

LinksPlatform's Platform.Collections Class Library

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Arrays
7 {
8     public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
9     {
10         protected readonly TReturnConstant _returnConstant;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
14             ↪ base(array, offset) => _returnConstant = returnConstant;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
18             ↪ returnConstant) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public TReturnConstant AddAndReturnConstant(TElement element) =>
22             ↪ _array.AddAndReturnConstant(ref _position, element, _returnConstant);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements) =>
26             ↪ _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements) =>
30             ↪ _array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements) =>
34             ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
35     }
36 }
```

1.2 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].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.Arrays
7 {
8     public class ArrayFiller<TElement>
9     {
10         protected readonly TElement[] _array;
11         protected long _position;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ArrayFiller(TElement[] array, long offset)
15         {
16             _array = array;
17             _position = offset;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ArrayFiller(TElement[] array) : this(array, 0) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _array[_position++] = element;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
28             ↪ _position, element, true);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
32             ↪ _array.AddFirstAndReturnConstant(ref _position, elements, true);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
36             ↪ _array.AddAllAndReturnConstant(ref _position, elements, true);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
40             ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
41     }
42 }
```

```

36         public bool AddSkipFirstAndReturnTrue(ICollection<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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static class ArrayPool
8      {
9          public static readonly int DefaultSizesAmount = 512;
10         public static readonly int DefaultMaxArraysPerSize = 32;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
17     }
18 }

```

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
13     ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
14     /// </remarks>
15     public class ArrayPool<T>
16     {
17         // May be use Default class for that later.
18         [ThreadStatic]
19         private static ArrayPool<T> _threadInstance;
20         internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
           ↪ ArrayPool<T>());
21
22         private readonly int _maxArraysPerSize;
23         private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
           ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public Disposable<T[]> AllocatedDisposable(long size) => (Allocate(size), Free);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
36         {
37             var destination = AllocatedDisposable(size);
38             T[] sourceArray = source;
39             if (!sourceArray.IsNullOrEmpty())
40             {
41                 T[] destinationArray = destination;
42                 Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
                   ↪ sourceArray.LongLength);
43                 source.Dispose();
44             }
45             return destination;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public virtual void Clear() => _pool.Clear();
50     }
51 }

```

```

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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static unsafe class CharArrayExtensions
8      {
9          /// <remarks>
10         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12         /// </remarks>
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static int GenerateHashCode(this char[] array, int offset, int length)
15         {
16             var hashSeed = 5381;
17             var hashAccumulator = hashSeed;
18             fixed (char* arrayPointer = &array[offset])
19             {
20                 for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
21                     ↪ < last; charPointer++)
22                 {
23                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
24                 }
25             }
26             return hashAccumulator + (hashSeed * 1566083941);
27         }
28
29         /// <remarks>
30         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
31         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
32         /// </remarks>
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
35             ↪ right, int rightOffset)

```

```

32 {
33     fixed (char* leftPointer = &left[leftOffset])
34     {
35         fixed (char* rightPointer = &right[rightOffset])
36         {
37             char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
38             if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
39                 ↪ rightPointerCopy, ref length))
40             {
41                 return false;
42             }
43             CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
44                 ↪ ref length);
45             return length <= 0;
46         }
47     }
48 }
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
51     ↪ int length)
52 {
53     while (length >= 10)
54     {
55         if ((* (int*)left != * (int*)right)
56             || (* (int*)(left + 2) != * (int*)(right + 2))
57             || (* (int*)(left + 4) != * (int*)(right + 4))
58             || (* (int*)(left + 6) != * (int*)(right + 6))
59             || (* (int*)(left + 8) != * (int*)(right + 8)))
60         {
61             return false;
62         }
63         left += 10;
64         right += 10;
65         length -= 10;
66     }
67     return true;
68 }
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
71     ↪ int length)
72 {
73     // This depends on the fact that the String objects are
74     // always zero terminated and that the terminating zero is not included
75     // in the length. For odd string sizes, the last compare will include
76     // the zero terminator.
77     while (length > 0)
78     {
79         if ((* (int*)left != * (int*)right)
80             {
81                 break;
82             }
83         left += 2;
84         right += 2;
85         length -= 2;
86     }
87 }

```

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

```

```

57  /// <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>
58  /// </summary>
59  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
60  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
61  /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
    → для сравнения.</para></param>
62  /// <param name="element"><para>Passing the argument by reference, if successful, it
    → will take the value array[index] otherwise default value.</para><para>Передаёт
    → аргумент по ссылке, в случае успеха он примет значение array[index] в противном
    → случае значение по умолчанию.</para></param>
63  /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
    → в противном случае false</para></returns>
64  [MethodImpl(MethodImplOptions.AggressiveInlining)]
65  public static bool TryGetElement<T>(this T[] array, long index, out T element)
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
116   cref="NotImplementedException"/> exception is thrown, but if the value of the shift
117   variable is 0 - an exact copy of the array is returned. Otherwise, an array is
118   returned with the shift of the elements.</para>
119 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
120   cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
121   возвращается точная копия массива. Иначе возвращается массив со сдвигом
122   элементов.</para>
123 /// </returns>
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static IList<T> ShiftRight<T>(this T[] array, long shift)
126 {
127     if (shift < 0)
128     {
129         throw new NotImplementedException();
130     }
131     if (shift == 0)
132     {
133         return array.Clone<T>();
134     }
135     else
136     {
137         var restrictions = new T[array.LongLength + shift];
138         Array.Copy(array, 0L, restrictions, shift, array.LongLength);
139         return restrictions;
140     }
141 }
142
143 /// <summary>
144 /// <para>Adding in array the passed element at the specified position and increments
145   position value by one.</para>
146 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
147   значение position на единицу.</para>
148 /// </summary>
149 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
150   массива.</para></typeparam>
151 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
152   который необходимо добавить элемент.</para></param>
153 /// <param name="position"><para>A reference to the position of type int where the
154   element will be added.</para><para>Ссылка на позицию типа int, в которую будет
155   добавлен элемент.</para></param>
156 /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
157   который нужно добавить в массив.</para></param>
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static void Add<T>(this T[] array, ref int position, T element) =>
160   array[position++] = element;
161
162 /// <summary>
163 /// <para>Adding in array the passed element at the specified position and increments
164   position value by one.</para>
165 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
166   значение position на единицу.</para>
167 /// </summary>
168 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
169   массива.</para></typeparam>
170 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
171   который необходимо добавить элемент.</para></param>
172 /// <param name="position"><para>A reference to the position of type long where the
173   element will be added.</para><para>Ссылка на позицию типа long, в которую будет
174   добавлен элемент.</para></param>
175 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
176   который необходимо добавить в массив.</para></param>
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 public static void Add<T>(this T[] array, ref long position, T element) =>
179   array[position++] = element;
180
181 /// <summary>
182 /// <para>Adding in array the passed element, at the specified position, increments
183   position value by one and returns the value of the passed constant.</para>
184 /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
185   значение position на единицу и возвращает значение переданной константы.</para>
186 /// </summary>
187 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
188   массива.</para></typeparam>
189 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
190   возвращаемой константы.</para></typeparam>

```

```

165  /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
166  /// <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>
167  /// <param name="element"><para>The element to add to the array.</para><para>Элемент
    → который необходимо добавить в массив.</para></param>
168  /// <param name="returnConstant"><para>The constant value that will be
    → returned.</para><para>Значение константы, которое будет возвращено.</para></param>
169  /// <returns>
170  /// <para>The constant value passed as an argument.</para>
171  /// <para>Значение константы, переданное в качестве аргумента.</para>
172  /// </returns>
173  [MethodImpl(MethodImplOptions.AggressiveInlining)]
174  public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, TElement element, TReturnConstant
    → returnConstant)
175  {
176      array.Add(ref position, element);
177      return returnConstant;
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 /// <summary>
212 /// <para>Adding in array all elements from the passed collection, at the specified
213   → 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>
220 /// <param name="elements"><para>List, whose elements will be added to the
   → array.</para><para>Список, элементы которого будут добавлены 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 добавлены 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 добавлены 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 }
270
271 /// <summary>
272 /// <para>Adding in array all elements of the collection, skipping the first position
    → and increments position value by one.</para>
273 /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию и
    → увеличивает значение position на единицу.</para>
274 /// </summary>
275 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
276 /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    → необходимо добавить элементы.</para></param>
277 /// <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>
278 /// <param name="elements"><para>List, whose elements will be added to the
    → array.</para><para>Список, элементы которого будут добавлены в
    → массив.</para></param>
279 [MethodImpl(MethodImplOptions.AggressiveInlining)]
280 public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
    → => array.AddSkipFirst(ref position, elements, 1);
281
282 /// <summary>
283 /// <para>Adding in array all but the first element, skipping a specified number of
    → positions and increments position value by one.</para>
284 /// <para>Добавляет в массив все элементы коллекции, кроме первого, пропуская
    → определенное количество позиций и увеличивает значение position на единицу.</para>
285 /// </summary>
286 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
287 /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    → необходимо добавить элементы.</para></param>
288 /// <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>
289 /// <param name="elements"><para>List, whose elements will be added to the
    → array.</para><para>Список, элементы которого будут добавлены в
    → массив.</para></param>
290 /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
    → пропускаемых элементов.</para></param>
291 [MethodImpl(MethodImplOptions.AggressiveInlining)]
292 public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
    → int skip)
293 {
294     for (var i = skip; i < elements.Count; i++)
295     {
296         array.Add(ref position, elements[i]);
297     }
298 }
299 }
300 }

```

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

```

1 using System;
2 using System.Collections.Concurrent;
3 using System.Collections.Generic;
4 using System.Numerics;

```

```

5 using System.Runtime.CompilerServices;
6 using System.Threading.Tasks;
7 using Platform.Exceptions;
8 using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>
16     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
17     ///   ↳ 64 бит в массиве значений.
18     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
19     ///   ↳ байт в 8 байт.
20     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
21     ///   ↳ помощью которой можно быстро
22     /// проверять есть ли значения непосредственно далее (ниже по уровню).
23     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
24     /// </remarks>
25     public class BitString : IEquatable<BitString>
26     {
27         private static readonly byte[] [] _bitsSetIn16Bits;
28         private long[] _array;
29         private long _length;
30         private long _minPositiveWord;
31         private long _maxPositiveWord;
32
33         public bool this[long index]
34         {
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             get => Get(index);
37             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38             set => Set(index, value);
39         }
40
41         public long Length
42         {
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             get => _length;
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             set
47             {
48                 if (_length == value)
49                 {
50                     return;
51                 }
52                 Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
53                 // Currently we never shrink the array
54                 if (value > _length)
55                 {
56                     var words = GetWordsCountFromIndex(value);
57                     var oldWords = GetWordsCountFromIndex(_length);
58                     if (words > _array.LongLength)
59                     {
60                         var copy = new long[words];
61                         Array.Copy(_array, copy, _array.LongLength);
62                         _array = copy;
63                     }
64                     else
65                     {
66                         // What is going on here?
67                         Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
68                     }
69                     // What is going on here?
70                     var mask = (int)(_length % 64);
71                     if (mask > 0)
72                     {
73                         _array[oldWords - 1] &= (1L << mask) - 1;
74                     }
75                 }
76                 else
77                 {
78                     // Looks like minimum and maximum positive words are not updated
79                     throw new NotImplementedException();
80                 }
81                 _length = value;
82             }
83         }
84     }
85 }

```

```

81
82 #region Constructors
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 static BitString()
86 {
87     _bitsSetIn16Bits = new byte[65536][];
88     int i, c, k;
89     byte bitIndex;
90     for (i = 0; i < 65536; i++)
91     {
92         // Calculating size of array (number of positive bits)
93         for (c = 0, k = 1; k <= 65536; k <= 1)
94         {
95             if ((i & k) == k)
96             {
97                 c++;
98             }
99         }
100         var array = new byte[c];
101         // Adding positive bits indices into array
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(OL, _array.LongLength, _array.LongLength /
    ↪ threads);
166     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    ↪ MaxDegreeOfParallelism = threads }, range =>
    {
167         var maximum = range.Item2;
168         for (var i = range.Item1; i < maximum; i++)
169         {
170             _array[i] = ~_array[i];
171         }
172     });
173     MarkBordersAsAllBitsSet();
174     TryShrinkBorders();
175     return this;
176 }
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public BitString VectorNot()
180 {
181     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
182     {
183         return Not();
184     }
185     var step = Vector<long>.Count;
186     if (_array.Length < step)
187     {
188         return Not();
189     }
190     VectorNotLoop(_array, step, 0, _array.Length);
191     MarkBordersAsAllBitsSet();
192     TryShrinkBorders();
193     return this;
194 }
195
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 public BitString ParallelVectorNot()
198 {
199     var threads = Environment.ProcessorCount / 2;
200     if (threads <= 1)
201     {
202         return VectorNot();
203     }
204     if (!Vector.IsHardwareAccelerated)
205     {
206         return ParallelNot();
207     }
208     var step = Vector<long>.Count;
209     if (_array.Length < (step * threads))
210     {
211         return VectorNot();
212     }
213     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
214     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
    ↪ range.Item1, range.Item2));
215     MarkBordersAsAllBitsSet();
216     TryShrinkBorders();
217     return this;
218 }
219
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
222 {
223     var i = start;
224     var range = maximum - start - 1;
225     var stop = range - (range % step);
226     for (; i < stop; i += step)
227     {
228         (~new Vector<long>(array, i)).CopyTo(array, i);
229     }
230     for (; i < maximum; i++)
231     {
232         array[i] = ~array[i];
233     }

```

```

234     }
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public BitString And(BitString other)
239 {
240     EnsureBitStringHasTheSameSize(other, nameof(other));
241     GetCommonOuterBorders(this, other, out long from, out long to);
242     var otherArray = other._array;
243     for (var i = from; i <= to; i++)
244     {
245         _array[i] &= otherArray[i];
246         RefreshBordersByWord(i);
247     }
248     return this;
249 }
250
251 [MethodImpl(MethodImplOptions.AggressiveInlining)]
252 public BitString ParallelAnd(BitString other)
253 {
254     var threads = Environment.ProcessorCount / 2;
255     if (threads <= 1)
256     {
257         return And(other);
258     }
259     EnsureBitStringHasTheSameSize(other, nameof(other));
260     GetCommonOuterBorders(this, other, out long from, out long to);
261     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
262     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
263         ↪ MaxDegreeOfParallelism = threads }, range =>
264     {
265         var maximum = range.Item2;
266         for (var i = range.Item1; i < maximum; i++)
267         {
268             _array[i] &= other._array[i];
269         }
270     });
271     MarkBordersAsAllBitsSet();
272     TryShrinkBorders();
273     return this;
274 }
275
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 public BitString VectorAnd(BitString other)
278 {
279     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
280     {
281         return And(other);
282     }
283     var step = Vector<long>.Count;
284     if (_array.Length < step)
285     {
286         return And(other);
287     }
288     EnsureBitStringHasTheSameSize(other, nameof(other));
289     GetCommonOuterBorders(this, other, out int from, out int to);
290     VectorAndLoop(_array, other._array, step, from, to + 1);
291     MarkBordersAsAllBitsSet();
292     TryShrinkBorders();
293     return this;
294 }
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 public BitString ParallelVectorAnd(BitString other)
298 {
299     var threads = Environment.ProcessorCount / 2;
300     if (threads <= 1)
301     {
302         return VectorAnd(other);
303     }
304     if (!Vector.IsHardwareAccelerated)
305     {
306         return ParallelAnd(other);
307     }
308     var step = Vector<long>.Count;
309     if (_array.Length < (step * threads))
310     {
311         return VectorAnd(other);

```

```

311     }
312     EnsureBitStringHasTheSameSize(other, nameof(other));
313     GetCommonOuterBorders(this, other, out int from, out int to);
314     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
315     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
        ↪ step, range.Item1, range.Item2));
316     MarkBordersAsAllBitsSet();
317     TryShrinkBorders();
318     return this;
319 }
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
    ↪ int maximum)
323 {
324     var i = start;
325     var range = maximum - start - 1;
326     var stop = range - (range % step);
327     for (; i < stop; i += step)
328     {
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     }
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 {
415         ↪ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
416         ↪ step, range.Item1, range.Item2));
417     MarkBordersAsAllBitsSet();
418     TryShrinkBorders();
419     return this;
420 }
421
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
424     ↪ int maximum)
425 {
426     var i = start;
427     var range = maximum - start - 1;
428     var stop = range - (range % step);
429     for (; i < stop; i += step)
430     {
431         (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
432     }
433     for (; i < maximum; i++)
434     {
435         array[i] |= otherArray[i];
436     }
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 public BitString Xor(BitString other)
441 {
442     EnsureBitStringHasTheSameSize(other, nameof(other));
443     GetCommonOuterBorders(this, other, out long from, out long to);
444     for (var i = from; i <= to; i++)
445     {
446         _array[i] ^= other._array[i];
447         RefreshBordersByWord(i);
448     }
449     return this;
450 }
451
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 public BitString ParallelXor(BitString other)
454 {
455     var threads = Environment.ProcessorCount / 2;
456     if (threads <= 1)
457     {
458         return Xor(other);
459     }
460     EnsureBitStringHasTheSameSize(other, nameof(other));
461     GetCommonOuterBorders(this, other, out long from, out long to);
462     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);

```



```

460 Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
461     ↪ MaxDegreeOfParallelism = threads }, range =>
462 {
463     var maximum = range.Item2;
464     for (var i = range.Item1; i < maximum; i++)
465     {
466         _array[i] ^= other._array[i];
467     }
468 });
469 MarkBordersAsAllBitsSet();
470 TryShrinkBorders();
471 return this;
472 }
473
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public BitString VectorXor(BitString other)
476 {
477     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
478     {
479         return Xor(other);
480     }
481     var step = Vector<long>.Count;
482     if (_array.Length < step)
483     {
484         return Xor(other);
485     }
486     EnsureBitStringHasTheSameSize(other, nameof(other));
487     GetCommonOuterBorders(this, other, out int from, out int to);
488     VectorXorLoop(_array, other._array, step, from, to + 1);
489     MarkBordersAsAllBitsSet();
490     TryShrinkBorders();
491     return this;
492 }
493
494 [MethodImpl(MethodImplOptions.AggressiveInlining)]
495 public BitString ParallelVectorXor(BitString other)
496 {
497     var threads = Environment.ProcessorCount / 2;
498     if (threads <= 1)
499     {
500         return VectorXor(other);
501     }
502     if (!Vector.IsHardwareAccelerated)
503     {
504         return ParallelXor(other);
505     }
506     var step = Vector<long>.Count;
507     if (_array.Length < (step * threads))
508     {
509         return VectorXor(other);
510     }
511     EnsureBitStringHasTheSameSize(other, nameof(other));
512     GetCommonOuterBorders(this, other, out int from, out int to);
513     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
514     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
515         ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
516         ↪ step, range.Item1, range.Item2));
517     MarkBordersAsAllBitsSet();
518     TryShrinkBorders();
519     return this;
520 }
521
522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
523 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
524     ↪ int maximum)
525 {
526     var i = start;
527     var range = maximum - start - 1;
528     var stop = range - (range % step);
529     for (; i < stop; i += step)
530     {
531         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
532     }
533     for (; i < maximum; i++)
534     {
535         array[i] ^= otherArray[i];
536     }
537 }

```

```

534 [MethodImpl(MethodImplOptions.AggressiveInlining)]
535 private void RefreshBordersByWord(long wordIndex)
536 {
537     if (_array[wordIndex] == 0)
538     {
539         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
540         {
541             _minPositiveWord++;
542         }
543         if (wordIndex == _maxPositiveWord && wordIndex != 0)
544         {
545             _maxPositiveWord--;
546         }
547     }
548 }
549 else
550 {
551     if (wordIndex < _minPositiveWord)
552     {
553         _minPositiveWord = wordIndex;
554     }
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)
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);
713         }
714     }
715     return result;
716 }
717
718 [MethodImpl(MethodImplOptions.AggressiveInlining)]
719 public long GetFirstSetBitIndex()
720 {
721     var i = _minPositiveWord;
722     var word = _array[i];
723     if (word != 0)
724     {
725         return GetFirstSetBitForWord(i, word);
726     }
727     return -1;
728 }
729
730 [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 public long GetLastSetBitIndex()
732 {
733     var i = _maxPositiveWord;
734     var word = _array[i];
735     if (word != 0)
736     {
737         return GetLastSetBitForWord(i, word);
738     }
739     return -1;
740 }
741
742 [MethodImpl(MethodImplOptions.AggressiveInlining)]
743 public long CountSetBits()
744 {
745     var total = 0L;
746     GetBorders(out long from, out long to);
747     for (var i = from; i <= to; i++)
748     {
749         var word = _array[i];
750         if (word != 0)
751         {
752             total += CountSetBitsForWord(word);
753         }
754     }
755     return total;
756 }
757
758 [MethodImpl(MethodImplOptions.AggressiveInlining)]
759 public bool HaveCommonBits(BitString other)
760 {
761     EnsureBitStringHasTheSameSize(other, nameof(other));
762     GetCommonInnerBorders(this, other, out long from, out long to);
763     var otherArray = other._array;
764     for (var i = from; i <= to; i++)
765     {
766         var left = _array[i];
767         var right = otherArray[i];
768         if (left != 0 && right != 0 && (left & right) != 0)
769         {
770             return true;

```

```

771     }
772 }
773 return false;
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 public long CountCommonBits(BitString other)
778 {
779     EnsureBitStringHasTheSameSize(other, nameof(other));
780     GetCommonInnerBorders(this, other, out long from, out long to);
781     var total = 0L;
782     var otherArray = other._array;
783     for (var i = from; i <= to; i++)
784     {
785         var left = _array[i];
786         var right = otherArray[i];
787         var combined = left & right;
788         if (combined != 0)
789         {
790             total += CountSetBitsForWord(combined);
791         }
792     }
793     return total;
794 }
795
796 [MethodImpl(MethodImplOptions.AggressiveInlining)]
797 public List<long> GetCommonIndices(BitString other)
798 {
799     EnsureBitStringHasTheSameSize(other, nameof(other));
800     GetCommonInnerBorders(this, other, out long from, out long to);
801     var result = new List<long>();
802     var otherArray = other._array;
803     for (var i = from; i <= to; i++)
804     {
805         var left = _array[i];
806         var right = otherArray[i];
807         var combined = left & right;
808         if (combined != 0)
809         {
810             AppendAllSetBitIndices(result, i, combined);
811         }
812     }
813     return result;
814 }
815
816 [MethodImpl(MethodImplOptions.AggressiveInlining)]
817 public List<ulong> GetCommonUInt64Indices(BitString other)
818 {
819     EnsureBitStringHasTheSameSize(other, nameof(other));
820     GetCommonBorders(this, other, out ulong from, out ulong to);
821     var result = new List<ulong>();
822     var otherArray = other._array;
823     for (var i = from; i <= to; i++)
824     {
825         var left = _array[i];
826         var right = otherArray[i];
827         var combined = left & right;
828         if (combined != 0)
829         {
830             AppendAllSetBitIndices(result, i, combined);
831         }
832     }
833     return result;
834 }
835
836 [MethodImpl(MethodImplOptions.AggressiveInlining)]
837 public long GetFirstCommonBitIndex(BitString other)
838 {
839     EnsureBitStringHasTheSameSize(other, nameof(other));
840     GetCommonInnerBorders(this, other, out long from, out long to);
841     var otherArray = other._array;
842     for (var i = from; i <= to; i++)
843     {
844         var left = _array[i];
845         var right = otherArray[i];
846         var combined = left & right;
847         if (combined != 0)
848         {
849             return GetFirstSetBitForWord(i, combined);

```

```

850     }
851 }
852 return -1;
853 }
854
855 [MethodImpl(MethodImplOptions.AggressiveInlining)]
856 public long GetLastCommonBitIndex(BitString other)
857 {
858     EnsureBitStringHasTheSameSize(other, nameof(other));
859     GetCommonInnerBorders(this, other, out long from, out long to);
860     var otherArray = other._array;
861     for (var i = to; i >= from; i--)
862     {
863         var left = _array[i];
864         var right = otherArray[i];
865         var combined = left & right;
866         if (combined != 0)
867         {
868             return GetLastSetBitForWord(i, combined);
869         }
870     }
871     return -1;
872 }
873
874 [MethodImpl(MethodImplOptions.AggressiveInlining)]
875 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
    ↪ false;
876
877 [MethodImpl(MethodImplOptions.AggressiveInlining)]
878 public bool Equals(BitString other)
879 {
880     if (_length != other._length)
881     {
882         return false;
883     }
884     var otherArray = other._array;
885     if (_array.Length != otherArray.Length)
886     {
887         return false;
888     }
889     if (_minPositiveWord != other._minPositiveWord)
890     {
891         return false;
892     }
893     if (_maxPositiveWord != other._maxPositiveWord)
894     {
895         return false;
896     }
897     GetCommonBorders(this, other, out ulong from, out ulong to);
898     for (var i = from; i <= to; i++)
899     {
900         if (_array[i] != otherArray[i])
901         {
902             return false;
903         }
904     }
905     return true;
906 }
907
908 [MethodImpl(MethodImplOptions.AggressiveInlining)]
909 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
910 {
911     Ensure.Always.ArgumentNotNull(other, argumentName);
912     if (_length != other._length)
913     {
914         throw new ArgumentException("Bit string must be the same size.", argumentName);
915     }
916 }
917
918 [MethodImpl(MethodImplOptions.AggressiveInlining)]
919 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
920
921 [MethodImpl(MethodImplOptions.AggressiveInlining)]
922 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
923
924 [MethodImpl(MethodImplOptions.AggressiveInlining)]
925 private void GetBorders(out long from, out long to)
926 {
927     from = _minPositiveWord;

```

```

928         to = _maxPositiveWord;
929     }
930
931     [MethodImpl(MethodImplOptions.AggressiveInlining)]
932     private void GetBorders(out ulong from, out ulong to)
933     {
934         from = (ulong)_minPositiveWord;
935         to = (ulong)_maxPositiveWord;
936     }
937
938     [MethodImpl(MethodImplOptions.AggressiveInlining)]
939     private void SetBorders(long from, long to)
940     {
941         _minPositiveWord = from;
942         _maxPositiveWord = to;
943     }
944
945     [MethodImpl(MethodImplOptions.AggressiveInlining)]
946     private Range<long> GetValidIndexRange() => (0, _length - 1);
947
948     [MethodImpl(MethodImplOptions.AggressiveInlining)]
949     private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
950
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     }
957
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     }
964
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     }
971
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     }
978
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     }
985
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 }
1006
1007 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1008 private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1009 ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1010 {
1011     for (var j = 0; j < bits00to15.Length; j++)
1012     {
1013         result.Add(bits00to15[j] + (i * 64));
1014     }
1015     for (var j = 0; j < bits16to31.Length; j++)
1016     {
1017         result.Add(bits16to31[j] + 16UL + (i * 64));
1018     }
1019     for (var j = 0; j < bits32to47.Length; j++)
1020     {
1021         result.Add(bits32to47[j] + 32UL + (i * 64));
1022     }
1023     for (var j = 0; j < bits48to63.Length; j++)
1024     {
1025         result.Add(bits48to63[j] + 48UL + (i * 64));
1026     }
1027 }
1028
1029 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1030 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1031 ↪ bits32to47, byte[] bits48to63)
1032 {
1033     if (bits00to15.Length > 0)
1034     {
1035         return bits00to15[0] + (i * 64);
1036     }
1037     if (bits16to31.Length > 0)
1038     {
1039         return bits16to31[0] + 16 + (i * 64);
1040     }
1041     if (bits32to47.Length > 0)
1042     {
1043         return bits32to47[0] + 32 + (i * 64);
1044     }
1045     return bits48to63[0] + 48 + (i * 64);
1046 }
1047
1048 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1049 private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1050 ↪ bits32to47, byte[] bits48to63)
1051 {
1052     if (bits48to63.Length > 0)
1053     {
1054         return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1055     }
1056     if (bits32to47.Length > 0)
1057     {
1058         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1059     }
1060     if (bits16to31.Length > 0)
1061     {
1062         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1063     }
1064     return bits00to15[bits00to15.Length - 1] + (i * 64);
1065 }
1066
1067 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1068 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
1069 ↪ byte[] bits32to47, out byte[] bits48to63)
1070 {
1071     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1072     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];

```



```

1069         bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1070         bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1071     }
1072
1073     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1074     public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
1075         ↪ out long to)
1076     {
1077         from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1078         to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1079     }
1080
1081     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1082     public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
1083         ↪ out long to)
1084     {
1085         from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1086         to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1087     }
1088
1089     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1090     public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
1091         ↪ out int to)
1092     {
1093         from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1094         to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1095     }
1096
1097     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1098     public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
1099         ↪ ulong to)
1100     {
1101         from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1102         to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1103     }
1104
1105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1106     public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1107
1108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1109     public static long GetWordIndexFromIndex(long index) => index >> 6;
1110
1111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1112     public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1113
1114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1115     public override int GetHashCode() => base.GetHashCode();
1116
1117     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1118     public override string ToString() => base.ToString();
1119 }
1120 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Random;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections
7 {
8     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

```

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

```

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
12         ↪ value) ? value : default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
16         ↪ value) ? value : default;
17     }
18 }

```

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

```

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

```

```

35         if (string.IsNullOrEmpty(argument))
36         {
37             throw new ArgumentException(message, argumentName);
38         }
39     }
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
43         ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
44         ↪ argument, argumentName, null);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
48         ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
49
50     #endregion
51
52     #region OnDebug
53
54     [Conditional("DEBUG")]
55     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
56         ↪ ICollection<T> argument, string argumentName, string message) =>
57         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
58
59     [Conditional("DEBUG")]
60     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
61         ↪ ICollection<T> argument, string argumentName) =>
62         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
63
64     [Conditional("DEBUG")]
65     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
66         ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
67
68     [Conditional("DEBUG")]
69     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
70         ↪ root, string argument, string argumentName, string message) =>
71         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
72
73     [Conditional("DEBUG")]
74     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
75         ↪ root, string argument, string argumentName) =>
76         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
77
78     [Conditional("DEBUG")]
79     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
80         ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
81         ↪ null, null);
82
83     #endregion
84 }

```

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

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections
8  {
9      public static class ICollectionExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
13             ↪ null || collection.Count == 0;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
17         {
18             var equalityComparer = EqualityComparer<T>.Default;
19             return collection.All(item => equalityComparer.Equals(item, default));
20         }
21     }
22 }

```

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

```

1  using System;
2  using System.Collections.Generic;

```

```

3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections
8 {
9     public static class IDictionaryExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
13        ↪ dictionary, TKey key)
14        {
15            dictionary.TryGetValue(key, out TValue value);
16            return value;
17        }
18
19        [MethodImpl(MethodImplOptions.AggressiveInlining)]
20        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
21        ↪ TKey key, Func<TKey, TValue> valueFactory)
22        {
23            if (!dictionary.TryGetValue(key, out TValue value))
24            {
25                value = valueFactory(key);
26                dictionary.Add(key, value);
27                return value;
28            }
29            return value;
30        }
31    }
32 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Lists
5 {
6     public static class CharListExtensions
7     {
8         /// <summary>
9         /// <para>Generates a hash code for the entire list based on the values of its
10        ↪ elements.</para>
11        /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
12        /// </summary>
13        /// <param name="list"><para>The list to be hashed.</para><para>Список для
14        ↪ хеширования.</para></param>
15        /// <returns>
16        /// <para>The hash code of the list.</para>
17        /// <para>Хэш-код списка.</para>
18        /// </returns>
19        /// <remarks>
20        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
21        ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
22        /// </remarks>
23        [MethodImpl(MethodImplOptions.AggressiveInlining)]
24        public static int GenerateHashCode(this IList<char> list)
25        {
26            var hashSeed = 5381;
27            var hashAccumulator = hashSeed;
28            for (var i = 0; i < list.Count; i++)
29            {
30                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
31            }
32            return hashAccumulator + (hashSeed * 1566083941);
33        }
34
35        /// <summary>
36        /// <para>Compares two lists for equality.</para>
37        /// <para>Сравнивает два списка на равенство.</para>
38        /// </summary>
39        /// <param name="left"><para>The first compared list.</para><para>Первый список для
40        ↪ сравнения.</para></param>
41        /// <param name="right"><para>The second compared list.</para><para>Второй список для
42        ↪ сравнения.</para></param>
43        /// <returns>
44        /// <para>True, if the passed lists are equal to each other otherwise false.</para>
45        /// <para>True, если переданные списки равны друг другу, иначе false.</para>
46        /// </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)]
56 public static bool ContentEqualTo(this IList<char> left, IList<char> right)
57 {
58     for (var i = left.Count - 1; i >= 0; --i)
59     {
60         if (left[i] != right[i])
61         {
62             return false;
63         }
64     }
65     return true;
66 }
67 }
68 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Lists
5 {
6     public class IListComparer<T> : IComparer<IList<T>>
7     {
8         /// <summary>
9         /// <para>Compares two lists.</para>
10        /// <para>Сравнивает два списка.</para>
11        /// </summary>
12        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            → списка.</para></typeparam>
13        /// <param name="left"><para>The first compared list.</para><para>Первый список для
            → сравнения.</para></param>
14        /// <param name="right"><para>The second compared list.</para><para>Второй список для
            → сравнения.</para></param>
15        /// <returns>
16        /// <para>
17        ///     A signed integer that indicates the relative values of <paramref name="left" />
            → and <paramref name="right" /> lists' elements, as shown in the following table.
18        ///     <list type="table">
19        ///         <listheader>
20        ///             <term>Value</term>
21        ///             <description>Meaning</description>
22        ///         </listheader>
23        ///         <item>
24        ///             <term>Is less than zero</term>
25        ///             <description>First non equal element of <paramref name="left" /> list is
            → less than first not equal element of <paramref name="right" /> list.</description>
26        ///         </item>
27        ///         <item>
28        ///             <term>Zero</term>
29        ///             <description>All elements of <paramref name="left" /> list equals to all
            → elements of <paramref name="right" /> list.</description>
30        ///         </item>
31        ///         <item>
32        ///             <term>Is greater than zero</term>
33        ///             <description>First non equal element of <paramref name="left" /> list is
            → greater than first not equal element of <paramref name="right" /> list.</description>
34        ///         </item>
35        ///     </list>
36        /// </para>

```

```

37     /// <para>
38     ///     Целое число со знаком, которое указывает относительные значения элементов
    → списков <paramref name="left" /> и <paramref name="right" /> как показано в
    → следующей таблице.
39     ///     <list type="table">
40     ///         <listheader>
41     ///             <term>Значение</term>
42     ///             <description>Смысл</description>
43     ///         </listheader>
44     ///         <item>
45     ///             <term>Меньше нуля</term>
46     ///             <description>Первый не равный элемент <paramref name="left" /> списка
    → меньше первого неравного элемента <paramref name="right" /> списка.</description>
47     ///         </item>
48     ///         <item>
49     ///             <term>Ноль</term>
50     ///             <description>Все элементы <paramref name="left" /> списка равны всем
    → элементам <paramref name="right" /> списка.</description>
51     ///         </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
11         /// <para>→ within the list's boundaries, otherwise it returns default value of type T.</para>
12         /// <para>Получает элемент из указанного индекса, если список не является null и индекс
13         /// <para>→ находится в границах списка, в противном случае он возвращает значение по умолчанию
14         /// <para>→ типа T.</para>
15         /// </summary>
16         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
17         /// <para>→ списка.</para></typeparam>
18         /// <param name="list"><para>The checked list.</para><para>Проверяемый
19         /// <para>→ список.</para></param>
20         /// <param name="index"><para>The index of element.</para><para>Индекс
21         /// <para>→ элемента.</para></param>
22         /// <returns>
23         /// <para>If the specified index is within list's boundaries, then - list[index],
24         /// <para>→ otherwise the default value.</para>
25         /// <para>Если указанный индекс находится в пределах границ списка, тогда - list[index],
26         /// <para>→ иначе же значение по умолчанию.</para>
27         /// </returns>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
30         /// <para>→ list.Count > index ? list[index] : default;
31
32         /// <summary>
33         /// <para>Checks if a list is passed, checks its length, and if successful, copies the
34         /// <para>→ value of list [index] into the element variable. Otherwise, the element variable has
35         /// <para>→ a default value.</para>
36         /// <para>Проверяет, передан ли список, сверяет его длину и в случае успеха копирует
37         /// <para>→ значение list[index] в переменную element. Иначе переменная element имеет значение
38         /// <para>→ по умолчанию.</para>
39         /// </summary>
40         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
41         /// <para>→ списка.</para></typeparam>
42         /// <param name="list"><para>The checked list.</para><para>Список для
43         /// <para>→ проверки.</para></param>
44         /// <param name="index"><para>The index of element.</para><para>Индекс
45         /// <para>→ элемента.</para></param>
46         /// <param name="element"><para>Variable for passing the index
47         /// <para>→ value.</para><para>Переменная для передачи значения индекса.</para></param>
48         /// <returns>
49         /// <para>True on success, false otherwise.</para>
50         /// <para>True в случае успеха, иначе false.</para>
51         /// </returns>
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
54         {
55             if (list != null && list.Count > index)
56             {
57                 element = list[index];
58                 return true;
59             }
60             else
61             {
62                 element = default;
63                 return false;
64             }
65         }
66
67         /// <summary>
68         /// <para>Adds a value to the list.</para>
69         /// <para>Добавляет значение в список.</para>
70         /// </summary>
71         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
72         /// <para>→ списка.</para></typeparam>
73         /// <param name="list"><para>The list to add the value to.</para><para>Список в который
74         /// <para>→ нужно добавить значение.</para></param>
75         /// <param name="element"><para>The item to add to the list.</para><para>Элемент который
76         /// <para>→ нужно добавить в список.</para></param>
77         /// <returns>
78         /// <para>True value in any case.</para>
79         /// <para>Значение true в любом случае.</para>
80         /// </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
73   ↳ списка.</para></typeparam>
74 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
75   ↳ нужно добавить значение.</para></param>
76 /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
77   ↳ который нужно добавить в список</para></param>
78 /// <returns>
79 /// <para>True value in any case.</para>
80 /// <para>Значение true в любом случае.</para>
81 /// </returns>
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
84 {
85     list.AddFirst(elements);
86     return true;
87 }
88
89 /// <summary>
90 /// <para>Adds a value to the list at the first index.</para>
91 /// <para>Добавляет значение в список по первому индексу.</para>
92 /// </summary>
93 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
94   ↳ списка.</para></typeparam>
95 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
96   ↳ нужно добавить значение.</para></param>
97 /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
98   ↳ который нужно добавить в список</para></param>
99 [MethodImpl(MethodImplOptions.AggressiveInlining)]
100 public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
101     list.Add(elements[0]);
102
103 /// <summary>
104 /// <para>Adds all elements from other list to this list and returns true.</para>
105 /// <para>Добавляет все элементы из другого списка в этот список и возвращает
106   ↳ true.</para>
107 /// </summary>
108 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
109   ↳ списка.</para></typeparam>
110 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
111   ↳ нужно добавить значения.</para></param>
112 /// <param name="elements"><para>List of values to add.</para><para>Список значений
113   ↳ которые необходимо добавить.</para></param>
114 /// <returns>
115 /// <para>True value in any case.</para>
116 /// <para>Значение true в любом случае.</para>
117 /// </returns>
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
120 {
121     list.AddAll(elements);
122     return true;
123 }
124
125 /// <summary>
126 /// <para>Adds all elements from other list to this list.</para>
127 /// <para>Добавляет все элементы из другого списка в этот список.</para>
128 /// </summary>
129 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
130   ↳ списка.</para></typeparam>
131 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
132   ↳ нужно добавить значения.</para></param>
133 /// <param name="elements"><para>The list of values to add.</para><para>Список значений
134   ↳ которые необходимо добавить.</para></param>
135 [MethodImpl(MethodImplOptions.AggressiveInlining)]
136 public static void AddAll<T>(this IList<T> list, IList<T> elements)
137 {

```



```

124     for (var i = 0; i < elements.Count; i++)
125     {
126         list.Add(elements[i]);
127     }
128 }
129
130 /// <summary>
131 /// <para>Adds values to the list skipping the first element.</para>
132 /// <para>Добавляет значения в список пропуская первый элемент.</para>
133 /// </summary>
134 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
135 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
136 /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
137 /// <returns>
138 /// <para>True value in any case.</para>
139 /// <para>Значение true в любом случае.</para>
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)]
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
261   ↳ predicate. If no list is passed, null is returned.</para>
262   ↳ <para>Создаёт массив, копируя из списка все элементы которые удовлетворяют
263     ↳ предикату. Если список не передан, возвращается null.</para>
264   ↳ </summary>
265   ↳ <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
266     ↳ списка.</para></typeparam>
267   ↳ <param name="list"><para>The list to copy from.</para><para>Список для копирования.</para></param>
268   ↳ <param name="predicate"><para>A function that determines whether an element should
269     ↳ be copied.</para><para>Функция определяющая должен ли копироваться
270     ↳ элемент.</para></param>
271   ↳ </returns>
272   ↳ <para>An array with copied elements from the list.</para>
273   ↳ <para>Массив с скопированными элементами из списка.</para>
274   ↳ </returns>
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
277 {
278     if (list == null)
279     {
280         return null;
281     }
282     var result = new List<T>(list.Count);
283     for (var i = 0; i < list.Count; i++)
284     {
285         if (predicate(list[i]))
286         {
287             result.Add(list[i]);
288         }
289     }
290     return result.ToArray();
291 }
292
293 /// <summary>
294 /// <para>Copies all the elements of the list into an array and returns it.</para>
295   ↳ <para>Копирует все элементы списка в массив и возвращает его.</para>
296   ↳ </summary>
297   ↳ <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
298     ↳ списка.</para></typeparam>
299   ↳ <param name="list"><para>The list to copy from.</para><para>Список для
300     ↳ копирования.</para></param>
301   ↳ </returns>
302   ↳ <para>An array with all the elements of the passed list.</para>
303   ↳ <para>Массив со всеми элементами переданного списка.</para>
304   ↳ </returns>
305 [MethodImpl(MethodImplOptions.AggressiveInlining)]
306 public static T[] ToArray<T>(this IList<T> list)
307 {
308     var array = new T[list.Count];
309     list.CopyTo(array, 0);
310     return array;
311 }
312
313 /// <summary>
314 /// <para>Executes the passed action for each item in the list.</para>
315   ↳ <para>Выполняет переданное действие для каждого элемента в списке.</para>
316   ↳ </summary>
317   ↳ <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
318     ↳ списка.</para></typeparam>
319   ↳ <param name="list"><para>The list of elements for which the action will be
320     ↳ executed.</para><para>Список элементов для которых будет выполняться
321     ↳ 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>
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
377  → списков <paramref name="left" /> и <paramref name="right" /> как показано в
378  → следующей таблице.
379  /// <list type="table">
380  /// <listheader>
381  /// <term>Значение</term>
382  /// <description>Смысл</description>
383  /// </listheader>
384  /// <item>
385  /// <term>Меньше нуля</term>
386  /// <description>Первый не равный элемент <paramref name="left" /> списка
387  → меньше первого неравного элемента <paramref name="right" /> списка.</description>
388  /// </item>
389  /// <item>
390  /// <term>Ноль</term>
391  /// <description>Все элементы <paramref name="left" /> списка равны всем
392  → элементам <paramref name="right" /> списка.</description>
393  /// </item>
394  /// <item>
395  /// <term>Больше нуля</term>
396  /// <description>Первый не равный элемент <paramref name="left" /> списка
397  → больше первого неравного элемента <paramref name="right" /> списка.</description>
398  /// </item>
399  /// </list>
400  /// </para>
401  /// </returns>
402  [MethodImpl(MethodImplOptions.AggressiveInlining)]
403  public static int CompareTo<T>(this IList<T> left, IList<T> right)
404  {
405      var comparer = Comparer<T>.Default;
406      var leftCount = left.GetCountOrZero();
407      var rightCount = right.GetCountOrZero();
408      var intermediateResult = leftCount.CompareTo(rightCount);
409      for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
410      {
411          intermediateResult = comparer.Compare(left[i], right[i]);
412      }
413      return intermediateResult;
414  }
415  /// <summary>
416  /// <para>Skips one element in the list and builds an array from the remaining
417  → elements.</para>
418  /// <para>Пропускает один элемент списка и составляет из оставшихся элементов
419  → массив.</para>
420  /// </summary>
421  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
422  → списка.</para></typeparam>
423  /// <param name="list"><para>The list to copy from.</para><para>Список для
424  → копирования.</para></param>
425  /// <returns>
426  /// <para>If the list is empty, returns an empty array, otherwise - an array with a
427  → missing first element.</para>
428  /// <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
429  → первым элементом.</para>
430  /// </returns>
431  [MethodImpl(MethodImplOptions.AggressiveInlining)]
432  public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
433  /// <summary>
434  /// <para>Skips the specified number of elements in the list and builds an array from
435  → the remaining elements.</para>
436  /// <para>Пропускает указанное количество элементов списка и составляет из оставшихся
437  → элементов массив.</para>
438  /// </summary>
439  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
440  → списка.</para></typeparam>
441  /// <param name="list"><para>The list to copy from.</para><para>Список для
442  → копирования.</para></param>
443  /// <param name="skip"><para>The number of items to skip.</para><para>Количество
444  → пропускаемых элементов.</para></param>
445  /// <returns>
446  /// <para>If the list is empty, or the number of skipped elements is greater than the
447  → list, returns an empty array, otherwise - an array with the specified number of
448  → missing elements.</para>

```

```

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

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Lists
5 {
6     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
16         ↪ заполняться.</para></param>
17         /// <param name="returnConstant"><para>The value for the constant returned by
18         ↪ corresponding methods.</para><para>Значение для константы возвращаемой
19         ↪ соответствующими методами.</para></param>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
22         {
23             _list = list;
24             _returnConstant = returnConstant;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public ListFiller(List<TElement> list) : this(list, default) { }
29
30         /// <summary>
31         /// <para>Adds an item to the end of the list.</para>
32         /// <para>Добавляет элемент в конец списка.</para>
33         /// </summary>
34         /// <param name="element"><para>Element to add.</para><para>Добавляемый
35         ↪ элемент.</para></param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public void Add(TElement element) => _list.Add(element);
38
39         /// <summary>
40         /// <para>Adds an item to the end of the list and return true.</para>
41         /// <para>Добавляет элемент в конец списка и возвращает true.</para>
42         /// </summary>
43         /// <param name="element"><para>Element to add.</para><para>Добавляемый
44         ↪ элемент.</para></param>
45         /// <returns>
46         /// <para>True value in any case.</para>
47         /// <para>Значение true в любом случае.</para>
48         /// </returns>
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
51
52         /// <summary>
53         /// <para>Adds a value to the list at the first index and return true.</para>
54         /// <para>Добавляет значение в список по первому индексу и возвращает true.</para>
55         /// </summary>
56         /// <param name="element"><para>Element to add.</para><para>Добавляемый
57         ↪ элемент.</para></param>
58         /// <returns>
59         /// <para>True value in any case.</para>
60         /// <para>Значение true в любом случае.</para>
61         /// </returns>
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
64             ↪ _list.AddFirstAndReturnTrue(elements);
65
66         /// <summary>
67         /// <para>Adds all elements from other list to this list and returns true.</para>
68         /// <para>Добавляет все элементы из другого списка в этот список и возвращает
69         ↪ true.</para>
70         /// </summary>
71         /// <param name="elements"><para>List of values to add.</para><para>Список значений
72         ↪ которые необходимо добавить.</para></param>
73         /// <returns>
74         /// <para>True value in any case.</para>
75         /// <para>Значение true в любом случае.</para>
76         /// </returns>
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
79             ↪ _list.AddAllAndReturnTrue(elements);
80     }
81 }

```

```

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>
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;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class CharSegment : Segment<char>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
    ↪     length) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override int GetHashCode()
18         {
19             // Base can be not an array, but still IList<char>
20             if (Base is char[] baseArray)
21             {
22                 return baseArray.GenerateHashCode(Offset, Length);
23             }
24             else
25             {
26                 return this.GenerateHashCode();
27             }
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public override bool Equals(Segment<char> other)
32         {
33             bool contentEqualityComparer(IList<char> left, IList<char> right)
34             {
35                 // Base can be not an array, but still IList<char>
36                 if (Base is char[] baseArray && other.Base is char[] otherArray)
37                 {
38                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
39                 }
40                 else
41                 {
42                     return left.ContentEqualTo(right);
43                 }
44             }
45             return this.EqualTo(other, contentEqualityComparer);
46         }
47
48         public override bool Equals(object obj) => obj is Segment<char> charSegment ?
    ↪     Equals(charSegment) : false;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public static implicit operator string(CharSegment segment)
52         {
53             if (!(segment.Base is char[] array))
54             {
55                 array = segment.Base.ToArray();
56             }
57             return new string(array, segment.Offset, segment.Length);
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         public override string ToString() => this;

```

```

62     }
63 }

```

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

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Segments
11 {
12     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
13     {
14         public IList<T> Base
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19         public int Offset
20         {
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             get;
23         }
24         public int Length
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public Segment(IList<T> @base, int offset, int length)
32         {
33             Base = @base;
34             Offset = offset;
35             Length = length;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public override int GetHashCode() => this.GenerateHashCode();
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
46             ↪ false;
47
48         #region IList
49         public T this[int i]
50         {
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             get => Base[Offset + i];
53             [MethodImpl(MethodImplOptions.AggressiveInlining)]
54             set => Base[Offset + i] = value;
55         }
56
57         public int Count
58         {
59             [MethodImpl(MethodImplOptions.AggressiveInlining)]
60             get => Length;
61         }
62
63         public bool IsReadOnly
64         {
65             [MethodImpl(MethodImplOptions.AggressiveInlining)]
66             get => true;
67         }
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public int IndexOf(T item)
71         {
72             var index = Base.IndexOf(item);
73             if (index >= Offset)
74             {
75                 var actualIndex = index - Offset;
76                 if (actualIndex < Length)

```

```

77         {
78             return actualIndex;
79         }
80     }
81     return -1;
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public void Insert(int index, T item) => throw new NotSupportedException();
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public void RemoveAt(int index) => throw new NotSupportedException();
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public void Add(T item) => throw new NotSupportedException();
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public void Clear() => throw new NotSupportedException();
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

```

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

```

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) =>
15             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
19     }
20 }

```

```

18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public virtual void WalkAll(ICollection<T> elements)
20     {
21         for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
22             ↪ offset <= maxOffset; offset++)
23         {
24             for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
25                 ↪ offset; length <= maxLength; length++)
26             {
27                 Iteration(CreateSegment(elements, offset, length));
28             }
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected abstract TSegment CreateSegment(ICollection<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(ICollection<T> elements, int offset, int length)
12             ↪ => new Segment<T>(elements, offset, length);
13     }
14 }

```

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) =>
11             ↪ walker.WalkAll(@string.ToCharArray());
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char> walker,
15             ↪ string @string) where TSegment : Segment<char> =>
16             ↪ walker.WalkAll(@string.ToCharArray());
17     }
18 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Segments.Walkers
8 {
9     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
10         ↪ DuplicateSegmentsWalkerBase<T, TSegment>
11         where TSegment : Segment<T>
12     {
13         public static readonly bool DefaultResetDictionaryOnEachWalk;
14
15         private readonly bool _resetDictionaryOnEachWalk;
16         protected IDictionary<TSegment, long> Dictionary;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

18     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
19         ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
20         : base(minimumStringSegmentLength)
21     {
22         Dictionary = dictionary;
23         _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
24     }
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
28         ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
29         ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
33         ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
34         ↪ DefaultResetDictionaryOnEachWalk) { }
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
38         ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
39         ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
40         { }
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
44         ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
48         ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public override void WalkAll(ICollection<T> elements)
52     {
53         if (_resetDictionaryOnEachWalk)
54         {
55             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
56             Dictionary = new Dictionary<TSegment, long>((int)capacity);
57         }
58         base.WalkAll(elements);
59     }
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     protected override long GetSegmentFrequency(TSegment segment) =>
63         ↪ Dictionary.GetOrDefault(segment);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
67         ↪ Dictionary[segment] = frequency;
68 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
9          ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
13             ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
14             ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
18             ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
19             ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
23             ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
24             ↪ DefaultResetDictionaryOnEachWalk) { }
25     }
26 }

```

```

18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
20         ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
21         ↪ resetDictionaryOnEachWalk) { }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
25         ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
29         ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
30 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Segments.Walkers
6 {
7     public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
8         ↪ TSegment>
9         where TSegment : Segment<T>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
13             ↪ base(minimumStringSegmentLength) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override void Iteration(TSegment segment)
20         {
21             var frequency = GetSegmentFrequency(segment);
22             if (frequency == 1)
23             {
24                 OnDuplicateFound(segment);
25             }
26             SetSegmentFrequency(segment, frequency + 1);
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected abstract void OnDuplicateFound(TSegment segment);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected abstract long GetSegmentFrequency(TSegment segment);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
37     }
38 }

```

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

```

1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 namespace Platform.Collections.Segments.Walkers
4 {
5     public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
6         ↪ Segment<T>>
7     {
8     }
9 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Sets
7 {
8     public static class ISetExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

11     public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
15         ↪ set.Remove(element);
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
19     {
20         set.Add(element);
21         return true;
22     }
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
26     {
27         AddFirst(set, elements);
28         return true;
29     }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
33         ↪ set.Add(elements[0]);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
37     {
38         set.AddAll(elements);
39         return true;
40     }
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     public static void AddAll<T>(this ISet<T> set, IList<T> elements)
44     {
45         for (var i = 0; i < elements.Count; i++)
46         {
47             set.Add(elements[i]);
48         }
49     }
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
53     {
54         set.AddSkipFirst(elements);
55         return true;
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
60         ↪ set.AddSkipFirst(elements, 1);
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
64     {
65         for (var i = skip; i < elements.Count; i++)
66         {
67             set.Add(elements[i]);
68         }
69     }
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static bool DoNotContains<T>(this ISet<T> set, T element) =>
73         ↪ !set.Contains(element);
74 }

```

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)]
30     public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
31         ↪ _set.AddFirstAndReturnTrue(elements);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
35         ↪ _set.AddAllAndReturnTrue(elements);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
39         ↪ _set.AddSkipFirstAndReturnTrue(elements);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public TReturnConstant AddAndReturnConstant(TElement element)
43     {
44         _set.Add(element);
45         return _returnConstant;
46     }
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
50     {
51         _set.AddFirst(elements);
52         return _returnConstant;
53     }
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
57     {
58         _set.AddAll(elements);
59         return _returnConstant;
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements)
64     {
65         _set.AddSkipFirst(elements);
66         return _returnConstant;
67     }
68 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Stacks
7 {
8     public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
9     {
10         public bool IsEmpty
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get => Count <= 0;
14         }
15     }
16 }

```

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

```

1 using System.Runtime.CompilerServices;
2

```



```

3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Stacks
6  {
7      public interface IStack<TElement>
8      {
9          bool IsEmpty
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         void Push(TElement element);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         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 :
20             ↪ stack.Pop();
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
24             ↪ stack.Peek();
25     }
26 }

```

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() :
12             ↪ default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
16             ↪ : default;
17     }
18 }

```

```
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.IsNullOrEmpty(@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) =>
37             ↪ string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
38             ↪ Math.Min(@string.Length, maxLength));
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public static string TrimSingle(this string @string, char charToTrim)
42         {
43             if (!string.IsNullOrEmpty(@string))
44             {
45                 if (@string.Length == 1)
46                 {
47                     if (@string[0] == charToTrim)
48                     {
49                         return "";
50                     }
51                     else
52                     {
53                         return @string;
54                     }
55                 }
56                 else
57                 {
58                     var left = 0;
59                     var right = @string.Length - 1;
60                     if (@string[left] == charToTrim)
61                     {
62                         left++;
63                     }
64                     if (@string[right] == charToTrim)
65                     {
66                         right--;
67                     }
68                     return @string.Substring(left, right - left + 1);
69                 }
70             }
71             else
72             {
73                 return @string;
74             }
75         }
76     }
77 }
```

```

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
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get;
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             set;
19         }
20
21         public Dictionary<object, Node> ChildNodes
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
25         }
26
27         public Node this[object key]
28         {
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             get => GetChild(key) ?? AddChild(key);
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             set => SetChildValue(value, key);
33         }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public Node(object value) => Value = value;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public Node() : this(null) { }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public Node GetChild(params object[] keys)
46         {
47             var node = this;
48             for (var i = 0; i < keys.Length; i++)
49             {
50                 node.ChildNodes.TryGetValue(keys[i], out node);
51                 if (node == null)
52                 {
53                     return null;
54                 }
55             }
56             return node;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public Node AddChild(object key) => AddChild(key, new Node(null));
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public Node AddChild(object key, object value) => AddChild(key, new Node(value));
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public Node AddChild(object key, Node child)
70         {
71             ChildNodes.Add(key, child);
72             return child;
73         }
74
75         [MethodImpl(MethodImplOptions.AggressiveInlining)]
76         public Node SetChild(params object[] keys) => SetChildValue(null, keys);

```

```

77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public Node SetChild(object key) => SetChildValue(null, key);
79
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public Node SetChildValue(object value, params object[] keys)
82     {
83         var node = this;
84         for (var i = 0; i < keys.Length; i++)
85         {
86             node = SetChildValue(value, keys[i]);
87         }
88         node.Value = value;
89         return node;
90     }
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public Node SetChildValue(object value, object key)
94     {
95         if (!ChildNodes.TryGetValue(key, out Node child))
96         {
97             child = AddChild(key, value);
98         }
99         child.Value = value;
100         return child;
101     }
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 }

```

```

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         {
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,
147     ↪ BitString, BitString> test)
148 {
149     const int n = 5654;
150     var x = new BitString(n);
151     var y = new BitString(n);
152     while (x.Equals(y))
153     {
154         x.SetRandomBits();
155         y.SetRandomBits();
156     }
157     var w = new BitString(x);
158     var v = new BitString(y);
159     Assert.False(x.Equals(y));
160     Assert.False(w.Equals(v));
161     Assert.True(x.Equals(w));
162     Assert.True(y.Equals(v));
163     test(x, y, w, v);
164     Assert.True(x.Equals(w));
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

```

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

```

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

- ./csharp/Platform.Collections.Tests/ArrayTests.cs, 52
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 52
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 54
- ./csharp/Platform.Collections.Tests/ListTests.cs, 55
- ./csharp/Platform.Collections.Tests/StringTests.cs, 55
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayPool.cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayString.cs, 3
- ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs, 3
- ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 4
- ./csharp/Platform.Collections/BitString.cs, 10
- ./csharp/Platform.Collections/BitStringExtensions.cs, 25
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 25
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 26
- ./csharp/Platform.Collections/EnsureExtensions.cs, 26
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 27
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 27
- ./csharp/Platform.Collections/Lists/CharIListExtensions.cs, 28
- ./csharp/Platform.Collections/Lists/IListComparer.cs, 29
- ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 30
- ./csharp/Platform.Collections/Lists/IListExtensions.cs, 30
- ./csharp/Platform.Collections/Lists/ListFiller.cs, 39
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 41
- ./csharp/Platform.Collections/Segments/Segment.cs, 42
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 43
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 43
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 44
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 44
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 44
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 45
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 46
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 46
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 46
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 47
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 48
- ./csharp/Platform.Collections/Stacks/IStack.cs, 48
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 49
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 49
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 49
- ./csharp/Platform.Collections/StringExtensions.cs, 50
- ./csharp/Platform.Collections/Trees/Node.cs, 51