LinksPlatform's Platform.Collections Class Library

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

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

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

namespace Platform.Collections.Arrays
{
    public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
    {
        protected readonly TReturnConstant _returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
        ↪  base(array, offset) => _returnConstant = returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
        ↪  returnConstant) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element) =>
        ↪  _array.AddAndReturnConstant(ref _position, element, _returnConstant);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements) =>
        ↪  _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements) =>
        ↪  _array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements) =>
        ↪  _array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
    }
}
```

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

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

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

namespace Platform.Collections.Arrays
{
    public class ArrayFiller<TElement>
    {
        protected readonly TElement[] _array;
        protected long _position;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array, long offset)
        {
            _array = array;
            _position = offset;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayFiller(TElement[] array) : this(array, 0) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TElement element) => _array[_position++] = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
        ↪  _position, element, true);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
        ↪  _array.AddFirstAndReturnConstant(ref _position, elements, true);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAllAndReturnTrue(IList<TElement> elements) =>
        ↪  _array.AddAllAndReturnConstant(ref _position, elements, true);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
36        public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
   ↪   _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
37      }
38  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   namespace Platform.Collections.Arrays
4   {
5       public static class ArrayPool
6       {
7           public static readonly int DefaultSizesAmount = 512;
8           public static readonly int DefaultMaxArraysPerSize = 32;
9
10          /// <summary>
11          /// <para>Allocation of an array of a specified size from the array pool.</para>
12          /// <para>Выделение массива указанного размера из пула массивов.</para>
13          /// </summary>
14          /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
   ↪   массива.</para></typeparam>
15          /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
   ↪   массива.</para></param>
16          /// <returns>
17          /// <para>The array from a pool of arrays.</para>
18          /// <para>Массив из пулла массивов.</para>
19          /// </returns>
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
22
23          /// <summary>
24          /// <para>Freeing an array into an array pool.</para>
25          /// <para>Освобождение массива в пул массивов.</para>
26          /// </summary>
27          /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
   ↪   массива.</para></typeparam>
28          /// <param name="array"><para>The array to be freed into the pull.</para><para>Массив
   ↪   который нужно освобоить в пулл.</para></param>
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
31      }
32  }
```

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   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Collections.Arrays
10  {
11      /// <remarks>
12      /// Original idea from
   ↪   http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
13      /// </remarks>
14      public class ArrayPool<T>
15      {
16          // May be use Default class for that later.
17          [ThreadStatic]
18          private static ArrayPool<T> _threadInstance;
19          internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
   ↪   ArrayPool<T>());
20
21          private readonly int _maxArraysPerSize;
22          private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
   ↪   Stack<T[]>>(ArrayPool.DefaultSizesAmount);
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
26
27          [MethodImpl(MethodImplOptions.AggressiveInlining)]
28          public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
32
```

```
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public Disposable<T[]> Resize(Disposable<T[]> source, long size)
35          {
36              var destination = AllocateDisposable(size);
37              T[] sourceArray = source;
38              if (!sourceArray.IsNullOrEmpty())
39              {
40                  T[] destinationArray = destination;
41                  Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
                    ↪ sourceArray.LongLength);
42                  source.Dispose();
43              }
44              return destination;
45          }
46
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public virtual void Clear() => _pool.Clear();
49
50          [MethodImpl(MethodImplOptions.AggressiveInlining)]
51          public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
            ↪ _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
52
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          public virtual void Free(T[] array)
55          {
56              if (array.IsNullOrEmpty())
57              {
58                  return;
59              }
60              var stack = _pool.GetOrAdd(array.LongLength, size => new
                ↪ Stack<T[]>(_maxArraysPerSize));
61              if (stack.Count == _maxArraysPerSize) // Stack is full
62              {
63                  return;
64              }
65              stack.Push(array);
66          }
67      }
68  }
```

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

```
1   using System.Runtime.CompilerServices;
2   using Platform.Collections.Segments;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Collections.Arrays
7   {
8       public class ArrayString<T> : Segment<T>
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public ArrayString(int length) : base(new T[length], 0, length) { }
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public ArrayString(T[] array) : base(array, 0, array.Length) { }
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public ArrayString(T[] array, int length) : base(array, 0, length) { }
18      }
19  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   namespace Platform.Collections.Arrays
4   {
5       public static unsafe class CharArrayExtensions
6       {
7           /// <summary>
8           /// <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>
9           /// <para>Генерирует хэш-код сегмента массива с указанным смещением и длиной. Хэш-код
                ↪ генерируется на основе значений элементов массива входящих в указанный
                ↪ сегмент.</para>
10          /// </summary>
11          /// <param name="array"><para>The array to hash.</para><para>Массив для
                ↪ хеширования.</para></param>
```

```csharp
        /// <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>
        /// <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>
        /// <returns>
        /// <para>The hash code of the segment in the array.</para>
        /// <para>Хэш-код сегмента в массиве.</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 char[] array, int offset, int length)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            fixed (char* arrayPointer = &array[offset])
            {
                for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
                  < last; charPointer++)
                {
                    hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
                }
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        /// <summary>
        /// <para>Checks if all elements of two lists are equal.</para>
        /// <para>Проверяет равны ли все элементы двух списков.</para>
        /// </summary>
        /// <param name="left"><para>The first compared array.</para><para>Первый массив для
        ///   сравнения.</para></param>
        /// <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>
        /// <param name="length"><para>The number of checked elements.</para><para>Количество
        ///   проверяемых элементов.</para></param>
        /// <param name="right"><para>The second compared array.</para><para>Второй массив для
        ///   сравнения.</para></param>
        /// <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>
        /// <returns>
        /// <para>True if the segments of the passed arrays are equal to each other otherwise
        ///   false.</para>
        /// <para>True, если сегменты переданных массивов равны друг другу, иначе же
        ///   false.</para>
        /// </returns>
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
        ///   a3eda37d3d4cd10/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;
                }
            }
        }
```

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

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        public static class GenericArrayExtensions
8        {
9            /// <summary>
10           /// <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>
11           /// <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>
12           /// </summary>
13           /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
   ↪       массива.</para></typeparam>
14           /// <param name="array"><para>Array that will participate in
   ↪       verification.</para><para>Массив который будет учавствовать в
   ↪       проверке.</para></param>
15           /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
   ↪       сравнения.</para></param>
16           /// <returns><para>Array element or default value.</para><para>Элемент массива или же
   ↪       значение по умолчанию.</para></returns>
17           [MethodImpl(MethodImplOptions.AggressiveInlining)]
18           public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
   ↪       array.Length > index ? array[index] : default;
19
20           /// <summary>
21           /// <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>
```

```csharp
        /// <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>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///   массива.</para></typeparam>
        /// <param name="array"><para>Array that will participate in
        ///   verification.</para><para>Массив который будет учавствовать в
        ///   проверке.</para></param>
        /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
        ///   для сравнения.</para></param>
        /// <returns><para>Array element or default value.</para><para>Элемент массива или же
        ///   значение по умолчанию.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
            array.LongLength > index ? array[index] : default;

        /// <summary>
        /// <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 true.</para>
        /// <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 true.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///   массива.</para></typeparam>
        /// <param name="array"><para>Array that will participate in
        ///   verification.</para><para>Массив который будет учавствовать в
        ///   проверке.</para></param>
        /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
        ///   сравнения.</para></param>
        /// <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>
        /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
        ///   в противном случае false</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool TryGetElement<T>(this T[] array, int index, out T element)
        {
            if (array != null && array.Length > index)
            {
                element = array[index];
                return true;
            }
            else
            {
                element = default;
                return false;
            }
        }

        /// <summary>
        /// <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 true.</para>
        /// <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 true.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///   массива.</para></typeparam>
        /// <param name="array"><para>Array that will participate in
        ///   verification.</para><para>Массив который будет учавствовать в
        ///   проверке.</para></param>
        /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
        ///   для сравнения.</para></param>
        /// <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>
        /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
        ///   в противном случае false</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool TryGetElement<T>(this T[] array, long index, out T element)
```

```csharp
66          {
67              if (array != null && array.LongLength > index)
68              {
69                  element = array[index];
70                  return true;
71              }
72              else
73              {
74                  element = default;
75                  return false;
76              }
77          }
78
79          /// <summary>
80          /// <para>Copying  of elements from one array to another array.</para>
81          /// <para>Копирует  элементы из одного массива в другой массив.</para>
82          /// </summary>
83          /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
     массива.</para></typeparam>
84          /// <param name="array"><para>The array  to copy.</para><para>Массив который необходимо
     скопировать.</para></param>
85          /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
87          public static T[] Clone<T>(this T[] array)
88          {
89              var copy = new T[array.LongLength];
90              Array.Copy(array, 0L, copy, 0L, array.LongLength);
91              return copy;
92          }
93
94          /// <summary>
95          /// <para>Shifts all the elements of the array by one position to the right.</para>
96          /// <para>Сдвигает вправо все элементы массива на одну позицию.</para>
97          /// </summary>
98          /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
     массива.</para></typeparam>
99          /// <param name="array"><para>The array to copy from.</para><para>Массив для
     копирования.</para></param>
100          /// <returns>
101          /// <para>Array with a shift of elements by one position.</para>
102          /// <para>Массив со сдвигом элементов на одну позицию.</para>
103          /// </returns>
104          [MethodImpl(MethodImplOptions.AggressiveInlining)]
105          public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
106
107          /// <summary>
108          /// <para>Shifts all elements of the array to the right by the specified number of
     elements.</para>
109          /// <para>Сдвигает вправо все элементы массива на указанное количество элементов.</para>
110          /// </summary>
111          /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
     массива.</para></typeparam>
112          /// <param name="array"><para>The array to copy from.</para><para>Массив для
     копирования.</para></param>
113          /// <param name="skip"><para>The number of items to shift.</para><para>Количество
     сдвигаемых элементов.</para></param>
114          /// <returns>
115          /// <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>
116          /// <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>
117          /// </returns>
118          [MethodImpl(MethodImplOptions.AggressiveInlining)]
119          public static IList<T> ShiftRight<T>(this T[] array, long shift)
120          {
121              if (shift < 0)
122              {
123                  throw new NotImplementedException();
124              }
125              if (shift == 0)
126              {
127                  return array.Clone<T>();
128              }
```

```csharp
                else
                {
                    var restrictions = new T[array.LongLength + shift];
                    Array.Copy(array, 0L, restrictions, shift, array.LongLength);
                    return restrictions;
                }
        }

        /// <summary>
        /// <para>Adding in array the passed element at the specified position and increments
        ///     position value by one.</para>
        /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
        ///     значение position на единицу.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
        ///     массива.<para></typeparam>
        /// <param name="array"><para>The array to add the element to.</para><para>Массив в
        ///     который необходимо добавить элемент.</para></param>
        /// <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>
        /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
        ///     который нужно добавить в массив.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Add<T>(this T[] array, ref int position, T element) =>
            array[position++] = element;

        /// <summary>
        /// <para>Adding in array the passed element at the specified position and increments
        ///     position value by one.</para>
        /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
        ///     значение position на единицу.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
        ///     массива.<para></typeparam>
        /// <param name="array"><para>The array to add the element to.</para><para>Массив в
        ///     который необходимо добавить элемент.</para></param>
        /// <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>
        /// <param name="element"><para>The element to add to the array</para><para>Элемент
        ///     который необходимо добавить в массив.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Add<T>(this T[] array, ref long position, T element) =>
            array[position++] = element;

        /// <summary>
        /// <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>
        /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
        ///     значение position на единицу и возвращает значение переданной константы.</para>
        /// </summary>
        /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
        ///     массива.</para></typeparam>
        /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
        ///     возвращаемой константы.</para></typeparam>
        /// <param name="array"><para>The array to add the element to.</para><para>Массив в
        ///     который необходимо добавить элемент.</para></param>
        /// <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>
        /// <param name="element"><para>The element to add to the array.</para><para>Элемент
        ///     который необходимо добавить в массив.</para></param>
        /// <param name="returnConstant"><para>The constant value that will be
        ///     returned.</para><para>Значение константы, которое будет возвращено.</para></param>
        /// <returns>
        /// <para>The constant value passed as an argument.</para>
        /// <para>Значение константы, переданное в качестве аргумента.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
            TElement[] array, ref long position, TElement element, TReturnConstant
            returnConstant)
        {
            array.Add(ref position, element);
            return returnConstant;
```

```csharp
178             }
179
180             /// <summary>
181             /// <para>Adds the first element from the passed collection to the array, at the
        ↪  specified position and increments position value by one.</para>
182             /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
        ↪  позицию и увеличивает значение position на единицу.</para>
183             /// </summary>
184             /// <typeparam name="T"><para>Array element type.</para><para>Тип элементов
        ↪  массива.</para></typeparam>
185             /// <param name="array"><para>The array to add the element to.</para><para>Массив в
        ↪  который необходимо добавить элемент.</para></param>
186             /// <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>
187             /// <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>
188             [MethodImpl(MethodImplOptions.AggressiveInlining)]
189             public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
        ↪  array[position++] = elements[0];
190
191             /// <summary>
192             /// <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>
193             /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
        ↪  позицию, увеличивает значение position на единицу и возвращает значение переданной
        ↪  константы.</para>
194             /// </summary>
195             /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
        ↪  массива.</para></typeparam>
196             /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
        ↪  возвращаемой константы.</para></typeparam>
197             /// <param name="array"><para>The array to add the element to.</para><para>Массив в
        ↪  который необходимо добавить элемент.</para></param>
198             /// <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>
199             /// <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>
200             /// <param name="returnConstant"><para>The constant value that will be
        ↪  returned.</para><para>Значение константы, которое будет возвращено.</para></param>
201             /// <returns>
202             /// <para>The constant value passed as an argument.</para>
203             /// <para>Значение константы, переданное в качестве аргумента.</para>
204             /// </returns>
205             [MethodImpl(MethodImplOptions.AggressiveInlining)]
206             public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
        ↪  TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
        ↪  returnConstant)
207             {
208                 array.AddFirst(ref position, elements);
209                 return returnConstant;
210             }
211
212             /// <summary>
213             /// <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>
214             /// <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>
215             /// </summary>
216             /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
        ↪  массива.</para></typeparam>
217             /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
        ↪  возвращаемой константы.</para></typeparam>
218             /// <param name="array"><para>The array to add the element to.</para><para>Массив в
        ↪  который необходимо добавить элементы.</para></param>
219             /// <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>
```

```csharp
220      /// <param name="elements"><para>List, whose elements will be added to the
     ↪    array.</para><para>Список, элементы которого будут добавленны в
     ↪    массив.</para></param>
221      /// <param name="returnConstant"><para>The constant value that will be
     ↪    returned.</para><para>Значение константы, которое будет возвращено.</para></param>
222      /// <returns>
223      /// <para>The constant value passed as an argument.</para>
224      /// <para>Значение константы, переданное в качестве аргумента.</para>
225      /// </returns>
226      [MethodImpl(MethodImplOptions.AggressiveInlining)]
227      public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
     ↪    TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
     ↪    returnConstant)
228      {
229          array.AddAll(ref position, elements);
230          return returnConstant;
231      }
232
233      /// <summary>
234      /// <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>
235      /// <para>Добавляет в массив все элементы коллекции, начиная с определенной позиции и
     ↪    увеличивает значение position на количество добавленных элементов.</para>
236      /// </summary>
237      /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
     ↪    массива.</para></typeparam>
238      /// <param name="array"><para>The array to add the element to.</para><para>Массив в
     ↪    который необходимо добавить элементы.</para></param>
239      /// <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>
240      /// <param name="elements"><para>List, whose elements will be added to the
     ↪    array.</para><para>Список, элементы которого будут добавленны в
     ↪    массив.</para></param>
241      [MethodImpl(MethodImplOptions.AggressiveInlining)]
242      public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
243      {
244          for (var i = 0; i < elements.Count; i++)
245          {
246              array.Add(ref position, elements[i]);
247          }
248      }
249
250      /// <summary>
251      /// <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>
252      /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию,
     ↪    увеличивает значение position на единицу и возвращает значение переданной
     ↪    константы.</para>
253      /// </summary>
254      /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
     ↪    массива.</para></typeparam>
255      /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
     ↪    возвращаемой константы.</para></typeparam>
256      /// <param name="array"><para>The array to add items to.</para><para>Массив в который
     ↪    необходимо добавить элементы.</para></param>
257      /// <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>
258      /// <param name="elements"><para>List, whose elements will be added to the
     ↪    array.</para><para>Список, элементы которого будут добавленны в
     ↪    массив.</para></param>
259      /// <param name="returnConstant"><para>The constant value that will be
     ↪    returned.</para><para>Значение константы, которое будет возвращено.</para></param>
260      /// <returns>
261      /// <para>The constant value passed as an argument.</para>
262      /// <para>Значение константы, переданное в качестве аргумента.</para>
263      /// </returns>
264      [MethodImpl(MethodImplOptions.AggressiveInlining)]
265      public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
     ↪    TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
     ↪    TReturnConstant returnConstant)
266      {
267          array.AddSkipFirst(ref position, elements);
268          return returnConstant;
269      }
```

```csharp
        /// <summary>
        /// <para>Adding in array all elements of the collection, skipping the first position
        ///  and increments position value by one.</para>
        /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию и
        ///  увеличивает значение position на единицу.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///  массива.</para></typeparam>
        /// <param name="array"><para>The array to add items to.</para><para>Массив в который
        ///  необходимо добавить элементы.</para></param>
        /// <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>
        /// <param name="elements"><para>List, whose elements will be added to the
        ///  array.</para><para>Список, элементы которого будут добавленны в
        ///  массив.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
         => array.AddSkipFirst(ref position, elements, 1);

        /// <summary>
        /// <para>Adding in array all but the first element, skipping a specified number of
        ///  positions and increments position value by one.</para>
        /// <para>Добавляет в массив все элементы коллекции, кроме первого, пропуская
        ///  определенное количество позиций и увеличивает значение position на единицу.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///  массива.</para></typeparam>
        /// <param name="array"><para>The array to add items to.</para><para>Массив в который
        ///  необходимо добавить элементы.</para></param>
        /// <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>
        /// <param name="elements"><para>List, whose elements will be added to the
        ///  array.</para><para>Список, элементы которого будут добавленны в
        ///  массив.</para></param>
        /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
        ///  пропускаемых элементов.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
         int skip)
        {
            for (var i = skip; i < elements.Count; i++)
            {
                array.Add(ref position, elements[i]);
            }
        }
    }
}
```

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

```csharp
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Numerics;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;
using Platform.Exceptions;
using Platform.Ranges;

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

namespace Platform.Collections
{
    /// <remarks>
    /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
    ///  64 бит в массиве значений.
    /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
    ///  байт в 8 байт.
    /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
    ///  помощью которой можно быстро
    /// проверять есть ли значения непосредственно далее (ниже по уровню).
    /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
    /// </remarks>
    public class BitString : IEquatable<BitString>
```

```csharp
    {
        private static readonly byte[][] _bitsSetIn16Bits;
        private long[] _array;
        private long _length;
        private long _minPositiveWord;
        private long _maxPositiveWord;

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

        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

        [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
```

```csharp
102                for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <<= 1)
103                {
104                    if ((i & k) == k)
105                    {
106                        array[c++] = bitIndex;
107                    }
108                    bitIndex++;
109                }
110                _bitsSetIn16Bits[i] = array;
111            }
112        }
113
114        [MethodImpl(MethodImplOptions.AggressiveInlining)]
115        public BitString(BitString other)
116        {
117            Ensure.Always.ArgumentNotNull(other, nameof(other));
118            _length = other._length;
119            _array = new long[GetWordsCountFromIndex(_length)];
120            _minPositiveWord = other._minPositiveWord;
121            _maxPositiveWord = other._maxPositiveWord;
122            Array.Copy(other._array, _array, _array.LongLength);
123        }
124
125        [MethodImpl(MethodImplOptions.AggressiveInlining)]
126        public BitString(long length)
127        {
128            Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
129            _length = length;
130            _array = new long[GetWordsCountFromIndex(_length)];
131            MarkBordersAsAllBitsReset();
132        }
133
134        [MethodImpl(MethodImplOptions.AggressiveInlining)]
135        public BitString(long length, bool defaultValue)
136            : this(length)
137        {
138            if (defaultValue)
139            {
140                SetAll();
141            }
142        }
143
144        #endregion
145
146        [MethodImpl(MethodImplOptions.AggressiveInlining)]
147        public BitString Not()
148        {
149            for (var i = 0L; i < _array.LongLength; i++)
150            {
151                _array[i] = ~_array[i];
152                RefreshBordersByWord(i);
153            }
154            return this;
155        }
156
157        [MethodImpl(MethodImplOptions.AggressiveInlining)]
158        public BitString ParallelNot()
159        {
160            var threads = Environment.ProcessorCount / 2;
161            if (threads <= 1)
162            {
163                return Not();
164            }
165            var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
      ↪   threads);
166            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
      ↪   MaxDegreeOfParallelism = threads }, range =>
167            {
168                var maximum = range.Item2;
169                for (var i = range.Item1; i < maximum; i++)
170                {
171                    _array[i] = ~_array[i];
172                }
173            });
174            MarkBordersAsAllBitsSet();
175            TryShrinkBorders();
176            return this;
177        }
178
```

```csharp
        [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;
        }

        [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;
        }

        [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];
            }
        }

        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelAnd(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
```

```csharp
            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;
        }

        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorAnd(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorAnd(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelAnd(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
                return VectorAnd(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 => VectorAndLoop(_array, other._array,
            ↪  step, range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorAndLoop(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)
            {
```

```csharp
329                     (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
330                 }
331                 for (; i < maximum; i++)
332                 {
333                     array[i] &= otherArray[i];
334                 }
335             }
336
337             [MethodImpl(MethodImplOptions.AggressiveInlining)]
338             public BitString Or(BitString other)
339             {
340                 EnsureBitStringHasTheSameSize(other, nameof(other));
341                 GetCommonOuterBorders(this, other, out long from, out long to);
342                 for (var i = from; i <= to; i++)
343                 {
344                     _array[i] |= other._array[i];
345                     RefreshBordersByWord(i);
346                 }
347                 return this;
348             }
349
350             [MethodImpl(MethodImplOptions.AggressiveInlining)]
351             public BitString ParallelOr(BitString other)
352             {
353                 var threads = Environment.ProcessorCount / 2;
354                 if (threads <= 1)
355                 {
356                     return Or(other);
357                 }
358                 EnsureBitStringHasTheSameSize(other, nameof(other));
359                 GetCommonOuterBorders(this, other, out long from, out long to);
360                 var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
361                 Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                 ↪  MaxDegreeOfParallelism = threads }, range =>
362                 {
363                     var maximum = range.Item2;
364                     for (var i = range.Item1; i < maximum; i++)
365                     {
366                         _array[i] |= other._array[i];
367                     }
368                 });
369                 MarkBordersAsAllBitsSet();
370                 TryShrinkBorders();
371                 return this;
372             }
373
374             [MethodImpl(MethodImplOptions.AggressiveInlining)]
375             public BitString VectorOr(BitString other)
376             {
377                 if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
378                 {
379                     return Or(other);
380                 }
381                 var step = Vector<long>.Count;
382                 if (_array.Length < step)
383                 {
384                     return Or(other);
385                 }
386                 EnsureBitStringHasTheSameSize(other, nameof(other));
387                 GetCommonOuterBorders(this, other, out int from, out int to);
388                 VectorOrLoop(_array, other._array, step, from, to + 1);
389                 MarkBordersAsAllBitsSet();
390                 TryShrinkBorders();
391                 return this;
392             }
393
394             [MethodImpl(MethodImplOptions.AggressiveInlining)]
395             public BitString ParallelVectorOr(BitString other)
396             {
397                 var threads = Environment.ProcessorCount / 2;
398                 if (threads <= 1)
399                 {
400                     return VectorOr(other);
401                 }
402                 if (!Vector.IsHardwareAccelerated)
403                 {
404                     return ParallelOr(other);
405                 }
```

```csharp
406             var step = Vector<long>.Count;
407             if (_array.Length < (step * threads))
408             {
409                 return VectorOr(other);
410             }
411             EnsureBitStringHasTheSameSize(other, nameof(other));
412             GetCommonOuterBorders(this, other, out int from, out int to);
413             var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
414             Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
     ↪   MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
     ↪   step, range.Item1, range.Item2));
415             MarkBordersAsAllBitsSet();
416             TryShrinkBorders();
417             return this;
418         }
419
420         [MethodImpl(MethodImplOptions.AggressiveInlining)]
421         static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
     ↪   int maximum)
422         {
423             var i = start;
424             var range = maximum - start - 1;
425             var stop = range - (range % step);
426             for (; i < stop; i += step)
427             {
428                 (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
429             }
430             for (; i < maximum; i++)
431             {
432                 array[i] |= otherArray[i];
433             }
434         }
435
436         [MethodImpl(MethodImplOptions.AggressiveInlining)]
437         public BitString Xor(BitString other)
438         {
439             EnsureBitStringHasTheSameSize(other, nameof(other));
440             GetCommonOuterBorders(this, other, out long from, out long to);
441             for (var i = from; i <= to; i++)
442             {
443                 _array[i] ^= other._array[i];
444                 RefreshBordersByWord(i);
445             }
446             return this;
447         }
448
449         [MethodImpl(MethodImplOptions.AggressiveInlining)]
450         public BitString ParallelXor(BitString other)
451         {
452             var threads = Environment.ProcessorCount / 2;
453             if (threads <= 1)
454             {
455                 return Xor(other);
456             }
457             EnsureBitStringHasTheSameSize(other, nameof(other));
458             GetCommonOuterBorders(this, other, out long from, out long to);
459             var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
460             Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
     ↪   MaxDegreeOfParallelism = threads }, range =>
461             {
462                 var maximum = range.Item2;
463                 for (var i = range.Item1; i < maximum; i++)
464                 {
465                     _array[i] ^= other._array[i];
466                 }
467             });
468             MarkBordersAsAllBitsSet();
469             TryShrinkBorders();
470             return this;
471         }
472
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         public BitString VectorXor(BitString other)
475         {
476             if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
477             {
478                 return Xor(other);
479             }
```

```csharp
480            var step = Vector<long>.Count;
481            if (_array.Length < step)
482            {
483                return Xor(other);
484            }
485            EnsureBitStringHasTheSameSize(other, nameof(other));
486            GetCommonOuterBorders(this, other, out int from, out int to);
487            VectorXorLoop(_array, other._array, step, from, to + 1);
488            MarkBordersAsAllBitsSet();
489            TryShrinkBorders();
490            return this;
491        }
492
493        [MethodImpl(MethodImplOptions.AggressiveInlining)]
494        public BitString ParallelVectorXor(BitString other)
495        {
496            var threads = Environment.ProcessorCount / 2;
497            if (threads <= 1)
498            {
499                return VectorXor(other);
500            }
501            if (!Vector.IsHardwareAccelerated)
502            {
503                return ParallelXor(other);
504            }
505            var step = Vector<long>.Count;
506            if (_array.Length < (step * threads))
507            {
508                return VectorXor(other);
509            }
510            EnsureBitStringHasTheSameSize(other, nameof(other));
511            GetCommonOuterBorders(this, other, out int from, out int to);
512            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
513            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    ↪   MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
    ↪   step, range.Item1, range.Item2));
514            MarkBordersAsAllBitsSet();
515            TryShrinkBorders();
516            return this;
517        }
518
519        [MethodImpl(MethodImplOptions.AggressiveInlining)]
520        static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
    ↪   int maximum)
521        {
522            var i = start;
523            var range = maximum - start - 1;
524            var stop = range - (range % step);
525            for (; i < stop; i += step)
526            {
527                (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
528            }
529            for (; i < maximum; i++)
530            {
531                array[i] ^= otherArray[i];
532            }
533        }
534
535        [MethodImpl(MethodImplOptions.AggressiveInlining)]
536        private void RefreshBordersByWord(long wordIndex)
537        {
538            if (_array[wordIndex] == 0)
539            {
540                if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
541                {
542                    _minPositiveWord++;
543                }
544                if (wordIndex == _maxPositiveWord && wordIndex != 0)
545                {
546                    _maxPositiveWord--;
547                }
548            }
549            else
550            {
551                if (wordIndex < _minPositiveWord)
552                {
553                    _minPositiveWord = wordIndex;
554                }
```

```csharp
555                    if (wordIndex > _maxPositiveWord)
556                    {
557                        _maxPositiveWord = wordIndex;
558                    }
559                }
560            }
561
562            [MethodImpl(MethodImplOptions.AggressiveInlining)]
563            public bool TryShrinkBorders()
564            {
565                GetBorders(out long from, out long to);
566                while (from <= to && _array[from] == 0)
567                {
568                    from++;
569                }
570                if (from > to)
571                {
572                    MarkBordersAsAllBitsReset();
573                    return true;
574                }
575                while (to >= from && _array[to] == 0)
576                {
577                    to--;
578                }
579                if (to < from)
580                {
581                    MarkBordersAsAllBitsReset();
582                    return true;
583                }
584                var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
585                if (bordersUpdated)
586                {
587                    SetBorders(from, to);
588                }
589                return bordersUpdated;
590            }
591
592            [MethodImpl(MethodImplOptions.AggressiveInlining)]
593            public bool Get(long index)
594            {
595                Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
596                return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
597            }
598
599            [MethodImpl(MethodImplOptions.AggressiveInlining)]
600            public void Set(long index, bool value)
601            {
602                if (value)
603                {
604                    Set(index);
605                }
606                else
607                {
608                    Reset(index);
609                }
610            }
611
612            [MethodImpl(MethodImplOptions.AggressiveInlining)]
613            public void Set(long index)
614            {
615                Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
616                var wordIndex = GetWordIndexFromIndex(index);
617                var mask = GetBitMaskFromIndex(index);
618                _array[wordIndex] |= mask;
619                RefreshBordersByWord(wordIndex);
620            }
621
622            [MethodImpl(MethodImplOptions.AggressiveInlining)]
623            public void Reset(long index)
624            {
625                Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
626                var wordIndex = GetWordIndexFromIndex(index);
627                var mask = GetBitMaskFromIndex(index);
628                _array[wordIndex] &= ~mask;
629                RefreshBordersByWord(wordIndex);
630            }
631
632            [MethodImpl(MethodImplOptions.AggressiveInlining)]
633            public bool Add(long index)
```

```csharp
634              {
635                  var wordIndex = GetWordIndexFromIndex(index);
636                  var mask = GetBitMaskFromIndex(index);
637                  if ((_array[wordIndex] & mask) == 0)
638                  {
639                      _array[wordIndex] |= mask;
640                      RefreshBordersByWord(wordIndex);
641                      return true;
642                  }
643                  else
644                  {
645                      return false;
646                  }
647              }
648
649              [MethodImpl(MethodImplOptions.AggressiveInlining)]
650              public void SetAll(bool value)
651              {
652                  if (value)
653                  {
654                      SetAll();
655                  }
656                  else
657                  {
658                      ResetAll();
659                  }
660              }
661
662              [MethodImpl(MethodImplOptions.AggressiveInlining)]
663              public void SetAll()
664              {
665                  const long fillValue = unchecked((long)0xffffffffffffffff);
666                  var words = GetWordsCountFromIndex(_length);
667                  for (var i = 0; i < words; i++)
668                  {
669                      _array[i] = fillValue;
670                  }
671                  MarkBordersAsAllBitsSet();
672              }
673
674              [MethodImpl(MethodImplOptions.AggressiveInlining)]
675              public void ResetAll()
676              {
677                  const long fillValue = 0;
678                  GetBorders(out long from, out long to);
679                  for (var i = from; i <= to; i++)
680                  {
681                      _array[i] = fillValue;
682                  }
683                  MarkBordersAsAllBitsReset();
684              }
685
686              [MethodImpl(MethodImplOptions.AggressiveInlining)]
687              public List<long> GetSetIndices()
688              {
689                  var result = new List<long>();
690                  GetBorders(out long from, out long to);
691                  for (var i = from; i <= to; i++)
692                  {
693                      var word = _array[i];
694                      if (word != 0)
695                      {
696                          AppendAllSetBitIndices(result, i, word);
697                      }
698                  }
699                  return result;
700              }
701
702              [MethodImpl(MethodImplOptions.AggressiveInlining)]
703              public List<ulong> GetSetUInt64Indices()
704              {
705                  var result = new List<ulong>();
706                  GetBorders(out ulong from, out ulong to);
707                  for (var i = from; i <= to; i++)
708                  {
709                      var word = _array[i];
710                      if (word != 0)
711                      {
712                          AppendAllSetBitIndices(result, i, word);
```

```csharp
                    }
                }
                return result;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public long GetFirstSetBitIndex()
            {
                var i = _minPositiveWord;
                var word = _array[i];
                if (word != 0)
                {
                    return GetFirstSetBitForWord(i, word);
                }
                return -1;
            }

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

            [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;
            }

            [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;
            }

            [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;
        }

        [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;
        }

        [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;
        }

        [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;
        }

        [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);
                }
            }
        }
```

```csharp
                return -1;
            }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
        ↪  false;

        [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;
        }

        [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);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out long from, out long to)
        {
            from = _minPositiveWord;
            to = _maxPositiveWord;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out ulong from, out ulong to)
        {
            from = (ulong)_minPositiveWord;
            to = (ulong)_maxPositiveWord;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void SetBorders(long from, long to)
        {
            _minPositiveWord = from;
            _maxPositiveWord = to;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private Range<long> GetValidIndexRange() => (0, _length - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
949         private static Range<long> GetValidLengthRange() => (0, long.MaxValue);

951         [MethodImpl(MethodImplOptions.AggressiveInlining)]
952         private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
      ↪  wordValue)
953         {
954             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
          ↪  bits32to47, out byte[] bits48to63);
955             AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
          ↪  bits48to63);
956         }

958         [MethodImpl(MethodImplOptions.AggressiveInlining)]
959         private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
      ↪  wordValue)
960         {
961             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
          ↪  bits32to47, out byte[] bits48to63);
962             AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
          ↪  bits48to63);
963         }

965         [MethodImpl(MethodImplOptions.AggressiveInlining)]
966         private static long CountSetBitsForWord(long word)
967         {
968             GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
          ↪  out byte[] bits48to63);
969             return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
          ↪  bits48to63.LongLength;
970         }

972         [MethodImpl(MethodImplOptions.AggressiveInlining)]
973         private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
974         {
975             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
          ↪  bits32to47, out byte[] bits48to63);
976             return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
977         }

979         [MethodImpl(MethodImplOptions.AggressiveInlining)]
980         private static long GetLastSetBitForWord(long wordIndex, long wordValue)
981         {
982             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
          ↪  bits32to47, out byte[] bits48to63);
983             return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
984         }

986         [MethodImpl(MethodImplOptions.AggressiveInlining)]
987         private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
      ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
988         {
989             for (var j = 0; j < bits00to15.Length; j++)
990             {
991                 result.Add(bits00to15[j] + (i * 64));
992             }
993             for (var j = 0; j < bits16to31.Length; j++)
994             {
995                 result.Add(bits16to31[j] + 16 + (i * 64));
996             }
997             for (var j = 0; j < bits32to47.Length; j++)
998             {
999                 result.Add(bits32to47[j] + 32 + (i * 64));
1000            }
1001            for (var j = 0; j < bits48to63.Length; j++)
1002            {
1003                result.Add(bits48to63[j] + 48 + (i * 64));
1004            }
1005        }

1007        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1008        private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
      ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1009        {
1010            for (var j = 0; j < bits00to15.Length; j++)
1011            {
1012                result.Add(bits00to15[j] + (i * 64));
1013            }
```

```csharp
                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));
                }
            }
        }

        [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);
        }

        [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);
        }

        [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];
        }

        [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);
        }

        [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);
        }
```

```csharp
1087        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1088        public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
     ↪   out int to)
1089        {
1090            from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1091            to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1092        }
1093
1094        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1095        public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
     ↪   ulong to)
1096        {
1097            from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1098            to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1099        }
1100
1101        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1102        public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1103
1104        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1105        public static long GetWordIndexFromIndex(long index) => index >> 6;
1106
1107        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1108        public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1109
1110        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1111        public override int GetHashCode() => base.GetHashCode();
1112
1113        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1114        public override string ToString() => base.ToString();
1115    }
1116 }
```

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

```csharp
1  using System.Runtime.CompilerServices;
2  using Platform.Random;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections
7  {
8      public static class BitStringExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void SetRandomBits(this BitString @string)
12         {
13             for (var i = 0; i < @string.Length; i++)
14             {
15                 var value = RandomHelpers.Default.NextBoolean();
16                 @string.Set(i, value);
17             }
18         }
19     }
20 }
```

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

```csharp
1  using System.Collections.Concurrent;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Concurrent
8  {
9      public static class ConcurrentQueueExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
13         {
14             while (queue.TryDequeue(out T item))
15             {
16                 yield return item;
17             }
18         }
19     }
20 }
```

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

```csharp
1  using System.Collections.Concurrent;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Concurrent
7  {
8      public static class ConcurrentStackExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
            value) ? value : default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
            value) ? value : default;
15     }
16 }
```

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Runtime.CompilerServices;
5  using Platform.Exceptions;
6  using Platform.Exceptions.ExtensionRoots;
7
8  #pragma warning disable IDE0060 // Remove unused parameter
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     public static class EnsureExtensions
14     {
15         #region Always
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ICollection<T> argument, string argumentName, string message)
19         {
20             if (argument.IsNullOrEmpty())
21             {
22                 throw new ArgumentException(message, argumentName);
23             }
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
            argumentName, null);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument, string argumentName, string message)
34         {
35             if (string.IsNullOrWhiteSpace(argument))
36             {
37                 throw new ArgumentException(message, argumentName);
38             }
39         }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
            argument, argumentName, null);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
46
47         #endregion
48
49         #region OnDebug
50
```

```csharp
        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName, string message) =>
        ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName) =>
        ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
        ↪  ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ↪  root, string argument, string argumentName, string message) =>
        ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
        ↪  root, string argument, string argumentName) =>
        ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);

        [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

```csharp
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
{
    public static class ICollectionExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
        ↪  null || collection.Count == 0;

        [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

```csharp
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
{
    public static class IDictionaryExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
        ↪  dictionary, TKey key)
        {
            dictionary.TryGetValue(key, out TValue value);
            return value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
        ↪  TKey key, Func<TKey, TValue> valueFactory)
```

```
20          {
21              if (!dictionary.TryGetValue(key, out TValue value))
22              {
23                  value = valueFactory(key);
24                  dictionary.Add(key, value);
25                  return value;
26              }
27              return value;
28          }
29      }
30  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public static class CharIListExtensions
7      {
8          /// <summary>
9          /// <para>Generates a hash code for the entire list based on the values of its
             elements.</para>
10         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
11         /// </summary>
12         /// <param name="list"><para>The list to be hashed.</para><para>Список для
             хеширования.</para></param>
13         /// <returns>
14         /// <para>The hash code of the list.</para>
15         /// <para>Хэш-код списка.</para>
16         /// </returns>
17         /// <remarks>
18         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
             a3eda37d3d4cd10/mscorlib/system/string.cs#L833
19         /// </remarks>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static int GenerateHashCode(this IList<char> list)
22         {
23             var hashSeed = 5381;
24             var hashAccumulator = hashSeed;
25             for (var i = 0; i < list.Count; i++)
26             {
27                 hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
28             }
29             return hashAccumulator + (hashSeed * 1566083941);
30         }
31
32         /// <summary>
33         /// <para>Compares two lists for equality.</para>
34         /// <para>Сравнивает два списка на равенство.</para>
35         /// </summary>
36         /// <param name="left"><para>The first compared list.</para><para>Первый список для
             сравнения.</para></param>
37         /// <param name="right"><para>The second compared list.</para><para>Второй список для
             сравнения.</para></param>
38         /// <returns>
39         /// <para>True, if the passed lists are equal to each other otherwise false.</para>
40         /// <para>True, если переданные списки равны друг другу, иначе false.</para>
41         /// </returns>
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         public static bool EqualTo(this IList<char> left, IList<char> right) =>
             left.EqualTo(right, ContentEqualTo);
44
45         /// <summary>
46         /// <para>Compares each element in the list for equality.</para>
47         /// <para>Сравнивает на равенство каждый элемент списка.</para>
48         /// </summary>
49         /// <param name="left"><para>The first compared list.</para><para>Первый список для
             сравнения.</para></param>
50         /// <param name="right"><para>The second compared list.</para><para>Второй список для
             сравнения.</para></param>
51         /// <returns>
52         /// <para>If at least one element of one list is not equal to the corresponding element
             from another list returns false, otherwise - true.</para>
53         /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
             из другого списка возвращает false, иначе - true.</para>
54         /// </returns>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
          public static bool ContentEqualTo(this IList<char> left, IList<char> right)
          {
              for (var i = left.Count - 1; i >= 0; --i)
              {
                  if (left[i] != right[i])
                  {
                      return false;
                  }
              }
              return true;
          }
      }
  }
```

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

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

namespace Platform.Collections.Lists
{
    public class IListComparer<T> : IComparer<IList<T>>
    {
        /// <summary>
        /// <para>Compares two lists.</para>
        /// <para>Сравнивает два списка.<para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪ списка.</para></typeparam>
        /// <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>
        ///     A signed integer that indicates the relative values of <paramref name="left" />
        ↪ and <paramref name="right" /> lists' elements, as shown in the following table.
        ///     <list type="table">
        ///         <listheader>
        ///             <term>Value</term>
        ///             <description>Meaning</description>
        ///         </listheader>
        ///         <item>
        ///             <term>Is less than zero</term>
        ///             <description>First non equal element of <paramref name="left" /> list is
        ↪ less than first not equal element of <paramref name="right" /> list.</description>
        ///         </item>
        ///         <item>
        ///             <term>Zero</term>
        ///             <description>All elements of <paramref name="left" /> list equals to all
        ↪ elements of <paramref name="right" /> list.</description>
        ///         </item>
        ///         <item>
        ///             <term>Is greater than zero</term>
        ///             <description>First non equal element of <paramref name="left" /> list is
        ↪ greater than first not equal element of <paramref name="right" /> list.</description>
        ///         </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>
```

```
52    ///            <item>
53    ///                <term>Больше нуля</term>
54    ///                <description>Первый не равный элемент <paramref name="left" /> списка
      ↪  больше первого неравного элемента <paramref name="right" /> списка.</description>
55    ///            </item>
56    ///        </list>
57    /// </para>
58    /// </returns>
59    [MethodImpl(MethodImplOptions.AggressiveInlining)]
60    public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
61    }
62 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
7      {
8          /// <summary>
9          /// <para>Compares two lists for equality.</para>
10         /// <para>Сравнивает два списка на равенство.</para>
11         /// </summary>
12         /// <param name="left"><para>The first compared list.</para><para>Первый список для
      ↪  сравнения.</para></param>
13         /// <param name="right"><para>The second compared list.</para><para>Второй список для
      ↪  сравнения.</para></param>
14         /// <returns>
15         /// <para>If the passed lists are equal to each other, true is returned, otherwise
      ↪  false.</para>
16         /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
      ↪  false.</para>
17         /// </returns>
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
20
21         /// <summary>
22         /// <para>Generates a hash code for the entire list based on the values of its
      ↪  elements.</para>
23         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
24         /// </summary>
25         /// <param name="list"><para>Hash list.</para><para>Список для
      ↪  хеширования.</para></param>
26         /// <returns>
27         /// <para>The hash code of the list.</para>
28         /// <para>Хэш-код списка.</para>
29         /// </returns>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public int GetHashCode(IList<T> list) => list.GenerateHashCode();
32     }
33 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Collections.Lists
6  {
7      public static class IListExtensions
8      {
9          /// <summary>
10         /// <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>
11         /// <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>
12         /// </summary>
13         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
      ↪  списка.</typeparam>
14         /// <param name="list"><para>The checked list.</para><para>Проверяемый
      ↪  список.</para></param>
15         /// <param name="index"><para>The index of element.</para><para>Индекс
      ↪  элемента.</para></param>
16         /// <returns>
```

```csharp
17      /// <para>If the specified index is within list's boundaries, then - list[index],
        ↪  otherwise the default value.</para>
18      /// <para>Если указанный индекс находится в пределах границ списка, тогда - list[index],
        ↪  иначе же значение по умолчанию.</para>
19      /// </returns>
20      [MethodImpl(MethodImplOptions.AggressiveInlining)]
21      public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
        ↪  list.Count > index ? list[index] : default;
22
23      /// <summary>
24      /// <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>
25      /// <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>
26      /// </summary>
27      /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪  списка.</para></typeparam>
28      /// <param name="list"><para>The checked list.</para><para>Список для
        ↪  проверки.</para></param>
29      /// <param name="index"><para>The index of element..</para><para>Индекс
        ↪  элемента.</para></param>
30      /// <param name="element"><para>Variable for passing the index
        ↪  value.</para><para>Переменная для передачи значения индекса.</para></param>
31      /// <returns>
32      /// <para>True on success, false otherwise.</para>
33      /// <para>True в случае успеха, иначе false.</para>
34      /// </returns>
35      [MethodImpl(MethodImplOptions.AggressiveInlining)]
36      public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
37      {
38          if (list != null && list.Count > index)
39          {
40              element = list[index];
41              return true;
42          }
43          else
44          {
45              element = default;
46              return false;
47          }
48      }
49
50      /// <summary>
51      /// <para>Adds a value to the list.</para>
52      /// <para>Добавляет значение в список.</para>
53      /// </summary>
54      /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪  списка.</para></typeparam>
55      /// <param name="list"><para>The list to add the value to.</para><para>Список в который
        ↪  нужно добавить значение.</para></param>
56      /// <param name="element"><para>The item to add to the list.</para><para>Элемент который
        ↪  нужно добавить в список.</para></param>
57      /// <returns>
58      /// <para>True value in any case.</para>
59      /// <para>Значение true в любом случае.</para>
60      /// </returns>
61      [MethodImpl(MethodImplOptions.AggressiveInlining)]
62      public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
63      {
64          list.Add(element);
65          return true;
66      }
67
68      /// <summary>
69      /// <para>Adds the value with first index from other list to this list.</para>
70      /// <para>Добавляет в этот список значение с первым индексом из другого списка.</para>
71      /// </summary>
72      /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪  списка.</para></typeparam>
73      /// <param name="list"><para>The list to add the value to.</para><para>Список в который
        ↪  нужно добавить значение.</para></param>
74      /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
        ↪  который нужно добавить в список</para></param>
75      /// <returns>
76      /// <para>True value in any case.</para>
```

```csharp
        /// <para>Значение true в любом случае.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
        {
            list.AddFirst(elements);
            return true;
        }

        /// <summary>
        /// <para>Adds a value to the list at the first index.</para>
        /// <para>Добавляет значение в список по первому индексу.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪    списка.</para></typeparam>
        /// <param name="list"><para>The list to add the value to.</para><para>Список в который
        ↪    нужно добавить значение.</para></param>
        /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
        ↪    который нужно добавить в список</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
        ↪    list.Add(elements[0]);

        /// <summary>
        /// <para>Adds all elements from other list to this list and returns true.</para>
        /// <para>Добавляет все элементы из другого списка в этот список и возвращает
        ↪    true.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪    списка.</para></typeparam>
        /// <param name="list"><para>The list to add the values to.</para><para>Список в который
        ↪    нужно добавить значения.</para></param>
        /// <param name="elements"><para>List of values to add.</para><para>Список значений
        ↪    которые необходимо добавить.</para></param>
        /// <returns>
        /// <para>True value in any case.</para>
        /// <para>Значение true в любом случае.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
        {
            list.AddAll(elements);
            return true;
        }

        /// <summary>
        /// <para>Adds all elements from other list to this list.</para>
        /// <para>Добавляет все элементы из другого списка в этот список.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪    списка.</para></typeparam>
        /// <param name="list"><para>The list to add the values to.</para><para>Список в который
        ↪    нужно добавить значения.</para></param>
        /// <param name="elements"><para>The list of values to add.</para><para>Список значений
        ↪    которые необходимо добавить.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAll<T>(this IList<T> list, IList<T> elements)
        {
            for (var i = 0; i < elements.Count; i++)
            {
                list.Add(elements[i]);
            }
        }

        /// <summary>
        /// <para>Adds values to the list skipping the first element.</para>
        /// <para>Добавляет значения в список пропуская первый элемент.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪    списка.</para></typeparam>
        /// <param name="list"><para>The list to add the values to.</para><para>Список в который
        ↪    нужно добавить значения.</para></param>
        /// <param name="elements"><para>The list of values to add.</para><para>Список значений
        ↪    которые необходимо добавить.</para></param>
        /// <returns>
        /// <para>True value in any case.</para>
        /// <para>Значение true в любом случае.</para>
```

```csharp
140         /// </returns>
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
143         {
144             list.AddSkipFirst(elements);
145             return true;
146         }
147
148         /// <summary>
149         /// <para>Adds values to the list skipping the first element.</para>
150         /// <para>Добавляет значения в список пропуская первый элемент.</para>
151         /// </summary>
152         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
      ↪ списка.</para></typeparam>
153         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
      ↪ нужно добавить значения.</para></param>
154         /// <param name="elements"><para>List of values to add.</para><para>Список значений
      ↪ которые необходимо добавить.</para></param>
155         [MethodImpl(MethodImplOptions.AggressiveInlining)]
156         public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
      ↪ list.AddSkipFirst(elements, 1);
157
158         /// <summary>
159         /// <para>Adds values to the list skipping a specified number of first elements.</para>
160         /// <para>Добавляет в список значения пропуская определенное количество первых
      ↪ элементов.</para>
161         /// </summary>
162         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
      ↪ списка.</para></typeparam>
163         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
      ↪ нужно добавить значения.</para></param>
164         /// <param name="elements"><para>List of values to add.</para><para>Список значений
      ↪ которые необходимо добавить.</para></param>
165         /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
      ↪ пропускаемых элементов.</para></param>
166         [MethodImpl(MethodImplOptions.AggressiveInlining)]
167         public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
168         {
169             for (var i = skip; i < elements.Count; i++)
170             {
171                 list.Add(elements[i]);
172             }
173         }
174
175         /// <summary>
176         /// <para>Reads the number of elements in the list.</para>
177         /// <para>Считывает число элементов списка.</para>
178         /// </summary>
179         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
      ↪ списка.</para></typeparam>
180         /// <param name="list"><para>The checked list.</para><para>Список для
      ↪ проверки.</para></param>
181         /// <returns>
182         /// <para>The number of items contained in the list or 0.</para>
183         /// <para>Число элементов содержащихся в списке или же 0.</para>
184         /// </returns>
185         [MethodImpl(MethodImplOptions.AggressiveInlining)]
186         public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
187
188         /// <summary>
189         /// <para>Compares two lists for equality.</para>
190         /// <para>Сравнивает два списка на равенство.</para>
191         /// </summary>
192         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
      ↪ списка.</para></typeparam>
193         /// <param name="left"><para>The first compared list.</para><para>Первый список для
      ↪ сравнения.</para></param>
194         /// <param name="right"><para>The second compared list.</para><para>Второй список для
      ↪ сравнения.</para></param>
195         /// <returns>
196         /// <para>If the passed lists are equal to each other, true is returned, otherwise
      ↪ false.</para>
197         /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
      ↪ false.</para>
198         /// </returns>
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
200        public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
    →   right, ContentEqualTo);
201
202        /// <summary>
203        /// <para>Compares two lists for equality.</para>
204        /// <para>Сравнивает два списка на равенство.</para>
205        /// </summary>
206        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    →   списка.</para></typeparam>
207        /// <param name="left"><para>The first compared list.</para><para>Первый список для
    →   проверки.</para></param>
208        /// <param name="right"><para>The second compared list.</para><para>Второй список для
    →   сравнения.</para></param>
209        /// <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>
210        /// <returns>
211        /// <para>If the passed lists are equal to each other, true is returned, otherwise
    →   false.</para>
212        /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
    →   false.</para>
213        /// </returns>
214        [MethodImpl(MethodImplOptions.AggressiveInlining)]
215        public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
    →   IList<T>, bool> contentEqualityComparer)
216        {
217            if (ReferenceEquals(left, right))
218            {
219                return true;
220            }
221            var leftCount = left.GetCountOrZero();
222            var rightCount = right.GetCountOrZero();
223            if (leftCount == 0 && rightCount == 0)
224            {
225                return true;
226            }
227            if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
228            {
229                return false;
230            }
231            return contentEqualityComparer(left, right);
232        }
233
234        /// <summary>
235        /// <para>Compares each element in the list for identity.</para>
236        /// <para>Сравнивает на равенство каждый элемент списка.</para>
237        /// </summary>
238        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    →   списка.</para></typeparam>
239        /// <param name="left"><para>The first compared list.</para><para>Первый список для
    →   сравнения.</para></param>
240        /// <param name="right"><para>The second compared list.</para><para>Второй список для
    →   сравнения.</para></param>
241        /// <returns>
242        /// <para>If at least one element of one list is not equal to the corresponding element
    →   from another list returns false, otherwise - true.</para>
243        /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
    →   из другого списка возвращает false, иначе - true.</para>
244        /// </returns>
245        [MethodImpl(MethodImplOptions.AggressiveInlining)]
246        public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
247        {
248            var equalityComparer = EqualityComparer<T>.Default;
249            for (var i = left.Count - 1; i >= 0; --i)
250            {
251                if (!equalityComparer.Equals(left[i], right[i]))
252                {
253                    return false;
254                }
255            }
256            return true;
257        }
258
259        /// <summary>
260        /// <para>Creates an array by copying all elements from the list that satisfy the
    →   predicate. If no list is passed, null is returned.</para>
```

```csharp
261         /// <para>Создаёт массив, копируя из списка все элементы которые удовлетворяют
            ↪  предикату. Если список не передан, возвращается null.</para>
262         /// </summary>
263         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
264         /// <param name="list">The list to copy from.<para>Список для копирования.</para></param>
265         /// <param name="predicate"><para>A function that determines whether an element should
            ↪  be copied.</para><para>Функция определяющая должен ли копироваться
            ↪  элемент.</para></param>
266         /// <returns>
267         /// <para>An array with copied elements from the list.</para>
268         /// <para>Массив с скопированными элементами из списка.</para>
269         /// </returns>
270         [MethodImpl(MethodImplOptions.AggressiveInlining)]
271         public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
272         {
273             if (list == null)
274             {
275                 return null;
276             }
277             var result = new List<T>(list.Count);
278             for (var i = 0; i < list.Count; i++)
279             {
280                 if (predicate(list[i]))
281                 {
282                     result.Add(list[i]);
283                 }
284             }
285             return result.ToArray();
286         }
287
288         /// <summary>
289         /// <para>Copies all the elements of the list into an array and returns it.</para>
290         /// <para>Копирует все элементы списка в массив и возвращает его.</para>
291         /// </summary>
292         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
293         /// <param name="list"><para>The list to copy from.</para><para>Список для
            ↪  копирования.</para></param>
294         /// <returns>
295         /// <para>An array with all the elements of the passed list.</para>
296         /// <para>Массив со всеми элементами переданного списка.</para>
297         /// </returns>
298         [MethodImpl(MethodImplOptions.AggressiveInlining)]
299         public static T[] ToArray<T>(this IList<T> list)
300         {
301             var array = new T[list.Count];
302             list.CopyTo(array, 0);
303             return array;
304         }
305
306         /// <summary>
307         /// <para>Executes the passed action for each item in the list.</para>
308         /// <para>Выполняет переданное действие для каждого элемента в списке.</para>
309         /// </summary>
310         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪  списка.</para></typeparam>
311         /// <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>
312         /// <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>
313         [MethodImpl(MethodImplOptions.AggressiveInlining)]
314         public static void ForEach<T>(this IList<T> list, Action<T> action)
315         {
316             for (var i = 0; i < list.Count; i++)
317             {
318                 action(list[i]);
319             }
320         }
321
322         /// <summary>
323         /// <para>Generates a hash code for the entire list based on the values of its
            ↪  elements.</para>
324         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
325         /// </summary>
```

```csharp
326         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪ списка.</para></typeparam>
327         /// <param name="list"><para>Hash list.</para><para>Список для
            ↪ хеширования.</para></param>
328         /// <returns>
329         /// <para>The hash code of the list.</para>
330         /// <para>Хэш-код списка.</para>
331         /// </returns>
332         /// <remarks>
333         /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
            ↪ -overridden-system-object-gethashcode
334         /// </remarks>
335         [MethodImpl(MethodImplOptions.AggressiveInlining)]
336         public static int GenerateHashCode<T>(this IList<T> list)
337         {
338             var hashAccumulator = 17;
339             for (var i = 0; i < list.Count; i++)
340             {
341                 hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
342             }
343             return hashAccumulator;
344         }
345
346         /// <summary>
347         /// <para>Compares two lists.</para>
348         /// <para>Сравнивает два списка.</para>
349         /// </summary>
350         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            ↪ списка.</para></typeparam>
351         /// <param name="left"><para>The first compared list.</para><para>Первый список для
            ↪ сравнения.</para></param>
352         /// <param name="right"><para>The second compared list.</para><para>Второй список для
            ↪ сравнения.</para></param>
353         /// <returns>
354         /// <para>
355         ///     A signed integer that indicates the relative values of <paramref name="left" />
            ↪ and <paramref name="right" /> lists' elements, as shown in the following table.
356         ///     <list type="table">
357         ///         <listheader>
358         ///             <term>Value</term>
359         ///             <description>Meaning</description>
360         ///         </listheader>
361         ///         <item>
362         ///             <term>Is less than zero</term>
363         ///             <description>First non equal element of <paramref name="left" /> list is
            ↪ less than first not equal element of <paramref name="right" /> list.</description>
364         ///         </item>
365         ///         <item>
366         ///             <term>Zero</term>
367         ///             <description>All elements of <paramref name="left" /> list equals to all
            ↪ elements of <paramref name="right" /> list.</description>
368         ///         </item>
369         ///         <item>
370         ///             <term>Is greater than zero</term>
371         ///             <description>First non equal element of <paramref name="left" /> list is
            ↪ greater than first not equal element of <paramref name="right" /> list.</description>
372         ///         </item>
373         ///     </list>
374         /// </para>
375         /// <para>
376         ///     Целое число со знаком, которое указывает относительные значения элементов
            ↪ списков <paramref name="left" /> и <paramref name="right" /> как показано в
            ↪ следующей таблице.
377         ///     <list type="table">
378         ///         <listheader>
379         ///             <term>Значение</term>
380         ///             <description>Смысл</description>
381         ///         </listheader>
382         ///         <item>
383         ///             <term>Меньше нуля</term>
384         ///             <description>Первый не равный элемент <paramref name="left" /> списка
            ↪ меньше первого неравного элемента <paramref name="right" /> списка.</description>
385         ///         </item>
386         ///         <item>
387         ///             <term>Ноль</term>
388         ///             <description>Все элементы <paramref name="left" /> списка равны всем
            ↪ элементам <paramref name="right" /> списка.</description>
```

```csharp
        ///         </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 static int CompareTo<T>(this IList<T> left, IList<T> right)
        {
            var comparer = Comparer<T>.Default;
            var leftCount = left.GetCountOrZero();
            var rightCount = right.GetCountOrZero();
            var intermediateResult = leftCount.CompareTo(rightCount);
            for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
            {
                intermediateResult = comparer.Compare(left[i], right[i]);
            }
            return intermediateResult;
        }

        /// <summary>
        /// <para>Skips one element in the list and builds an array from the remaining
        ///     elements.</para>
        /// <para>Пропускает один элемент списка и составляет из оставшихся элементов
        ///     массив.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ///     списка.</para></typeparam>
        /// <param name="list"><para>The list to copy from.</para><para>Список для
        ///     копирования.</para></param>
        /// <returns>
        /// <para>If the list is empty, returns an empty array, otherwise - an array with a
        ///     missing first element.</para>
        /// <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
        ///     первым элементом.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);

        /// <summary>
        /// <para>Skips the specified number of elements in the list and builds an array from
        ///     the remaining elements.</para>
        /// <para>Пропускает указанное количество элементов списка и составляет из оставшихся
        ///     элементов массив.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ///     списка.</para></typeparam>
        /// <param name="list"><para>The list to copy from.</para><para>Список для
        ///     копирования.</para></param>
        /// <param name="skip"><para>The number of items to skip.</para><para>Количество
        ///     пропускаемых элементов.</para></param>
        /// <returns>
        /// <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>
        /// <para>Если список пуст, или количество пропускаемых элементов больше списка -
        ///     возвращает пустой массив, иначе - массив с указанным количеством пропущенных
        ///     элементов.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] SkipFirst<T>(this IList<T> list, int skip)
        {
            if (list.IsNullOrEmpty() || list.Count <= skip)
            {
                return Array.Empty<T>();
            }
            var result = new T[list.Count - skip];
            for (int r = skip, w = 0; r < list.Count; r++, w++)
            {
                result[w] = list[r];
            }
            return result;
        }
```

```csharp
450        /// <summary>
451        /// <para>Shifts all the elements of the list by one position to the right.</para>
452        /// <para>Сдвигает вправо все элементы списка на одну позицию.</para>
453        /// </summary>
454        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
     ↪  списка.</para></typeparam>
455        /// <param name="list"><para>The list to copy from.</para><para>Список для
     ↪  копирования.</para></param>
456        /// <returns>
457        /// <para>Array with a shift of elements by one position.</para>
458        /// <para>Массив со сдвигом элементов на одну позицию.</para>
459        /// </returns>
460        [MethodImpl(MethodImplOptions.AggressiveInlining)]
461        public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
462
463        /// <summary>
464        /// <para>Shifts all elements of the list to the right by the specified number of
     ↪  elements.</para>
465        /// <para>Сдвигает вправо все элементы списка на указанное количество элементов.</para>
466        /// </summary>
467        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
     ↪  списка.</para></typeparam>
468        /// <param name="list"><para>The list to copy from.</para><para>Список для
     ↪  копирования.</para></param>
469        /// <param name="skip"><para>The number of items to shift.</para><para>Количество
     ↪  сдвигаемых элементов.</para></param>
470        /// <returns>
471        /// <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>
472        /// <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>
473        /// </returns>
474        [MethodImpl(MethodImplOptions.AggressiveInlining)]
475        public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
476        {
477            if (shift < 0)
478            {
479                throw new NotImplementedException();
480            }
481            if (shift == 0)
482            {
483                return list.ToArray();
484            }
485            else
486            {
487                var result = new T[list.Count + shift];
488                for (int r = 0, w = shift; r < list.Count; r++, w++)
489                {
490                    result[w] = list[r];
491                }
492                return result;
493            }
494        }
495    }
496 }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public class ListFiller<TElement, TReturnConstant>
7      {
8          protected readonly List<TElement> _list;
9          protected readonly TReturnConstant _returnConstant;
10
11         /// <summary>
12         /// <para>Initializes a new instance of the ListFiller class.</para>
13         /// <para>Инициализирует новый экземпляр класса ListFiller.</para>
14         /// </summary>
15         /// <param name="list"><para>The list to be filled.</para><para>Список который будет
     ↪  заполняться.</para></param>
```

```csharp
16          /// <param name="returnConstant"><para>The value for the constant returned by
      ↪   corresponding methods.</para><para>Значение для константы возвращаемой
      ↪   соответствующими методами.</para></param>
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          public ListFiller(List<TElement> list, TReturnConstant returnConstant)
19          {
20              _list = list;
21              _returnConstant = returnConstant;
22          }
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          public ListFiller(List<TElement> list) : this(list, default) { }
26
27          /// <summary>
28          /// <para>Adds an item to the end of the list.</para>
29          /// <para>Добавляет элемент в конец списка.</para>
30          /// </summary>
31          /// <param name="element"><para>Element to add.</para><para>Добавляемый
      ↪   элемент.</para></param>
32          [MethodImpl(MethodImplOptions.AggressiveInlining)]
33          public void Add(TElement element) => _list.Add(element);
34
35          /// <summary>
36          /// <para>Adds an item to the end of the list and return true.</para>
37          /// <para>Добавляет элемент в конец списка и возвращает true.</para>
38          /// </summary>
39          /// <param name="element"><para>Element to add.</para><para>Добавляемый
      ↪   элемент.</para></param>
40          /// <returns>
41          /// <para>True value in any case.</para>
42          /// <para>Значение true в любом случае.</para>
43          /// </returns>
44          [MethodImpl(MethodImplOptions.AggressiveInlining)]
45          public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
46
47          /// <summary>
48          /// <para>Adds a value to the list at the first index and return true.</para>
49          /// <para>Добавляет значение в список по первому индексу и возвращает true.</para>
50          /// </summary>
51          /// <param name="element"><para>Element to add.</para><para>Добавляемый
      ↪   элемент.</para></param>
52          /// <returns>
53          /// <para>True value in any case.</para>
54          /// <para>Значение true в любом случае.</para>
55          /// </returns>
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
      ↪   _list.AddFirstAndReturnTrue(elements);
58
59          /// <summary>
60          /// <para>Adds all elements from other list to this list and returns true.</para>
61          /// <para>Добавляет все элементы из другого списка в этот список и возвращает
      ↪   true.</para>
62          /// </summary>
63          /// <param name="elements"><para>List of values to add.</para><para>Список значений
      ↪   которые необходимо добавить.</para></param>
64          /// <returns>
65          /// <para>True value in any case.</para>
66          /// <para>Значение true в любом случае.</para>
67          /// </returns>
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public bool AddAllAndReturnTrue(IList<TElement> elements) =>
      ↪   _list.AddAllAndReturnTrue(elements);
70
71          /// <summary>
72          /// <para>Adds values to the list skipping the first element.</para>
73          /// <para>Добавляет значения в список пропуская первый элемент.</para>
74          /// </summary>
75          /// <param name="elements"><para>The list of values to add.</para><para>Список значений
      ↪   которые необходимо добавить.</para></param>
76          /// <returns>
77          /// <para>True value in any case.</para>
78          /// <para>Значение true в любом случае.</para>
79          /// </returns>
80          [MethodImpl(MethodImplOptions.AggressiveInlining)]
81          public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
      ↪   _list.AddSkipFirstAndReturnTrue(elements);
```

```
82
83        /// <summary>
84        /// <para>Adds an item to the end of the list and return constant.</para>
85        /// <para>Добавляет элемент в конец списка и возвращает константу.</para>
86        /// </summary>
87        /// <param name="element"><para>Element to add.</para><para>Добавляемый
   ↪    элемент.</para></param>
88        /// <returns>
89        /// <para>Constant value in any case.</para>
90        /// <para>Значение константы в любом случае.</para>
91        /// </returns>
92        [MethodImpl(MethodImplOptions.AggressiveInlining)]
93        public TReturnConstant AddAndReturnConstant(TElement element)
94        {
95            _list.Add(element);
96            return _returnConstant;
97        }
98
99        /// <summary>
100       /// <para>Adds a value to the list at the first index and return constant.</para>
101       /// <para>Добавляет значение в список по первому индексу и возвращает константу.</para>
102       /// </summary>
103       /// <param name="element"><para>Element to add.</para><para>Добавляемый
   ↪    элемент.</para></param>
104       /// <returns>
105       /// <para>Constant value in any case.</para>
106       /// <para>Значение константы в любом случае.</para>
107       /// <returns></returns>
108       [MethodImpl(MethodImplOptions.AggressiveInlining)]
109       public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
110       {
111           _list.AddFirst(elements);
112           return _returnConstant;
113       }
114
115       /// <summary>
116       /// <para>Adds all elements from other list to this list and returns constant.</para>
117       /// <para>Добавляет все элементы из другого списка в этот список и возвращает
   ↪    константу.</para>
118       /// </summary>
119       /// <param name="elements"><para>List of values to add.</para><para>Список значений
   ↪    которые необходимо добавить.</para></param>
120       /// <returns>
121       /// <para>Constant value in any case.</para>
122       /// <para>Значение константы в любом случае.</para>
123       /// <returns>
124       [MethodImpl(MethodImplOptions.AggressiveInlining)]
125       public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
126       {
127           _list.AddAll(elements);
128           return _returnConstant;
129       }
130
131       /// <summary>
132       /// <para>Adds values to the list skipping the first element and return constant
   ↪    value.</para>
133       /// <para>Добавляет значения в список пропуская первый элемент и возвращает значение
   ↪    константы.</para>
134       /// </summary>
135       /// <param name="elements"><para>The list of values to add.</para><para>Список значений
   ↪    которые необходимо добавить.</para></param>
136       /// <returns>
137       /// <para>constant value in any case.</para>
138       /// <para>Значение константы в любом случае.</para>
139       /// </returns>
140       [MethodImpl(MethodImplOptions.AggressiveInlining)]
141       public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
142       {
143           _list.AddSkipFirst(elements);
144           return _returnConstant;
145       }
146   }
147 }
```

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

```
1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
```

```csharp
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
{
    public class CharSegment : Segment<char>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
        ↪  length) { }

        [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();
            }
        }

        [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);
        }

        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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => this;
    }
}
```

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

```csharp
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
{
    public class Segment<T> : IEquatable<Segment<T>>, IList<T>
    {
        public IList<T> Base
        {
```

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        public int Offset
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        public int Length
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Segment(IList<T> @base, int offset, int length)
        {
            Base = @base;
            Offset = offset;
            Length = length;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => this.GenerateHashCode();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Equals(Segment<T> other) => this.EqualTo(other);

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

        #region IList

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

        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Length;
        }

        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Insert(int index, T item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(T item) => throw new NotSupportedException();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();
```

```csharp
95
96          [MethodImpl(MethodImplOptions.AggressiveInlining)]
97          public bool Contains(T item) => IndexOf(item) >= 0;
98
99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100         public void CopyTo(T[] array, int arrayIndex)
101         {
102             for (var i = 0; i < Length; i++)
103             {
104                 array.Add(ref arrayIndex, this[i]);
105             }
106         }
107
108         [MethodImpl(MethodImplOptions.AggressiveInlining)]
109         public bool Remove(T item) => throw new NotSupportedException();
110
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public IEnumerator<T> GetEnumerator()
113         {
114             for (var i = 0; i < Length; i++)
115             {
116                 yield return this[i];
117             }
118         }
119
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
122
123         #endregion
124     }
125 }
```

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

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      public abstract class AllSegmentsWalkerBase
6      {
7          public static readonly int DefaultMinimumStringSegmentLength = 2;
8      }
9  }
```

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
9          where TSegment : Segment<T>
10     {
11         private readonly int _minimumStringSegmentLength;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
       ↪   _minimumStringSegmentLength = minimumStringSegmentLength;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public virtual void WalkAll(IList<T> elements)
21         {
22             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
               ↪   offset <= maxOffset; offset++)
23             {
24                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                   ↪   offset; length <= maxLength; length++)
25                 {
26                     Iteration(CreateSegment(elements, offset, length));
27                 }
28             }
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
```

```
33
34          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35          protected abstract void Iteration(TSegment segment);
36      }
37  }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
9      {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
            ↪   => new Segment<T>(elements, offset, length);
12      }
13  }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Segments.Walkers
6  {
7      public static class AllSegmentsWalkerExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
            ↪   walker.WalkAll(@string.ToCharArray());
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
            ↪   string @string) where TSegment : Segment<char> =>
            ↪   walker.WalkAll(@string.ToCharArray());
14      }
15  }
```

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      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
        ↪   DuplicateSegmentsWalkerBase<T, TSegment>
10          where TSegment : Segment<T>
11      {
12          public static readonly bool DefaultResetDictionaryOnEachWalk;
13
14          private readonly bool _resetDictionaryOnEachWalk;
15          protected IDictionary<TSegment, long> Dictionary;
16
17          [MethodImpl(MethodImplOptions.AggressiveInlining)]
18          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪   dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
19              : base(minimumStringSegmentLength)
20          {
21              Dictionary = dictionary;
22              _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
23          }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪   dictionary, int minimumStringSegmentLength) : this(dictionary,
            ↪   minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
            ↪   dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
            ↪   DefaultResetDictionaryOnEachWalk) { }
30
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
            bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
            Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
            { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
            this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [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);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override long GetSegmentFrequency(TSegment segment) =>
            Dictionary.GetOrDefault(segment);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
            Dictionary[segment] = frequency;
    }
}
```

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

```csharp
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
{
    public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
        DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
            base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            dictionary, int minimumStringSegmentLength) : base(dictionary,
            minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
            DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
            bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
            resetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
            base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
    }
}
```

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Segments.Walkers
{
    public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
      TSegment>
        where TSegment : Segment<T>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
          base(minimumStringSegmentLength) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Iteration(TSegment segment)
        {
            var frequency = GetSegmentFrequency(segment);
            if (frequency == 1)
            {
                OnDublicateFound(segment);
            }
            SetSegmentFrequency(segment, frequency + 1);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void OnDublicateFound(TSegment segment);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract long GetSegmentFrequency(TSegment segment);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
    }
}
```

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

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

namespace Platform.Collections.Segments.Walkers
{
    public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
      Segment<T>>
    {
    }
}
```

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

```csharp
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
{
    public static class ISetExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
          set.Remove(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
        {
            set.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
        {
            AddFirst(set, elements);
```

```
27              return true;
28          }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
    ↪       set.Add(elements[0]);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
35          {
36              set.AddAll(elements);
37              return true;
38          }
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public static void AddAll<T>(this ISet<T> set, IList<T> elements)
42          {
43              for (var i = 0; i < elements.Count; i++)
44              {
45                  set.Add(elements[i]);
46              }
47          }
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
51          {
52              set.AddSkipFirst(elements);
53              return true;
54          }
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
    ↪       set.AddSkipFirst(elements, 1);
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
61          {
62              for (var i = skip; i < elements.Count; i++)
63              {
64                  set.Add(elements[i]);
65              }
66          }
67
68          [MethodImpl(MethodImplOptions.AggressiveInlining)]
69          public static bool DoNotContains<T>(this ISet<T> set, T element) =>
    ↪       !set.Contains(element);
70      }
71  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Collections.Sets
7   {
8       public class SetFiller<TElement, TReturnConstant>
9       {
10          protected readonly ISet<TElement> _set;
11          protected readonly TReturnConstant _returnConstant;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
15          {
16              _set = set;
17              _returnConstant = returnConstant;
18          }
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          public SetFiller(ISet<TElement> set) : this(set, default) { }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public void Add(TElement element) => _set.Add(element);
25
26          [MethodImpl(MethodImplOptions.AggressiveInlining)]
27          public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);
28
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
            public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
            ↪   _set.AddFirstAndReturnTrue(elements);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool AddAllAndReturnTrue(IList<TElement> elements) =>
            ↪   _set.AddAllAndReturnTrue(elements);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
            ↪   _set.AddSkipFirstAndReturnTrue(elements);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TReturnConstant AddAndReturnConstant(TElement element)
            {
                _set.Add(element);
                return _returnConstant;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
            {
                _set.AddFirst(elements);
                return _returnConstant;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
            {
                _set.AddAll(elements);
                return _returnConstant;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
            {
                _set.AddSkipFirst(elements);
                return _returnConstant;
            }
        }
    }
```

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

```csharp
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
{
    public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
    {
        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Count <= 0;
        }
    }
}
```

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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public interface IStack<TElement>
    {
        bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Push(TElement element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Pop();
```

```
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          TElement Peek();
23      }
24  }
```

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

```
1   using System.Runtime.CompilerServices;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Stacks
6   {
7       public static class IStackExtensions
8       {
9           [MethodImpl(MethodImplOptions.AggressiveInlining)]
10          public static void Clear<T>(this IStack<T> stack)
11          {
12              while (!stack.IsEmpty)
13              {
14                  _ = stack.Pop();
15              }
16          }
17
18          [MethodImpl(MethodImplOptions.AggressiveInlining)]
19          public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
            ↪   stack.Pop();
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
            ↪   stack.Peek();
23      }
24  }
```

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

```
1   using Platform.Interfaces;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Stacks
6   {
7       public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
8       {
9       }
10  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Collections.Stacks
7   {
8       public static class StackExtensions
9       {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
            ↪   default;
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
            ↪   : default;
15      }
16  }
```

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

```
1   using System;
2   using System.Globalization;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Collections
8   {
9       public static class StringExtensions
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public static string CapitalizeFirstLetter(this string @string)
```

```
13                   {
14                       if (string.IsNullOrWhiteSpace(@string))
15                       {
16                           return @string;
17                       }
18                       var chars = @string.ToCharArray();
19                       for (var i = 0; i < chars.Length; i++)
20                       {
21                           var category = char.GetUnicodeCategory(chars[i]);
22                           if (category == UnicodeCategory.UppercaseLetter)
23                           {
24                               return @string;
25                           }
26                           if (category == UnicodeCategory.LowercaseLetter)
27                           {
28                               chars[i] = char.ToUpper(chars[i]);
29                               return new string(chars);
30                           }
31                       }
32                       return @string;
33                   }
34
35                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
36                   public static string Truncate(this string @string, int maxLength) =>
    ↪        string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
    ↪        Math.Min(@string.Length, maxLength));
37
38                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
39                   public static string TrimSingle(this string @string, char charToTrim)
40                   {
41                       if (!string.IsNullOrEmpty(@string))
42                       {
43                           if (@string.Length == 1)
44                           {
45                               if (@string[0] == charToTrim)
46                               {
47                                   return "";
48                               }
49                               else
50                               {
51                                   return @string;
52                               }
53                           }
54                           else
55                           {
56                               var left = 0;
57                               var right = @string.Length - 1;
58                               if (@string[left] == charToTrim)
59                               {
60                                   left++;
61                               }
62                               if (@string[right] == charToTrim)
63                               {
64                                   right--;
65                               }
66                               return @string.Substring(left, right - left + 1);
67                           }
68                       }
69                       else
70                       {
71                           return @string;
72                       }
73                   }
74               }
75       }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   // ReSharper disable ForCanBeConvertedToForeach
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Collections.Trees
8   {
9       public class Node
10      {
11          private Dictionary<object, Node> _childNodes;
12
13          public object Value
```

```csharp
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public Dictionary<object, Node> ChildNodes
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node(object value) => Value = value;

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool ContainsChild(params object[] keys) => GetChild(keys) != null;

        [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;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key) => AddChild(key, new Node(null));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, object value) => AddChild(key, new Node(value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, Node child)
        {
            ChildNodes.Add(key, child);
            return child;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(params object[] keys) => SetChildValue(null, keys);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(object key) => SetChildValue(null, key);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChildValue(object value, params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node = SetChildValue(value, keys[i]);
            }
            node.Value = value;
            return node;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
94        public Node SetChildValue(object value, object key)
95        {
96            if (!ChildNodes.TryGetValue(key, out Node child))
97            {
98                child = AddChild(key, value);
99            }
100           child.Value = value;
101           return child;
102       }
103   }
104 }
```

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

```
1  using Xunit;
2  using Platform.Collections.Arrays;
3
4  namespace Platform.Collections.Tests
5  {
6      public class ArrayTests
7      {
8          [Fact]
9          public void GetElementTest()
10         {
11             var nullArray = (int[])null;
12             Assert.Equal(0, nullArray.GetElementOrDefault(1));
13             Assert.False(nullArray.TryGetElement(1, out int element));
14             Assert.Equal(0, element);
15             var array = new int[] { 1, 2, 3 };
16             Assert.Equal(3, array.GetElementOrDefault(2));
17             Assert.True(array.TryGetElement(2, out element));
18             Assert.Equal(3, element);
19             Assert.Equal(0, array.GetElementOrDefault(10));
20             Assert.False(array.TryGetElement(10, out element));
21             Assert.Equal(0, element);
22         }
23     }
24 }
```

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

```
1  using System;
2  using System.Collections;
3  using Xunit;
4  using Platform.Random;
5
6  namespace Platform.Collections.Tests
7  {
8      public static class BitStringTests
9      {
10         [Fact]
11         public static void BitGetSetTest()
12         {
13             const int n = 250;
14             var bitArray = new BitArray(n);
15             var bitString = new BitString(n);
16             for (var i = 0; i < n; i++)
17             {
18                 var value = RandomHelpers.Default.NextBoolean();
19                 bitArray.Set(i, value);
20                 bitString.Set(i, value);
21                 Assert.Equal(value, bitArray.Get(i));
22                 Assert.Equal(value, bitString.Get(i));
23             }
24         }
25
26         [Fact]
27         public static void BitVectorNotTest()
28         {
29             TestToOperationsWithSameMeaning((x, y, w, v) =>
30             {
31                 x.VectorNot();
32                 w.Not();
33             });
34         }
35
36         [Fact]
37         public static void BitParallelNotTest()
38         {
39             TestToOperationsWithSameMeaning((x, y, w, v) =>
40             {
```

```csharp
41              x.ParallelNot();
42              w.Not();
43          });
44      }
45
46      [Fact]
47      public static void BitParallelVectorNotTest()
48      {
49          TestToOperationsWithSameMeaning((x, y, w, v) =>
50          {
51              x.ParallelVectorNot();
52              w.Not();
53          });
54      }
55
56      [Fact]
57      public static void BitVectorAndTest()
58      {
59          TestToOperationsWithSameMeaning((x, y, w, v) =>
60          {
61              x.VectorAnd(y);
62              w.And(v);
63          });
64      }
65
66      [Fact]
67      public static void BitParallelAndTest()
68      {
69          TestToOperationsWithSameMeaning((x, y, w, v) =>
70          {
71              x.ParallelAnd(y);
72              w.And(v);
73          });
74      }
75
76      [Fact]
77      public static void BitParallelVectorAndTest()
78      {
79          TestToOperationsWithSameMeaning((x, y, w, v) =>
80          {
81              x.ParallelVectorAnd(y);
82              w.And(v);
83          });
84      }
85
86      [Fact]
87      public static void BitVectorOrTest()
88      {
89          TestToOperationsWithSameMeaning((x, y, w, v) =>
90          {
91              x.VectorOr(y);
92              w.Or(v);
93          });
94      }
95
96      [Fact]
97      public static void BitParallelOrTest()
98      {
99          TestToOperationsWithSameMeaning((x, y, w, v) =>
100         {
101             x.ParallelOr(y);
102             w.Or(v);
103         });
104     }
105
106     [Fact]
107     public static void BitParallelVectorOrTest()
108     {
109         TestToOperationsWithSameMeaning((x, y, w, v) =>
110         {
111             x.ParallelVectorOr(y);
112             w.Or(v);
113         });
114     }
115
116     [Fact]
117     public static void BitVectorXorTest()
118     {
```

```
119         TestToOperationsWithSameMeaning((x, y, w, v) =>
120         {
121             x.VectorXor(y);
122             w.Xor(v);
123         });
124     }
125
126     [Fact]
127     public static void BitParallelXorTest()
128     {
129         TestToOperationsWithSameMeaning((x, y, w, v) =>
130         {
131             x.ParallelXor(y);
132             w.Xor(v);
133         });
134     }
135
136     [Fact]
137     public static void BitParallelVectorXorTest()
138     {
139         TestToOperationsWithSameMeaning((x, y, w, v) =>
140         {
141             x.ParallelVectorXor(y);
142             w.Xor(v);
143         });
144     }
145
146     private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
    ↪ BitString, BitString> test)
147     {
148         const int n = 5654;
149         var x = new BitString(n);
150         var y = new BitString(n);
151         while (x.Equals(y))
152         {
153             x.SetRandomBits();
154             y.SetRandomBits();
155         }
156         var w = new BitString(x);
157         var v = new BitString(y);
158         Assert.False(x.Equals(y));
159         Assert.False(w.Equals(v));
160         Assert.True(x.Equals(w));
161         Assert.True(y.Equals(v));
162         test(x, y, w, v);
163         Assert.True(x.Equals(w));
164     }
165     }
166 }
```

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

```
1  using Xunit;
2  using Platform.Collections.Segments;
3
4  namespace Platform.Collections.Tests
5  {
6      public static class CharsSegmentTests
7      {
8          [Fact]
9          public static void GetHashCodeEqualsTest()
10         {
11             const string testString = "test test";
12             var testArray = testString.ToCharArray();
13             var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
14             var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
15             Assert.Equal(firstHashCode, secondHashCode);
16         }
17
18         [Fact]
19         public static void EqualsTest()
20         {
21             const string testString = "test test";
22             var testArray = testString.ToCharArray();
23             var first = new CharSegment(testArray, 0, 4);
24             var second = new CharSegment(testArray, 5, 4);
25             Assert.True(first.Equals(second));
26         }
27     }
28 }
```

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

```csharp
1   using System.Collections.Generic;
2   using Xunit;
3   using Platform.Collections.Lists;
4
5
6   namespace Platform.Collections.Tests
7   {
8       public class ListTests
9       {
10          [Fact]
11          public void GetElementTest()
12          {
13              var nullList = (IList<int>)null;
14              Assert.Equal(0, nullList.GetElementOrDefault(1));
15              Assert.False(nullList.TryGetElement(1, out int element));
16              Assert.Equal(0, element);
17              var list = new List<int>() { 1, 2, 3 };
18              Assert.Equal(3, list.GetElementOrDefault(2));
19              Assert.True(list.TryGetElement(2, out element));
20              Assert.Equal(3, element);
21              Assert.Equal(0, list.GetElementOrDefault(10));
22              Assert.False(list.TryGetElement(10, out element));
23              Assert.Equal(0, element);
24          }
25      }
26  }
```

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

```csharp
1   using Xunit;
2
3   namespace Platform.Collections.Tests
4   {
5       public static class StringTests
6       {
7           [Fact]
8           public static void CapitalizeFirstLetterTest()
9           {
10              Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
11              Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
12              Assert.Equal("  Hello", "  hello".CapitalizeFirstLetter());
13          }
14
15          [Fact]
16          public static void TrimSingleTest()
17          {
18              Assert.Equal("", "'".TrimSingle('\''));
19              Assert.Equal("", "''".TrimSingle('\''));
20              Assert.Equal("hello", "'hello'".TrimSingle('\''));
21              Assert.Equal("hello", "hello'".TrimSingle('\''));
22              Assert.Equal("hello", "'hello".TrimSingle('\''));
23          }
24      }
25  }
```

Index