209     /// </para>
1210     /// <para></para>
1211     /// </summary>
1212     /// <returns>
1213     /// <para>The result.</para>
1214     /// <para></para>
1215     /// </returns>
1216     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1217     public List<ulong> GetSetUInt64Indices()
1218     {
1219         var result = new List<ulong>();
1220         GetBorders(out ulong from, out ulong to);
1221         for (var i = from; i <= to; i++)
1222         {
1223             var word = _array[i];
1224             if (word != 0)
1225             {
1226                 AppendAllSetBitIndices(result, i, word);
1227             }
1228         }
1229         return result;
1230     }
1231
1232     /// <summary>
1233     /// <para>
1234     /// Gets the first set bit index.
1235     /// </para>
1236     /// <para></para>
1237     /// </summary>
1238     /// <returns>
1239     /// <para>The long</para>
1240     /// <para></para>
1241     /// </returns>
1242     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1243     public long GetFirstSetBitIndex()
1244     {
1245         var i = _minPositiveWord;
1246         var word = _array[i];
1247         if (word != 0)
1248         {
1249             return GetFirstSetBitForWord(i, word);
1250         }
1251         return -1;
1252     }
1253
1254     /// <summary>
1255     /// <para>
1256     /// Gets the last set bit index.
1257     /// </para>
1258     /// <para></para>
1259     /// </summary>

```



```

1260 /// <returns>
1261 /// <para>The long</para>
1262 /// <para></para>
1263 /// </returns>
1264 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1265 public long GetLastSetBitIndex()
1266 {
1267     var i = _maxPositiveWord;
1268     var word = _array[i];
1269     if (word != 0)
1270     {
1271         return GetLastSetBitForWord(i, word);
1272     }
1273     return -1;
1274 }
1275
1276 /// <summary>
1277 /// <para>
1278 /// Counts the set bits.
1279 /// </para>
1280 /// <para></para>
1281 /// </summary>
1282 /// <returns>
1283 /// <para>The total.</para>
1284 /// <para></para>
1285 /// </returns>
1286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1287 public long CountSetBits()
1288 {
1289     var total = 0L;
1290     GetBorders(out long from, out long to);
1291     for (var i = from; i <= to; i++)
1292     {
1293         var word = _array[i];
1294         if (word != 0)
1295         {
1296             total += CountSetBitsForWord(word);
1297         }
1298     }
1299     return total;
1300 }
1301
1302 /// <summary>
1303 /// <para>
1304 /// Determines whether this instance have common bits.
1305 /// </para>
1306 /// <para></para>
1307 /// </summary>
1308 /// <param name="other">
1309 /// <para>The other.</para>
1310 /// <para></para>
1311 /// </param>
1312 /// <returns>
1313 /// <para>The bool</para>
1314 /// <para></para>
1315 /// </returns>
1316 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1317 public bool HaveCommonBits(BitString other)
1318 {
1319     EnsureBitStringHasTheSameSize(other, nameof(other));
1320     GetCommonInnerBorders(this, other, out long from, out long to);
1321     var otherArray = other._array;
1322     for (var i = from; i <= to; i++)
1323     {
1324         var left = _array[i];
1325         var right = otherArray[i];
1326         if (left != 0 && right != 0 && (left & right) != 0)
1327         {
1328             return true;
1329         }
1330     }
1331     return false;
1332 }
1333
1334 /// <summary>
1335 /// <para>
1336 /// Counts the common bits using the specified other.
1337 /// </para>

```

```

1338     /// <para></para>
1339     /// </summary>
1340     /// <param name="other">
1341     /// <para>The other.</para>
1342     /// <para></para>
1343     /// </param>
1344     /// <returns>
1345     /// <para>The total.</para>
1346     /// <para></para>
1347     /// </returns>
1348     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1349     public long CountCommonBits(BitString other)
1350     {
1351         EnsureBitStringHasTheSameSize(other, nameof(other));
1352         GetCommonInnerBorders(this, other, out long from, out long to);
1353         var total = 0L;
1354         var otherArray = other._array;
1355         for (var i = from; i <= to; i++)
1356         {
1357             var left = _array[i];
1358             var right = otherArray[i];
1359             var combined = left & right;
1360             if (combined != 0)
1361             {
1362                 total += CountSetBitsForWord(combined);
1363             }
1364         }
1365         return total;
1366     }
1367
1368     /// <summary>
1369     /// <para>
1370     /// Gets the common indices using the specified other.
1371     /// </para>
1372     /// <para></para>
1373     /// </summary>
1374     /// <param name="other">
1375     /// <para>The other.</para>
1376     /// <para></para>
1377     /// </param>
1378     /// <returns>
1379     /// <para>The result.</para>
1380     /// <para></para>
1381     /// </returns>
1382     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1383     public List<long> GetCommonIndices(BitString other)
1384     {
1385         EnsureBitStringHasTheSameSize(other, nameof(other));
1386         GetCommonInnerBorders(this, other, out long from, out long to);
1387         var result = new List<long>();
1388         var otherArray = other._array;
1389         for (var i = from; i <= to; i++)
1390         {
1391             var left = _array[i];
1392             var right = otherArray[i];
1393             var combined = left & right;
1394             if (combined != 0)
1395             {
1396                 AppendAllSetBitIndices(result, i, combined);
1397             }
1398         }
1399         return result;
1400     }
1401
1402     /// <summary>
1403     /// <para>
1404     /// Gets the common u int 64 indices using the specified other.
1405     /// </para>
1406     /// <para></para>
1407     /// </summary>
1408     /// <param name="other">
1409     /// <para>The other.</para>
1410     /// <para></para>
1411     /// </param>
1412     /// <returns>
1413     /// <para>The result.</para>
1414     /// <para></para>
1415     /// </returns>

```

```

1416 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1417 public List<ulong> GetCommonUInt64Indices(BitString other)
1418 {
1419     EnsureBitStringHasTheSameSize(other, nameof(other));
1420     GetCommonBorders(this, other, out ulong from, out ulong to);
1421     var result = new List<ulong>();
1422     var otherArray = other._array;
1423     for (var i = from; i <= to; i++)
1424     {
1425         var left = _array[i];
1426         var right = otherArray[i];
1427         var combined = left & right;
1428         if (combined != 0)
1429         {
1430             AppendAllSetBitIndices(result, i, combined);
1431         }
1432     }
1433     return result;
1434 }
1435
1436 /// <summary>
1437 /// <para>
1438 /// Gets the first common bit index using the specified other.
1439 /// </para>
1440 /// <para></para>
1441 /// </summary>
1442 /// <param name="other">
1443 /// <para>The other.</para>
1444 /// <para></para>
1445 /// </param>
1446 /// <returns>
1447 /// <para>The long</para>
1448 /// <para></para>
1449 /// </returns>
1450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1451 public long GetFirstCommonBitIndex(BitString other)
1452 {
1453     EnsureBitStringHasTheSameSize(other, nameof(other));
1454     GetCommonInnerBorders(this, other, out long from, out long to);
1455     var otherArray = other._array;
1456     for (var i = from; i <= to; i++)
1457     {
1458         var left = _array[i];
1459         var right = otherArray[i];
1460         var combined = left & right;
1461         if (combined != 0)
1462         {
1463             return GetFirstSetBitForWord(i, combined);
1464         }
1465     }
1466     return -1;
1467 }
1468
1469 /// <summary>
1470 /// <para>
1471 /// Gets the last common bit index using the specified other.
1472 /// </para>
1473 /// <para></para>
1474 /// </summary>
1475 /// <param name="other">
1476 /// <para>The other.</para>
1477 /// <para></para>
1478 /// </param>
1479 /// <returns>
1480 /// <para>The long</para>
1481 /// <para></para>
1482 /// </returns>
1483 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1484 public long GetLastCommonBitIndex(BitString other)
1485 {
1486     EnsureBitStringHasTheSameSize(other, nameof(other));
1487     GetCommonInnerBorders(this, other, out long from, out long to);
1488     var otherArray = other._array;
1489     for (var i = to; i >= from; i--)
1490     {
1491         var left = _array[i];
1492         var right = otherArray[i];
1493         var combined = left & right;

```

```

1494         if (combined != 0)
1495         {
1496             return GetLastSetBitForWord(i, combined);
1497         }
1498     }
1499     return -1;
1500 }
1501
1502 /// <summary>
1503 /// <para>
1504 /// Determines whether this instance equals.
1505 /// </para>
1506 /// <para></para>
1507 /// </summary>
1508 /// <param name="obj">
1509 /// <para>The obj.</para>
1510 /// <para></para>
1511 /// </param>
1512 /// <returns>
1513 /// <para>The bool</para>
1514 /// <para></para>
1515 /// </returns>
1516 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1517 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
    ↪ false;
1518
1519 /// <summary>
1520 /// <para>
1521 /// Determines whether this instance equals.
1522 /// </para>
1523 /// <para></para>
1524 /// </summary>
1525 /// <param name="other">
1526 /// <para>The other.</para>
1527 /// <para></para>
1528 /// </param>
1529 /// <returns>
1530 /// <para>The bool</para>
1531 /// <para></para>
1532 /// </returns>
1533 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1534 public bool Equals(BitString other)
1535 {
1536     if (_length != other._length)
1537     {
1538         return false;
1539     }
1540     var otherArray = other._array;
1541     if (_array.Length != otherArray.Length)
1542     {
1543         return false;
1544     }
1545     if (_minPositiveWord != other._minPositiveWord)
1546     {
1547         return false;
1548     }
1549     if (_maxPositiveWord != other._maxPositiveWord)
1550     {
1551         return false;
1552     }
1553     GetCommonBorders(this, other, out ulong from, out ulong to);
1554     for (var i = from; i <= to; i++)
1555     {
1556         if (_array[i] != otherArray[i])
1557         {
1558             return false;
1559         }
1560     }
1561     return true;
1562 }
1563
1564 /// <summary>
1565 /// <para>
1566 /// Ensures the bit string has the same size using the specified other.
1567 /// </para>
1568 /// <para></para>
1569 /// </summary>
1570 /// <param name="other">

```

```

1571    /// <para>The other.</para>
1572    /// <para></para>
1573    /// </param>
1574    /// <param name="argumentName">
1575    /// <para>The argument name.</para>
1576    /// <para></para>
1577    /// </param>
1578    /// <exception cref="ArgumentException">
1579    /// <para>Bit string must be the same size. </para>
1580    /// <para></para>
1581    /// </exception>
1582    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1583    private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
1584    {
1585        Ensure.Always.ArgumentNotNull(other, argumentName);
1586        if (_length != other._length)
1587        {
1588            throw new ArgumentException("Bit string must be the same size.", argumentName);
1589        }
1590    }
1591
1592    /// <summary>
1593    /// <para>
1594    /// Marks the borders as all bits reset.
1595    /// </para>
1596    /// <para></para>
1597    /// </summary>
1598    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1599    private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
1600
1601    /// <summary>
1602    /// <para>
1603    /// Marks the borders as all bits set.
1604    /// </para>
1605    /// <para></para>
1606    /// </summary>
1607    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1608    private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
1609
1610    /// <summary>
1611    /// <para>
1612    /// Gets the borders using the specified from.
1613    /// </para>
1614    /// <para></para>
1615    /// </summary>
1616    /// <param name="from">
1617    /// <para>The from.</para>
1618    /// <para></para>
1619    /// </param>
1620    /// <param name="to">
1621    /// <para>The to.</para>
1622    /// <para></para>
1623    /// </param>
1624    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1625    private void GetBorders(out long from, out long to)
1626    {
1627        from = _minPositiveWord;
1628        to = _maxPositiveWord;
1629    }
1630
1631    /// <summary>
1632    /// <para>
1633    /// Gets the borders using the specified from.
1634    /// </para>
1635    /// <para></para>
1636    /// </summary>
1637    /// <param name="from">
1638    /// <para>The from.</para>
1639    /// <para></para>
1640    /// </param>
1641    /// <param name="to">
1642    /// <para>The to.</para>
1643    /// <para></para>
1644    /// </param>
1645    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1646    private void GetBorders(out ulong from, out ulong to)
1647    {
1648        from = (ulong)_minPositiveWord;

```

```

1649         to = (ulong)_maxPositiveWord;
1650     }
1651
1652     /// <summary>
1653     /// <para>
1654     /// Sets the borders using the specified from.
1655     /// </para>
1656     /// <para></para>
1657     /// </summary>
1658     /// <param name="from">
1659     /// <para>The from.</para>
1660     /// <para></para>
1661     /// </param>
1662     /// <param name="to">
1663     /// <para>The to.</para>
1664     /// <para></para>
1665     /// </param>
1666     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1667     private void SetBorders(long from, long to)
1668     {
1669         _minPositiveWord = from;
1670         _maxPositiveWord = to;
1671     }
1672
1673     /// <summary>
1674     /// <para>
1675     /// Gets the valid index range.
1676     /// </para>
1677     /// <para></para>
1678     /// </summary>
1679     /// <returns>
1680     /// <para>A range of long</para>
1681     /// <para></para>
1682     /// </returns>
1683     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1684     private Range<long> GetValidIndexRange() => (0, _length - 1);
1685
1686     /// <summary>
1687     /// <para>
1688     /// Gets the valid length range.
1689     /// </para>
1690     /// <para></para>
1691     /// </summary>
1692     /// <returns>
1693     /// <para>A range of long</para>
1694     /// <para></para>
1695     /// </returns>
1696     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1697     private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
1698
1699     /// <summary>
1700     /// <para>
1701     /// Appends the all set bit indices using the specified result.
1702     /// </para>
1703     /// <para></para>
1704     /// </summary>
1705     /// <param name="result">
1706     /// <para>The result.</para>
1707     /// <para></para>
1708     /// </param>
1709     /// <param name="wordIndex">
1710     /// <para>The word index.</para>
1711     /// <para></para>
1712     /// </param>
1713     /// <param name="wordValue">
1714     /// <para>The word value.</para>
1715     /// <para></para>
1716     /// </param>
1717     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1718     private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
1719     ↪ wordValue)
1720     {
1721         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
1722         ↪ bits32to47, out byte[] bits48to63);
1723         AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
1724         ↪ bits48to63);
1725     }

```

```

1724     /// <summary>
1725     /// <para>
1726     /// Appends the all set bit indices using the specified result.
1727     /// </para>
1728     /// <para></para>
1729     /// </summary>
1730     /// <param name="result">
1731     /// <para>The result.</para>
1732     /// <para></para>
1733     /// </param>
1734     /// <param name="wordIndex">
1735     /// <para>The word index.</para>
1736     /// <para></para>
1737     /// </param>
1738     /// <param name="wordValue">
1739     /// <para>The word value.</para>
1740     /// <para></para>
1741     /// </param>
1742     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1743     private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
        ↪ wordValue)
1744     {
1745         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
        ↪ bits32to47, out byte[] bits48to63);
1746         AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
        ↪ bits48to63);
1747     }
1748
1749     /// <summary>
1750     /// <para>
1751     /// Counts the set bits for word using the specified word.
1752     /// </para>
1753     /// <para></para>
1754     /// </summary>
1755     /// <param name="word">
1756     /// <para>The word.</para>
1757     /// <para></para>
1758     /// </param>
1759     /// <returns>
1760     /// <para>The long</para>
1761     /// <para></para>
1762     /// </returns>
1763     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1764     private static long CountSetBitsForWord(long word)
1765     {
1766         GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
        ↪ out byte[] bits48to63);
1767         return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
        ↪ bits48to63.LongLength;
1768     }
1769
1770     /// <summary>
1771     /// <para>
1772     /// Gets the first set bit for word using the specified word index.
1773     /// </para>
1774     /// <para></para>
1775     /// </summary>
1776     /// <param name="wordIndex">
1777     /// <para>The word index.</para>
1778     /// <para></para>
1779     /// </param>
1780     /// <param name="wordValue">
1781     /// <para>The word value.</para>
1782     /// <para></para>
1783     /// </param>
1784     /// <returns>
1785     /// <para>The long</para>
1786     /// <para></para>
1787     /// </returns>
1788     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1789     private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
1790     {
1791         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
        ↪ bits32to47, out byte[] bits48to63);
1792         return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
1793     }
1794

```

```

1795     /// <summary>
1796     /// <para>
1797     /// Gets the last set bit for word using the specified word index.
1798     /// </para>
1799     /// <para></para>
1800     /// </summary>
1801     /// <param name="wordIndex">
1802     /// <para>The word index.</para>
1803     /// <para></para>
1804     /// </param>
1805     /// <param name="wordValue">
1806     /// <para>The word value.</para>
1807     /// <para></para>
1808     /// </param>
1809     /// <returns>
1810     /// <para>The long</para>
1811     /// <para></para>
1812     /// </returns>
1813     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1814     private static long GetLastSetBitForWord(long wordIndex, long wordValue)
1815     {
1816         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪ bits32to47, out byte[] bits48to63);
1817         return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
1818     }
1819
1820     /// <summary>
1821     /// <para>
1822     /// Appends the all set bit indices using the specified result.
1823     /// </para>
1824     /// <para></para>
1825     /// </summary>
1826     /// <param name="result">
1827     /// <para>The result.</para>
1828     /// <para></para>
1829     /// </param>
1830     /// <param name="i">
1831     /// <para>The .</para>
1832     /// <para></para>
1833     /// </param>
1834     /// <param name="bits00to15">
1835     /// <para>The bits 00to 15.</para>
1836     /// <para></para>
1837     /// </param>
1838     /// <param name="bits16to31">
1839     /// <para>The bits 16to 31.</para>
1840     /// <para></para>
1841     /// </param>
1842     /// <param name="bits32to47">
1843     /// <para>The bits 32to 47.</para>
1844     /// <para></para>
1845     /// </param>
1846     /// <param name="bits48to63">
1847     /// <para>The bits 48to 63.</para>
1848     /// <para></para>
1849     /// </param>
1850     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1851     private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
            ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1852     {
1853         for (var j = 0; j < bits00to15.Length; j++)
1854         {
1855             result.Add(bits00to15[j] + (i * 64));
1856         }
1857         for (var j = 0; j < bits16to31.Length; j++)
1858         {
1859             result.Add(bits16to31[j] + 16 + (i * 64));
1860         }
1861         for (var j = 0; j < bits32to47.Length; j++)
1862         {
1863             result.Add(bits32to47[j] + 32 + (i * 64));
1864         }
1865         for (var j = 0; j < bits48to63.Length; j++)
1866         {
1867             result.Add(bits48to63[j] + 48 + (i * 64));
1868         }
1869     }
1870

```



```

1871     /// <summary>
1872     /// <para>
1873     /// Appends the all set indices using the specified result.
1874     /// </para>
1875     /// <para></para>
1876     /// </summary>
1877     /// <param name="result">
1878     /// <para>The result.</para>
1879     /// <para></para>
1880     /// </param>
1881     /// <param name="i">
1882     /// <para>The .</para>
1883     /// <para></para>
1884     /// </param>
1885     /// <param name="bits00to15">
1886     /// <para>The bits 00to 15.</para>
1887     /// <para></para>
1888     /// </param>
1889     /// <param name="bits16to31">
1890     /// <para>The bits 16to 31.</para>
1891     /// <para></para>
1892     /// </param>
1893     /// <param name="bits32to47">
1894     /// <para>The bits 32to 47.</para>
1895     /// <para></para>
1896     /// </param>
1897     /// <param name="bits48to63">
1898     /// <para>The bits 48to 63.</para>
1899     /// <para></para>
1900     /// </param>
1901     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1902     private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1903     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1904     {
1905         for (var j = 0; j < bits00to15.Length; j++)
1906         {
1907             result.Add(bits00to15[j] + (i * 64));
1908         }
1909         for (var j = 0; j < bits16to31.Length; j++)
1910         {
1911             result.Add(bits16to31[j] + 16UL + (i * 64));
1912         }
1913         for (var j = 0; j < bits32to47.Length; j++)
1914         {
1915             result.Add(bits32to47[j] + 32UL + (i * 64));
1916         }
1917         for (var j = 0; j < bits48to63.Length; j++)
1918         {
1919             result.Add(bits48to63[j] + 48UL + (i * 64));
1920         }
1921     }
1922     /// <summary>
1923     /// <para>
1924     /// Gets the first set bit using the specified i.
1925     /// </para>
1926     /// <para></para>
1927     /// </summary>
1928     /// <param name="i">
1929     /// <para>The .</para>
1930     /// <para></para>
1931     /// </param>
1932     /// <param name="bits00to15">
1933     /// <para>The bits 00to 15.</para>
1934     /// <para></para>
1935     /// </param>
1936     /// <param name="bits16to31">
1937     /// <para>The bits 16to 31.</para>
1938     /// <para></para>
1939     /// </param>
1940     /// <param name="bits32to47">
1941     /// <para>The bits 32to 47.</para>
1942     /// <para></para>
1943     /// </param>
1944     /// <param name="bits48to63">
1945     /// <para>The bits 48to 63.</para>
1946     /// <para></para>
1947     /// </param>

```

```

1948     /// <returns>
1949     /// <para>The long</para>
1950     /// <para></para>
1951     /// </returns>
1952     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1953     private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
    ↪ bits32to47, byte[] bits48to63)
1954     {
1955         if (bits00to15.Length > 0)
1956         {
1957             return bits00to15[0] + (i * 64);
1958         }
1959         if (bits16to31.Length > 0)
1960         {
1961             return bits16to31[0] + 16 + (i * 64);
1962         }
1963         if (bits32to47.Length > 0)
1964         {
1965             return bits32to47[0] + 32 + (i * 64);
1966         }
1967         return bits48to63[0] + 48 + (i * 64);
1968     }
1969
1970     /// <summary>
1971     /// <para>
1972     /// Gets the last set bit using the specified i.
1973     /// </para>
1974     /// <para></para>
1975     /// </summary>
1976     /// <param name="i">
1977     /// <para>The .</para>
1978     /// <para></para>
1979     /// </param>
1980     /// <param name="bits00to15">
1981     /// <para>The bits 00to 15.</para>
1982     /// <para></para>
1983     /// </param>
1984     /// <param name="bits16to31">
1985     /// <para>The bits 16to 31.</para>
1986     /// <para></para>
1987     /// </param>
1988     /// <param name="bits32to47">
1989     /// <para>The bits 32to 47.</para>
1990     /// <para></para>
1991     /// </param>
1992     /// <param name="bits48to63">
1993     /// <para>The bits 48to 63.</para>
1994     /// <para></para>
1995     /// </param>
1996     /// <returns>
1997     /// <para>The long</para>
1998     /// <para></para>
1999     /// </returns>
2000     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2001     private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
    ↪ bits32to47, byte[] bits48to63)
2002     {
2003         if (bits48to63.Length > 0)
2004         {
2005             return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
2006         }
2007         if (bits32to47.Length > 0)
2008         {
2009             return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
2010         }
2011         if (bits16to31.Length > 0)
2012         {
2013             return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
2014         }
2015         return bits00to15[bits00to15.Length - 1] + (i * 64);
2016     }
2017
2018     /// <summary>
2019     /// <para>
2020     /// Gets the bits using the specified word.
2021     /// </para>
2022     /// <para></para>
2023     /// </summary>

```

```

2024    /// <param name="word">
2025    /// <para>The word.</para>
2026    /// <para></para>
2027    /// </param>
2028    /// <param name="bits00to15">
2029    /// <para>The bits 00to 15.</para>
2030    /// <para></para>
2031    /// </param>
2032    /// <param name="bits16to31">
2033    /// <para>The bits 16to 31.</para>
2034    /// <para></para>
2035    /// </param>
2036    /// <param name="bits32to47">
2037    /// <para>The bits 32to 47.</para>
2038    /// <para></para>
2039    /// </param>
2040    /// <param name="bits48to63">
2041    /// <para>The bits 48to 63.</para>
2042    /// <para></para>
2043    /// </param>
2044    [MethodImpl(MethodImplOptions.AggressiveInlining)]
2045    private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    → byte[] bits32to47, out byte[] bits48to63)
2046    {
2047        bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
2048        bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
2049        bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
2050        bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
2051    }
2052
2053    /// <summary>
2054    /// <para>
2055    /// Gets the common inner borders using the specified left.
2056    /// </para>
2057    /// <para></para>
2058    /// </summary>
2059    /// <param name="left">
2060    /// <para>The left.</para>
2061    /// <para></para>
2062    /// </param>
2063    /// <param name="right">
2064    /// <para>The right.</para>
2065    /// <para></para>
2066    /// </param>
2067    /// <param name="from">
2068    /// <para>The from.</para>
2069    /// <para></para>
2070    /// </param>
2071    /// <param name="to">
2072    /// <para>The to.</para>
2073    /// <para></para>
2074    /// </param>
2075    [MethodImpl(MethodImplOptions.AggressiveInlining)]
2076    public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    → out long to)
2077    {
2078        from = Math.Max(left._minPositiveWord, right._minPositiveWord);
2079        to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
2080    }
2081
2082    /// <summary>
2083    /// <para>
2084    /// Gets the common outer borders using the specified left.
2085    /// </para>
2086    /// <para></para>
2087    /// </summary>
2088    /// <param name="left">
2089    /// <para>The left.</para>
2090    /// <para></para>
2091    /// </param>
2092    /// <param name="right">
2093    /// <para>The right.</para>
2094    /// <para></para>
2095    /// </param>
2096    /// <param name="from">
2097    /// <para>The from.</para>
2098    /// <para></para>
2099    /// </param>

```

```

2100     /// <param name="to">
2101     /// <para>The to.</para>
2102     /// <para></para>
2103     /// </param>
2104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2105     public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
2106     ↪ out long to)
2107     {
2108         from = Math.Min(left._minPositiveWord, right._minPositiveWord);
2109         to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
2110     }
2111     /// <summary>
2112     /// <para>
2113     /// Gets the common outer borders using the specified left.
2114     /// </para>
2115     /// <para></para>
2116     /// </summary>
2117     /// <param name="left">
2118     /// <para>The left.</para>
2119     /// <para></para>
2120     /// </param>
2121     /// <param name="right">
2122     /// <para>The right.</para>
2123     /// <para></para>
2124     /// </param>
2125     /// <param name="from">
2126     /// <para>The from.</para>
2127     /// <para></para>
2128     /// </param>
2129     /// <param name="to">
2130     /// <para>The to.</para>
2131     /// <para></para>
2132     /// </param>
2133     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2134     public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
2135     ↪ out int to)
2136     {
2137         from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
2138         to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
2139     }
2140     /// <summary>
2141     /// <para>
2142     /// Gets the common borders using the specified left.
2143     /// </para>
2144     /// <para></para>
2145     /// </summary>
2146     /// <param name="left">
2147     /// <para>The left.</para>
2148     /// <para></para>
2149     /// </param>
2150     /// <param name="right">
2151     /// <para>The right.</para>
2152     /// <para></para>
2153     /// </param>
2154     /// <param name="from">
2155     /// <para>The from.</para>
2156     /// <para></para>
2157     /// </param>
2158     /// <param name="to">
2159     /// <para>The to.</para>
2160     /// <para></para>
2161     /// </param>
2162     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2163     public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
2164     ↪ ulong to)
2165     {
2166         from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
2167         to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
2168     }
2169     /// <summary>
2170     /// <para>
2171     /// Gets the words count from index using the specified index.
2172     /// </para>
2173     /// <para></para>
2174     /// </summary>

```

```

2175     /// <param name="index">
2176     /// <para>The index.</para>
2177     /// <para></para>
2178     /// </param>
2179     /// <returns>
2180     /// <para>The long</para>
2181     /// <para></para>
2182     /// </returns>
2183     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2184     public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
2185
2186     /// <summary>
2187     /// <para>
2188     /// Gets the word index from index using the specified index.
2189     /// </para>
2190     /// <para></para>
2191     /// </summary>
2192     /// <param name="index">
2193     /// <para>The index.</para>
2194     /// <para></para>
2195     /// </param>
2196     /// <returns>
2197     /// <para>The long</para>
2198     /// <para></para>
2199     /// </returns>
2200     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2201     public static long GetWordIndexFromIndex(long index) => index >> 6;
2202
2203     /// <summary>
2204     /// <para>
2205     /// Gets the bit mask from index using the specified index.
2206     /// </para>
2207     /// <para></para>
2208     /// </summary>
2209     /// <param name="index">
2210     /// <para>The index.</para>
2211     /// <para></para>
2212     /// </param>
2213     /// <returns>
2214     /// <para>The long</para>
2215     /// <para></para>
2216     /// </returns>
2217     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2218     public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
2219
2220     /// <summary>
2221     /// <para>
2222     /// Gets the hash code.
2223     /// </para>
2224     /// <para></para>
2225     /// </summary>
2226     /// <returns>
2227     /// <para>The int</para>
2228     /// <para></para>
2229     /// </returns>
2230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2231     public override int GetHashCode() => base.GetHashCode();
2232
2233     /// <summary>
2234     /// <para>
2235     /// Returns the string.
2236     /// </para>
2237     /// <para></para>
2238     /// </summary>
2239     /// <returns>
2240     /// <para>The string</para>
2241     /// <para></para>
2242     /// </returns>
2243     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2244     public override string ToString() => base.ToString();
2245 }
2246

```

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
36        ↪ value) ? value : default;
37
38        /// <summary>
39        /// <para>
40        /// Peeks 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 PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
58        ↪ value) ? value : default;
59    }
60 }
```

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 }
```

```

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,
55      ↪ ICollection<T> argument, string argumentName, string message)
56      {
57          if (argument.IsNullOrEmpty())
58          {
59              throw new ArgumentException(message, argumentName);
60          }
61      }
62
63      /// <summary>
64      /// <para>
65      /// Arguments the not empty using the specified root.
66      /// </para>
67      /// <para></para>
68      /// </summary>
69      /// <typeparam name="T">
70      /// <para>The .</para>
71      /// <para></para>
72      /// </typeparam>
73      /// <param name="root">
74      /// <para>The root.</para>
75      /// <para></para>
76      /// </param>
77      /// <param name="argument">
78      /// <para>The argument.</para>
79      /// <para></para>
80      /// </param>
81      /// <param name="argumentName">
82      /// <para>The argument name.</para>
83      /// <para></para>
84      /// </param>
85      [MethodImpl(MethodImplOptions.AggressiveInlining)]
86      public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
87      ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
88      ↪ argumentName, null);
89
90      /// <summary>
91      /// <para>
92      /// Arguments the not empty using the specified root.
93      /// </para>
94      /// <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,
    ↪    string argument, string argumentName, string message)
136    {
137        if (string.IsNullOrEmpty(argument))
138        {
139            throw new ArgumentException(message, argumentName);
140        }
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,
    ↪    string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
    ↪    argument, argumentName, null);
163
164    /// <summary>

```

```

165     /// <para>
166     /// Arguments the not empty and not white space using the specified root.
167     /// </para>
168     /// <para></para>
169     /// </summary>
170     /// <param name="root">
171     /// <para>The root.</para>
172     /// <para></para>
173     /// </param>
174     /// <param name="argument">
175     /// <para>The argument.</para>
176     /// <para></para>
177     /// </param>
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
180         ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
181
182     #endregion
183
184     #region OnDebug
185
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>
208     /// <param name="message">
209     /// <para>The message.</para>
210     /// <para></para>
211     /// </param>
212     [Conditional("DEBUG")]
213     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
214         ↪ ICollection<T> argument, string argumentName, string message) =>
215         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
216
217     /// <summary>
218     /// <para>
219     /// Arguments the not empty using the specified root.
220     /// </para>
221     /// <para></para>
222     /// </summary>
223     /// <typeparam name="T">
224     /// <para>The .</para>
225     /// <para></para>
226     /// </typeparam>
227     /// <param name="root">
228     /// <para>The root.</para>
229     /// <para></para>
230     /// </param>
231     /// <param name="argument">
232     /// <para>The argument.</para>
233     /// <para></para>
234     /// </param>
235     /// <param name="argumentName">
236     /// <para>The argument name.</para>
237     /// <para></para>
238     /// </param>
239     [Conditional("DEBUG")]
240     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
241         ↪ ICollection<T> argument, string argumentName) =>
242         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);

```

```

238
239     /// <summary>
240     /// <para>
241     /// Arguments the not empty using the specified root.
242     /// </para>
243     /// <para></para>
244     /// </summary>
245     /// <typeparam name="T">
246     /// <para>The .</para>
247     /// <para></para>
248     /// </typeparam>
249     /// <param name="root">
250     /// <para>The root.</para>
251     /// <para></para>
252     /// </param>
253     /// <param name="argument">
254     /// <para>The argument.</para>
255     /// <para></para>
256     /// </param>
257     [Conditional("DEBUG")]
258     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
259     ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
260
261     /// <summary>
262     /// <para>
263     /// Arguments the not empty and not white space using the specified root.
264     /// </para>
265     /// <para></para>
266     /// </summary>
267     /// <param name="root">
268     /// <para>The root.</para>
269     /// <para></para>
270     /// </param>
271     /// <param name="argument">
272     /// <para>The argument.</para>
273     /// <para></para>
274     /// </param>
275     /// <param name="argumentName">
276     /// <para>The argument name.</para>
277     /// <para></para>
278     /// </param>
279     /// <param name="message">
280     /// <para>The message.</para>
281     /// <para></para>
282     /// </param>
283     [Conditional("DEBUG")]
284     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
285     ↪ root, string argument, string argumentName, string message) =>
286     ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
287
288     /// <summary>
289     /// <para>
290     /// Arguments the not empty and not white space using the specified root.
291     /// </para>
292     /// <para></para>
293     /// </summary>
294     /// <param name="root">
295     /// <para>The root.</para>
296     /// <para></para>
297     /// </param>
298     /// <param name="argument">
299     /// <para>The argument.</para>
300     /// <para></para>
301     /// </param>
302     /// <param name="argumentName">
303     /// <para>The argument name.</para>
304     /// <para></para>
305     /// </param>
306     [Conditional("DEBUG")]
307     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
308     ↪ root, string argument, string argumentName) =>
309     ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
310
311     /// <summary>
312     /// <para>
313     /// Arguments the not empty and not white space using the specified root.
314     /// </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>Метод принимает коллекцию элементов <see cref="ICollection<T>" /> типа.</para>
23         /// </param>
24         /// <returns>
25         /// <para>Returns a <see cref="bool" /> type variable equal to False if the collection is
26         ↪ empty else returns true.</para>
27         /// <para>Возвращает переменную типа <see cref="bool" /> равной false если коллекция
28         ↪ пустая иначе возвращает true.</para>
29         /// </returns>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
32         ↪ null || collection.Count == 0;
33
34         /// <summary>
35         /// <para>
36         /// <para>Determines whether all equal to default.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         /// <typeparam name="T">
41         /// <para>The .</para>
42         /// <para></para>
43         /// </typeparam>
44         /// <param name="collection">
45         /// <para>The collection.</para>
46         /// <para></para>
47         /// </param>
48         /// <returns>
49         /// <para>The bool</para>
50         /// <para></para>
51         /// </returns>
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
54         {
55             var equalityComparer = EqualityComparer<T>.Default;
56             return collection.All(item => equalityComparer.Equals(item, default));
57         }
58     }
59 }

```

```
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 GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
45        → dictionary, TKey key)
46        {
47            dictionary.TryGetValue(key, out TValue value);
48            return value;
49        }
50
51        /// <summary>
52        /// <para>
53        /// Gets the or add using the specified dictionary.
54        /// </para>
55        /// <para></para>
56        /// </summary>
57        /// <typeparam name="TKey">
58        /// <para>The key.</para>
59        /// <para></para>
60        /// </typeparam>
61        /// <typeparam name="TValue">
62        /// <para>The value.</para>
63        /// <para></para>
64        /// </typeparam>
65        /// <param name="dictionary">
66        /// <para>The dictionary.</para>
67        /// <para></para>
68        /// </param>
69        /// <param name="key">
70        /// <para>The key.</para>
71        /// <para></para>
72        /// </param>
73        /// <param name="valueFactory">
74        /// <para>The value factory.</para>
75        /// <para></para>
```

```

75     /// </param>
76     /// <returns>
77     /// <para>The value.</para>
78     /// <para></para>
79     /// </returns>
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
82     ↪ TKey key, Func<TKey, TValue> valueFactory)
83     {
84         if (!dictionary.TryGetValue(key, out TValue value))
85         {
86             value = valueFactory(key);
87             dictionary.Add(key, value);
88             return value;
89         }
90         return value;
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
16         ↪ elements.</para>
17         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
18         /// </summary>
19         /// <param name="list"><para>The list to be hashed.</para><para>Список для
20         ↪ хеширования.</para></param>
21         /// <returns>
22         /// <para>The hash code of the list.</para>
23         /// <para>Хэш-код списка.</para>
24         /// </returns>
25         /// <remarks>
26         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c7831a3eda37d3d4cd10/mscorlib/system/string.cs#L833
27         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
28         /// </remarks>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public static int GenerateHashCode(this IList<char> list)
31         {
32             var hashSeed = 5381;
33             var hashAccumulator = hashSeed;
34             for (var i = 0; i < list.Count; i++)
35             {
36                 hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
37             }
38             return hashAccumulator + (hashSeed * 1566083941);
39         }
40
41         /// <summary>
42         /// <para>Compares two lists for equality.</para>
43         /// <para>Сравнивает два списка на равенство.</para>
44         /// </summary>
45         /// <param name="left"><para>The first compared list.</para><para>Первый список для
46         ↪ сравнения.</para></param>
47         /// <param name="right"><para>The second compared list.</para><para>Второй список для
48         ↪ сравнения.</para></param>
49         /// <returns>
50         /// <para>True, if the passed lists are equal to each other otherwise false.</para>
51         /// <para>True, если переданные списки равны друг другу, иначе false.</para>
52         /// </returns>
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public static bool EqualTo(this IList<char> left, IList<char> right) =>
55             ↪ left.EqualTo(right, ContentEqualTo);
56
57         /// <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
46     ///     ↪ списков <paramref name="left" /> и <paramref name="right" /> как показано в
47     ///     ↪ следующей таблице.
48     ///     <list type="table">
49     ///         <listheader>
50     ///             <term>Значение</term>
51     ///             <description>Смысл</description>
52     ///         </listheader>
53     ///         <item>
54     ///             <term>Меньше нуля</term>
55     ///             <description>Первый не равный элемент <paramref name="left" /> списка
56     ///             ↪ меньше первого неравного элемента <paramref name="right" /> списка.</description>
57     ///         </item>
58     ///         <item>
59     ///             <term>Ноль</term>
60     ///             <description>Все элементы <paramref name="left" /> списка равны всем
61     ///             ↪ элементам <paramref name="right" /> списка.</description>
62     ///         </item>
63     ///         <item>
64     ///             <term>Больше нуля</term>
65     ///             <description>Первый не равный элемент <paramref name="left" /> списка
66     ///             ↪ больше первого неравного элемента <paramref name="right" /> списка.</description>
67     ///         </item>
68     ///     </list>
69     /// </para>
70     /// </returns>
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
73 }
74 }

```

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>

```



```

37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public int GetHashCode(IList<T> list) => list.GenerateHashCode();
39 }
40 }

```

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         /// <summary>
39         /// <para>Checks if a list is passed, checks its length, and if successful, copies the
40         ///     → value of list [index] into the element variable. Otherwise, the element variable has
41         ///     → a default value.</para>
42         /// <para>Проверяет, передан ли список, сверяет его длину и в случае успеха копирует
43         ///     → значение list[index] в переменную element. Иначе переменная element имеет значение
44         ///     → по умолчанию.</para>
45         /// </summary>
46         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
47         ///     → списка.</para></typeparam>
48         /// <param name="list"><para>The checked list.</para><para>Список для
49         ///     → проверки.</para></param>
50         /// <param name="index"><para>The index of element.</para><para>Индекс
51         ///     → элемента.</para></param>
52         /// <param name="element"><para>Variable for passing the index
53         ///     → value.</para><para>Переменная для передачи значения индекса.</para></param>
54         /// <returns>
55         /// <para>True on success, false otherwise.</para>
56         /// <para>True в случае успеха, иначе false.</para>
57         /// </returns>
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
60         {
61             if (list != null && list.Count > index)
62             {
63                 element = list[index];
64                 return true;
65             }
66             else
67             {
68                 element = default;
69                 return false;
70             }
71         }
72     }
73 }

```

```

55 /// <summary>
56 /// <para>Adds a value to the list.</para>
57 /// <para>Добавляет значение в список.</para>
58 /// </summary>
59 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
60 → списка.</para></typeparam>
61 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
62 → нужно добавить значение.</para></param>
63 /// <param name="element"><para>The item to add to the list.</para><para>Элемент который
64 → нужно добавить в список.</para></param>
65 /// <returns>
66 /// <para>True value in any case.</para>
67 /// <para>Значение true в любом случае.</para>
68 /// </returns>
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
71 {
72     list.Add(element);
73     return true;
74 }
75
76 /// <summary>
77 /// <para>Adds the value with first index from other list to this list.</para>
78 /// <para>Добавляет в этот список значение с первым индексом из другого списка.</para>
79 /// </summary>
80 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
81 → списка.</para></typeparam>
82 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
83 → нужно добавить значение.</para></param>
84 /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
85 → который нужно добавить в список</para></param>
86 /// <returns>
87 /// <para>True value in any case.</para>
88 /// <para>Значение true в любом случае.</para>
89 /// </returns>
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
92 {
93     list.AddFirst(elements);
94     return true;
95 }
96
97 /// <summary>
98 /// <para>Adds a value to the list at the first index.</para>
99 /// <para>Добавляет значение в список по первому индексу.</para>
100 /// </summary>
101 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
102 → списка.</para></typeparam>
103 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
104 → нужно добавить значение.</para></param>
105 /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
106 → который нужно добавить в список</para></param>
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
109     list.Add(elements[0]);
110
111 /// <summary>
112 /// <para>Adds all elements from other list to this list and returns true.</para>
113 /// <para>Добавляет все элементы из другого списка в этот список и возвращает
114 → true.</para>
115 /// </summary>
116 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
117 → списка.</para></typeparam>
118 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
119 → нужно добавить значения.</para></param>
120 /// <param name="elements"><para>List of values to add.</para><para>Список значений
121 → которые необходимо добавить.</para></param>
122 /// <returns>
123 /// <para>True value in any case.</para>
124 /// <para>Значение true в любом случае.</para>
125 /// </returns>
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
128 {
129     list.AddAll(elements);
130     return true;
131 }

```

```

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
    → проверки.</para></param>
214 /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
215 /// <param name="contentEqualityComparer"><para>Function to test two lists for their
    → content equality.</para><para>Функция для проверки двух списков на равенство их
    → содержимого.</para></param>
216 /// <returns>
217 /// <para>If the passed lists are equal to each other, true is returned, otherwise
    → false.</para>
218 /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
    → false.</para>
219 /// </returns>
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
    → IList<T>, bool> contentEqualityComparer)
222 {
223     if (ReferenceEquals(left, right))
224     {
225         return true;
226     }
227     var leftCount = left.GetCountOrZero();
228     var rightCount = right.GetCountOrZero();
229     if (leftCount == 0 && rightCount == 0)
230     {
231         return true;
232     }
233     if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
234     {
235         return false;
236     }
237     return contentEqualityComparer(left, right);
238 }
239
240 /// <summary>

```

```

241 /// <para>Compares each element in the list for identity.</para>
242 /// <para>Сравнивает на равенство каждый элемент списка.</para>
243 /// </summary>
244 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
245 /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>
246 /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
247 /// <returns>
248 /// <para>If at least one element of one list is not equal to the corresponding element
    → from another list returns false, otherwise - true.</para>
249 /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
    → из другого списка возвращает false, иначе - true.</para>
250 /// </returns>
251 [MethodImpl(MethodImplOptions.AggressiveInlining)]
252 public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
253 {
254     var equalityComparer = EqualityComparer<T>.Default;
255     for (var i = left.Count - 1; i >= 0; --i)
256     {
257         if (!equalityComparer.Equals(left[i], right[i]))
258         {
259             return false;
260         }
261     }
262     return true;
263 }
264
265 /// <summary>
266 /// <para>Creates an array by copying all elements from the list that satisfy the
    → predicate. If no list is passed, null is returned.</para>
267 /// <para>Создаёт массив, копируя из списка все элементы которые удовлетворяют
    → предикату. Если список не передан, возвращается null.</para>
268 /// </summary>
269 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
270 /// <param name="list"><para>The list to copy from.</para><para>Список для копирования.</para></param>
271 /// <param name="predicate"><para>A function that determines whether an element should
    → be copied.</para><para>Функция определяющая должен ли копироваться
    → элемент.</para></param>
272 /// <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
    → списка.</para></typeparam>
299 /// <param name="list"><para>The list to copy from.</para><para>Список для
    → копирования.</para></param>
300 /// <returns>
301 /// <para>An array with all the elements of the passed list.</para>
302 /// <para>Массив со всеми элементами переданного списка.</para>
303 /// </returns>
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 public static T[] ToArray<T>(this IList<T> list)

```

```

306 {
307     var array = new T[list.Count];
308     list.CopyTo(array, 0);
309     return array;
310 }
311
312 /// <summary>
313 /// <para>Executes the passed action for each item in the list.</para>
314 /// <para>Выполняет переданное действие для каждого элемента в списке.</para>
315 /// </summary>
316 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
317 /// <param name="list"><para>The list of elements for which the action will be
    ↳ executed.</para><para>Список элементов для которых будет выполняться
    ↳ действие.</para></param>
318 /// <param name="action"><para>A function that will be called for each element of the
    ↳ list.</para><para>Функция которая будет вызываться для каждого элемента
    ↳ списка.</para></param>
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public static void ForEach<T>(this IList<T> list, Action<T> action)
321 {
322     for (var i = 0; i < list.Count; i++)
323     {
324         action(list[i]);
325     }
326 }
327
328 /// <summary>
329 /// <para>Generates a hash code for the entire list based on the values of its
    ↳ elements.</para>
330 /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
331 /// </summary>
332 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
333 /// <param name="list"><para>Hash list.</para><para>Список для
    ↳ хеширования.</para></param>
334 /// <returns>
335 /// <para>The hash code of the list.</para>
336 /// <para>Хэш-код списка.</para>
337 /// </returns>
338 /// <remarks>
339 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
    ↳ -overridden-system-object-gethashcode
340 /// </remarks>
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 public static int GenerateHashCode<T>(this IList<T> list)
343 {
344     var hashAccumulator = 17;
345     for (var i = 0; i < list.Count; i++)
346     {
347         hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
348     }
349     return hashAccumulator;
350 }
351
352 /// <summary>
353 /// <para>Compares two lists.</para>
354 /// <para>Сравнивает два списка.</para>
355 /// </summary>
356 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
357 /// <param name="left"><para>The first compared list.</para><para>Первый список для
    ↳ сравнения.</para></param>
358 /// <param name="right"><para>The second compared list.</para><para>Второй список для
    ↳ сравнения.</para></param>
359 /// <returns>
360 /// <para>
361 ///     A signed integer that indicates the relative values of <paramref name="left" />
    ↳ and <paramref name="right" /> lists' elements, as shown in the following table.
362 ///     <list type="table">
363 ///         <listheader>
364 ///             <term>Value</term>
365 ///             <description>Meaning</description>
366 ///         </listheader>
367 ///         <item>
368 ///             <term>Is less than zero</term>

```

```

369     <description>First non equal element of <paramref name="left" /> list is
    → less than first not equal element of <paramref name="right" /> list.</description>
370 </item>
371 <item>
372 <term>Zero</term>
373 <description>All elements of <paramref name="left" /> list equals to all
    → elements of <paramref name="right" /> list.</description>
374 </item>
375 <item>
376 <term>Is greater than zero</term>
377 <description>First non equal element of <paramref name="left" /> list is
    → greater than first not equal element of <paramref name="right" /> list.</description>
378 </item>
379 </list>
380 </para>
381 <para>
382 <table>
    Целое число со знаком, которое указывает относительные значения элементов
    → списков <paramref name="left" /> и <paramref name="right" /> как показано в
    → следующей таблице.
383 <thead>
384 <tr>
385 <th>Значение</th>
386 <th>Смысл</th>
387 </tr>
388 <tbody>
389 <tr>
390 <td>Меньше нуля</td>
    <description>Первый не равный элемент <paramref name="left" /> списка
    → меньше первого неравного элемента <paramref name="right" /> списка.</description>
391 </tr>
392 <tr>
393 <td>Ноль</td>
    <description>Все элементы <paramref name="left" /> списка равны всем
    → элементам <paramref name="right" /> списка.</description>
394 </tr>
395 <tr>
396 <td>Больше нуля</td>
    <description>Первый не равный элемент <paramref name="left" /> списка
    → больше первого неравного элемента <paramref name="right" /> списка.</description>
397 </tr>
398 </tbody>
399 </table>
400 </para>
401 </returns>
402 [MethodImpl(MethodImplOptions.AggressiveInlining)]
403 public static int CompareTo<T>(this IList<T> left, IList<T> right)
404 {
405     var comparer = Comparer<T>.Default;
406     var leftCount = left.GetCountOrZero();
407     var rightCount = right.GetCountOrZero();
408     var intermediateResult = leftCount.CompareTo(rightCount);
409     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
410     {
411         intermediateResult = comparer.Compare(left[i], right[i]);
412     }
413     return intermediateResult;
414 }
415
416 <summary>
417 <para>Skips one element in the list and builds an array from the remaining
    → elements.</para>
418 <para>Пропускает один элемент списка и составляет из оставшихся элементов
    → массив.</para>
419 </summary>
420 <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
421 <param name="list"><para>The list to copy from.</para><para>Список для
    → копирования.</para></param>
422 </returns>
423 <para>If the list is empty, returns an empty array, otherwise - an array with a
    → missing first element.</para>
424 <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
    → первым элементом.</para>
425 </returns>
426 [MethodImpl(MethodImplOptions.AggressiveInlining)]
427 public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
428
429 <summary>
430

```

```

431 /// <para>Skips the specified number of elements in the list and builds an array from
432   ↳ the remaining elements.</para>
433 /// <para>Пропускает указанное количество элементов списка и составляет из оставшихся
434   ↳ элементов массив.</para>
435 /// </summary>
436 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
437   ↳ списка.</para></typeparam>
438 /// <param name="list"><para>The list to copy from.</para><para>Список для
439   ↳ копирования.</para></param>
440 /// <param name="skip"><para>The number of items to skip.</para><para>Количество
441   ↳ пропускаемых элементов.</para></param>
442 /// <returns>
443 /// <para>If the list is empty, or the number of skipped elements is greater than the
444   ↳ list, returns an empty array, otherwise - an array with the specified number of
445   ↳ missing elements.</para>
446 /// <para>Если список пуст, или количество пропускаемых элементов больше списка -
447   ↳ возвращает пустой массив, иначе - массив с указанным количеством пропущенных
448   ↳ элементов.</para>
449 /// </returns>
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public static T[] SkipFirst<T>(this IList<T> list, int skip)
452 {
453     if (list.IsNullOrEmpty() || list.Count <= skip)
454     {
455         return Array.Empty<T>();
456     }
457     var result = new T[list.Count - skip];
458     for (int r = skip, w = 0; r < list.Count; r++, w++)
459     {
460         result[w] = list[r];
461     }
462     return result;
463 }
464
465 /// <summary>
466 /// <para>Shifts all the elements of the list by one position to the right.</para>
467 /// <para>Сдвигает вправо все элементы списка на одну позицию.</para>
468 /// </summary>
469 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
470   ↳ списка.</para></typeparam>
471 /// <param name="list"><para>The list to copy from.</para><para>Список для
472   ↳ копирования.</para></param>
473 /// <returns>
474 /// <para>Array with a shift of elements by one position.</para>
475 /// <para>Массив со сдвигом элементов на одну позицию.</para>
476 /// </returns>
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
479
480 /// <summary>
481 /// <para>Shifts all elements of the list to the right by the specified number of
482   ↳ elements.</para>
483 /// <para>Сдвигает вправо все элементы списка на указанное количество элементов.</para>
484 /// </summary>
485 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
486   ↳ списка.</para></typeparam>
487 /// <param name="list"><para>The list to copy from.</para><para>Список для
488   ↳ копирования.</para></param>
489 /// <param name="shift"><para>The number of items to shift.</para><para>Количество
490   ↳ сдвигаемых элементов.</para></param>
491 /// <returns>
492 /// <para>If the value of the shift variable is less than zero - an <see
493   ↳ cref="NotImplementedException"/> exception is thrown, but if the value of the shift
494   ↳ variable is 0 - an exact copy of the array is returned. Otherwise, an array is
495   ↳ returned with the shift of the elements.</para>
496 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
497   ↳ cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
498   ↳ возвращается точная копия массива. Иначе возвращается массив со сдвигом
499   ↳ элементов.</para>
500 /// </returns>
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
503 {
504     if (shift < 0)
505     {
506         throw new NotImplementedException();
507     }
508 }

```



```

486     }
487     if (shift == 0)
488     {
489         return list.ToArray();
490     }
491     else
492     {
493         var result = new T[list.Count + shift];
494         for (int r = 0, w = shift; r < list.Count; r++, w++)
495         {
496             result[w] = list[r];
497         }
498         return result;
499     }
500 }
501 }
502 }

```

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
34         ↪ заполняться.</para></param>
35         /// <param name="returnConstant"><para>The value for the constant returned by
36         ↪ corresponding methods.</para><para>Значение для константы возвращаемой
37         ↪ соответствующими методами.</para></param>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
40         {
41             _list = list;
42             _returnConstant = returnConstant;
43         }
44
45         /// <summary>
46         /// <para>
47         /// Initializes a new <see cref="ListFiller"/> instance.
48         /// </para>
49         /// <para></para>
50         /// </summary>
51         /// <param name="list">
52         /// <para>A list.</para>
53         /// <para></para>
54         /// </param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public ListFiller(List<TElement> list) : this(list, default) { }
57
58         /// <summary>
59         /// <para>Adds an item to the end of the list.</para>
60         /// <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(ICollection<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(ICollection<TElement> elements) =>
    → _list.AddAllAndReturnTrue(elements);
98
99     /// <summary>
100    /// <para>Adds values to the list skipping the first element.</para>
101    /// <para>Добавляет значения в список пропуская первый элемент.</para>
102    /// </summary>
103    /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
104    /// <returns>
105    /// <para>True value in any case.</para>
106    /// <para>Значение true в любом случае.</para>
107    /// </returns>
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
    → _list.AddSkipFirstAndReturnTrue(elements);
110
111    /// <summary>
112    /// <para>Adds an item to the end of the list and return constant.</para>
113    /// <para>Добавляет элемент в конец списка и возвращает константу.</para>
114    /// </summary>
115    /// <param name="element"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
116    /// <returns>
117    /// <para>Constant value in any case.</para>
118    /// <para>Значение константы в любом случае.</para>
119    /// </returns>
120    [MethodImpl(MethodImplOptions.AggressiveInlining)]
121    public TReturnConstant AddAndReturnConstant(TElement element)
122    {
123        _list.Add(element);
124        return _returnConstant;
    }

```

```

125     }
126
127     /// <summary>
128     /// <para>Adds a value to the list at the first index and return constant.</para>
129     /// <para>Добавляет значение в список по первому индексу и возвращает константу.</para>
130     /// </summary>
131     /// <param name="element"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
132     /// <returns>
133     /// <para>Constant value in any case.</para>
134     /// <para>Значение константы в любом случае.</para>
135     /// </returns>
136     [MethodImpl(MethodImplOptions.AggressiveInlining)]
137     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
138     {
139         _list.AddFirst(elements);
140         return _returnConstant;
141     }
142
143     /// <summary>
144     /// <para>Adds all elements from other list to this list and returns constant.</para>
145     /// <para>Добавляет все элементы из другого списка в этот список и возвращает
    → константу.</para>
146     /// </summary>
147     /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
148     /// <returns>
149     /// <para>Constant value in any case.</para>
150     /// <para>Значение константы в любом случае.</para>
151     /// </returns>
152     [MethodImpl(MethodImplOptions.AggressiveInlining)]
153     public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
154     {
155         _list.AddAll(elements);
156         return _returnConstant;
157     }
158
159     /// <summary>
160     /// <para>Adds values to the list skipping the first element and return constant
    → value.</para>
161     /// <para>Добавляет значения в список пропуская первый элемент и возвращает значение
    → константы.</para>
162     /// </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(ICollection<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(ICollection<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 ICollection<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(ICollection<char> left, ICollection<char> right)
83         {
84             // Base can be not an array, but still ICollection<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 относительного исходного списка (индекс с которого
33         ↪ начинается этот сегмент).</para>
34         /// </summary>
35         public int Offset
36         {

```

```

34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     get;
36 }
37 /// <summary>
38 /// <para>Gets the length of a segment.</para>
39 /// <para>Возвращает длину сегмента.</para>
40 /// </summary>
41 public int Length
42 {
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     get;
45 }
46
47 /// <summary>
48 /// <para>Initializes a new instance of the <see cref="Segment"/> class, using the
49   → <paramref name="base"/> list, <paramref name="offset"/> of the segment and its
50   → <paramref name="length" />.</para>
51 /// <para>Инициализирует новый экземпляр класса <see cref="Segment"/>, используя список
52   → <paramref name="base"/>, <paramref name="offset"/> сегмента и его <paramref
53   → name="length"/>.</para>
54 /// </summary>
55 /// <param name="base"><para>The reference to the original list containing the elements
56   → of this segment.</para><para>Ссылка на исходный список в котором находятся элементы
57   → этого сегмента.</para></param>
58 /// <param name="offset"><para>The offset relative to the <paramref name="base"/> list
59   → from which the segment starts.</para><para>Смещение относительно списка <paramref
60   → name="base"/>, с которого начинается сегмент.</para></param>
61 /// <param name="length"><para>The segment's length.</para><para>Длина
62   → сегмента.</para></param>
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public Segment(IList<T> @base, int offset, int length)
65 {
66     Base = @base;
67     Offset = offset;
68     Length = length;
69 }
70
71 /// <summary>
72 /// <para>Gets the hash code of the current <see cref="Segment"/> instance.</para>
73 /// <para>Возвращает хэш-код текущего экземпляра <see cref="Segment"/>.</para>
74 /// </summary>
75 /// <returns></returns>
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public override int GetHashCode() => this.GenerateHashCode();
78
79 /// <summary>
80 /// <para>Returns a value indicating whether the current <see cref="Segment"/> is equal
81   → to another <see cref="Segment" />.</para>
82 /// <para>Возвращает значение определяющее, равен ли текущий <see cref="Segment"/>
83   → другому <see cref="Segment"/>.</para>
84 /// </summary>
85 /// <param name="other"><para>An <see cref="Segment"/> object to compare with the
86   → current <see cref="Segment"/>.</para><para>Объект <see cref="Segment"/> для
87   → сравнения с текущим <see cref="Segment"/>.</para></param>
88 /// <returns>
89 /// <para><see langword="true"/> if the current <see cref="Segment"/> is equal to the
90   → <paramref name="other"/> parameter; otherwise, <see langword="false"/>.</para>
91 /// <para><see langword="true"/>, если текущий <see cref="Segment"/> равен параметру
92   → <paramref name="other"/>, в противном случае - <see langword="false"/>.</para>
93 /// </returns>
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 public virtual bool Equals(Segment<T> other) => this.EqualTo(other);

```

```

96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
    ↳ false;
98
99 #region IList
100
101 /// <summary>
102 /// <para>
103 /// The value.
104 /// </para>
105 /// <para></para>
106 /// </summary>
107 public T this[int i]
108 {
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     get => Base[Offset + i];
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     set => Base[Offset + i] = value;
113 }
114
115 /// <summary>
116 /// <para>Gets the number of elements contained in the <see cref="Segment"/>.</para>
117 /// <para>Возвращает число элементов, содержащихся в <see cref="Segment"/>.</para>
118 /// </summary>
119 /// <value>
120 /// <para>The number of elements contained in the <see cref="Segment"/>.</para>
121 /// <para>Число элементов, содержащихся в <see cref="Segment"/>.</para>
122 /// </value>
123 public int Count
124 {
125     [MethodImpl(MethodImplOptions.AggressiveInlining)]
126     get => Length;
127 }
128
129 /// <summary>
130 /// <para>Gets a value indicating whether the <see cref="Segment"/> is read-only.</para>
131 /// <para>Возвращает значение, указывающее, является ли <see cref="Segment"/> доступным
    ↳ только для чтения.</para>
132 /// </summary>
133 /// <value>
134 /// <para><see langword="true"/> if the <see cref="Segment"/> is read-only; otherwise,
    ↳ <see langword="false"/>.</para>
135 /// <para>Значение <see langword="true"/>, если <see cref="Segment"/> доступен только
    ↳ для чтения, в противном случае - значение <see langword="false"/>.</para>
136 /// </value>
137 /// <remarks>
138 /// <para>Any <see cref="Segment"/> is read-only.</para>
139 /// <para>Любой <see cref="Segment"/> доступен только для чтения.</para>
140 /// </remarks>
141 public bool IsReadOnly
142 {
143     [MethodImpl(MethodImplOptions.AggressiveInlining)]
144     get => true;
145 }
146
147 /// <summary>
148 /// <para>Determines the index of a specific item in the <see cref="Segment"/>.</para>
149 /// <para>Определяет индекс конкретного элемента в <see cref="Segment"/>.</para>
150 /// </summary>
151 /// <param name="item"><para>The object to locate in the <see
    ↳ cref="Segment"/>.</para><para>Элемент для поиска в <see
    ↳ cref="Segment"/>.</para></param>
152 /// <returns>
153 /// <para>The index of <paramref name="item"/> if found in the segment; otherwise,
    ↳ -1.</para>
154 /// <para>Индекс <paramref name="item"/>, если он найден в сегменте; в противном случае
    ↳ - значение -1.</para>
155 /// </returns>
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 public int IndexOf(T item)
158 {
159     var index = Base.IndexOf(item);
160     if (index >= Offset)
161     {
162         var actualIndex = index - Offset;
163         if (actualIndex < Length)
164         {
165             return actualIndex;

```

```

166     }
167 }
168 return -1;
169 }
170
171 /// <summary>
172 /// <para>Inserts an item to the <see cref="Segment"/> at the specified index.</para>
173 /// <para>Вставляет элемент в <see cref="Segment"/> по указанному индексу.</para>
174 /// </summary>
175 /// <param name="index"><para>The zero-based index at which <paramref name="item"/>
  → should be inserted.</para><para>Отсчитываемый от нуля индекс, по которому следует
  → вставить элемент <paramref name="item"/>.</para></param>
176 /// <param name="item"><para>The element to insert into the <see
  → cref="Segment"/>.</para><para>Элемент, вставляемый в <see
  → cref="Segment"/>.</para></param>
177 /// <exception cref="NotSupportedException">
178 /// <para>The <see cref="Segment"/> is read-only.</para>
179 /// <para><see cref="Segment"/> доступен только для чтения.</para>
180 /// </exception>
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public void Insert(int index, T item) => throw new NotSupportedException();
183
184 /// <summary>
185 /// <para>Removes the <see cref="Segment"/> item at the specified index.</para>
186 /// <para>Удаляет элемент <see cref="Segment"/> по указанному индексу.</para>
187 /// </summary>
188 /// <param name="index"><para>The zero-based index of the item to
  → remove.</para><para>Отсчитываемый от нуля индекс элемента для
  → удаления.</para></param>
189 /// <exception cref="NotSupportedException">
190 /// <para>The <see cref="Segment"/> is read-only.</para>
191 /// <para><see cref="Segment"/> доступен только для чтения.</para>
192 /// </exception>
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public void RemoveAt(int index) => throw new NotSupportedException();
195
196 /// <summary>
197 /// <para>Adds an item to the <see cref="Segment"/>.</para>
198 /// <para>Добавляет элемент в <see cref="Segment"/>.</para>
199 /// </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         /// <summary>
19         /// <para>
20         /// The minimum string segment length.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         private readonly int _minimumStringSegmentLength;
25
26         /// <summary>
27         /// <para>
28         /// Initializes a new <see cref="AllSegmentsWalkerBase"/> instance.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         /// <param name="minimumStringSegmentLength">
33         /// <para>A minimum string segment length.</para>
34         /// <para></para>
35         /// </param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
38             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
39
40         /// <summary>
41         /// <para>
42         /// Initializes a new <see cref="AllSegmentsWalkerBase"/> instance.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
48
49         /// <summary>
50         /// <para>
51         /// Walks the all using the specified elements.
52         /// </para>
53         /// <para></para>
54         /// </summary>
55         /// <param name="elements">
```

```

55     /// <para>The elements.</para>
56     /// <para></para>
57     /// </param>
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public virtual void WalkAll(ICollection<T> elements)
60     {
61         for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
62             ↪ offset <= maxOffset; offset++)
63         {
64             for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
65                 ↪ offset; length <= maxLength; length++)
66             {
67                 Iteration(CreateSegment(elements, offset, length));
68             }
69         }
70     }
71     /// <summary>
72     /// <para>
73     /// Creates the segment using the specified elements.
74     /// </para>
75     /// </summary>
76     /// <param name="elements">
77     /// <para>The elements.</para>
78     /// </param>
79     /// <param name="offset">
80     /// <para>The offset.</para>
81     /// </param>
82     /// <param name="length">
83     /// <para>The length.</para>
84     /// </param>
85     /// <returns>
86     /// <para>The segment</para>
87     /// </returns>
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);
90     /// <summary>
91     /// <para>
92     /// Iterations the segment.
93     /// </para>
94     /// </summary>
95     /// <param name="segment">
96     /// <para>The segment.</para>
97     /// </param>
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     protected abstract void Iteration(TSegment segment);
100 }
101 }

```

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     /// </summary>
13     /// <seealso cref="AllSegmentsWalkerBase{T, Segment{T}}"/>
14     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
15     {
16         /// <summary>
17         /// <para>
18         /// Creates the segment using the specified elements.
19         /// </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     ↪ => new Segment<T>(elements, offset, length);
42 }

```

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        /// <para></para>
38        /// </summary>
39        /// <typeparam name="TSegment">
40        /// <para>The segment.</para>
41        /// <para></para>
42        /// </typeparam>
43        /// <param name="walker">
44        /// <para>The walker.</para>
45        /// <para></para>
46        /// </param>
47        /// <param name="@string">
48        /// <para>The string.</para>
49        /// <para></para>
50        /// </param>
51        [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

51         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
        ↪     string @string) where TSegment : Segment<char> =>
        ↪     walker.WalkAll(@string.ToCharArray());
52     }
53 }

```

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> :
        ↪     DuplicateSegmentsWalkerBase<T, TSegment>
        ↪     where TSegment : Segment<T>
17     {
18         /// <summary>
19         /// <para>
20         /// The default reset dictionary on each walk.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         public static readonly bool DefaultResetDictionaryOnEachWalk;
25
26         /// <summary>
27         /// <para>
28         /// The reset dictionary on each walk.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         private readonly bool _resetDictionaryOnEachWalk;
33         /// <summary>
34         /// <para>
35         /// The dictionary.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         protected IDictionary<TSegment, long> Dictionary;
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         /// <param name="resetDictionaryOnEachWalk">
56         /// <para>A reset dictionary on each walk.</para>
57         /// <para></para>
58         /// </param>
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪     dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
        ↪     : base(minimumStringSegmentLength)
61         {
62             Dictionary = dictionary;
63             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
64         }
65
66         /// <summary>
67         /// <para>
68         /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.

```

```

71     /// </para>
72     /// <para></para>
73     /// </summary>
74     /// <param name="dictionary">
75     /// <para>A dictionary.</para>
76     /// <para></para>
77     /// </param>
78     /// <param name="minimumStringSegmentLength">
79     /// <para>A minimum string segment length.</para>
80     /// <para></para>
81     /// </param>
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
        ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

84
85     /// <summary>
86     /// <para>
87     /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
88     /// </para>
89     /// <para></para>
90     /// </summary>
91     /// <param name="dictionary">
92     /// <para>A dictionary.</para>
93     /// <para></para>
94     /// </param>
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
        ↪ DefaultResetDictionaryOnEachWalk) { }

97
98     /// <summary>
99     /// <para>
100    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
101    /// </para>
102    /// <para></para>
103    /// </summary>
104    /// <param name="minimumStringSegmentLength">
105    /// <para>A minimum string segment length.</para>
106    /// <para></para>
107    /// </param>
108    /// <param name="resetDictionaryOnEachWalk">
109    /// <para>A reset dictionary on each walk.</para>
110    /// <para></para>
111    /// </param>
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
        ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
        ↪ { }

114
115    /// <summary>
116    /// <para>
117    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
118    /// </para>
119    /// <para></para>
120    /// </summary>
121    /// <param name="minimumStringSegmentLength">
122    /// <para>A minimum string segment length.</para>
123    /// <para></para>
124    /// </param>
125    [MethodImpl(MethodImplOptions.AggressiveInlining)]
126    protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

127
128    /// <summary>
129    /// <para>
130    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
131    /// </para>
132    /// <para></para>
133    /// </summary>
134    [MethodImpl(MethodImplOptions.AggressiveInlining)]
135    protected DictionaryBasedDuplicateSegmentsWalkerBase() :
        ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

136
137    /// <summary>
138    /// <para>

```

```

139     /// Walks the all using the specified elements.
140     /// </para>
141     /// <para></para>
142     /// </summary>
143     /// <param name="elements">
144     /// <para>The elements.</para>
145     /// <para></para>
146     /// </param>
147     [MethodImpl(MethodImplOptions.AggressiveInlining)]
148     public override void WalkAll(ICollection<T> elements)
149     {
150         if (_resetDictionaryOnEachWalk)
151         {
152             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
153             Dictionary = new Dictionary<TSegment, long>((int)capacity);
154         }
155         base.WalkAll(elements);
156     }
157
158     /// <summary>
159     /// <para>
160     /// Gets the segment frequency using the specified segment.
161     /// </para>
162     /// <para></para>
163     /// </summary>
164     /// <param name="segment">
165     /// <para>The segment.</para>
166     /// <para></para>
167     /// </param>
168     /// <returns>
169     /// <para>The long</para>
170     /// <para></para>
171     /// </returns>
172     [MethodImpl(MethodImplOptions.AggressiveInlining)]
173     protected override long GetSegmentFrequency(TSegment segment) =>
174         Dictionary.TryGetValue(segment, out long frequency);
175
176     /// <summary>
177     /// <para>
178     /// Sets the segment frequency using the specified segment.
179     /// </para>
180     /// <para></para>
181     /// </summary>
182     /// <param name="segment">
183     /// <para>The segment.</para>
184     /// <para></para>
185     /// </param>
186     /// <param name="frequency">
187     /// <para>The frequency.</para>
188     /// <para></para>
189     /// </param>
190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
191     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
192         Dictionary[segment] = frequency;
193 }

```

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.

```

```

20    /// </para>
21    /// <para></para>
22    /// </summary>
23    /// <param name="dictionary">
24    /// <para>A dictionary.</para>
25    /// <para></para>
26    /// </param>
27    /// <param name="minimumStringSegmentLength">
28    /// <para>A minimum string segment length.</para>
29    /// <para></para>
30    /// </param>
31    /// <param name="resetDictionaryOnEachWalk">
32    /// <para>A reset dictionary on each walk.</para>
33    /// <para></para>
34    /// </param>
35    [MethodImpl(MethodImplOptions.AggressiveInlining)]
36    protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
    ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
37
38    /// <summary>
39    /// <para>
40    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
41    /// </para>
42    /// <para></para>
43    /// </summary>
44    /// <param name="dictionary">
45    /// <para>A dictionary.</para>
46    /// <para></para>
47    /// </param>
48    /// <param name="minimumStringSegmentLength">
49    /// <para>A minimum string segment length.</para>
50    /// <para></para>
51    /// </param>
52    [MethodImpl(MethodImplOptions.AggressiveInlining)]
53    protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
    ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
54
55    /// <summary>
56    /// <para>
57    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
58    /// </para>
59    /// <para></para>
60    /// </summary>
61    /// <param name="dictionary">
62    /// <para>A dictionary.</para>
63    /// <para></para>
64    /// </param>
65    [MethodImpl(MethodImplOptions.AggressiveInlining)]
66    protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
67
68    /// <summary>
69    /// <para>
70    /// 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,
    ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪ resetDictionaryOnEachWalk) { }
84
85    /// <summary>
86    /// <para>
87    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
88    /// </para>

```



```

89     /// <para></para>
90     /// </summary>
91     /// <param name="minimumStringSegmentLength">
92     /// <para>A minimum string segment length.</para>
93     /// <para></para>
94     /// </param>
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
97         ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
98
99     /// <summary>
100    /// <para>
101    /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
102    /// </para>
103    /// <para></para>
104    /// </summary>
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected DictionaryBasedDuplicateSegmentsWalkerBase() :
107        ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
108    }
109 }

```

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>
36         /// <para></para>
37         /// </summary>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
40
41         /// <summary>
42         /// <para>
43         /// Iterations the segment.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="segment">
48         /// <para>The segment.</para>
49         /// <para></para>
50         /// </param>
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected override void Iteration(TSegment segment)
53         {
54             var frequency = GetSegmentFrequency(segment);
55             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,
13         ↪ Segment<T>>
14     {
15     }
16 }

```

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) =>
57            ↪ set.Remove(element);
58
59        /// <summary>
60        /// <para>
61        /// Determines whether add and return true.
62        /// </para>
63        /// <para></para>
64        /// </summary>
65        /// <typeparam name="T">
66        /// <para>The .</para>
67        /// <para></para>
68        /// </typeparam>
69        /// <param name="set">
70        /// <para>The set.</para>
71        /// <para></para>
72        /// </param>
73        /// <param name="element">
74        /// <para>The element.</para>
75        /// <para></para>
76        /// </param>
77        /// <returns>
78        /// <para>The bool</para>
79        /// <para></para>
80        /// </returns>
81        [MethodImpl(MethodImplOptions.AggressiveInlining)]
82        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>
140    /// Determines whether add all and return true.
141    /// </para>
142    /// <para></para>
143    /// </summary>
144    /// <typeparam name="T">
145    /// <para>The .</para>
146    /// <para></para>
147    /// </typeparam>
148    /// <param name="set">
149    /// <para>The set.</para>
150    /// <para></para>
151    /// </param>
152    /// <param name="elements">
153    /// <para>The elements.</para>
154    /// <para></para>
155    /// </param>
156    /// <returns>
157    /// <para>The bool</para>
158    /// <para></para>
159    /// </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></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></para>
83      /// </summary>
84      /// <param name="element">
85      /// <para>The element.</para>
86      /// <para></para>
87      /// </param>
88      /// <returns>
89      /// <para>The bool</para>

```

```

90     /// <para></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></para>
100    /// </summary>
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 bool AddFirstAndReturnTrue(IList<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></para>
118    /// </summary>
119    /// <param name="elements">
120    /// <para>The elements.</para>
121    /// <para></para>
122    /// </param>
123    /// <returns>
124    /// <para>The bool</para>
125    /// <para></para>
126    /// </returns>
127    [MethodImpl(MethodImplOptions.AggressiveInlining)]
128    public bool AddAllAndReturnTrue(IList<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></para>
136    /// </summary>
137    /// <param name="elements">
138    /// <para>The elements.</para>
139    /// <para></para>
140    /// </param>
141    /// <returns>
142    /// <para>The bool</para>
143    /// <para></para>
144    /// </returns>
145    [MethodImpl(MethodImplOptions.AggressiveInlining)]
146    public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
147        ↪ _set.AddSkipFirstAndReturnTrue(elements);
148
149    /// <summary>
150    /// <para>
151    /// Adds the and return constant using the specified element.
152    /// </para>
153    /// <para></para>
154    /// </summary>
155    /// <param name="element">
156    /// <para>The element.</para>
157    /// <para></para>
158    /// </param>
159    /// <returns>
160    /// <para>The return constant.</para>
161    /// <para></para>
162    /// </returns>
163    [MethodImpl(MethodImplOptions.AggressiveInlining)]
164    public TReturnConstant AddAndReturnConstant(TElement element)
165    {
166        ↪ _set.Add(element);
167        ↪ return _returnConstant;
168    }

```



```

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 :
58             ↪ stack.Pop();
59
60         /// <summary>
61         /// <para>
62         /// Peeks the or default using the specified stack.
63         /// </para>
64         /// <para></para>
65         /// </summary>
66         /// <typeparam name="T">
67         /// <para>The .</para>
68         /// <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                 {
118                     left++;
119                 }
120                 if (@string[right] == charToTrim)
121                 {
122                     right--;
123                 }
124                 return @string.Substring(left, right - left + 1);
125             }
126         }
127         else
128         {
129             return @string;
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        /// <summary>
18        /// <para>
19        /// The child nodes.
20        /// </para>
21        /// <para></para>
22        /// </summary>
23        private Dictionary<object, Node> _childNodes;
24
25        /// <summary>
26        /// <para>
27        /// Gets or sets the value value.
28        /// </para>
29        /// <para></para>
30        /// </summary>
31        public object Value
32        {
33            [MethodImpl(MethodImplOptions.AggressiveInlining)]
34            get;
35            [MethodImpl(MethodImplOptions.AggressiveInlining)]
36            set;
37        }
38
39        /// <summary>
40        /// <para>
41        /// Gets the child nodes value.
42        /// </para>
43        /// <para></para>
44        /// </summary>
45        public Dictionary<object, Node> ChildNodes
46        {
47            [MethodImpl(MethodImplOptions.AggressiveInlining)]
48            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
49        }
50
51        /// <summary>
52        /// <para>
53        /// The key.
54        /// </para>
55        /// <para></para>
56        /// </summary>
57        public Node this[object key]
58        {
59            [MethodImpl(MethodImplOptions.AggressiveInlining)]
60            get => GetChild(key) ?? AddChild(key);
61            [MethodImpl(MethodImplOptions.AggressiveInlining)]
62            set => SetChildValue(value, key);
63        }
64
65        /// <summary>
66        /// <para>
67        /// Initializes a new <see cref="Node"/> instance.
68        /// </para>
69        /// <para></para>
70        /// </summary>
71        /// <param name="value">
72        /// <para>A value.</para>
73        /// <para></para>
74        /// </param>
75        [MethodImpl(MethodImplOptions.AggressiveInlining)]
76        public Node(object value) => Value = value;
77
78        /// <summary>
79        /// <para>
80        /// Initializes a new <see cref="Node"/> instance.
81        /// </para>
82        /// <para></para>
83        /// </summary>

```

```

84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public Node() : this(null) { }
86
87 /// <summary>
88 /// <para>
89 /// Determines whether this instance contains child.
90 /// </para>
91 /// <para></para>
92 /// </summary>
93 /// <param name="keys">
94 /// <para>The keys.</para>
95 /// <para></para>
96 /// </param>
97 /// <returns>
98 /// <para>The bool</para>
99 /// <para></para>
100 /// </returns>
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
103
104 /// <summary>
105 /// <para>
106 /// Gets the child using the specified keys.
107 /// </para>
108 /// <para></para>
109 /// </summary>
110 /// <param name="keys">
111 /// <para>The keys.</para>
112 /// <para></para>
113 /// </param>
114 /// <returns>
115 /// <para>The node.</para>
116 /// <para></para>
117 /// </returns>
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public Node GetChild(params object[] keys)
120 {
121     var node = this;
122     for (var i = 0; i < keys.Length; i++)
123     {
124         node.ChildNodes.TryGetValue(keys[i], out node);
125         if (node == null)
126         {
127             return null;
128         }
129     }
130     return node;
131 }
132
133 /// <summary>
134 /// <para>
135 /// Gets the child value using the specified keys.
136 /// </para>
137 /// <para></para>
138 /// </summary>
139 /// <param name="keys">
140 /// <para>The keys.</para>
141 /// <para></para>
142 /// </param>
143 /// <returns>
144 /// <para>The object</para>
145 /// <para></para>
146 /// </returns>
147 [MethodImpl(MethodImplOptions.AggressiveInlining)]
148 public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
149
150 /// <summary>
151 /// <para>
152 /// Adds the child using the specified key.
153 /// </para>
154 /// <para></para>
155 /// </summary>
156 /// <param name="key">
157 /// <para>The key.</para>
158 /// <para></para>
159 /// </param>
160 /// <returns>
161 /// <para>The node</para>

```



```

162     /// <para></para>
163     /// </returns>
164     [MethodImpl(MethodImplOptions.AggressiveInlining)]
165     public Node AddChild(object key) => AddChild(key, new Node(null));
166
167     /// <summary>
168     /// <para>
169     /// Adds the child using the specified key.
170     /// </para>
171     /// <para></para>
172     /// </summary>
173     /// <param name="key">
174     /// <para>The key.</para>
175     /// <para></para>
176     /// </param>
177     /// <param name="value">
178     /// <para>The value.</para>
179     /// <para></para>
180     /// </param>
181     /// <returns>
182     /// <para>The node</para>
183     /// <para></para>
184     /// </returns>
185     [MethodImpl(MethodImplOptions.AggressiveInlining)]
186     public Node AddChild(object key, object value) => AddChild(key, new Node(value));
187
188     /// <summary>
189     /// <para>
190     /// Adds the child using the specified key.
191     /// </para>
192     /// <para></para>
193     /// </summary>
194     /// <param name="key">
195     /// <para>The key.</para>
196     /// <para></para>
197     /// </param>
198     /// <param name="child">
199     /// <para>The child.</para>
200     /// <para></para>
201     /// </param>
202     /// <returns>
203     /// <para>The child.</para>
204     /// <para></para>
205     /// </returns>
206     [MethodImpl(MethodImplOptions.AggressiveInlining)]
207     public Node AddChild(object key, Node child)
208     {
209         ChildNodes.Add(key, child);
210         return child;
211     }
212
213     /// <summary>
214     /// <para>
215     /// Sets the child using the specified keys.
216     /// </para>
217     /// <para></para>
218     /// </summary>
219     /// <param name="keys">
220     /// <para>The keys.</para>
221     /// <para></para>
222     /// </param>
223     /// <returns>
224     /// <para>The node</para>
225     /// <para></para>
226     /// </returns>
227     [MethodImpl(MethodImplOptions.AggressiveInlining)]
228     public Node SetChild(params object[] keys) => SetChildValue(null, keys);
229
230     /// <summary>
231     /// <para>
232     /// Sets the child using the specified key.
233     /// </para>
234     /// <para></para>
235     /// </summary>
236     /// <param name="key">
237     /// <para>The key.</para>
238     /// <para></para>
239     /// </param>

```

```

240     /// <returns>
241     /// <para>The node</para>
242     /// <para></para>
243     /// </returns>
244     [MethodImpl(MethodImplOptions.AggressiveInlining)]
245     public Node SetChild(object key) => SetChildValue(null, key);
246
247     /// <summary>
248     /// <para>
249     /// Sets the child value using the specified value.
250     /// </para>
251     /// <para></para>
252     /// </summary>
253     /// <param name="value">
254     /// <para>The value.</para>
255     /// <para></para>
256     /// </param>
257     /// <param name="keys">
258     /// <para>The keys.</para>
259     /// <para></para>
260     /// </param>
261     /// <returns>
262     /// <para>The node.</para>
263     /// <para></para>
264     /// </returns>
265     [MethodImpl(MethodImplOptions.AggressiveInlining)]
266     public Node SetChildValue(object value, params object[] keys)
267     {
268         var node = this;
269         for (var i = 0; i < keys.Length; i++)
270         {
271             node = SetChildValue(value, keys[i]);
272         }
273         node.Value = value;
274         return node;
275     }
276
277     /// <summary>
278     /// <para>
279     /// Sets the child value using the specified value.
280     /// </para>
281     /// <para></para>
282     /// </summary>
283     /// <param name="value">
284     /// <para>The value.</para>
285     /// <para></para>
286     /// </param>
287     /// <param name="key">
288     /// <para>The key.</para>
289     /// <para></para>
290     /// </param>
291     /// <returns>
292     /// <para>The child.</para>
293     /// <para></para>
294     /// </returns>
295     [MethodImpl(MethodImplOptions.AggressiveInlining)]
296     public Node SetChildValue(object value, object key)
297     {
298         if (!ChildNodes.TryGetValue(key, out Node child))
299         {
300             child = AddChild(key, value);
301         }
302         child.Value = value;
303         return child;
304     }
305 }
306 }

```

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
230     /// <summary>
231     /// <para>
232     /// Tests the to operations with same meaning using the specified test.
233     /// </para>
234     /// <para></para>
235     /// </summary>
236     /// <param name="test">
237     /// <para>The test.</para>
238     /// <para></para>
239     /// </param>
240     private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
241     ↪ BitString, BitString> test)
242     {
243         const int n = 5654;
244         var x = new BitString(n);
245         var y = new BitString(n);
246         while (x.Equals(y))
247         {
248             x.SetRandomBits();
249             y.SetRandomBits();
250         }
251         var w = new BitString(x);
252         var v = new BitString(y);
253         Assert.False(x.Equals(y));
254         Assert.False(w.Equals(v));
255         Assert.True(x.Equals(w));
256         Assert.True(y.Equals(v));
257         test(x, y, w, v);
258         Assert.True(x.Equals(w));
259     }
260 }

```

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         /// <summary>
22         /// <para>
23         /// The elfen lied.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         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.
28 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.
29 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!""
30 Pfui, stößt den Kopf an harten Stein! Elfe, gelt, du hast genug? Gukuk!";
31
32         /// <summary>
33         /// <para>
34         /// The example text.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         private static readonly string _exampleText =
                @"([english version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```


39 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?

40 [![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)

41 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?

42 [![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)

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

44 [![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)

45 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?

46 [![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)

47 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?

48 [![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)

49 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?

50 [![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)

51 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?

52 [![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)

53 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?

54 [![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)

55 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?

56 [![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)

57 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?

58 [![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)

```

59 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
   ↳ рекурсии или фрактала?
60 [![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)
61 Последовательность? Массив? Список? Множество? Объект? Таблица? Цвета? Символы? Буквы?
   ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
62 [![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)
63 ...
64 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
   ↳ ion-500.gif
   ↳ "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
   ↳ -animation-500.gif)";

65
66
67     /// <summary>
68     /// <para>
69     /// The exam rple text.
70     /// </para>
71     /// <para></para>
72     /// </summary>
73     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.";

74
75     /// <summary>
76     /// <para>
77     /// Tests that console tests.
78     /// </para>
79     /// <para></para>
80     /// </summary>
81     [Fact]
82     public void ConsoleTests()
83     {
84         string text = elfen_lied;
85
86         var iterationsCounter = new IterationsCounter();
87         iterationsCounter.WalkAll(text);
88         var result = iterationsCounter.IterationsCount;
89         Console.WriteLine($"TextLength: {text.Length}. Iterations: {result}.");
90
91         {
92             var start = new Stopwatch();
93             start.Start();
94
95             var walker = new Walker4();
96             walker.WalkAll(text);
97
98             //foreach (var (key, value) in walker.PublicDictionary)
99             //{
100                 // Console.WriteLine($"{key} {value}");
101             //}
102
103             start.Stop();
104             Console.WriteLine($"{{start.ElapsedMilliseconds}}ms");
105         }
106
107         {
108             var start = new Stopwatch();
109             start.Start();
110
111             var walker = new Walker2();
112             walker.WalkAll(text);
113
114             //foreach (var (key, value) in walker._cache)
115             //{
116                 // Console.WriteLine($"{key} {value}");
117             //}

```

```

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

```

```

193     /// <para></para>
194     /// </summary>
195     private Node _currentNode;
196
197     /// <summary>
198     /// <para>
199     /// Walks the all using the specified elements.
200     /// </para>
201     /// <para></para>
202     /// </summary>
203     /// <param name="elements">
204     /// <para>The elements.</para>
205     /// <para></para>
206     /// </param>
207     public override void WalkAll(IList<char> elements)
208     {
209         _rootNode = new Node();
210
211         base.WalkAll(elements);
212
213         Console.WriteLine(_rootNode.Value);
214     }
215
216     /// <summary>
217     /// <para>
218     /// Ons the duplicate found using the specified segment.
219     /// </para>
220     /// <para></para>
221     /// </summary>
222     /// <param name="segment">
223     /// <para>The segment.</para>
224     /// <para></para>
225     /// </param>
226     protected override void OnDuplicateFound(CharSegment segment)
227     {
228
229     }
230
231     /// <summary>
232     /// <para>
233     /// Gets the segment frequency using the specified segment.
234     /// </para>
235     /// <para></para>
236     /// </summary>
237     /// <param name="segment">
238     /// <para>The segment.</para>
239     /// <para></para>
240     /// </param>
241     /// <returns>
242     /// <para>The long</para>
243     /// <para></para>
244     /// </returns>
245     protected override long GetSegmentFrequency(CharSegment segment)
246     {
247         for (int i = 0; i < segment.Length; i++)
248         {
249             var element = segment[i];
250
251             _currentNode = _currentNode[element];
252         }
253
254         if (_currentNode.Value is int)
255         {
256             return (int)_currentNode.Value;
257         }
258         else
259         {
260             return 0;
261         }
262     }
263
264     /// <summary>
265     /// <para>
266     /// Sets the segment frequency using the specified segment.
267     /// </para>
268     /// <para></para>
269     /// </summary>
270     /// <param name="segment">
271     /// <para>The segment.</para>

```

```

272     /// <para></para>
273     /// </param>
274     /// <param name="frequency">
275     /// <para>The frequency.</para>
276     /// <para></para>
277     /// </param>
278     protected override void SetSegmentFrequency(CharSegment segment, long frequency) =>
279         ↪ _currentNode.Value = frequency;
280
281     /// <summary>
282     /// <para>
283     /// Iterations the segment.
284     /// </para>
285     /// <para></para>
286     /// </summary>
287     /// <param name="segment">
288     /// <para>The segment.</para>
289     /// <para></para>
290     /// </param>
291     protected override void Iteration(CharSegment segment)
292     {
293         _currentNode = _rootNode;
294
295         base.Iteration(segment);
296     }
297
298     // Too much memory, but fast
299     /// <summary>
300     /// <para>
301     /// Represents the walker.
302     /// </para>
303     /// <para></para>
304     /// </summary>
305     /// <seealso cref="ConsolePrintedDuplicateWalkerBase"/>
306     public class Walker2 : ConsolePrintedDuplicateWalkerBase
307     {
308         /// <summary>
309         /// <para>
310         /// The cache.
311         /// </para>
312         /// <para></para>
313         /// </summary>
314         public Dictionary<string, long> _cache;
315         /// <summary>
316         /// <para>
317         /// The current key.
318         /// </para>
319         /// <para></para>
320         /// </summary>
321         private string _currentKey;
322         /// <summary>
323         /// <para>
324         /// The total duplicates.
325         /// </para>
326         /// <para></para>
327         /// </summary>
328         private int _totalDuplicates;
329
330         /// <summary>
331         /// <para>
332         /// Walks the all using the specified elements.
333         /// </para>
334         /// <para></para>
335         /// </summary>
336         /// <param name="elements">
337         /// <para>The elements.</para>
338         /// <para></para>
339         /// </param>
340         public override void WalkAll(ICollection<char> elements)
341         {
342             _cache = new Dictionary<string, long>();
343
344             base.WalkAll(elements);
345
346             Console.WriteLine($"Unique string segments: {_cache.Count}. Total duplicates:
347                 ↪ {_totalDuplicates}");
348         }

```

```

348
349     /// <summary>
350     /// <para>
351     /// Ons the dublicate found using the specified segment.
352     /// </para>
353     /// <para></para>
354     /// </summary>
355     /// <param name="segment">
356     /// <para>The segment.</para>
357     /// <para></para>
358     /// </param>
359     protected override void OnDuplicateFound(CharSegment segment)
360     {
361         _totalDuplicates++;
362     }
363
364     /// <summary>
365     /// <para>
366     /// Gets the segment frequency using the specified segment.
367     /// </para>
368     /// <para></para>
369     /// </summary>
370     /// <param name="segment">
371     /// <para>The segment.</para>
372     /// <para></para>
373     /// </param>
374     /// <returns>
375     /// <para>The long</para>
376     /// <para></para>
377     /// </returns>
378     protected override long GetSegmentFrequency(CharSegment segment) =>
379     ↪ _cache.GetOrDefault(_currentKey);
380
381     /// <summary>
382     /// <para>
383     /// Sets the segment frequency using the specified segment.
384     /// </para>
385     /// <para></para>
386     /// </summary>
387     /// <param name="segment">
388     /// <para>The segment.</para>
389     /// <para></para>
390     /// </param>
391     /// <param name="frequency">
392     /// <para>The frequency.</para>
393     /// <para></para>
394     /// </param>
395     protected override void SetSegmentFrequency(CharSegment segment, long frequency) =>
396     ↪ _cache[_currentKey] = frequency;
397
398     /// <summary>
399     /// <para>
400     /// Iterations the segment.
401     /// </para>
402     /// <para></para>
403     /// </summary>
404     /// <param name="segment">
405     /// <para>The segment.</para>
406     /// <para></para>
407     /// </param>
408     protected override void Iteration(CharSegment segment)
409     {
410         _currentKey = segment;
411         base.Iteration(segment);
412     }
413
414     /// <summary>
415     /// <para>
416     /// Represents the walker.
417     /// </para>
418     /// <para></para>
419     /// </summary>
420     /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase<char, CharSegment>" />
421     public class Walker4 : DictionaryBasedDuplicateSegmentsWalkerBase<char, CharSegment>
422     {
423         /// <summary>

```

```

424 /// <para>
425 /// Gets the public dictionary value.
426 /// </para>
427 /// <para></para>
428 /// </summary>
429 public IDictionary<CharSegment, long> PublicDictionary
430 {
431     get
432     {
433         return Dictionary;
434     }
435 }
436
437 /// <summary>
438 /// <para>
439 /// Initializes a new <see cref="Walker4"/> instance.
440 /// </para>
441 /// <para></para>
442 /// </summary>
443 public Walker4()
444     : base(DefaultMinimumStringSegmentLength, resetDictionaryOnEachWalk: true)
445 {
446 }
447
448 /// <summary>
449 /// <para>
450 /// The total duplicates.
451 /// </para>
452 /// <para></para>
453 /// </summary>
454 private int _totalDuplicates;
455
456 /// <summary>
457 /// <para>
458 /// Walks the all using the specified elements.
459 /// </para>
460 /// <para></para>
461 /// </summary>
462 /// <param name="elements">
463 /// <para>The elements.</para>
464 /// <para></para>
465 /// </param>
466 public override void WalkAll(ICollection<char> elements)
467 {
468     _totalDuplicates = 0;
469
470     base.WalkAll(elements);
471     Console.WriteLine($"Unique string segments: {Dictionary.Count}. Total duplicates:
472         ↪ {_totalDuplicates}.");
473 }
474
475 /// <summary>
476 /// <para>
477 /// Creates the segment using the specified elements.
478 /// </para>
479 /// <para></para>
480 /// </summary>
481 /// <param name="elements">
482 /// <para>The elements.</para>
483 /// <para></para>
484 /// </param>
485 /// <param name="offset">
486 /// <para>The offset.</para>
487 /// <para></para>
488 /// </param>
489 /// <param name="length">
490 /// <para>The length.</para>
491 /// <para></para>
492 /// </param>
493 /// <returns>
494 /// <para>The char segment</para>
495 /// <para></para>
496 /// </returns>
497 protected override CharSegment CreateSegment(ICollection<char> elements, int offset, int
498     ↪ length) => new CharSegment(elements, offset, length);
499
500 /// <summary>
501 /// <para>

```

```

500     /// Ons the duplicate found using the specified segment.
501     /// </para>
502     /// <para></para>
503     /// </summary>
504     /// <param name="segment">
505     /// <para>The segment.</para>
506     /// <para></para>
507     /// </param>
508     protected override void OnDuplicateFound(CharSegment segment)
509     {
510         _totalDuplicates++;
511     }
512 }
513
514     /// <summary>
515     /// <para>
516     /// Represents the iterations counter.
517     /// </para>
518     /// <para></para>
519     /// </summary>
520     /// <seealso cref="AllSegmentsWalkerBase{char}"/>
521     public class IterationsCounter : AllSegmentsWalkerBase<char>
522     {
523         /// <summary>
524         /// <para>
525         /// The iterations count.
526         /// </para>
527         /// <para></para>
528         /// </summary>
529         public long IterationsCount;
530
531         /// <summary>
532         /// <para>
533         /// Iterations the segment.
534         /// </para>
535         /// <para></para>
536         /// </summary>
537         /// <param name="segment">
538         /// <para>The segment.</para>
539         /// <para></para>
540         /// </param>
541         protected override void Iteration(Segment<char> segment) => IterationsCount++;
542     }
543 }

```


Index

- ./csharp/Platform.Collections.Tests/ArrayTests.cs, 98
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 99
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 102
- ./csharp/Platform.Collections.Tests/ListTests.cs, 103
- ./csharp/Platform.Collections.Tests/StringTests.cs, 103
- ./csharp/Platform.Collections.Tests/WalkersTests.cs, 104
- ./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, 8
- ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 10
- ./csharp/Platform.Collections/BitString.cs, 16
- ./csharp/Platform.Collections/BitStringExtensions.cs, 45
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 46
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 46
- ./csharp/Platform.Collections/EnsureExtensions.cs, 47
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 52
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 53
- ./csharp/Platform.Collections/Lists/CharIListExtensions.cs, 54
- ./csharp/Platform.Collections/Lists/IListComparer.cs, 55
- ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 56
- ./csharp/Platform.Collections/Lists/IListExtensions.cs, 57
- ./csharp/Platform.Collections/Lists/ListFiller.cs, 65
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 67
- ./csharp/Platform.Collections/Segments/Segment.cs, 69
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 73
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 74
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 75
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 76
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 77
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 79
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 81
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 82
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 82
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 86
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 89
- ./csharp/Platform.Collections/Stacks/IStack.cs, 90
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 91
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 92
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 92
- ./csharp/Platform.Collections/StringExtensions.cs, 93
- ./csharp/Platform.Collections/Trees/Node.cs, 94