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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Arrays
{
    /// <summary>
    /// <para>Represents <see cref="T:TElement[]"/> array filler with additional methods that return a given constant of type <typeparamref cref="TReturnConstant"/>.</para>
    /// <para>Представляет заполнитель массива <see cref="T:TElement[]"/> с дополнительными методами, возвращающими заданную константу типа <typeparamref cref="TReturnConstant"/>.</para>
    /// </summary>
    /// <typeparam name="TElement"><para>The elements' type.</para><para>Тип элементов массива.</para></typeparam>
    /// <typeparam name="TReturnConstant"><para>The return constant's type.</para><para>Тип возвращаемой константы.</para></typeparam>
    public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
    {
        /// <summary>
        /// <para>
        /// The return constant.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TReturnConstant _returnConstant;

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the specified array, the offset from which filling will start and the constant returned when elements are being filled.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя указанный массив, смещение с которого начнётся заполнение и константу возвращаемую при заполнении элементов.</para>
        /// </summary>
        /// <param name="array"><para>The array to fill.</para><para>Массив для заполнения.</para></param>
        /// <param name="offset"><para>The offset from which to start the array filling.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
        /// <param name="returnConstant"><para>The constant's value.</para><para>Значение константы.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) : base(array, offset) => _returnConstant = returnConstant;

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the specified array and the constant returned when elements are being filled. Filling will start from the beginning of the array.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя указанный массив и константу возвращаемую при заполнении элементов. Заполнение начнётся с начала массива.</para>
        /// </summary>
        /// <param name="array"><para>The array to fill.</para><para>Массив для заполнения.</para></param>
        /// <param name="returnConstant"><para>The constant's value.</para><para>Значение константы.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0, returnConstant) { }

        /// <summary>
        /// <para>Adds an item into the array and returns the constant.</para>
        /// <para>Добавляет элемент в массив и возвращает константу.</para>
        /// </summary>
        /// <param name="element"><para>The element to add.</para><para>Добавляемый элемент.</para></param>
        /// <returns>
        /// <para>The constant's value.</para>
        /// <para>Значение константы.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element) =>
            _array.AddAndReturnConstant(ref _position, element, _returnConstant);

        /// <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(IList<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(IList<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(IList<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          /// <summary>
29          /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
            ↪  specified array as the array to fill and the offset from which to start
            ↪  filling.</para>
30          /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
            ↪  указанный массив в качестве заполняемого и смещение с которого начнётся
            ↪  заполнение.</para>
31          /// </summary>
32          /// <param name="array"><para>The array to fill.</para><para>Массив для
            ↪  заполнения.</para></param>
33          /// <param name="offset"><para>The offset from which to start filling the
            ↪  array.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          public ArrayFiller(TElement[] array, long offset)
36          {
37              _array = array;
38              _position = offset;
39          }
40
41          /// <summary>
42          /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
            ↪  specified array. Filling will start from the beginning of the array.</para>
43          /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
            ↪  указанный массив. Заполнение начнётся с начала массива.</para>
44          /// </summary>
45          /// <param name="array"><para>The array to fill.</para><para>Массив для
            ↪  заполнения.</para></param>
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          public ArrayFiller(TElement[] array) : this(array, 0) { }
48
49          /// <summary>
50          /// <para>Adds an item into the array.</para>
51          /// <para>Добавляет элемент в массив.</para>
52          /// </summary>
53          /// <param name="element"><para>The element to add.</para><para>Добавляемый
            ↪  элемент.</para></param>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public void Add(TElement element) => _array[_position++] = element;
56
57          /// <summary>
58          /// <para>Adds an item into the array and returns <see langword="true"/>.</para>
59          /// <para>Добавляет элемент в массив и возвращает <see langword="true"/>.</para>
60          /// </summary>
61          /// <param name="element"><para>The element to add.</para><para>Добавляемый
            ↪  элемент.</para></param>
62          /// <returns>
63          /// <para>The <see langword="true"/> value.</para>
64          /// <para>Значение <see langword="true"/>.</para>
65          /// </returns>
66          [MethodImpl(MethodImplOptions.AggressiveInlining)]
67          public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
            ↪  _position, element, true);
68
69          /// <summary>
70          /// <para>Adds the first element from the specified list to the array to fill and
            ↪  returns <see langword="true"/>.</para>
71          /// <para>Добавляет первый элемент из указанного списка в заполняемый массив и
            ↪  возвращает <see langword="true"/>.</para>
72          /// </summary>
73          /// <param name="elements"><para>The list from which the first item will be
            ↪  added.</para><para>Список из которого будет добавлен первый элемент.</para></param>
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 AddFirstAndReturnTrue(IList<TElement> elements) =>
            ↪  _array.AddFirstAndReturnConstant(ref _position, elements, true);
80
81          /// <summary>
82          /// <para>Adds all elements from the specified list to the array to fill and returns
            ↪  <see langword="true"/>.</para>
83          /// <para>Добавляет все элементы из указанного списка в заполняемый массив и возвращает
            ↪  <see langword="true"/>.</para>
84          /// </summary>
```

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

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

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

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

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Disposables;
5   using Platform.Collections.Stacks;
6
7   namespace Platform.Collections.Arrays
8   {
9       /// <summary>
10      /// <para>Represents a set of arrays ready for reuse.</para>
11      /// <para>Представляет собой набор массивов готовых к повторному использованию.</para>
12      /// </summary>
13      /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
14      /// <remarks>
15      /// Original idea from
            ↪  http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
16      /// </remarks>
17      public class ArrayPool<T>
18      {
19          // May be use Default class for that later.
20          /// <summary>
21          /// <para>
22          /// The thread instance.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          [ThreadStatic]
27          private static ArrayPool<T> _threadInstance;
28          /// <summary>
29          /// <para>
30          /// Gets the thread instance value.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
            ↪  ArrayPool<T>());
35
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><para>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
            ↪  количество массивов на каждый размер по умолчанию.</para>
62          /// </summary>
63          [MethodImpl(MethodImplOptions.AggressiveInlining)]
64          public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
65
66          /// <summary>
67          /// <para>Retrieves an array from the pool, which will automatically return to the pool
            ↪  when the container is disposed.</para>
68          /// <para>Извлекает из пула массив, который автоматически вернётся в пул при
            ↪  высвобождении контейнера.</para>
69          /// </summary>
70          /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
            ↪  массива.</para></param>
71          /// <returns>
72          /// <para>The disposable container containing either a new array or an array from the
            ↪  pool.</para>
73          /// <para>Высвобождаемый контейнер содержащий либо новый массив, либо массив из
            ↪  пула.</para>
74          /// </returns>
75          [MethodImpl(MethodImplOptions.AggressiveInlining)]
76          public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
77
78          /// <summary>
79          /// <para>Replaces the array with another array from the pool with the specified
            ↪  size.</para>
80          /// <para>Заменяет массив на другой массив из пула с указанным размером.</para>
81          /// </summary>
82          /// <param name="source"><para>The source array.</para><para>Исходный
            ↪  массив.</para></param>
83          /// <param name="size"><para>A new array size.</para><para>Новый размер
            ↪  массива.</para></param>
84          /// <returns>
85          /// <para>An array with a new size.</para>
86          /// <para>Массив с новым размером.</para>
87          /// </returns>
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          public Disposable<T[]> Resize(Disposable<T[]> source, long size)
90          {
91              var destination = AllocateDisposable(size);
92              T[] sourceArray = source;
93              if (!sourceArray.IsNullOrEmpty())
94              {
95                  T[] destinationArray = destination;
96                  Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
                    ↪  sourceArray.LongLength);
97                  source.Dispose();
98              }
99              return destination;
100         }
101
102         /// <summary>
103         /// <para>Clears the pool.</para>
104         /// <para>Очищает пул.</para>
105         /// </summary>
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         public virtual void Clear() => _pool.Clear();
108
109         /// <summary>
110         /// <para>Retrieves an array with the specified size from the pool.</para>
111         /// <para>Извлекает из пула массив с указанным размером.</para>
112         /// </summary>
113         /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
            ↪  массива.</para></param>
114         /// <returns>
115         /// <para>An array from the pool or a new array.</para>
116         /// <para>Массив из пула или новый массив.</para>
117         /// </returns>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
            ↪  _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
120
121         /// <summary>
122         /// <para>Frees the array to the pool for later reuse.</para>
123         /// <para>Освобождает массив в пул для последующего повторного использования.</para>
124         /// </summary>
```

```
        /// <param name="array"><para>The array to be freed into the pool.</para><para>Массив
        ↪  который нужно освободить в пул.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Free(T[] array)
        {
            if (array.IsNullOrEmpty())
            {
                return;
            }
            var stack = _pool.GetOrAdd(array.LongLength, size => new
            ↪  Stack<T[]>(_maxArraysPerSize));
            if (stack.Count == _maxArraysPerSize) // Stack is full
            {
                return;
            }
            stack.Push(array);
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Collections.Segments;

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

namespace Platform.Collections.Arrays
{
    /// <summary>
    /// <para>
    /// Represents the array string.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Segment{T}"/>
    public class ArrayString<T> : Segment<T>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArrayString"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="length">
        /// <para>A length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(int length) : base(new T[length], 0, length) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArrayString"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>A array.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(T[] array) : base(array, 0, array.Length) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArrayString"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>A array.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>A length.</para>
        /// <para></para>
        /// </param>
        [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
            ↪  length. The hash code is generated based on the values of the array elements
            ↪  included in the specified segment.</para>
15          /// <para>Генерирует хэш-код сегмента массива с указанным смещением и длиной. Хэш-код
            ↪  генерируется на основе значений элементов массива входящих в указанный
            ↪  сегмент.</para>
16          /// </summary>
17          /// <param name="array"><para>The array to hash.</para><para>Массив для
            ↪  хеширования.</para></param>
18          /// <param name="offset"><para>The offset from which reading of the specified number of
            ↪  elements in the array starts.</para><para>Смещение, с которого начинается чтение
            ↪  указанного количества элементов в массиве.</para></param>
19          /// <param name="length"><para>The number of array elements used to calculate the
            ↪  hash.</para><para>Количество элементов массива, на основе которых будет вычислен
            ↪  хэш.</para></param>
20          /// <returns>
21          /// <para>The hash code of the segment in the array.</para>
22          /// <para>Хэш-код сегмента в массиве.</para>
23          /// </returns>
24          /// <remarks>
25          /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
            ↪  a3eda37d3d4cd10/mscorlib/system/string.cs#L833
26          /// </remarks>
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public static int GenerateHashCode(this char[] array, int offset, int length)
29          {
30              var hashSeed = 5381;
31              var hashAccumulator = hashSeed;
32              fixed (char* arrayPointer = &array[offset])
33              {
34                  for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
                    ↪  < last; charPointer++)
35                  {
36                      hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
37                  }
38              }
39              return hashAccumulator + (hashSeed * 1566083941);
40          }
41
42          /// <summary>
43          /// <para>Checks if all elements of two lists are equal.</para>
44          /// <para>Проверяет равны ли все элементы двух списков.</para>
45          /// </summary>
46          /// <param name="left"><para>The first compared array.</para><para>Первый массив для
            ↪  сравнения.</para></param>
47          /// <param name="leftOffset"><para>The offset from which reading of the specified number
            ↪  of elements in the first array starts.</para><para>Смещение, с которого начинается
            ↪  чтение элементов в первом массиве.</para></param>
48          /// <param name="length"><para>The number of checked elements.</para><para>Количество
            ↪  проверяемых элементов.</para></param>
49          /// <param name="right"><para>The second compared array.</para><para>Второй массив для
            ↪  сравнения.</para></param>
50          /// <param name="rightOffset"><para>The offset from which reading of the specified
            ↪  number of elements in the second array starts.</para><para>Смещение, с которого
            ↪  начинается чтение элементов в втором массиве.</para></param>
51          /// <returns>
52          /// <para><see langword="true"/> if the segments of the passed arrays are equal to each
            ↪  other otherwise <see langword="false"/>.</para>
```

```
        /// <para><see langword="true"/>, если сегменты переданных массивов равны друг другу, иначе же <see langword="false"/>.</para>
        /// </returns>
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L364
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[] right, int rightOffset)
        {
            fixed (char* leftPointer = &left[leftOffset])
            {
                fixed (char* rightPointer = &right[rightOffset])
                {
                    char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
                    if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref rightPointerCopy, ref length))
                    {
                        return false;
                    }
                    CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy, ref length);
                    return length <= 0;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether check arrays main part for equality.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref int length)
        {
            while (length >= 10)
            {
                if ((*(int*)left != *(int*)right)
                  || (*(int*)(left + 2) != *(int*)(right + 2))
                  || (*(int*)(left + 4) != *(int*)(right + 4))
                  || (*(int*)(left + 6) != *(int*)(right + 6))
                  || (*(int*)(left + 8) != *(int*)(right + 8)))
                {
                    return false;
                }
                left += 10;
                right += 10;
                length -= 10;
            }
            return true;
        }

        /// <summary>
        /// <para>
        /// Checks the arrays remainder for equality using the specified left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <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
               ↪ variable type int, and if the array length is greater than the index - return
               ↪ array[index], otherwise - default value.</para>
15         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
               ↪ помощью переменной index, и если длина массива больше индекса - возвращает
               ↪ array[index], иначе - значение по умолчанию.</para>
16         /// </summary>
17         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
               ↪ массива.</para></typeparam>
18         /// <param name="array"><para>Array that will participate in
               ↪ verification.</para><para>Массив который будет учавствовать в
               ↪ проверке.</para></param>
19         /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
               ↪ сравнения.</para></param>
20         /// <returns><para>Array element or default value.</para><para>Элемент массива или же
               ↪ значение по умолчанию.</para></returns>
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
               ↪ array.Length > index ? array[index] : default;
23
24         /// <summary>
25         /// <para>Checks whether the array exists, if so, checks the array length using the
               ↪ index variable type long, and if the array length is greater than the index - return
               ↪ array[index], otherwise - default value.</para>
26         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
               ↪ помощью переменной index, и если длина массива больше индекса - возвращает
               ↪ array[index], иначе - значение по умолчанию.</para>
27         /// </summary>
28         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
               ↪ массива.</para></typeparam>
```

```
29          /// <param name="array"><para>Array that will participate in
            ↪  verification.</para><para>Массив который будет учавствовать 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
            ↪  varible 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 учавствовать 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>Передает
            ↪  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 varible 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 учавствовать 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>Передает
            ↪  аргумент по ссылке, в случае успеха он примет значение array[index] в противном
            ↪  случае значение по умолчанию.</para></param>
67          /// <returns><para><see langword="true"/> if successful otherwise <see
            ↪  langword="false"/>.</para><para><see langword="true"/> в случае успеха, в противном
            ↪  случае <see langword="false"/></para></returns>
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static bool TryGetElement<T>(this T[] array, long index, out T element)
70          {
71              if (array != null && array.LongLength > index)
```

```
72              {
73                  element = array[index];
74                  return true;
75              }
76              else
77              {
78                  element = default;
79                  return false;
80              }
81          }
82
83          /// <summary>
84          /// <para>Copying of elements from one array to another array.</para>
85          /// <para>Копирует элементы из одного массива в другой массив.</para>
86          /// </summary>
87          /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
88          /// <param name="array"><para>The array  to copy.</para><para>Массив который необходимо
            ↪  скопировать.</para></param>
89          /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public static T[] Clone<T>(this T[] array)
92          {
93              var copy = new T[array.LongLength];
94              Array.Copy(array, 0L, copy, 0L, array.LongLength);
95              return copy;
96          }
97
98          /// <summary>
99          /// <para>Shifts all the elements of the array by one position to the right.</para>
100         /// <para>Сдвигает вправо все элементы массива на одну позицию.</para>
101         /// </summary>
102         /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
103         /// <param name="array"><para>The array to copy from.</para><para>Массив для
            ↪  копирования.</para></param>
104         /// <returns>
105         /// <para>Array with a shift of elements by one position.</para>
106         /// <para>Массив со сдвигом элементов на одну позицию.</para>
107         /// </returns>
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
110
111         /// <summary>
112         /// <para>Shifts all elements of the array to the right by the specified number of
            ↪  elements.</para>
113         /// <para>Сдвигает вправо все элементы массива на указанное количество элементов.</para>
114         /// </summary>
115         /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
116         /// <param name="array"><para>The array to copy from.</para><para>Массив для
            ↪  копирования.</para></param>
117         /// <param name="shift"><para>The number of items to shift.</para><para>Количество
            ↪  сдвигаемых элементов.</para></param>
118         /// <returns>
119         /// <para>If the value of the shift variable is less than zero - an <see
            ↪  cref="NotImplementedException"/> exception is thrown, but if the value of the shift
            ↪  variable is 0 - an exact copy of the array is returned. Otherwise, an array is
            ↪  returned with the shift of the elements.</para>
120         /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
            ↪  cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
            ↪  возвращается точная копия массива. Иначе возвращается массив со сдвигом
            ↪  элементов.</para>
121         /// </returns>
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         public static IList<T> ShiftRight<T>(this T[] array, long shift)
124         {
125             if (shift < 0)
126             {
127                 throw new NotImplementedException();
128             }
129             if (shift == 0)
130             {
131                 return array.Clone<T>();
132             }
133             else
134             {
```

```
135                 var restrictions = new T[array.LongLength + shift];
136                 Array.Copy(array, 0L, restrictions, shift, array.LongLength);
137                 return restrictions;
138             }
139         }
140 
141         /// <summary>
142         /// <para>Adding in array the passed element at the specified position and increments
            ↪  position value by one.</para>
143         /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
            ↪  значение position на единицу.</para>
144         /// </summary>
145         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
146         /// <param name="array"><para>The array to add the element to.</para><para>Массив в
            ↪  который необходимо добавить элемент.</para></param>
147         /// <param name="position"><para>A reference to the position of type int where the
            ↪  element will be added.</para><para>Ссылка на позицию типа int, в которую будет
            ↪  добавлен элемент.</para></param>
148         /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
            ↪  который нужно добавить в массив.</para></param>
149         [MethodImpl(MethodImplOptions.AggressiveInlining)]
150         public static void Add<T>(this T[] array, ref int position, T element) =>
            ↪  array[position++] = element;
151 
152         /// <summary>
153         /// <para>Adding in array the passed element at the specified position and increments
            ↪  position value by one.</para>
154         /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
            ↪  значение position на единицу.</para>
155         /// </summary>
156         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
157         /// <param name="array"><para>The array to add the element to.</para><para>Массив в
            ↪  который необходимо добавить элемент.</para></param>
158         /// <param name="position"><para>A reference to the position of type long where the
            ↪  element will be added.</para><para>Ссылка на позицию типа long, в которую будет
            ↪  добавлен элемент.</para></param>
159         /// <param name="element"><para>The element to add to the array</para><para>Элемент
            ↪  который необходимо добавить в массив.</para></param>
160         [MethodImpl(MethodImplOptions.AggressiveInlining)]
161         public static void Add<T>(this T[] array, ref long position, T element) =>
            ↪  array[position++] = element;
162 
163         /// <summary>
164         /// <para>Adding in array the passed element, at the specified position, increments
            ↪  position value by one and returns the value of the passed constant.</para>
165         /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
            ↪  значение position на единицу и возвращает значение переданной константы.</para>
166         /// </summary>
167         /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
            ↪  массива.</para></typeparam>
168         /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
            ↪  возвращаемой константы.</para></typeparam>
169         /// <param name="array"><para>The array to add the element to.</para><para>Массив в
            ↪  который необходимо добавить элемент.</para></param>
170         /// <param name="position"><para>Reference to the position to which the element will be
            ↪  added.</para><para>Ссылка на позицию, в которую будет добавлен
            ↪  элемент.</para></param>
171         /// <param name="element"><para>The element to add to the array.</para><para>Элемент
            ↪  который необходимо добавить в массив.</para></param>
172         /// <param name="returnConstant"><para>The constant value that will be
            ↪  returned.</para><para>Значение константы, которое будет возвращено.</para></param>
173         /// <returns>
174         /// <para>The constant value passed as an argument.</para>
175         /// <para>Значение константы, переданное в качестве аргумента.</para>
176         /// </returns>
177         [MethodImpl(MethodImplOptions.AggressiveInlining)]
178         public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, TElement element, TReturnConstant
            ↪  returnConstant)
179         {
180             array.Add(ref position, element);
181             return returnConstant;
182         }
183 
```

```
184         /// <summary>
185         /// <para>Adds the first element from the passed collection to the array, at the
            ↪  specified position and increments position value by one.</para>
186         /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
            ↪  позицию и увеличивает значение position на единицу.</para>
187         /// </summary>
188         /// <typeparam name="T"><para>Array element type.</para><para>Тип элементов
            ↪  массива.</para></typeparam>
189         /// <param name="array"><para>The array to add the element to.</para><para>Массив в
            ↪  который необходимо добавить элемент.</para></param>
190         /// <param name="position"><para>Reference to the position to which the element will be
            ↪  added.</para><para>Ссылка на позицию, в которую будет добавлен
            ↪  элемент.</para></param>
191         /// <param name="elements"><para>List, the first element of which will be added to the
            ↪  array.</para><para>Список, первый элемент которого будет добавлен в
            ↪  массив.</para></param>
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
            ↪  array[position++] = elements[0];
194
195         /// <summary>
196         /// <para>Adds the first element from the passed collection to the array, at the
            ↪  specified position, increments position value by one and returns the value of the
            ↪  passed constant.</para>
197         /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
            ↪  позицию, увеличивает значение position на единицу и возвращает значение переданной
            ↪  константы.</para>
198         /// </summary>
199         /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
            ↪  массива.</para></typeparam>
200         /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
            ↪  возвращаемой константы.</para></typeparam>
201         /// <param name="array"><para>The array to add the element to.</para><para>Массив в
            ↪  который необходимо добавить элемент.</para></param>
202         /// <param name="position"><para>Reference to the position to which the element will be
            ↪  added.</para><para>Ссылка на позицию, в которую будет добавлен
            ↪  элемент.</para></param>
203         /// <param name="elements"><para>List, the first element of which will be added to the
            ↪  array.</para><para>Список, первый элемент которого будет добавлен в
            ↪  массив.</para></param>
204         /// <param name="returnConstant"><para>The constant value that will be
            ↪  returned.</para><para>Значение константы, которое будет возвращено.</para></param>
205         /// <returns>
206         /// <para>The constant value passed as an argument.</para>
207         /// <para>Значение константы, переданное в качестве аргумента.</para>
208         /// </returns>
209         [MethodImpl(MethodImplOptions.AggressiveInlining)]
210         public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
            ↪  returnConstant)
211         {
212             array.AddFirst(ref position, elements);
213             return returnConstant;
214         }
215
216         /// <summary>
217         /// <para>Adding in array all elements from the passed collection, at the specified
            ↪  position, increases the position value by the number of elements added and returns
            ↪  the value of the passed constant.</para>
218         /// <para>Добавляет в массив все элементы из переданной коллекции, на указанную позицию,
            ↪  увеличивает значение position на количество добавленных элементов и возвращает
            ↪  значение переданной константы.</para>
219         /// </summary>
220         /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
            ↪  массива.</para></typeparam>
221         /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
            ↪  возвращаемой константы.</para></typeparam>
222         /// <param name="array"><para>The array to add the element to.</para><para>Массив в
            ↪  который необходимо добавить элементы.</para></param>
223         /// <param name="position"><para>Reference to the position from which elements will be
            ↪  added to the array.</para><para>Ссылка на позицию, начиная с которой будут
            ↪  добавляться элементы в массив.</para></param>
224         /// <param name="elements"><para>List, whose elements will be added to the
            ↪  array.</para><para>Список, элементы которого будут добавленны 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 добавленны 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 добавленны в
            ↪ массив.</para></param>
263         /// <param name="returnConstant"><para>The constant value that will be
            ↪ returned.</para><para>Значение константы, которое будет возвращено.</para></param>
264         /// <returns>
265         /// <para>The constant value passed as an argument.</para>
266         /// <para>Значение константы, переданное в качестве аргумента.</para>
267         /// </returns>
268         [MethodImpl(MethodImplOptions.AggressiveInlining)]
269         public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
            ↪ TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
            ↪ TReturnConstant returnConstant)
270         {
271             array.AddSkipFirst(ref position, elements);
272             return returnConstant;
273         }
274
275         /// <summary>
```

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

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

```
1  using System;
2  using System.Collections.Concurrent;
3  using System.Collections.Generic;
4  using System.Numerics;
5  using System.Runtime.CompilerServices;
6  using System.Threading.Tasks;
7  using Platform.Exceptions;
8  using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>
16     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
       ↪  64 бит в массиве значений.
17     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
       ↪  байт в 8 байт.
18     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
       ↪  помощью которой можно быстро
19     /// проверять есть ли значения непосредственно далее (ниже по уровню).
20     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
21     /// </remarks>
22     public class BitString : IEquatable<BitString>
23     {
24         /// <summary>
```

```
        /// <para>
        /// The bits set in 16 bits.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly byte[][] _bitsSetIn16Bits;
        /// <summary>
        /// <para>
        /// The array.
        /// </para>
        /// <para></para>
        /// </summary>
        private long[] _array;
        /// <summary>
        /// <para>
        /// The length.
        /// </para>
        /// <para></para>
        /// </summary>
        private long _length;
        /// <summary>
        /// <para>
        /// The min positive word.
        /// </para>
        /// <para></para>
        /// </summary>
        private long _minPositiveWord;
        /// <summary>
        /// <para>
        /// The max positive word.
        /// </para>
        /// <para></para>
        /// </summary>
        private long _maxPositiveWord;

        /// <summary>
        /// <para>
        /// The value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool this[long index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Get(index);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => Set(index, value);
        }

        /// <summary>
        /// <para>
        /// Gets or sets the length value.
        /// </para>
        /// <para></para>
        /// </summary>
        public long Length
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _length;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set
            {
                if (_length == value)
                {
                    return;
                }
                Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
                // Currently we never shrink the array
                if (value > _length)
                {
                    var words = GetWordsCountFromIndex(value);
                    var oldWords = GetWordsCountFromIndex(_length);
                    if (words > _array.LongLength)
                    {
                        var copy = new long[words];
                        Array.Copy(_array, copy, _array.LongLength);
                        _array = copy;
                    }
                    else
```

```
                {
                    // What is going on here?
                    Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
                }
                // What is going on here?
                var mask = (int)(_length % 64);
                if (mask > 0)
                {
                    _array[oldWords - 1] &= (1L << mask) - 1;
                }
            }
            else
            {
                // Looks like minimum and maximum positive words are not updated
                throw new NotImplementedException();
            }
            _length = value;
        }
    }

    #region Constructors

    /// <summary>
    /// <para>
    /// Initializes a new <see cref="BitString"/> instance.
    /// </para>
    /// <para></para>
    /// </summary>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static BitString()
    {
        _bitsSetIn16Bits = new byte[65536][];
        int i, c, k;
        byte bitIndex;
        for (i = 0; i < 65536; i++)
        {
            // Calculating size of array (number of positive bits)
            for (c = 0, k = 1; k <= 65536; k <<= 1)
            {
                if ((i & k) == k)
                {
                    c++;
                }
            }
            var array = new byte[c];
            // Adding positive bits indices into array
            for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <<= 1)
            {
                if ((i & k) == k)
                {
                    array[c++] = bitIndex;
                }
                bitIndex++;
            }
            _bitsSetIn16Bits[i] = array;
        }
    }

    /// <summary>
    /// <para>
    /// Initializes a new <see cref="BitString"/> instance.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="other">
    /// <para>A other.</para>
    /// <para></para>
    /// </param>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public BitString(BitString other)
    {
        Ensure.Always.ArgumentNotNull(other, nameof(other));
        _length = other._length;
        _array = new long[GetWordsCountFromIndex(_length)];
        _minPositiveWord = other._minPositiveWord;
        _maxPositiveWord = other._maxPositiveWord;
        Array.Copy(other._array, _array, _array.LongLength);
    }

```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BitString"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="length">
        /// <para>A length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString(long length)
        {
            Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
            _length = length;
            _array = new long[GetWordsCountFromIndex(_length)];
            MarkBordersAsAllBitsReset();
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BitString"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="length">
        /// <para>A length.</para>
        /// <para></para>
        /// </param>
        /// <param name="defaultValue">
        /// <para>A default value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString(long length, bool defaultValue)
            : this(length)
        {
            if (defaultValue)
            {
                SetAll();
            }
        }

        #endregion

        /// <summary>
        /// <para>
        /// Nots this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Not()
        {
            for (var i = 0L; i < _array.LongLength; i++)
            {
                _array[i] = ~_array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the not.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelNot()
```

```
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return Not();
            }
            var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
            ↪  threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range =>
            {
                var maximum = range.Item2;
                for (var i = range.Item1; i < maximum; i++)
                {
                    _array[i] = ~_array[i];
                }
            });
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Vectors the not.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString VectorNot()
        {
            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
            {
                return Not();
            }
            var step = Vector<long>.Count;
            if (_array.Length < step)
            {
                return Not();
            }
            VectorNotLoop(_array, step, 0, _array.Length);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the vector not.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorNot()
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorNot();
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelNot();
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
                return VectorNot();
            }
            var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
```

```
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
            ↪  range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Vectors the not loop using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <param name="step">
        /// <para>The step.</para>
        /// <para></para>
        /// </param>
        /// <param name="start">
        /// <para>The start.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximum">
        /// <para>The maximum.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorNotLoop(long[] array, int step, int start, int maximum)
        {
            var i = start;
            var range = maximum - start - 1;
            var stop = range - (range % step);
            for (; i < stop; i += step)
            {
                (~new Vector<long>(array, i)).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] = ~array[i];
            }
        }

        /// <summary>
        /// <para>
        /// Ands the other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString And(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                _array[i] &= otherArray[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the and using the specified other.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelAnd(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return And(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range =>
            {
                var maximum = range.Item2;
                for (var i = range.Item1; i < maximum; i++)
                {
                    _array[i] &= other._array[i];
                }
            });
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Vectors the and using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString VectorAnd(BitString other)
        {
            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
            {
                return And(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < step)
            {
                return And(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            VectorAndLoop(_array, other._array, step, from, to + 1);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the vector and using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <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 {
                    ↪  MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
                    ↪  step, range.Item1, range.Item2));
516                 MarkBordersAsAllBitsSet();
517                 TryShrinkBorders();
518                 return this;
519             }
520 
521             /// <summary>
522             /// <para>
523             /// Vectors the and loop using the specified array.
524             /// </para>
525             /// <para></para>
526             /// </summary>
527             /// <param name="array">
528             /// <para>The array.</para>
529             /// <para></para>
530             /// </param>
531             /// <param name="otherArray">
532             /// <para>The other array.</para>
533             /// <para></para>
534             /// </param>
535             /// <param name="step">
536             /// <para>The step.</para>
537             /// <para></para>
538             /// </param>
539             /// <param name="start">
540             /// <para>The start.</para>
541             /// <para></para>
542             /// </param>
543             /// <param name="maximum">
544             /// <para>The maximum.</para>
545             /// <para></para>
546             /// </param>
547             [MethodImpl(MethodImplOptions.AggressiveInlining)]
548             static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
                ↪  int maximum)
549             {
550                 var i = start;
551                 var range = maximum - start - 1;
552                 var stop = range - (range % step);
553                 for (; i < stop; i += step)
554                 {
555                     (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
556                 }
557                 for (; i < maximum; i++)
558                 {
559                     array[i] &= otherArray[i];
560                 }
561             }
562 
563             /// <summary>
564             /// <para>
```

```
        /// Ors the other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Or(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] |= other._array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the or using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelOr(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return Or(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range =>
            {
                var maximum = range.Item2;
                for (var i = range.Item1; i < maximum; i++)
                {
                    _array[i] |= other._array[i];
                }
            });
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Vectors the or using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString VectorOr(BitString other)
        {
            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
            {
                return Or(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < step)
            {
                return Or(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            VectorOrLoop(_array, other._array, step, from, to + 1);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the vector or using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorOr(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorOr(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelOr(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
                return VectorOr(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
            ↪  step, range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Vectors the or loop using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <param name="otherArray">
        /// <para>The other array.</para>
        /// <para></para>
        /// </param>
        /// <param name="step">
        /// <para>The step.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="start">
        /// <para>The start.</para>
        /// <para></para>
        /// </param>
        /// <param name="maximum">
        /// <para>The maximum.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
        ↪  int maximum)
        {
            var i = start;
            var range = maximum - start - 1;
            var stop = range - (range % step);
            for (; i < stop; i += step)
            {
                (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] |= otherArray[i];
            }
        }

        /// <summary>
        /// <para>
        /// Xors the other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Xor(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] ^= other._array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the xor using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelXor(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return Xor(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
```

```
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
            ↪  MaxDegreeOfParallelism = threads }, range =>
            {
                var maximum = range.Item2;
                for (var i = range.Item1; i < maximum; i++)
                {
                    _array[i] ^= other._array[i];
                }
            });
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Vectors the xor using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString VectorXor(BitString other)
        {
            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
            {
                return Xor(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < step)
            {
                return Xor(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            VectorXorLoop(_array, other._array, step, from, to + 1);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        /// <summary>
        /// <para>
        /// Parallels the vector xor using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bit string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorXor(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorXor(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelXor(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
```

```
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--;
```

```
                }
            }
            else
            {
                if (wordIndex < _minPositiveWord)
                {
                    _minPositiveWord = wordIndex;
                }
                if (wordIndex > _maxPositiveWord)
                {
                    _maxPositiveWord = wordIndex;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance try shrink borders.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The borders updated.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool TryShrinkBorders()
        {
            GetBorders(out long from, out long to);
            while (from <= to && _array[from] == 0)
            {
                from++;
            }
            if (from > to)
            {
                MarkBordersAsAllBitsReset();
                return true;
            }
            while (to >= from && _array[to] == 0)
            {
                to--;
            }
            if (to < from)
            {
                MarkBordersAsAllBitsReset();
                return true;
            }
            var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
            if (bordersUpdated)
            {
                SetBorders(from, to);
            }
            return bordersUpdated;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance get.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Get(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
        }

        /// <summary>
        /// <para>
        /// Sets the index.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(long index, bool value)
        {
            if (value)
            {
                Set(index);
            }
            else
            {
                Reset(index);
            }
        }

        /// <summary>
        /// <para>
        /// Sets the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            _array[wordIndex] |= mask;
            RefreshBordersByWord(wordIndex);
        }

        /// <summary>
        /// <para>
        /// Resets the index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            _array[wordIndex] &= ~mask;
            RefreshBordersByWord(wordIndex);
        }

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

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(long index)
        {
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            if ((_array[wordIndex] & mask) == 0)
            {
                _array[wordIndex] |= mask;
                RefreshBordersByWord(wordIndex);
                return true;
            }
            else
            {
                return false;
            }
        }

        /// <summary>
        /// <para>
        /// Sets the all using the specified value.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetAll(bool value)
        {
            if (value)
            {
                SetAll();
            }
            else
            {
                ResetAll();
            }
        }

        /// <summary>
        /// <para>
        /// Sets the all.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetAll()
        {
            const long fillValue = unchecked((long)0xffffffffffffffff);
            var words = GetWordsCountFromIndex(_length);
            for (var i = 0; i < words; i++)
            {
                _array[i] = fillValue;
            }
            MarkBordersAsAllBitsSet();
        }

        /// <summary>
        /// <para>
        /// Resets the all.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ResetAll()
        {
            const long fillValue = 0;
            GetBorders(out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] = fillValue;
            }
            MarkBordersAsAllBitsReset();
        }

        /// <summary>
        /// <para>
```

```
        /// Gets the set indices.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<long> GetSetIndices()
        {
            var result = new List<long>();
            GetBorders(out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    AppendAllSetBitIndices(result, i, word);
                }
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Gets the set u int 64 indices.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetSetUInt64Indices()
        {
            var result = new List<ulong>();
            GetBorders(out ulong from, out ulong to);
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    AppendAllSetBitIndices(result, i, word);
                }
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Gets the first set bit index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetFirstSetBitIndex()
        {
            var i = _minPositiveWord;
            var word = _array[i];
            if (word != 0)
            {
                return GetFirstSetBitForWord(i, word);
            }
            return -1;
        }

        /// <summary>
        /// <para>
        /// Gets the last set bit index.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetLastSetBitIndex()
        {
            var i = _maxPositiveWord;
            var word = _array[i];
            if (word != 0)
            {
                return GetLastSetBitForWord(i, word);
            }
            return -1;
        }

        /// <summary>
        /// <para>
        /// Counts the set bits.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long CountSetBits()
        {
            var total = 0L;
            GetBorders(out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    total += CountSetBitsForWord(word);
                }
            }
            return total;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance have common bits.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool HaveCommonBits(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                if (left != 0 && right != 0 && (left & right) != 0)
                {
                    return true;
                }
            }
            return false;
        }

        /// <summary>
        /// <para>
        /// Counts the common bits using the specified other.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long CountCommonBits(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var total = 0L;
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    total += CountSetBitsForWord(combined);
                }
            }
            return total;
        }

        /// <summary>
        /// <para>
        /// Gets the common indices using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<long> GetCommonIndices(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var result = new List<long>();
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    AppendAllSetBitIndices(result, i, combined);
                }
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Gets the common u int 64 indices using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetCommonUInt64Indices(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonBorders(this, other, out ulong from, out ulong to);
            var result = new List<ulong>();
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    AppendAllSetBitIndices(result, i, combined);
                }
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Gets the first common bit index using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetFirstCommonBitIndex(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    return GetFirstSetBitForWord(i, combined);
                }
            }
            return -1;
        }

        /// <summary>
        /// <para>
        /// Gets the last common bit index using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetLastCommonBitIndex(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = to; i >= from; i--)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
```

```
                if (combined != 0)
                {
                    return GetLastSetBitForWord(i, combined);
                }
            }
            return -1;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
        ↪  false;

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(BitString other)
        {
            if (_length != other._length)
            {
                return false;
            }
            var otherArray = other._array;
            if (_array.Length != otherArray.Length)
            {
                return false;
            }
            if (_minPositiveWord != other._minPositiveWord)
            {
                return false;
            }
            if (_maxPositiveWord != other._maxPositiveWord)
            {
                return false;
            }
            GetCommonBorders(this, other, out ulong from, out ulong to);
            for (var i = from; i <= to; i++)
            {
                if (_array[i] != otherArray[i])
                {
                    return false;
                }
            }
            return true;
        }

        /// <summary>
        /// <para>
        /// Ensures the bit string has the same size using the specified other.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
```

```
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentException">
        /// <para>Bit string must be the same size. </para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
        {
            Ensure.Always.ArgumentNotNull(other, argumentName);
            if (_length != other._length)
            {
                throw new ArgumentException("Bit string must be the same size.", argumentName);
            }
        }

        /// <summary>
        /// <para>
        /// Marks the borders as all bits reset.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);

        /// <summary>
        /// <para>
        /// Marks the borders as all bits set.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);

        /// <summary>
        /// <para>
        /// Gets the borders using the specified from.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out long from, out long to)
        {
            from = _minPositiveWord;
            to = _maxPositiveWord;
        }

        /// <summary>
        /// <para>
        /// Gets the borders using the specified from.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out ulong from, out ulong to)
        {
            from = (ulong)_minPositiveWord;
```

```
            to = (ulong)_maxPositiveWord;
        }

        /// <summary>
        /// <para>
        /// Sets the borders using the specified from.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void SetBorders(long from, long to)
        {
            _minPositiveWord = from;
            _maxPositiveWord = to;
        }

        /// <summary>
        /// <para>
        /// Gets the valid index range.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A range of long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Range<long> GetValidIndexRange() => (0, _length - 1);

        /// <summary>
        /// <para>
        /// Gets the valid length range.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A range of long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Range<long> GetValidLengthRange() => (0, long.MaxValue);

        /// <summary>
        /// <para>
        /// Appends the all set bit indices using the specified result.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="wordIndex">
        /// <para>The word index.</para>
        /// <para></para>
        /// </param>
        /// <param name="wordValue">
        /// <para>The word value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
        ↪  wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪  bits32to47, out byte[] bits48to63);
            AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
            ↪  bits48to63);
        }

```

```
        /// <summary>
        /// <para>
        /// Appends the all set bit indices using the specified result.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="wordIndex">
        /// <para>The word index.</para>
        /// <para></para>
        /// </param>
        /// <param name="wordValue">
        /// <para>The word value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
        ↪  wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪  bits32to47, out byte[] bits48to63);
            AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
            ↪  bits48to63);
        }

        /// <summary>
        /// <para>
        /// Counts the set bits for word using the specified word.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="word">
        /// <para>The word.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long CountSetBitsForWord(long word)
        {
            GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
            ↪  out byte[] bits48to63);
            return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
            ↪  bits48to63.LongLength;
        }

        /// <summary>
        /// <para>
        /// Gets the first set bit for word using the specified word index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="wordIndex">
        /// <para>The word index.</para>
        /// <para></para>
        /// </param>
        /// <param name="wordValue">
        /// <para>The word value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪  bits32to47, out byte[] bits48to63);
            return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
        }

```

```
        /// <summary>
        /// <para>
        /// Gets the last set bit for word using the specified word index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="wordIndex">
        /// <para>The word index.</para>
        /// <para></para>
        /// </param>
        /// <param name="wordValue">
        /// <para>The word value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetLastSetBitForWord(long wordIndex, long wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47, out byte[] bits48to63);
            return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
        }

        /// <summary>
        /// <para>
        /// Appends the all set bit indices using the specified result.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="i">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <param name="bits00to15">
        /// <para>The bits 00to 15.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits16to31">
        /// <para>The bits 16to 31.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits32to47">
        /// <para>The bits 32to 47.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits48to63">
        /// <para>The bits 48to 63.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15, byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
        {
            for (var j = 0; j < bits00to15.Length; j++)
            {
                result.Add(bits00to15[j] + (i * 64));
            }
            for (var j = 0; j < bits16to31.Length; j++)
            {
                result.Add(bits16to31[j] + 16 + (i * 64));
            }
            for (var j = 0; j < bits32to47.Length; j++)
            {
                result.Add(bits32to47[j] + 32 + (i * 64));
            }
            for (var j = 0; j < bits48to63.Length; j++)
            {
                result.Add(bits48to63[j] + 48 + (i * 64));
            }
        }

```

```
        /// <summary>
        /// <para>
        /// Appends the all set indices using the specified result.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="i">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <param name="bits00to15">
        /// <para>The bits 00to 15.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits16to31">
        /// <para>The bits 16to 31.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits32to47">
        /// <para>The bits 32to 47.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits48to63">
        /// <para>The bits 48to 63.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
        ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
        {
            for (var j = 0; j < bits00to15.Length; j++)
            {
                result.Add(bits00to15[j] + (i * 64));
            }
            for (var j = 0; j < bits16to31.Length; j++)
            {
                result.Add(bits16to31[j] + 16UL + (i * 64));
            }
            for (var j = 0; j < bits32to47.Length; j++)
            {
                result.Add(bits32to47[j] + 32UL + (i * 64));
            }
            for (var j = 0; j < bits48to63.Length; j++)
            {
                result.Add(bits48to63[j] + 48UL + (i * 64));
            }
        }

        /// <summary>
        /// <para>
        /// Gets the first set bit using the specified i.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="i">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <param name="bits00to15">
        /// <para>The bits 00to 15.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits16to31">
        /// <para>The bits 16to 31.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits32to47">
        /// <para>The bits 32to 47.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits48to63">
        /// <para>The bits 48to 63.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
        ↪  bits32to47, byte[] bits48to63)
        {
            if (bits00to15.Length > 0)
            {
                return bits00to15[0] + (i * 64);
            }
            if (bits16to31.Length > 0)
            {
                return bits16to31[0] + 16 + (i * 64);
            }
            if (bits32to47.Length > 0)
            {
                return bits32to47[0] + 32 + (i * 64);
            }
            return bits48to63[0] + 48 + (i * 64);
        }

        /// <summary>
        /// <para>
        /// Gets the last set bit using the specified i.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="i">
        /// <para>The .</para>
        /// <para></para>
        /// </param>
        /// <param name="bits00to15">
        /// <para>The bits 00to 15.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits16to31">
        /// <para>The bits 16to 31.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits32to47">
        /// <para>The bits 32to 47.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits48to63">
        /// <para>The bits 48to 63.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
        ↪  bits32to47, byte[] bits48to63)
        {
            if (bits48to63.Length > 0)
            {
                return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
            }
            if (bits32to47.Length > 0)
            {
                return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
            }
            if (bits16to31.Length > 0)
            {
                return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
            }
            return bits00to15[bits00to15.Length - 1] + (i * 64);
        }

        /// <summary>
        /// <para>
        /// Gets the bits using the specified word.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="word">
        /// <para>The word.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits00to15">
        /// <para>The bits 00to 15.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits16to31">
        /// <para>The bits 16to 31.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits32to47">
        /// <para>The bits 32to 47.</para>
        /// <para></para>
        /// </param>
        /// <param name="bits48to63">
        /// <para>The bits 48to 63.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
        ↪  byte[] bits32to47, out byte[] bits48to63)
        {
            bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
            bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
            bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
            bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
        }

        /// <summary>
        /// <para>
        /// Gets the common inner borders using the specified left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
        ↪  out long to)
        {
            from = Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        /// <summary>
        /// <para>
        /// Gets the common outer borders using the specified left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
        ↪  out long to)
        {
            from = Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        /// <summary>
        /// <para>
        /// Gets the common outer borders using the specified left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
        ↪  out int to)
        {
            from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        /// <summary>
        /// <para>
        /// Gets the common borders using the specified left.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="left">
        /// <para>The left.</para>
        /// <para></para>
        /// </param>
        /// <param name="right">
        /// <para>The right.</para>
        /// <para></para>
        /// </param>
        /// <param name="from">
        /// <para>The from.</para>
        /// <para></para>
        /// </param>
        /// <param name="to">
        /// <para>The to.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
        ↪  ulong to)
        {
            from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        /// <summary>
        /// <para>
        /// Gets the words count from index using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;

        /// <summary>
        /// <para>
        /// Gets the word index from index using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordIndexFromIndex(long index) => index >> 6;

        /// <summary>
        /// <para>
        /// Gets the bit mask from index using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);

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

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

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

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

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

```

namespace Platform.Collections
{
    /// <summary>
    /// <para>
    /// Represents the bit string extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class BitStringExtensions
    {
        /// <summary>
        /// <para>
        /// Sets the random bits using the specified string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@string">
        /// <para>The string.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void SetRandomBits(this BitString @string)
        {
            for (var i = 0; i < @string.Length; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                @string.Set(i, value);
            }
        }
    }
}
```

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

```
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Concurrent
{
    /// <summary>
    /// <para>
    /// Represents the concurrent queue extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ConcurrentQueueExtensions
    {
        /// <summary>
        /// <para>
        /// Dequeues the all using the specified queue.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="queue">
        /// <para>The queue.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
        {
            while (queue.TryDequeue(out T item))
            {
                yield return item;
            }
        }
    }
}
```

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

```
using System.Collections.Concurrent;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Concurrent
{
    /// <summary>
    /// <para>
    /// Represents the concurrent stack extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ConcurrentStackExtensions
    {
        /// <summary>
        /// <para>
        /// Pops the or default using the specified stack.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
        ↪  value) ? value : default;

        /// <summary>
        /// <para>
        /// Peeks the or default using the specified stack.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
        ↪  value) ? value : default;
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections
{
    /// <summary>
    /// <para>
    /// Represents the ensure extensions.
    /// </para>
```

```
    /// <para></para>
    /// </summary>
    public static class EnsureExtensions
    {
        #region Always

        /// <summary>
        /// <para>
        /// Arguments the not empty using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName, string message)
        {
            if (argument.IsNullOrEmpty())
            {
                throw new ArgumentException(message, argumentName);
            }
        }

        /// <summary>
        /// <para>
        /// Arguments the not empty using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
        ↪  argumentName, null);

        /// <summary>
        /// <para>
        /// Arguments the not empty using the specified root.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
        ↪  ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);

        /// <summary>
        /// <para>
        /// Arguments the not empty and not white space using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
        ↪  string argument, string argumentName, string message)
        {
            if (string.IsNullOrWhiteSpace(argument))
            {
                throw new ArgumentException(message, argumentName);
            }
        }

        /// <summary>
        /// <para>
        /// Arguments the not empty and not white space using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
        ↪  string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
        ↪  argument, argumentName, null);

        /// <summary>
```

```
        /// <para>
        /// Arguments the not empty and not white space using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
        ↪  string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);

        #endregion

        #region OnDebug

        /// <summary>
        /// <para>
        /// Arguments the not empty using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        /// <param name="message">
        /// <para>The message.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName, string message) =>
        ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);

        /// <summary>
        /// <para>
        /// Arguments the not empty using the specified root.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <param name="argumentName">
        /// <para>The argument name.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName) =>
        ↪  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,
            ↪  ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
259
260         /// <summary>
261         /// <para>
262         /// Arguments the not empty and not white space using the specified root.
263         /// </para>
264         /// <para></para>
265         /// </summary>
266         /// <param name="root">
267         /// <para>The root.</para>
268         /// <para></para>
269         /// </param>
270         /// <param name="argument">
271         /// <para>The argument.</para>
272         /// <para></para>
273         /// </param>
274         /// <param name="argumentName">
275         /// <para>The argument name.</para>
276         /// <para></para>
277         /// </param>
278         /// <param name="message">
279         /// <para>The message.</para>
280         /// <para></para>
281         /// </param>
282         [Conditional("DEBUG")]
283         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪  root, string argument, string argumentName, string message) =>
            ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
284
285         /// <summary>
286         /// <para>
287         /// Arguments the not empty and not white space using the specified root.
288         /// </para>
289         /// <para></para>
290         /// </summary>
291         /// <param name="root">
292         /// <para>The root.</para>
293         /// <para></para>
294         /// </param>
295         /// <param name="argument">
296         /// <para>The argument.</para>
297         /// <para></para>
298         /// </param>
299         /// <param name="argumentName">
300         /// <para>The argument name.</para>
301         /// <para></para>
302         /// </param>
303         [Conditional("DEBUG")]
304         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪  root, string argument, string argumentName) =>
            ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
305
306         /// <summary>
307         /// <para>
308         /// Arguments the not empty and not white space using the specified root.
309         /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="root">
        /// <para>The root.</para>
        /// <para></para>
        /// </param>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ↪  root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
        ↪  null, null);

        #endregion
    }
}
```

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

```
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections
{
    /// <summary>
    /// <para>Presents a set of methods for working with collections.</para>
    /// <para>Представляет набор методов для работы с коллекциями.</para>
    /// </summary>
    public static class ICollectionExtensions
    {
        /// <summary>
        /// <para>Checking collection for empty.</para>
        /// <para>Проверяет коллекцию на пустоту.</para>
        /// </summary>
        /// <param name="collection">
        /// <para>Method takes an elements collection of <see cref="ICollection<T>"/>
        ↪  type.</para>
        /// <para>Метода принимает колекцию элементов <see cref="ICollection<T>"/> типа.</para>
        /// </param>
        /// <returns>
        /// <para>Returns a <see cref="bool"/> type variable equal to False if the collection is
        ↪  empty else returns true.</para>
        /// <para>Возвращает переменную типа <see cref="bool"/> равной false если коллекция
        ↪  пустая иначе возвращает true.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
        ↪  null || collection.Count == 0;

        /// <summary>
        /// <para>
        /// Determines whether all equal to default.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="collection">
        /// <para>The collection.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AllEqualToDefault<T>(this ICollection<T> collection)
        {
            var equalityComparer = EqualityComparer<T>.Default;
            return collection.All(item => equalityComparer.Equals(item, default));
        }
    }
```

```
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections
{
    /// <summary>
    /// <para>
    /// Represents the dictionary extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class IDictionaryExtensions
    {
        /// <summary>
        /// <para>
        /// Gets the or default using the specified dictionary.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TKey">
        /// <para>The key.</para>
        /// <para></para>
        /// </typeparam>
        /// <typeparam name="TValue">
        /// <para>The value.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="dictionary">
        /// <para>The dictionary.</para>
        /// <para></para>
        /// </param>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The value.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
        ↪  dictionary, TKey key)
        {
            dictionary.TryGetValue(key, out TValue value);
            return value;
        }

        /// <summary>
        /// <para>
        /// Gets the or add using the specified dictionary.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TKey">
        /// <para>The key.</para>
        /// <para></para>
        /// </typeparam>
        /// <typeparam name="TValue">
        /// <para>The value.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="dictionary">
        /// <para>The dictionary.</para>
        /// <para></para>
        /// </param>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </param>
        /// <param name="valueFactory">
        /// <para>The value factory.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The value.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
        ↪  TKey key, Func<TKey, TValue> valueFactory)
        {
            if (!dictionary.TryGetValue(key, out TValue value))
            {
                value = valueFactory(key);
                dictionary.Add(key, value);
                return value;
            }
            return value;
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Lists
{
    /// <summary>
    /// <para>
    /// Represents the char list extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class CharIListExtensions
    {
        /// <summary>
        /// <para>Generates a hash code for the entire list based on the values of its
        ↪  elements.</para>
        /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
        /// </summary>
        /// <param name="list"><para>The list to be hashed.</para><para>Список для
        ↪  хеширования.</para></param>
        /// <returns>
        /// <para>The hash code of the list.</para>
        /// <para>Хэш-код списка.</para>
        /// </returns>
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
        ↪  a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GenerateHashCode(this IList<char> list)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            for (var i = 0; i < list.Count; i++)
            {
                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        /// <summary>
        /// <para>Compares two lists for equality.</para>
        /// <para>Сравнивает два списка на равенство.</para>
        /// </summary>
        /// <param name="left"><para>The first compared list.</para><para>Первый список для
        ↪  сравнения.</para></param>
        /// <param name="right"><para>The second compared list.</para><para>Второй список для
        ↪  сравнения.</para></param>
        /// <returns>
        /// <para>True, if the passed lists are equal to each other otherwise false.</para>
        /// <para>True, если переданные списки равны друг другу, иначе false.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo(this IList<char> left, IList<char> right) =>
        ↪  left.EqualTo(right, ContentEqualTo);

        /// <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>
```

```
        ///     </list>
        /// </para>
        /// <para>
        ///     Целое число со знаком, которое указывает относительные значения элементов списков <paramref name="left" /> и <paramref name="right" /> как показано в следующей таблице.
        ///     <list type="table">
        ///         <listheader>
        ///             <term>Значение</term>
        ///             <description>Смысл</description>
        ///         </listheader>
        ///         <item>
        ///             <term>Меньше нуля</term>
        ///             <description>Первый не равный элемент <paramref name="left" /> списка меньше первого неравного элемента <paramref name="right" /> списка.</description>
        ///         </item>
        ///         <item>
        ///             <term>Ноль</term>
        ///             <description>Все элементы <paramref name="left" /> списка равны всем элементам <paramref name="right" /> списка.</description>
        ///         </item>
        ///         <item>
        ///             <term>Больше нуля</term>
        ///             <description>Первый не равный элемент <paramref name="left" /> списка больше первого неравного элемента <paramref name="right" /> списка.</description>
        ///         </item>
        ///     </list>
        /// </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Lists
{
    /// <summary>
    /// <para>
    /// Represents the list equality comparer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IEqualityComparer{IList{T}}"/>
    public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
    {
        /// <summary>
        /// <para>Compares two lists for equality.</para>
        /// <para>Сравнивает два списка на равенство.</para>
        /// </summary>
        /// <param name="left"><para>The first compared list.</para><para>Первый список для сравнения.</para></param>
        /// <param name="right"><para>The second compared list.</para><para>Второй список для сравнения.</para></param>
        /// <returns>
        /// <para>If the passed lists are equal to each other, true is returned, otherwise false.</para>
        /// <para>Если переданные списки равны друг другу, возвращается true, иначе же false.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);

        /// <summary>
        /// <para>Generates a hash code for the entire list based on the values of its elements.</para>
        /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
        /// </summary>
        /// <param name="list"><para>Hash list.</para><para>Список для хеширования.</para></param>
        /// <returns>
        /// <para>The hash code of the list.</para>
        /// <para>Хэш-код списка.</para>
        /// </returns>
```

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

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Lists
{
    /// <summary>
    /// <para>
    /// Represents the list extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class IListExtensions
    {
        /// <summary>
        /// <para>Gets the element from specified index if the list is not null and the index is within the list's boundaries, otherwise it returns default value of type T.</para>
        /// <para>Получает элемент из указанного индекса, если список не является null и индекс находится в границах списка, в противном случае он возвращает значение по умолчанию типа T.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов списка.</para></typeparam>
        /// <param name="list"><para>The checked list.</para><para>Проверяемый список.</para></param>
        /// <param name="index"><para>The index of element.</para><para>Индекс элемента.</para></param>
        /// <returns>
        /// <para>If the specified index is within list's boundaries, then - list[index], otherwise the default value.</para>
        /// <para>Если указанный индекс находится в пределах границ списка, тогда - list[index], иначе же значение по умолчанию.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null && list.Count > index ? list[index] : default;

        /// <summary>
        /// <para>Checks if a list is passed, checks its length, and if successful, copies the value of list [index] into the element variable. Otherwise, the element variable has a default value.</para>
        /// <para>Проверяет, передан ли список, сверяет его длину и в случае успеха копирует значение list[index] в переменную element. Иначе переменная element имеет значение по умолчанию.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов списка.</para></typeparam>
        /// <param name="list"><para>The checked list.</para><para>Список для проверки.</para></param>
        /// <param name="index"><para>The index of element..</para><para>Индекс элемента.</para></param>
        /// <param name="element"><para>Variable for passing the index value.</para><para>Переменная для передачи значения индекса.</para></param>
        /// <returns>
        /// <para>True on success, false otherwise.</para>
        /// <para>True в случае успеха, иначе false.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
        {
            if (list != null && list.Count > index)
            {
                element = list[index];
                return true;
            }
            else
            {
                element = default;
                return false;
            }
        }
```

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

```
118         }
119
120         /// <summary>
121         /// <para>Adds all elements from other list to this list.</para>
122         /// <para>Добавляет все элементы из другого списка в этот список.</para>
123         /// </summary>
124         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
125         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
            ↪  нужно добавить значения.</para></param>
126         /// <param name="elements"><para>The list of values to add.</para><para>Список значений
            ↪  которые необходимо добавить.</para></param>
127         [MethodImpl(MethodImplOptions.AggressiveInlining)]
128         public static void AddAll<T>(this IList<T> list, IList<T> elements)
129         {
130             for (var i = 0; i < elements.Count; i++)
131             {
132                 list.Add(elements[i]);
133             }
134         }
135
136         /// <summary>
137         /// <para>Adds values to the list skipping the first element.</para>
138         /// <para>Добавляет значения в список пропуская первый элемент.</para>
139         /// </summary>
140         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
141         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
            ↪  нужно добавить значения.</para></param>
142         /// <param name="elements"><para>The list of values to add.</para><para>Список значений
            ↪  которые необходимо добавить.</para></param>
143         /// <returns>
144         /// <para>True value in any case.</para>
145         /// <para>Значение true в любом случае.</para>
146         /// </returns>
147         [MethodImpl(MethodImplOptions.AggressiveInlining)]
148         public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
149         {
150             list.AddSkipFirst(elements);
151             return true;
152         }
153
154         /// <summary>
155         /// <para>Adds values to the list skipping the first element.</para>
156         /// <para>Добавляет значения в список пропуская первый элемент.</para>
157         /// </summary>
158         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
159         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
            ↪  нужно добавить значения.</para></param>
160         /// <param name="elements"><para>List of values to add.</para><para>Список значений
            ↪  которые необходимо добавить.</para></param>
161         [MethodImpl(MethodImplOptions.AggressiveInlining)]
162         public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
            ↪  list.AddSkipFirst(elements, 1);
163
164         /// <summary>
165         /// <para>Adds values to the list skipping a specified number of first elements.</para>
166         /// <para>Добавляет в список значения пропуская определенное количество первых
            ↪  элементов.</para>
167         /// </summary>
168         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
169         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
            ↪  нужно добавить значения.</para></param>
170         /// <param name="elements"><para>List of values to add.</para><para>Список значений
            ↪  которые необходимо добавить.</para></param>
171         /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
            ↪  пропускаемых элементов.</para></param>
172         [MethodImpl(MethodImplOptions.AggressiveInlining)]
173         public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
174         {
175             for (var i = skip; i < elements.Count; i++)
176             {
177                 list.Add(elements[i]);
178             }
179         }
```

```
180
181         /// <summary>
182         /// <para>Reads the number of elements in the list.</para>
183         /// <para>Считывает число элементов списка.</para>
184         /// </summary>
185         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪ списка.</para></typeparam>
186         /// <param name="list"><para>The checked list.</para><para>Список для
            ↪ проверки.</para></param>
187         /// <returns>
188         /// <para>The number of items contained in the list or 0.</para>
189         /// <para>Число элементов содержащихся в списке или же 0.</para>
190         /// </returns>
191         [MethodImpl(MethodImplOptions.AggressiveInlining)]
192         public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
193
194         /// <summary>
195         /// <para>Compares two lists for equality.</para>
196         /// <para>Сравнивает два списка на равенство.</para>
197         /// </summary>
198         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪ списка.</para></typeparam>
199         /// <param name="left"><para>The first compared list.</para><para>Первый список для
            ↪ сравнения.</para></param>
200         /// <param name="right"><para>The second compared list.</para><para>Второй список для
            ↪ сравнения.</para></param>
201         /// <returns>
202         /// <para>If the passed lists are equal to each other, true is returned, otherwise
            ↪ false.</para>
203         /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
            ↪ false.</para>
204         /// </returns>
205         [MethodImpl(MethodImplOptions.AggressiveInlining)]
206         public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
            ↪ right, ContentEqualTo);
207
208         /// <summary>
209         /// <para>Compares two lists for equality.</para>
210         /// <para>Сравнивает два списка на равенство.</para>
211         /// </summary>
212         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪ списка.</para></typeparam>
213         /// <param name="left"><para>The first compared list.</para><para>Первый список для
            ↪ 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">The list to copy from.<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             ///     Целое число со знаком, которое указывает относительные значения элементов
                ↪  списков <paramref name="left" /> и <paramref name="right" /> как показано в
                ↪  следующей таблице.
383             ///     <list type="table">
384             ///         <listheader>
385             ///             <term>Значение</term>
386             ///             <description>Смысл</description>
387             ///         </listheader>
388             ///         <item>
389             ///             <term>Меньше нуля</term>
390             ///             <description>Первый не равный элемент <paramref name="left" /> списка
                ↪  меньше первого неравного элемента <paramref name="right" /> списка.</description>
391             ///         </item>
392             ///         <item>
393             ///             <term>Ноль</term>
394             ///             <description>Все элементы <paramref name="left" /> списка равны всем
                ↪  элементам <paramref name="right" /> списка.</description>
395             ///         </item>
396             ///         <item>
397             ///             <term>Больше нуля</term>
398             ///             <description>Первый не равный элемент <paramref name="left" /> списка
                ↪  больше первого неравного элемента <paramref name="right" /> списка.</description>
399             ///         </item>
400             ///     </list>
401             /// </para>
402             /// </returns>
403             [MethodImpl(MethodImplOptions.AggressiveInlining)]
404             public static int CompareTo<T>(this IList<T> left, IList<T> right)
405             {
406                 var comparer = Comparer<T>.Default;
407                 var leftCount = left.GetCountOrZero();
408                 var rightCount = right.GetCountOrZero();
409                 var intermediateResult = leftCount.CompareTo(rightCount);
410                 for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
411                 {
412                     intermediateResult = comparer.Compare(left[i], right[i]);
413                 }
414                 return intermediateResult;
415             }
416
417             /// <summary>
418             /// <para>Skips one element in the list and builds an array from the remaining
                ↪  elements.</para>
419             /// <para>Пропускает один элемент списка и составляет из оставшихся элементов
                ↪  массив.</para>
420             /// </summary>
421             /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
                ↪  списка.</para></typeparam>
422             /// <param name="list"><para>The list to copy from.</para><para>Список для
                ↪  копирования.</para></param>
423             /// <returns>
424             /// <para>If the list is empty, returns an empty array, otherwise - an array with a
                ↪  missing first element.</para>
425             /// <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
                ↪  первым элементом.</para>
426             /// </returns>
427             [MethodImpl(MethodImplOptions.AggressiveInlining)]
428             public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
429
430             /// <summary>
```

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

```
            }
            if (shift == 0)
            {
                return list.ToArray();
            }
            else
            {
                var result = new T[list.Count + shift];
                for (int r = 0, w = shift; r < list.Count; r++, w++)
                {
                    result[w] = list[r];
                }
                return result;
            }
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Lists
{
    /// <summary>
    /// <para>
    /// Represents the list filler.
    /// </para>
    /// <para></para>
    /// </summary>
    public class ListFiller<TElement, TReturnConstant>
    {
        /// <summary>
        /// <para>
        /// The list.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly List<TElement> _list;
        /// <summary>
        /// <para>
        /// The return constant.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TReturnConstant _returnConstant;

        /// <summary>
        /// <para>Initializes a new instance of the ListFiller class.</para>
        /// <para>Инициализирует новый экземпляр класса ListFiller.</para>
        /// </summary>
        /// <param name="list"><para>The list to be filled.</para><para>Список который будет
        ↪  заполняться.</para></param>
        /// <param name="returnConstant"><para>The value for the constant returned by
        ↪  corresponding methods.</para><para>Значение для константы возвращаемой
        ↪  соответствующими методами.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ListFiller(List<TElement> list, TReturnConstant returnConstant)
        {
            _list = list;
            _returnConstant = returnConstant;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ListFiller"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="list">
        /// <para>A list.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ListFiller(List<TElement> list) : this(list, default) { }

        /// <summary>
        /// <para>Adds an item to the end of the list.</para>
        /// <para>Добавляет элемент в конец списка.</para>
```

```
58          /// </summary>
59          /// <param name="element"><para>Element to add.</para><para>Добавляемый
            ↪  элемент.</para></param>
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          public void Add(TElement element) => _list.Add(element);
62
63          /// <summary>
64          /// <para>Adds an item to the end of the list and return true.</para>
65          /// <para>Добавляет элемент в конец списка и возвращает true.</para>
66          /// </summary>
67          /// <param name="element"><para>Element to add.</para><para>Добавляемый
            ↪  элемент.</para></param>
68          /// <returns>
69          /// <para>True value in any case.</para>
70          /// <para>Значение true в любом случае.</para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
74
75          /// <summary>
76          /// <para>Adds a value to the list at the first index and return true.</para>
77          /// <para>Добавляет значение в список по первому индексу и возвращает true.</para>
78          /// </summary>
79          /// <param name="elements"><para>Element to add.</para><para>Добавляемый
            ↪  элемент.</para></param>
80          /// <returns>
81          /// <para>True value in any case.</para>
82          /// <para>Значение true в любом случае.</para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
            ↪  _list.AddFirstAndReturnTrue(elements);
86
87          /// <summary>
88          /// <para>Adds all elements from other list to this list and returns true.</para>
89          /// <para>Добавляет все элементы из другого списка в этот список и возвращает
            ↪  true.</para>
90          /// </summary>
91          /// <param name="elements"><para>List of values to add.</para><para>Список значений
            ↪  которые необходимо добавить.</para></param>
92          /// <returns>
93          /// <para>True value in any case.</para>
94          /// <para>Значение true в любом случае.</para>
95          /// </returns>
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public bool AddAllAndReturnTrue(IList<TElement> elements) =>
            ↪  _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(IList<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(IList<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(IList<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(IList<TElement> elements)
170         {
171             _list.AddSkipFirst(elements);
172             return _returnConstant;
173         }
174     }
175 }
```

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

```
1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the char segment.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="Segment{char}"/>
18     public class CharSegment : Segment<char>
19     {
```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CharSegment"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@base">
        /// <para>A base.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>A offset.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>A length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
        ↪  length) { }

        /// <summary>
        /// <para>
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode()
        {
            // Base can be not an array, but still IList<char>
            if (Base is char[] baseArray)
            {
                return baseArray.GenerateHashCode(Offset, Length);
            }
            else
            {
                return this.GenerateHashCode();
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(Segment<char> other)
        {
            bool contentEqualityComparer(IList<char> left, IList<char> right)
            {
                // Base can be not an array, but still IList<char>
                if (Base is char[] baseArray && other.Base is char[] otherArray)
                {
                    return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
                }
                else
                {
                    return left.ContentEqualTo(right);
                }
            }
            return this.EqualTo(other, contentEqualityComparer);
        }

```

```
        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public override bool Equals(object obj) => obj is Segment<char> charSegment ?
        ↪  Equals(charSegment) : false;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator string(CharSegment segment)
        {
            if (!(segment.Base is char[] array))
            {
                array = segment.Base.ToArray();
            }
            return new string(array, segment.Offset, segment.Length);
        }

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

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

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;

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

namespace Platform.Collections.Segments
{
    /// <summary>
    /// <para>Represents the segment of an <see cref="IList"/>.</para>
    /// <para>Представляет сегмент <see cref="IList"/>.</para>
    /// </summary>
    /// <typeparam name="T"><para>The segment elements type.</para><para>Тип элементов
    ↪  сегмента.</para></typeparam>
    public class Segment<T> : IEquatable<Segment<T>>, IList<T>
    {
        /// <summary>
        /// <para>Gets the original list (this segment is a part of it).</para>
        /// <para>Возвращает исходный список (частью которого является этот сегмент).</para>
        /// </summary>
        public IList<T> Base
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        /// <summary>
        /// <para>Gets the offset relative to the source list (the index at which this segment
        ↪  starts).</para>
        /// <para>Возвращает смещение относительного исходного списка (индекс с которого
        ↪  начинается этот сегмент).</para>
        /// </summary>
        public int Offset
        {
```

```
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        /// <summary>
        /// <para>Gets the length of a segment.</para>
        /// <para>Возвращает длину сегмента.</para>
        /// </summary>
        public int Length
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>Initializes a new instance of the <see cref="Segment"/> class, using the <paramref name="base"/> list, <paramref name="offset"/> of the segment and its <paramref name="length" />.</para>
        /// <para>Инициализирует новый экземпляр класса <see cref="Segment"/>, используя список <paramref name="base"/>, <paramref name="offset"/> сегмента и его <paramref name="length"/>.</para>
        /// </summary>
        /// <param name="base"><para>The reference to the original list containing the elements of this segment.</para><para>Ссылка на исходный список в котором находятся элементы этого сегмента.</para></param>
        /// <param name="offset"><para>The offset relative to the <paramref name="base"/> list from which the segment starts.</para><para>Смещение относительно списка <paramref name="base"/>, с которого начинается сегмент.</para></param>
        /// <param name="length"><para>The segment's length.</para><para>Длина сегмента.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Segment(IList<T> @base, int offset, int length)
        {
            Base = @base;
            Offset = offset;
            Length = length;
        }

        /// <summary>
        /// <para>Gets the hash code of the current <see cref="Segment"/> instance.</para>
        /// <para>Возвращает хэш-код текущего экземпляра <see cref="Segment"/>.</para>
        /// </summary>
        /// <returns></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => this.GenerateHashCode();

        /// <summary>
        /// <para>Returns a value indicating whether the current <see cref="Segment"/> is equal to another <see cref="Segment" />.</para>
        /// <para>Возвращает значение определяющее, равен ли текущий <see cref="Segment"/> другому <see cref="Segment"/>.</para>
        /// </summary>
        /// <param name="other"><para>An <see cref="Segment"/> object to compare with the current <see cref="Segment"/>.</para><para>Объект <see cref="Segment"/> для сравнения с текущим <see cref="Segment"/>.<para></para></param>
        /// <returns>
        /// <para><see langword="true"/> if the current <see cref="Segment"/> is equal to the <paramref name="other"/> parameter; otherwise, <see langword="false"/>.</para>
        /// <para><see langword="true"/>, если текущий <see cref="Segment"/> равен параметру <paramref name="other"/>, в противном случае - <see langword="false"/>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Equals(Segment<T> other) => this.EqualTo(other);

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

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
        ↪ false;

        #region IList

        /// <summary>
        /// <para>
        /// The value.
        /// </para>
        /// <para></para>
        /// </summary>
        public T this[int i]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Base[Offset + i];
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => Base[Offset + i] = value;
        }

        /// <summary>
        /// <para>Gets the number of elements contained in the <see cref="Segment"/>.</para>
        /// <para>Возвращает число элементов, содержащихся в <see cref="Segment"/>.</para>
        /// </summary>
        /// <value>
        /// <para>The number of elements contained in the <see cref="Segment"/>.</para>
        /// <para>Число элементов, содержащихся в <see cref="Segment"/>.</para>
        /// </value>
        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Length;
        }

        /// <summary>
        /// <para>Gets a value indicating whether the <see cref="Segment"/> is read-only.</para>
        /// <para>Возвращает значение, указывающее, является ли <see cref="Segment"/> доступным
        ↪ только для чтения.</para>
        /// </summary>
        /// <value>
        /// <para><see langword="true"/> if the <see cref="Segment"/> is read-only; otherwise,
        ↪ <see langword="false"/>.</para>
        /// <para>Значение <see langword="true"/>, если <see cref="Segment"/> доступен только
        ↪ для чтения, в противном случае - значение <see langword="false"/>.</para>
        /// </value>
        /// <remarks>
        /// <para>Any <see cref="Segment"/> is read-only.</para>
        /// <para>Любой <see cref="Segment"/> доступен только для чтения.</para>
        /// </remarks>
        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        /// <summary>
        /// <para>Determines the index of a specific item in the <see cref="Segment"/>.</para>
        /// <para>Определяет индекс конкретного элемента в <see cref="Segment"/>.</para>
        /// </summary>
        /// <param name="item"><para>The object to locate in the <see
        ↪ cref="Segment"/>.</para><para>Элемент для поиска в <see
        ↪ cref="Segment"/>.</para></param>
        /// <returns>
        /// <para>The index of <paramref name="item"/> if found in the segment; otherwise,
        ↪ -1.</para>
        /// <para>Индекс <paramref name="item"/>, если он найден в сегменте; в противном случае
        ↪ - значение -1.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int IndexOf(T item)
        {
            var index = Base.IndexOf(item);
            if (index >= Offset)
            {
                var actualIndex = index - Offset;
                if (actualIndex < Length)
                {
                    return actualIndex;
```

```
                }
            }
            return -1;
        }

        /// <summary>
        /// <para>Inserts an item to the <see cref="Segment"/> at the specified index.</para>
        /// <para>Вставляет элемент в <see cref="Segment"/> по указанному индексу.</para>
        /// </summary>
        /// <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>
        /// <param name="item"><para>The element to insert into the <see
        ↪  cref="Segment"/>.</para><para>Элемент, вставляемый в <see
        ↪  cref="Segment"/>.</para></param>
        /// <exception cref="NotSupportedException">
        /// <para>The <see cref="Segment"/> is read-only.</para>
        /// <para><see cref="Segment"/> доступен только для чтения.</para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, T item) => throw new NotSupportedException();

        /// <summary>
        /// <para>Removes the <see cref="Segment"/> item at the specified index.</para>
        /// <para>Удаляет элемент <see cref="Segment"/> по указанному индексу.</para>
        /// </summary>
        /// <param name="index"><para>The zero-based index of the item to
        ↪  remove.</para><para>Отсчитываемый от нуля индекс элемента для
        ↪  удаления.</para></param>
        /// <exception cref="NotSupportedException">
        /// <para>The <see cref="Segment"/> is read-only.</para>
        /// <para><see cref="Segment"/> доступен только для чтения.</para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        /// <summary>
        /// <para>Adds an item to the <see cref="Segment"/>.</para>
        /// <para>Добавляет элемент в <see cref="Segment"/>.</para>
        /// </summary>
        /// <param name="item"><para>The element to add to the <see
        ↪  cref="Segment"/>.</para><para>Элемент, добавляемый в <see
        ↪  cref="Segment"/>.</para></param>
        /// <exception cref="NotSupportedException">
        /// <para>The <see cref="Segment"/> is read-only.</para>
        /// <para><see cref="Segment"/> доступен только для чтения.</para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(T item) => throw new NotSupportedException();

        /// <summary>
        /// <para>Removes all items from the <see cref="Segment"/>.</para>
        /// <para>Удаляет все элементы из <see cref="Segment"/>.</para>
        /// </summary>
        /// <exception cref="NotSupportedException">
        /// <para>The <see cref="Segment"/> is read-only.</para>
        /// <para><see cref="Segment"/> доступен только для чтения.</para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        /// <summary>
        /// <para>Determines whether the <see cref="Segment"/> contains a specific value.</para>
        /// <para>Определяет, содержит ли <see cref="Segment"/> определенное значение.</para>
        /// </summary>
        /// <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>
        /// <returns>
        /// <para><see langword="true"/> if the value is found in the <see cref="Segment"/>;
        ↪  otherwise, <see langword="false"/>.</para>
        /// <para>Значение <see langword="true"/>, если значение находится в <see
        ↪  cref="Segment"/>; в противном случае - <see langword="false"/>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Contains(T item) => IndexOf(item) >= 0;

```

```
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

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the all segments walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    public abstract class AllSegmentsWalkerBase
    {
        /// <summary>
        /// <para>
        /// The default minimum string segment length.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly int DefaultMinimumStringSegmentLength = 2;
    }
}
```

1.23 ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the all segments walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AllSegmentsWalkerBase"/>
    public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
        where TSegment : Segment<T>
    {
        /// <summary>
        /// <para>
        /// The minimum string segment length.
        /// </para>
        /// <para></para>
        /// </summary>
        private readonly int _minimumStringSegmentLength;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="AllSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
        ↪  _minimumStringSegmentLength = minimumStringSegmentLength;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="AllSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }

        /// <summary>
        /// <para>
        /// Walks the all using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
```

```
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void WalkAll(IList<T> elements)
        {
            for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
            ↪  offset <= maxOffset; offset++)
            {
                for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                ↪  offset; length <= maxLength; length++)
                {
                    Iteration(CreateSegment(elements, offset, length));
                }
            }
        }

        /// <summary>
        /// <para>
        /// Creates the segment using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The segment</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);

        /// <summary>
        /// <para>
        /// Iterations the segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void Iteration(TSegment segment);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the all segments walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AllSegmentsWalkerBase{T, Segment{T}}"/>
    public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
    {
        /// <summary>
        /// <para>
        /// Creates the segment using the specified elements.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A segment of t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
        ↪  => new Segment<T>(elements, offset, length);
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the all segments walker extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class AllSegmentsWalkerExtensions
    {
        /// <summary>
        /// <para>
        /// Walks the all using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <param name="@string">
        /// <para>The string.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
        ↪  walker.WalkAll(@string.ToCharArray());

        /// <summary>
        /// <para>
        /// Walks the all using the specified walker.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TSegment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="walker">
        /// <para>The walker.</para>
        /// <para></para>
        /// </param>
        /// <param name="@string">
        /// <para>The string.</para>
        /// <para></para>
        /// </param>
        [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, Segment

```
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>
17          where TSegment : Segment<T>
18      {
19          /// <summary>
20          /// <para>
21          /// The default reset dictionary on each walk.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          public static readonly bool DefaultResetDictionaryOnEachWalk;
26
27          /// <summary>
28          /// <para>
29          /// The reset dictionary on each walk.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          private readonly bool _resetDictionaryOnEachWalk;
34          /// <summary>
35          /// <para>
36          /// The dictionary.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          protected IDictionary<TSegment, long> Dictionary;
41
42          /// <summary>
43          /// <para>
44          /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          /// <param name="dictionary">
49          /// <para>A dictionary.</para>
50          /// <para></para>
51          /// </param>
52          /// <param name="minimumStringSegmentLength">
53          /// <para>A minimum string segment length.</para>
54          /// <para></para>
55          /// </param>
56          /// <param name="resetDictionaryOnEachWalk">
57          /// <para>A reset dictionary on each walk.</para>
58          /// <para></para>
59          /// </param>
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪  dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
62              : base(minimumStringSegmentLength)
63          {
64              Dictionary = dictionary;
65              _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
66          }
67
68          /// <summary>
69          /// <para>
70          /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dictionary">
        /// <para>A dictionary.</para>
        /// <para></para>
        /// </param>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪  dictionary, int minimumStringSegmentLength) : this(dictionary,
        ↪  minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dictionary">
        /// <para>A dictionary.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪  dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
        ↪  DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        /// <param name="resetDictionaryOnEachWalk">
        /// <para>A reset dictionary on each walk.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪  bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
        ↪  Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
        ↪  { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪  this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
        ↪  this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
```

```
        /// Walks the all using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override void WalkAll(IList<T> elements)
        {
            if (_resetDictionaryOnEachWalk)
            {
                var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
                Dictionary = new Dictionary<TSegment, long>((int)capacity);
            }
            base.WalkAll(elements);
        }

        /// <summary>
        /// <para>
        /// Gets the segment frequency using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long GetSegmentFrequency(TSegment segment) =>
        ↪ Dictionary.GetOrDefault(segment);

        /// <summary>
        /// <para>
        /// Sets the segment frequency using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequency">
        /// <para>The frequency.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
        ↪ Dictionary[segment] = frequency;
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the dictionary based duplicate segments walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{T, Segment{T}}"/>
    public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
    ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dictionary">
        /// <para>A dictionary.</para>
        /// <para></para>
        /// </param>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        /// <param name="resetDictionaryOnEachWalk">
        /// <para>A reset dictionary on each walk.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪  dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
        ↪  base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dictionary">
        /// <para>A dictionary.</para>
        /// <para></para>
        /// </param>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪  dictionary, int minimumStringSegmentLength) : base(dictionary,
        ↪  minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="dictionary">
        /// <para>A dictionary.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪  dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
        ↪  DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        /// <param name="resetDictionaryOnEachWalk">
        /// <para>A reset dictionary on each walk.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪  bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
        ↪  resetDictionaryOnEachWalk) { }

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

```
        /// <para></para>
        /// </summary>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DictionaryBasedDuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
        ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the duplicate segments walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="AllSegmentsWalkerBase{T, TSegment}"/>
    public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
    ↪ TSegment>
        where TSegment : Segment<T>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DuplicateSegmentsWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="minimumStringSegmentLength">
        /// <para>A minimum string segment length.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪ base(minimumStringSegmentLength) { }

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

        /// <summary>
        /// <para>
        /// Iterations the segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Iteration(TSegment segment)
        {
            var frequency = GetSegmentFrequency(segment);
            if (frequency == 1)
```

```
            {
                OnDublicateFound(segment);
            }
            SetSegmentFrequency(segment, frequency + 1);
        }

        /// <summary>
        /// <para>
        /// Ons the dublicate found using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void OnDublicateFound(TSegment segment);

        /// <summary>
        /// <para>
        /// Gets the segment frequency using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract long GetSegmentFrequency(TSegment segment);

        /// <summary>
        /// <para>
        /// Sets the segment frequency using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequency">
        /// <para>The frequency.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
    }
}
```

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

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

namespace Platform.Collections.Segments.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the duplicate segments walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DuplicateSegmentsWalkerBase{T, Segment{T}}"/>
    public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
    ↪  Segment<T>>
    {
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

```
5
6  namespace Platform.Collections.Sets
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the set extensions.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class ISetExtensions
15     {
16         /// <summary>
17         /// <para>
18         /// Adds the and return void using the specified set.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         /// <typeparam name="T">
23         /// <para>The .</para>
24         /// <para></para>
25         /// </typeparam>
26         /// <param name="set">
27         /// <para>The set.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="element">
31         /// <para>The element.</para>
32         /// <para></para>
33         /// </param>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
36
37         /// <summary>
38         /// <para>
39         /// Removes the and return void using the specified set.
40         /// </para>
41         /// <para></para>
42         /// </summary>
43         /// <typeparam name="T">
44         /// <para>The .</para>
45         /// <para></para>
46         /// </typeparam>
47         /// <param name="set">
48         /// <para>The set.</para>
49         /// <para></para>
50         /// </param>
51         /// <param name="element">
52         /// <para>The element.</para>
53         /// <para></para>
54         /// </param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
           ↪  set.Remove(element);
57
58         /// <summary>
59         /// <para>
60         /// Determines whether add and return true.
61         /// </para>
62         /// <para></para>
63         /// </summary>
64         /// <typeparam name="T">
65         /// <para>The .</para>
66         /// <para></para>
67         /// </typeparam>
68         /// <param name="set">
69         /// <para>The set.</para>
70         /// <para></para>
71         /// </param>
72         /// <param name="element">
73         /// <para>The element.</para>
74         /// <para></para>
75         /// </param>
76         /// <returns>
77         /// <para>The bool</para>
78         /// <para></para>
79         /// </returns>
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
```

```
        {
            set.Add(element);
            return true;
        }

        /// <summary>
        /// <para>
        /// Determines whether add first and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            AddFirst(set, elements);
            return true;
        }

        /// <summary>
        /// <para>
        /// Adds the first using the specified set.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
        ↪  set.Add(elements[0]);

        /// <summary>
        /// <para>
        /// Determines whether add all and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            set.AddAll(elements);
            return true;
        }

        /// <summary>
        /// <para>
        /// Adds the all using the specified set.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAll<T>(this ISet<T> set, IList<T> elements)
        {
            for (var i = 0; i < elements.Count; i++)
            {
                set.Add(elements[i]);
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether add skip first and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            set.AddSkipFirst(elements);
            return true;
        }

        /// <summary>
        /// <para>
        /// Adds the skip first using the specified set.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
```

```
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
        ↪ set.AddSkipFirst(elements, 1);

        /// <summary>
        /// <para>
        /// Adds the skip first using the specified set.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <param name="skip">
        /// <para>The skip.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
        {
            for (var i = skip; i < elements.Count; i++)
            {
                set.Add(elements[i]);
            }
        }

        /// <summary>
        /// <para>
        /// Determines whether do not contains.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="set">
        /// <para>The set.</para>
        /// <para></para>
        /// </param>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
        ↪ !set.Contains(element);
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Sets
{
    /// <summary>
    /// <para>
    /// Represents the set filler.
    /// </para>
```

```
    /// <para></para>
    /// </summary>
    public class SetFiller<TElement, TReturnConstant>
    {
        /// <summary>
        /// <para>
        /// The set.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ISet<TElement> _set;
        /// <summary>
        /// <para>
        /// The return constant.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TReturnConstant _returnConstant;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SetFiller"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="set">
        /// <para>A set.</para>
        /// <para></para>
        /// </param>
        /// <param name="returnConstant">
        /// <para>A return constant.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
        {
            _set = set;
            _returnConstant = returnConstant;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SetFiller"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="set">
        /// <para>A set.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SetFiller(ISet<TElement> set) : this(set, default) { }

        /// <summary>
        /// <para>
        /// Adds the element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TElement element) => _set.Add(element);

        /// <summary>
        /// <para>
        /// Determines whether this instance add and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);

        /// <summary>
        /// <para>
        /// Determines whether this instance add first and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
        ↪  _set.AddFirstAndReturnTrue(elements);

        /// <summary>
        /// <para>
        /// Determines whether this instance add all and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAllAndReturnTrue(IList<TElement> elements) =>
        ↪  _set.AddAllAndReturnTrue(elements);

        /// <summary>
        /// <para>
        /// Determines whether this instance add skip first and return true.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
        ↪  _set.AddSkipFirstAndReturnTrue(elements);

        /// <summary>
        /// <para>
        /// Adds the and return constant using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The return constant.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element)
        {
            _set.Add(element);
            return _returnConstant;
```

```
        }

        /// <summary>
        /// <para>
        /// Adds the first and return constant using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The return constant.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
        {
            _set.AddFirst(elements);
            return _returnConstant;
        }

        /// <summary>
        /// <para>
        /// Adds the all and return constant using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The return constant.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
        {
            _set.AddAll(elements);
            return _returnConstant;
        }

        /// <summary>
        /// <para>
        /// Adds the skip first and return constant using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The return constant.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
        {
            _set.AddSkipFirst(elements);
            return _returnConstant;
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    /// <summary>
    /// <para>
    /// Represents the default stack.
    /// </para>
```

```
    /// <para></para>
    /// </summary>
    /// <seealso cref="Stack{TElement}"/>
    /// <seealso cref="IStack{TElement}"/>
    public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
    {
        /// <summary>
        /// <para>
        /// Gets the is empty value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Count <= 0;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    /// <summary>
    /// <para>
    /// Defines the stack.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface IStack<TElement>
    {
        /// <summary>
        /// <para>
        /// Gets the is empty value.
        /// </para>
        /// <para></para>
        /// </summary>
        bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>
        /// Pushes the element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Push(TElement element);

        /// <summary>
        /// <para>
        /// Pops this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Pop();

        /// <summary>
        /// <para>
        /// Peeks this instance.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <returns>
        /// <para>The element</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Peek();
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    /// <summary>
    /// <para>
    /// Represents the stack extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class IStackExtensions
    {
        /// <summary>
        /// <para>
        /// Clears the stack.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Clear<T>(this IStack<T> stack)
        {
            while (!stack.IsEmpty)
            {
                _ = stack.Pop();
            }
        }

        /// <summary>
        /// <para>
        /// Pops the or default using the specified stack.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪  stack.Pop();

        /// <summary>
        /// <para>
        /// Peeks the or default using the specified stack.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
```

```
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪  stack.Peek();
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Collections.Stacks
{
    /// <summary>
    /// <para>
    /// Defines the stack factory.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IFactory{IStack{TElement}}"/>
    public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
    {
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    /// <summary>
    /// <para>
    /// Represents the stack extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class StackExtensions
    {
        /// <summary>
        /// <para>
        /// Pops the or default using the specified stack.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
        ↪  default;

        /// <summary>
        /// <para>
        /// Peeks the or default using the specified stack.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <typeparam name="T">
        /// <para>The .</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="stack">
        /// <para>The stack.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
        ↪  : default;
    }
}
```

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

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

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

namespace Platform.Collections
{
    /// <summary>
    /// <para>
    /// Represents the string extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class StringExtensions
    {
        /// <summary>
        /// <para>
        /// Capitalizes the first letter using the specified string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@string">
        /// <para>The string.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string CapitalizeFirstLetter(this string @string)
        {
            if (string.IsNullOrWhiteSpace(@string))
            {
                return @string;
            }
            var chars = @string.ToCharArray();
            for (var i = 0; i < chars.Length; i++)
            {
                var category = char.GetUnicodeCategory(chars[i]);
                if (category == UnicodeCategory.UppercaseLetter)
                {
                    return @string;
                }
                if (category == UnicodeCategory.LowercaseLetter)
                {
                    chars[i] = char.ToUpper(chars[i]);
                    return new string(chars);
                }
            }
            return @string;
        }

        /// <summary>
        /// <para>
        /// Truncates the string.
        /// </para>
        /// <para></para>
        /// </summary>
```

```
        /// <param name="@string">
        /// <para>The string.</para>
        /// <para></para>
        /// </param>
        /// <param name="maxLength">
        /// <para>The max length.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Truncate(this string @string, int maxLength) =>
        ↪  string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
        ↪  Math.Min(@string.Length, maxLength));

        /// <summary>
        /// <para>
        /// Trims the single using the specified string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@string">
        /// <para>The string.</para>
        /// <para></para>
        /// </param>
        /// <param name="charToTrim">
        /// <para>The char to trim.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string TrimSingle(this string @string, char charToTrim)
        {
            if (!string.IsNullOrEmpty(@string))
            {
                if (@string.Length == 1)
                {
                    if (@string[0] == charToTrim)
                    {
                        return "";
                    }
                    else
                    {
                        return @string;
                    }
                }
                else
                {
                    var left = 0;
                    var right = @string.Length - 1;
                    if (@string[left] == charToTrim)
                    {
                        left++;
                    }
                    if (@string[right] == charToTrim)
                    {
                        right--;
                    }
                    return @string.Substring(left, right - left + 1);
                }
            }
            else
            {
                return @string;
            }
        }
    }
}
```

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

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;

// ReSharper disable ForCanBeConvertedToForeach
```

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

namespace Platform.Collections.Trees
{
    /// <summary>
    /// <para>
    /// Represents the node.
    /// </para>
    /// <para></para>
    /// </summary>
    public class Node
    {
        /// <summary>
        /// <para>
        /// The child nodes.
        /// </para>
        /// <para></para>
        /// </summary>
        private Dictionary<object, Node> _childNodes;

        /// <summary>
        /// <para>
        /// Gets or sets the value value.
        /// </para>
        /// <para></para>
        /// </summary>
        public object Value
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets the child nodes value.
        /// </para>
        /// <para></para>
        /// </summary>
        public Dictionary<object, Node> ChildNodes
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
        }

        /// <summary>
        /// <para>
        /// The key.
        /// </para>
        /// <para></para>
        /// </summary>
        public Node this[object key]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => GetChild(key) ?? AddChild(key);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => SetChildValue(value, key);
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Node"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node(object value) => Value = value;

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

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node() : this(null) { }

        /// <summary>
        /// <para>
        /// Determines whether this instance contains child.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="keys">
        /// <para>The keys.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool ContainsChild(params object[] keys) => GetChild(keys) != null;

        /// <summary>
        /// <para>
        /// Gets the child using the specified keys.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="keys">
        /// <para>The keys.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The node.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node GetChild(params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node.ChildNodes.TryGetValue(keys[i], out node);
                if (node == null)
                {
                    return null;
                }
            }
            return node;
        }

        /// <summary>
        /// <para>
        /// Gets the child value using the specified keys.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="keys">
        /// <para>The keys.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The object</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;

        /// <summary>
        /// <para>
        /// Adds the child using the specified key.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The node</para>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key) => AddChild(key, new Node(null));

        /// <summary>
        /// <para>
        /// Adds the child using the specified key.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </param>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The node</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, object value) => AddChild(key, new Node(value));

        /// <summary>
        /// <para>
        /// Adds the child using the specified key.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </param>
        /// <param name="child">
        /// <para>The child.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The child.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, Node child)
        {
            ChildNodes.Add(key, child);
            return child;
        }

        /// <summary>
        /// <para>
        /// Sets the child using the specified keys.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="keys">
        /// <para>The keys.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The node</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(params object[] keys) => SetChildValue(null, keys);

        /// <summary>
        /// <para>
        /// Sets the child using the specified key.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="key">
        /// <para>The key.</para>
        /// <para></para>
        /// </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>
```

```
    /// </summary>
    public class ArrayTests
    {
        /// <summary>
        /// <para>
        /// Tests that get element test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void GetElementTest()
        {
            var nullArray = (int[])null;
            Assert.Equal(0, nullArray.GetElementOrDefault(1));
            Assert.False(nullArray.TryGetElement(1, out int element));
            Assert.Equal(0, element);
            var array = new int[] { 1, 2, 3 };
            Assert.Equal(3, array.GetElementOrDefault(2));
            Assert.True(array.TryGetElement(2, out element));
            Assert.Equal(3, element);
            Assert.Equal(0, array.GetElementOrDefault(10));
            Assert.False(array.TryGetElement(10, out element));
            Assert.Equal(0, element);
        }
    }
}
```

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

```
using System;
using System.Collections;
using Xunit;
using Platform.Random;

namespace Platform.Collections.Tests
{
    /// <summary>
    /// <para>
    /// Represents the bit string tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class BitStringTests
    {
        /// <summary>
        /// <para>
        /// Tests that bit get set test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void BitGetSetTest()
        {
            const int n = 250;
            var bitArray = new BitArray(n);
            var bitString = new BitString(n);
            for (var i = 0; i < n; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                bitArray.Set(i, value);
                bitString.Set(i, value);
                Assert.Equal(value, bitArray.Get(i));
                Assert.Equal(value, bitString.Get(i));
            }
        }

        /// <summary>
        /// <para>
        /// Tests that bit vector not test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void BitVectorNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorNot();
                w.Not();
            });
```

```
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);
```

```
            w.And(v);
        });
    }

    /// <summary>
    /// <para>
    /// Tests that bit vector or test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void BitVectorOrTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.VectorOr(y);
            w.Or(v);
        });
    }

    /// <summary>
    /// <para>
    /// Tests that bit parallel or test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void BitParallelOrTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelOr(y);
            w.Or(v);
        });
    }

    /// <summary>
    /// <para>
    /// Tests that bit parallel vector or test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void BitParallelVectorOrTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.ParallelVectorOr(y);
            w.Or(v);
        });
    }

    /// <summary>
    /// <para>
    /// Tests that bit vector xor test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void BitVectorXorTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
        {
            x.VectorXor(y);
            w.Xor(v);
        });
    }

    /// <summary>
    /// <para>
    /// Tests that bit parallel xor test.
    /// </para>
    /// <para></para>
    /// </summary>
    [Fact]
    public static void BitParallelXorTest()
    {
        TestToOperationsWithSameMeaning((x, y, w, v) =>
```

```
            {
                x.ParallelXor(y);
                w.Xor(v);
            });
        }

        /// <summary>
        /// <para>
        /// Tests that bit parallel vector xor test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void BitParallelVectorXorTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelVectorXor(y);
                w.Xor(v);
            });
        }

        /// <summary>
        /// <para>
        /// Tests the to operations with same meaning using the specified test.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="test">
        /// <para>The test.</para>
        /// <para></para>
        /// </param>
        private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
        ↪  BitString, BitString> test)
        {
            const int n = 5654;
            var x = new BitString(n);
            var y = new BitString(n);
            while (x.Equals(y))
            {
                x.SetRandomBits();
                y.SetRandomBits();
            }
            var w = new BitString(x);
            var v = new BitString(y);
            Assert.False(x.Equals(y));
            Assert.False(w.Equals(v));
            Assert.True(x.Equals(w));
            Assert.True(y.Equals(v));
            test(x, y, w, v);
            Assert.True(x.Equals(w));
        }
    }
}
```

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

```
using Xunit;
using Platform.Collections.Segments;

namespace Platform.Collections.Tests
{
    /// <summary>
    /// <para>
    /// Represents the chars segment tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class CharsSegmentTests
    {
        /// <summary>
        /// <para>
        /// Tests that get hash code equals test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void GetHashCodeEqualsTest()
        {
```

```
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
            var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
            Assert.Equal(firstHashCode, secondHashCode);
        }

        /// <summary>
        /// <para>
        /// Tests that equals test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void EqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var first = new CharSegment(testArray, 0, 4);
            var second = new CharSegment(testArray, 5, 4);
            Assert.True(first.Equals(second));
        }
    }
}
```

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

```
using System.Collections.Generic;
using Xunit;
using Platform.Collections.Lists;


namespace Platform.Collections.Tests
{
    /// <summary>
    /// <para>
    /// Represents the list tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public class ListTests
    {
        /// <summary>
        /// <para>
        /// Tests that get element test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void GetElementTest()
        {
            var nullList = (IList<int>)null;
            Assert.Equal(0, nullList.GetElementOrDefault(1));
            Assert.False(nullList.TryGetElement(1, out int element));
            Assert.Equal(0, element);
            var list = new List<int>() { 1, 2, 3 };
            Assert.Equal(3, list.GetElementOrDefault(2));
            Assert.True(list.TryGetElement(2, out element));
            Assert.Equal(3, element);
            Assert.Equal(0, list.GetElementOrDefault(10));
            Assert.False(list.TryGetElement(10, out element));
            Assert.Equal(0, element);
        }
    }
}
```

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

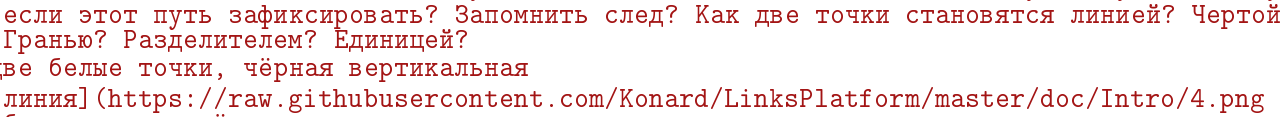
```
using Xunit;

namespace Platform.Collections.Tests
{
    /// <summary>
    /// <para>
    /// Represents the string tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class StringTests
    {
```

```
        /// <summary>
        /// <para>
        /// Tests that capitalize first letter test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CapitalizeFirstLetterTest()
        {
            Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
            Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
            Assert.Equal("  Hello", "  hello".CapitalizeFirstLetter());
        }

        /// <summary>
        /// <para>
        /// Tests that trim single test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void TrimSingleTest()
        {
            Assert.Equal("", "''".TrimSingle('\''));
            Assert.Equal("", "''".TrimSingle('\''));
            Assert.Equal("hello", "'hello'".TrimSingle('\''));
            Assert.Equal("hello", "hello'".TrimSingle('\''));
            Assert.Equal("hello", "'hello".TrimSingle('\''));
        }
    }
}
```

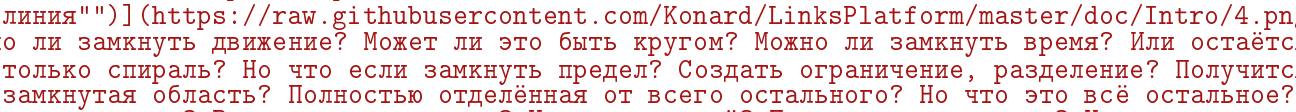
1.44 ./csharp/Platform.Collections.Tests/WalkersTests.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using Platform.Collections.Segments;
using Platform.Collections.Segments.Walkers;
using Platform.Collections.Trees;
using Xunit;
using Xunit.Abstractions;


namespace Platform.Collections.Tests
{
    /// <summary>
    /// <para>
    /// Represents the all repeating substrings in string.
    /// </para>
    /// <para></para>
    /// </summary>
    public class AllRepeatingSubstringsInString
    {
        /// <summary>
        /// <para>
        /// The elfen lied.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly string elfen_lied = @"Nacht im Dorf der Wächter rief: Elfe! Ein
        ↪  ganz kleines Elfchen im Walde schlief wohl um die Elfe! Und meint, es rief ihm aus
        ↪  dem Tal bei seinem Namen die Nachtigall, oder Silpelit hätt' ihm gerufen.
Reibt sich der Elf' die Augen aus, begibt sich vor sein Schneckenhaus und ist als wie ein
↪  trunken Mann, sein Schläflein war nicht voll getan, und humpelt also tippe tapp durch's
↪  Haselholz in's Tal hinab, schlupft an der Mauer hin so dicht, da sitzt der Glühwurm Licht an
↪  Licht.
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!""
Pfui, stößt den Kopf an harten Stein! Elfe, gelt, du hast genug? Gukuk!";
        /// <summary>
        /// <para>
        /// The example text.
        /// </para>
        /// <para></para>
        /// </summary>
        private static readonly string _exampleText =
    @"([english version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
```

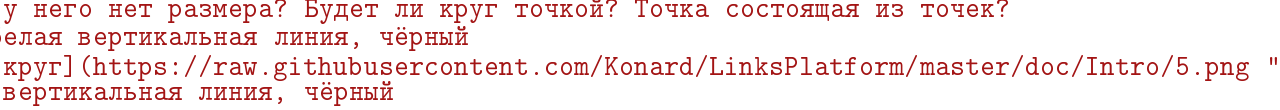
Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там где есть место для нового начала? Разве пустота это не характеристика пространства? Пространство это то, что можно чем-то наполнить?



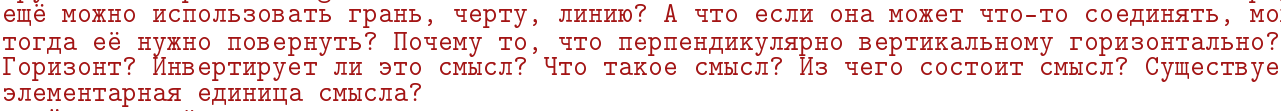
Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?



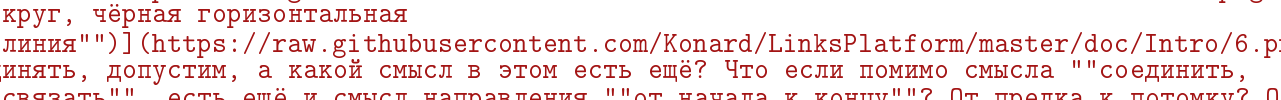
А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть так? Инверсия? Отражение? Сумма?



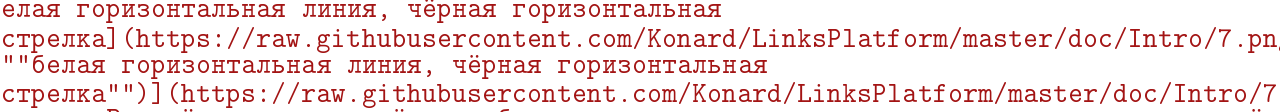
А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой? Гранью? Разделителем? Единицей?



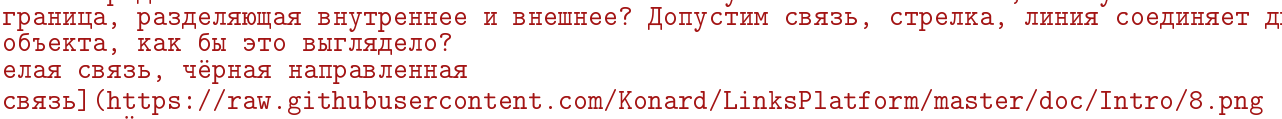
Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что можно делить? В каком направлении? 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?



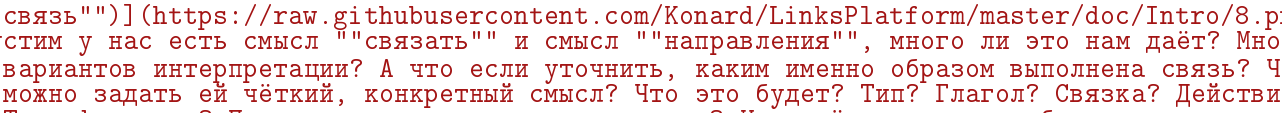
Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально? Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли элементарная единица смысла?



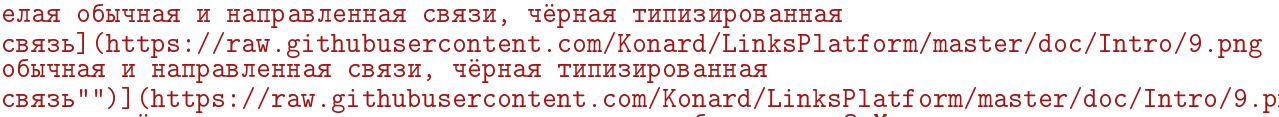
Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить, связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От родителя к ребёнку? От общего к частному?



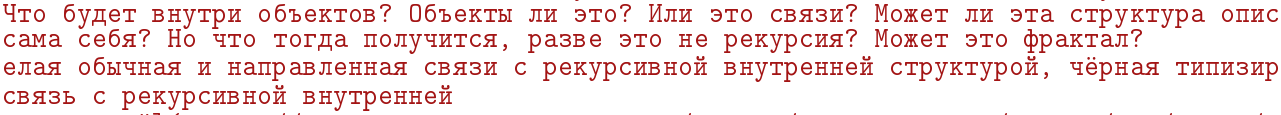
Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?



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?



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?



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


        /// <summary>
        /// <para>
        /// The exam тple text.
        /// </para>
        /// <para></para>
        /// </summary>
        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.";

        /// <summary>
        /// <para>
        /// Tests that console tests.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public void ConsoleTests()
        {
            string text = elfen_lied;

            var iterationsCounter = new IterationsCounter();
            iterationsCounter.WalkAll(text);
            var result = iterationsCounter.IterationsCount;
            Console.WriteLine($"TextLength: {text.Length}. Iterations: {result}.");

            {
                var start = new Stopwatch();
                start.Start();

                var walker = new Walker4();
                walker.WalkAll(text);

                //foreach (var (key, value) in walker.PublicDictionary)
                //{
                //    Console.WriteLine($"{key} {value}");
                //}

                start.Stop();
                Console.WriteLine($"{start.ElapsedMilliseconds}ms");
            }


            {
                var start = new Stopwatch();
                start.Start();

                var walker = new Walker2();
                walker.WalkAll(text);

                //foreach (var (key, value) in walker._cache)
                //{
                //    Console.WriteLine($"{key} {value}");
```

```
                //}

                start.Stop();
                Console.WriteLine($"{start.ElapsedMilliseconds}ms");
            }

            {
                var start = new Stopwatch();
                start.Start();

                var walker = new Walker1();
                walker.WalkAll(text);

                start.Stop();
                Console.WriteLine($"{start.ElapsedMilliseconds}ms");
            }
        }
    }

    /// <summary>
    /// <para>
    /// Represents the console printed dublicate walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="DuplicateSegmentsWalkerBase{char, CharSegment}"/>
    public abstract class ConsolePrintedDublicateWalkerBase : DuplicateSegmentsWalkerBase<char,
    ↪  CharSegment>
    {
        //protected override void OnDublicateFound(CharSegment segment) =>
        ↪  Console.WriteLine(segment);

        /// <summary>
        /// <para>
        /// Creates the segment using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The char segment</para>
        /// <para></para>
        /// </returns>
        protected override CharSegment CreateSegment(IList<char> elements, int offset, int
        ↪  length) => new CharSegment(elements, offset, length);
    }

    /// <summary>
    /// <para>
    /// Represents the walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ConsolePrintedDublicateWalkerBase"/>
    public class Walker1 : ConsolePrintedDublicateWalkerBase
    {
        /// <summary>
        /// <para>
        /// The root node.
        /// </para>
        /// <para></para>
        /// </summary>
        private Node _rootNode;
        /// <summary>
        /// <para>
        /// The current node.
        /// </para>
```

```
        /// <para></para>
        /// </summary>
        private Node _currentNode;

        /// <summary>
        /// <para>
        /// Walks the all using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        public override void WalkAll(IList<char> elements)
        {
            _rootNode = new Node();

            base.WalkAll(elements);

            Console.WriteLine(_rootNode.Value);
        }

        /// <summary>
        /// <para>
        /// Ons the dublicate found using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        protected override void OnDublicateFound(CharSegment segment)
        {

        }

        /// <summary>
        /// <para>
        /// Gets the segment frequency using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The long</para>
        /// <para></para>
        /// </returns>
        protected override long GetSegmentFrequency(CharSegment segment)
        {
            for (int i = 0; i < segment.Length; i++)
            {
                var element = segment[i];

                _currentNode = _currentNode[element];
            }

            if (_currentNode.Value is int)
            {
                return (int)_currentNode.Value;
            }
            else
            {
                return 0;
            }
        }

        /// <summary>
        /// <para>
        /// Sets the segment frequency using the specified segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="frequency">
        /// <para>The frequency.</para>
        /// <para></para>
        /// </param>
        protected override void SetSegmentFrequency(CharSegment segment, long frequency) =>
        ↪  _currentNode.Value = frequency;

        /// <summary>
        /// <para>
        /// Iterations the segment.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="segment">
        /// <para>The segment.</para>
        /// <para></para>
        /// </param>
        protected override void Iteration(CharSegment segment)
        {
            _currentNode = _rootNode;

            base.Iteration(segment);
        }
    }

    // Too much memory, but fast
    /// <summary>
    /// <para>
    /// Represents the walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ConsolePrintedDublicateWalkerBase"/>
    public class Walker2 : ConsolePrintedDublicateWalkerBase
    {
        /// <summary>
        /// <para>
        /// The cache.
        /// </para>
        /// <para></para>
        /// </summary>
        public Dictionary<string, long> _cache;
        /// <summary>
        /// <para>
        /// The current key.
        /// </para>
        /// <para></para>
        /// </summary>
        private string _currentKey;
        /// <summary>
        /// <para>
        /// The total duplicates.
        /// </para>
        /// <para></para>
        /// </summary>
        private int _totalDuplicates;

        /// <summary>
        /// <para>
        /// Walks the all using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        public override void WalkAll(IList<char> elements)
        {
            _cache = new Dictionary<string, long>();

            base.WalkAll(elements);

            Console.WriteLine($"Unique string segments: {_cache.Count}. Total duplicates:
            ↪  {_totalDuplicates}");
        }
```

```
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 OnDublicateFound(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) =>
          ↪  _cache.GetOrDefault(_currentKey);
379
380         /// <summary>
381         /// <para>
382         /// Sets the segment frequency using the specified segment.
383         /// </para>
384         /// <para></para>
385         /// </summary>
386         /// <param name="segment">
387         /// <para>The segment.</para>
388         /// <para></para>
389         /// </param>
390         /// <param name="frequency">
391         /// <para>The frequency.</para>
392         /// <para></para>
393         /// </param>
394         protected override void SetSegmentFrequency(CharSegment segment, long frequency) =>
          ↪  _cache[_currentKey] = frequency;
395
396         /// <summary>
397         /// <para>
398         /// Iterations the segment.
399         /// </para>
400         /// <para></para>
401         /// </summary>
402         /// <param name="segment">
403         /// <para>The segment.</para>
404         /// <para></para>
405         /// </param>
406         protected override void Iteration(CharSegment segment)
407         {
408             _currentKey = segment;
409
410             base.Iteration(segment);
411         }
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>
```

```
        /// <para>
        /// Gets the public dictionary value.
        /// </para>
        /// <para></para>
        /// </summary>
        public IDictionary<CharSegment, long> PublicDictionary
        {
            get
            {
                return Dictionary;
            }
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Walker4"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        public Walker4()
            : base(DefaultMinimumStringSegmentLength, resetDictionaryOnEachWalk: true)
        {
        }

        /// <summary>
        /// <para>
        /// The total duplicates.
        /// </para>
        /// <para></para>
        /// </summary>
        private int _totalDuplicates;

        /// <summary>
        /// <para>
        /// Walks the all using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        public override void WalkAll(IList<char> elements)
        {
            _totalDuplicates = 0;

            base.WalkAll(elements);
            Console.WriteLine($"Unique string segments: {Dictionary.Count}. Total duplicates:
            ↪  {_totalDuplicates}.");
        }

        /// <summary>
        /// <para>
        /// Creates the segment using the specified elements.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elements">
        /// <para>The elements.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <param name="length">
        /// <para>The length.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The char segment</para>
        /// <para></para>
        /// </returns>
        protected override CharSegment CreateSegment(IList<char> elements, int offset, int
        ↪  length) => new CharSegment(elements, offset, length);

        /// <summary>
        /// <para>
```

```
500             /// Ons the dublicate 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 OnDublicateFound(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