

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Stacks;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Arrays
10 {
11     /// <remarks>
12     /// Original idea from
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[]> AllocateDisposable(long size) => (Allocate(size), Free);

```

```

33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public Disposable<T[]> Resize(Disposable<T[]> source, long size)
35     {
36         var destination = AllocatedDisposable(size);
37         T[] sourceArray = source;
38         if (!sourceArray.IsNullOrEmpty())
39         {
40             T[] destinationArray = destination;
41             Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
42                 ↳ sourceArray.LongLength);
43             source.Dispose();
44         }
45         return destination;
46     }
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public virtual void Clear() => _pool.Clear();
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
53         ↳ _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public virtual void Free(T[] array)
57     {
58         if (array.IsNullOrEmpty())
59         {
60             return;
61         }
62         var stack = _pool.GetOrAdd(array.LongLength, size => new
63             ↳ Stack<T[]>(_maxArraysPerSize));
64         if (stack.Count == _maxArraysPerSize) // Stack is full
65         {
66             return;
67         }
68         stack.Push(array);
69     }
70 }

```

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

```

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

```

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

```

1 using System.Runtime.CompilerServices;
2
3 namespace Platform.Collections.Arrays
4 {
5     public static unsafe class CharArrayExtensions
6     {
7         /// <summary>
8         /// <para>Generates a hash code for an array segment with the specified offset and
9         ↳ length. The hash code is generated based on the values of the array elements
10         ↳ included in the specified segment.</para>
11         /// <para>Генерирует хэш-код сегмента массива с указанным смещением и длиной. Хэш-код
12         ↳ генерируется на основе значений элементов массива входящих в указанный
13         ↳ сегмент.</para>
14         /// </summary>
15         /// <param name="array"><para>The array to hash.</para><para>Массив для
16         ↳ хеширования.</para></param>

```

```

12  /// <param name="offset"><para>The offset from which reading of the specified number of
    ↳ elements in the array starts.</para><para>Смещение, с которого начинается чтение
13  /// <param name="length"><para>The number of array elements used to calculate the
    ↳ hash.</para><para>Количество элементов массива, на основе которых будет вычислен
    ↳ хэш.</para></param>
14  /// <returns>
15  /// <para>The hash code of the segment in the array.</para>
16  /// <para>Хэш-код сегмента в массиве.</para>
17  /// </returns>
18  /// <remarks>
19  /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    ↳ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
20  /// </remarks>
21  [MethodImpl(MethodImplOptions.AggressiveInlining)]
22  public static int GenerateHashCode(this char[] array, int offset, int length)
23  {
24      var hashSeed = 5381;
25      var hashAccumulator = hashSeed;
26      fixed (char* arrayPointer = &array[offset])
27      {
28          for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
    ↳ < last; charPointer++)
29          {
30              hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
31          }
32      }
33      return hashAccumulator + (hashSeed * 1566083941);
34  }
35
36  /// <summary>
37  /// <para>Checks if all elements of two lists are equal.</para>
38  /// <para>Проверяет равны ли все элементы двух списков.</para>
39  /// </summary>
40  /// <param name="left"><para>The first compared array.</para><para>Первый массив для
    ↳ сравнения.</para></param>
41  /// <param name="leftOffset"><para>The offset from which reading of the specified number
    ↳ of elements in the first array starts.</para><para>Смещение, с которого начинается
    ↳ чтение элементов в первом массиве.</para></param>
42  /// <param name="length"><para>The number of checked elements.</para><para>Количество
    ↳ проверяемых элементов.</para></param>
43  /// <param name="right"><para>The second compared array.</para><para>Второй массив для
    ↳ сравнения.</para></param>
44  /// <param name="rightOffset"><para>The offset from which reading of the specified
    ↳ number of elements in the second array starts.</para><para>Смещение, с которого
    ↳ начинается чтение элементов в втором массиве.</para></param>
45  /// <returns>
46  /// <para>True if the segments of the passed arrays are equal to each other otherwise
    ↳ false.</para>
47  /// <para>True, если сегменты переданных массивов равны друг другу, иначе же
    ↳ false.</para>
48  /// </returns>
49  /// <remarks>
50  /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    ↳ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
51  /// </remarks>
52  [MethodImpl(MethodImplOptions.AggressiveInlining)]
53  public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
    ↳ right, int rightOffset)
54  {
55      fixed (char* leftPointer = &left[leftOffset])
56      {
57          fixed (char* rightPointer = &right[rightOffset])
58          {
59              char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
60              if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
    ↳ rightPointerCopy, ref length))
61              {
62                  return false;
63              }
64              CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
    ↳ ref length);
65              return length <= 0;
66          }
67      }
68  }

```

```

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

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 namespace Platform.Collections.Arrays
6 {
7     public static class GenericArrayExtensions
8     {
9         /// <summary>
10         /// <para>Checks if an array exists, if so, checks the array length using the index
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>
17         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
18         ↪ массива.</para></typeparam>
19         /// <param name="array"><para>Array that will participate in
20         ↪ verification.</para><para>Массив который будет участвовать в
21         ↪ проверке.</para></param>
22         /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
23         ↪ сравнения.</para></param>
24         /// <returns><para>Array element or default value.</para><para>Элемент массива или же
25         ↪ значение по умолчанию.</para></returns>
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
28         ↪ array.Length > index ? array[index] : default;
29
30         /// <summary>
31         /// <para>Checks whether the array exists, if so, checks the array length using the
32         ↪ index variable type long, and if the array length is greater than the index - return
33         ↪ 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
84   → массива.</para></typeparam>
85 /// <param name="array"><para>The array to copy.</para><para>Массив который необходимо
86   → скопировать.</para></param>
87 /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 public static T[] Clone<T>(this T[] array)
90 {
91     var copy = new T[array.LongLength];
92     Array.Copy(array, 0L, copy, 0L, array.LongLength);
93     return copy;
94 }
95
96 /// <summary>
97 /// <para>Shifts all the elements of the array by one position to the right.</para>
98 /// <para>Сдвигает вправо все элементы массива на одну позицию.</para>
99 /// </summary>
100 /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
101   → массива.</para></typeparam>
102 /// <param name="array"><para>The array to copy from.</para><para>Массив для
103   → копирования.</para></param>
104 /// <returns>
105 /// <para>Array with a shift of elements by one position.</para>
106 /// <para>Массив со сдвигом элементов на одну позицию.</para>
107 /// </returns>
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
110
111 /// <summary>
112 /// <para>Shifts all elements of the array to the right by the specified number of
113   → elements.</para>
114 /// <para>Сдвигает вправо все элементы массива на указанное количество элементов.</para>
115 /// </summary>
116 /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
117   → массива.</para></typeparam>
118 /// <param name="array"><para>The array to copy from.</para><para>Массив для
119   → копирования.</para></param>
120 /// <param name="skip"><para>The number of items to shift.</para><para>Количество
121   → сдвигаемых элементов.</para></param>
122 /// <returns>
123 /// <para>If the value of the shift variable is less than zero - an <see
124   → cref="NotImplementedException"/> exception is thrown, but if the value of the shift
125   → variable is 0 - an exact copy of the array is returned. Otherwise, an array is
126   → returned with the shift of the elements.</para>
127 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
128   → cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
129   → возвращается точная копия массива. Иначе возвращается массив со сдвигом
130   → элементов.</para>
131 /// </returns>
132 [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 public static IList<T> ShiftRight<T>(this T[] array, long shift)
134 {
135     if (shift < 0)
136     {
137         throw new NotImplementedException();
138     }
139     if (shift == 0)
140     {
141         return array.Clone<T>();
142     }
143 }

```

```

129     else
130     {
131         var restrictions = new T[array.LongLength + shift];
132         Array.Copy(array, 0L, restrictions, shift, array.LongLength);
133         return restrictions;
134     }
135 }
136
137 /// <summary>
138 /// <para>Adding in array the passed element at the specified position and increments
139   ↳ position value by one.</para>
140 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
141   ↳ значение position на единицу.</para>
142 /// </summary>
143 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
144   ↳ массива.</para></typeparam>
145 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
146   ↳ который необходимо добавить элемент.</para></param>
147 /// <param name="position"><para>A reference to the position of type int where the
148   ↳ element will be added.</para><para>Ссылка на позицию типа int, в которую будет
149   ↳ добавлен элемент.</para></param>
150 /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
151   ↳ который нужно добавить в массив.</para></param>
152 [MethodImpl(MethodImplOptions.AggressiveInlining)]
153 public static void Add<T>(this T[] array, ref int position, T element) =>
154   ↳ array[position++] = element;
155
156 /// <summary>
157 /// <para>Adding in array the passed element at the specified position and increments
158   ↳ position value by one.</para>
159 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
160   ↳ значение position на единицу.</para>
161 /// </summary>
162 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
163   ↳ массива.</para></typeparam>
164 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
165   ↳ который необходимо добавить элемент.</para></param>
166 /// <param name="position"><para>A reference to the position of type long where the
167   ↳ element will be added.</para><para>Ссылка на позицию типа long, в которую будет
168   ↳ добавлен элемент.</para></param>
169 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
170   ↳ который необходимо добавить в массив.</para></param>
171 [MethodImpl(MethodImplOptions.AggressiveInlining)]
172 public static void Add<T>(this T[] array, ref long position, T element) =>
173   ↳ array[position++] = element;
174
175 /// <summary>
176 /// <para>Adding in array the passed element, at the specified position, increments
177   ↳ position value by one and returns the value of the passed constant.</para>
178 /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
179   ↳ значение position на единицу и возвращает значение переданной константы.</para>
180 /// </summary>
181 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
182   ↳ массива.</para></typeparam>
183 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
184   ↳ возвращаемой константы.</para></typeparam>
185 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
186   ↳ который необходимо добавить элемент.</para></param>
187 /// <param name="position"><para>Reference to the position to which the element will be
188   ↳ added.</para><para>Ссылка на позицию, в которую будет добавлен
189   ↳ элемент.</para></param>
190 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
191   ↳ который необходимо добавить в массив.</para></param>
192 /// <param name="returnConstant"><para>The constant value that will be
193   ↳ returned.</para><para>Значение константы, которое будет возвращено.</para></param>
194 /// <returns>
195 /// <para>The constant value passed as an argument.</para>
196 /// <para>Значение константы, переданное в качестве аргумента.</para>
197 /// </returns>
198 [MethodImpl(MethodImplOptions.AggressiveInlining)]
199 public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
200   ↳ TElement[] array, ref long position, TElement element, TReturnConstant
201   ↳ returnConstant)
202 {
203     array.Add(ref position, element);
204     return returnConstant;
205 }

```



```

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

```

```

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     /// A что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
    ↪ 64 бит в массиве значений.
17     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
    ↪ байт в 8 байт.
18     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
    ↪ помощью которой можно быстро
19     /// проверять есть ли значения непосредственно далее (ниже по уровню).
20     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
21     /// </remarks>
22     public class BitString : IEquatable<BitString>

```

```

23 {
24     private static readonly byte[][] _bitsSetIn16Bits;
25     private long[] _array;
26     private long _length;
27     private long _minPositiveWord;
28     private long _maxPositiveWord;
29
30     public bool this[long index]
31     {
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         get => Get(index);
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         set => Set(index, value);
36     }
37
38     public long Length
39     {
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         get => _length;
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         set
44         {
45             if (_length == value)
46             {
47                 return;
48             }
49             Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
50             // Currently we never shrink the array
51             if (value > _length)
52             {
53                 var words = GetWordsCountFromIndex(value);
54                 var oldWords = GetWordsCountFromIndex(_length);
55                 if (words > _array.LongLength)
56                 {
57                     var copy = new long[words];
58                     Array.Copy(_array, copy, _array.LongLength);
59                     _array = copy;
60                 }
61                 else
62                 {
63                     // What is going on here?
64                     Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
65                 }
66                 // What is going on here?
67                 var mask = (int)(_length % 64);
68                 if (mask > 0)
69                 {
70                     _array[oldWords - 1] &= (1L << mask) - 1;
71                 }
72             }
73             else
74             {
75                 // Looks like minimum and maximum positive words are not updated
76                 throw new NotImplementedException();
77             }
78             _length = value;
79         }
80     }
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(0L, _array.LongLength, _array.LongLength /
166 ↪ threads);
167     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
168 ↪ MaxDegreeOfParallelism = threads }, range =>
169     {
170         var maximum = range.Item2;
171         for (var i = range.Item1; i < maximum; i++)
172         {
173             _array[i] = ~_array[i];
174         }
175     });
176     MarkBordersAsAllBitsSet();
177     TryShrinkBorders();
178     return this;
179 }

```

```

179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public BitString VectorNot()
181 {
182     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
183     {
184         return Not();
185     }
186     var step = Vector<long>.Count;
187     if (_array.Length < step)
188     {
189         return Not();
190     }
191     VectorNotLoop(_array, step, 0, _array.Length);
192     MarkBordersAsAllBitsSet();
193     TryShrinkBorders();
194     return this;
195 }
196
197 [MethodImpl(MethodImplOptions.AggressiveInlining)]
198 public BitString ParallelVectorNot()
199 {
200     var threads = Environment.ProcessorCount / 2;
201     if (threads <= 1)
202     {
203         return VectorNot();
204     }
205     if (!Vector.IsHardwareAccelerated)
206     {
207         return ParallelNot();
208     }
209     var step = Vector<long>.Count;
210     if (_array.Length < (step * threads))
211     {
212         return VectorNot();
213     }
214     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
215     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
216         ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
217         ↪ range.Item1, range.Item2));
218     MarkBordersAsAllBitsSet();
219     TryShrinkBorders();
220     return this;
221 }
222
223 [MethodImpl(MethodImplOptions.AggressiveInlining)]
224 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
225 {
226     var i = start;
227     var range = maximum - start - 1;
228     var stop = range - (range % step);
229     for (; i < stop; i += step)
230     {
231         (~new Vector<long>(array, i)).CopyTo(array, i);
232     }
233     for (; i < maximum; i++)
234     {
235         array[i] = ~array[i];
236     }
237 }
238
239 [MethodImpl(MethodImplOptions.AggressiveInlining)]
240 public BitString And(BitString other)
241 {
242     EnsureBitStringHasTheSameSize(other, nameof(other));
243     GetCommonOuterBorders(this, other, out long from, out long to);
244     var otherArray = other._array;
245     for (var i = from; i <= to; i++)
246     {
247         _array[i] &= otherArray[i];
248         RefreshBordersByWord(i);
249     }
250     return this;
251 }
252
253 [MethodImpl(MethodImplOptions.AggressiveInlining)]
254 public BitString ParallelAnd(BitString other)
255 {
256     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 {
        ↪ MaxDegreeOfParallelism = threads }, range =>
263     {
264         var maximum = range.Item2;
265         for (var i = range.Item1; i < maximum; i++)
266         {
267             _array[i] &= other._array[i];
268         }
269     });
270     MarkBordersAsAllBitsSet();
271     TryShrinkBorders();
272     return this;
273 }
274
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public BitString VectorAnd(BitString other)
277 {
278     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
279     {
280         return And(other);
281     }
282     var step = Vector<long>.Count;
283     if (_array.Length < step)
284     {
285         return And(other);
286     }
287     EnsureBitStringHasTheSameSize(other, nameof(other));
288     GetCommonOuterBorders(this, other, out int from, out int to);
289     VectorAndLoop(_array, other._array, step, from, to + 1);
290     MarkBordersAsAllBitsSet();
291     TryShrinkBorders();
292     return this;
293 }
294
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 public BitString ParallelVectorAnd(BitString other)
297 {
298     var threads = Environment.ProcessorCount / 2;
299     if (threads <= 1)
300     {
301         return VectorAnd(other);
302     }
303     if (!Vector.IsHardwareAccelerated)
304     {
305         return ParallelAnd(other);
306     }
307     var step = Vector<long>.Count;
308     if (_array.Length < (step * threads))
309     {
310         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 {
362         ↪ MaxDegreeOfParallelism = threads }, range =>
363     {
364         var maximum = range.Item2;
365         for (var i = range.Item1; i < maximum; i++)
366         {
367             _array[i] |= other._array[i];
368         }
369     });
370     MarkBordersAsAllBitsSet();
371     TryShrinkBorders();
372     return this;
373 }
374
375 [MethodImpl(MethodImplOptions.AggressiveInlining)]
376 public BitString VectorOr(BitString other)
377 {
378     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
379     {
380         return Or(other);
381     }
382     var step = Vector<long>.Count;
383     if (_array.Length < step)
384     {
385         return Or(other);
386     }
387     EnsureBitStringHasTheSameSize(other, nameof(other));
388     GetCommonOuterBorders(this, other, out int from, out int to);
389     VectorOrLoop(_array, other._array, step, from, to + 1);
390     MarkBordersAsAllBitsSet();
391     TryShrinkBorders();
392     return this;
393 }
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 public BitString ParallelVectorOr(BitString other)
397 {
398     var threads = Environment.ProcessorCount / 2;
399     if (threads <= 1)
400     {
401         return VectorOr(other);
402     }
403     if (!Vector.IsHardwareAccelerated)
404     {
405         return ParallelOr(other);
406     }

```



```

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);
463     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
464         ↪ MaxDegreeOfParallelism = threads }, range =>
465     {
466         var maximum = range.Item2;
467         for (var i = range.Item1; i < maximum; i++)
468         {
469             _array[i] ^= other._array[i];
470         }
471     });
472     MarkBordersAsAllBitsSet();
473     TryShrinkBorders();
474     return this;
475 }
476
477 [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 public BitString VectorXor(BitString other)
479 {
480     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
481     {
482         return Xor(other);
483     }
484 }

```

```

480     var step = Vector<long>.Count;
481     if (_array.Length < step)
482     {
483         return Xor(other);
484     }
485     EnsureBitStringHasTheSameSize(other, nameof(other));
486     GetCommonOuterBorders(this, other, out int from, out int to);
487     VectorXorLoop(_array, other._array, step, from, to + 1);
488     MarkBordersAsAllBitsSet();
489     TryShrinkBorders();
490     return this;
491 }
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]
494 public BitString ParallelVectorXor(BitString other)
495 {
496     var threads = Environment.ProcessorCount / 2;
497     if (threads <= 1)
498     {
499         return VectorXor(other);
500     }
501     if (!Vector.IsHardwareAccelerated)
502     {
503         return ParallelXor(other);
504     }
505     var step = Vector<long>.Count;
506     if (_array.Length < (step * threads))
507     {
508         return VectorXor(other);
509     }
510     EnsureBitStringHasTheSameSize(other, nameof(other));
511     GetCommonOuterBorders(this, other, out int from, out int to);
512     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
513     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
514         ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
515         ↪ step, range.Item1, range.Item2));
516     MarkBordersAsAllBitsSet();
517     TryShrinkBorders();
518     return this;
519 }
520
521 [MethodImpl(MethodImplOptions.AggressiveInlining)]
522 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
523     ↪ int maximum)
524 {
525     var i = start;
526     var range = maximum - start - 1;
527     var stop = range - (range % step);
528     for (; i < stop; i += step)
529     {
530         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
531     }
532     for (; i < maximum; i++)
533     {
534         array[i] ^= otherArray[i];
535     }
536 }
537
538 [MethodImpl(MethodImplOptions.AggressiveInlining)]
539 private void RefreshBordersByWord(long wordIndex)
540 {
541     if (_array[wordIndex] == 0)
542     {
543         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
544         {
545             _minPositiveWord++;
546         }
547         if (wordIndex == _maxPositiveWord && wordIndex != 0)
548         {
549             _maxPositiveWord--;
550         }
551     }
552     else
553     {
554         if (wordIndex < _minPositiveWord)
555         {
556             _minPositiveWord = wordIndex;
557         }
558     }
559 }

```

```

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,
    ↳ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1009 {
1010     for (var j = 0; j < bits00to15.Length; j++)
1011     {
1012         result.Add(bits00to15[j] + (i * 64));
1013     }

```



```

1014     for (var j = 0; j < bits16to31.Length; j++)
1015     {
1016         result.Add(bits16to31[j] + 16UL + (i * 64));
1017     }
1018     for (var j = 0; j < bits32to47.Length; j++)
1019     {
1020         result.Add(bits32to47[j] + 32UL + (i * 64));
1021     }
1022     for (var j = 0; j < bits48to63.Length; j++)
1023     {
1024         result.Add(bits48to63[j] + 48UL + (i * 64));
1025     }
1026 }
1027
1028 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
    ↪ bits32to47, byte[] bits48to63)
1030 {
1031     if (bits00to15.Length > 0)
1032     {
1033         return bits00to15[0] + (i * 64);
1034     }
1035     if (bits16to31.Length > 0)
1036     {
1037         return bits16to31[0] + 16 + (i * 64);
1038     }
1039     if (bits32to47.Length > 0)
1040     {
1041         return bits32to47[0] + 32 + (i * 64);
1042     }
1043     return bits48to63[0] + 48 + (i * 64);
1044 }
1045
1046 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1047 private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
    ↪ bits32to47, byte[] bits48to63)
1048 {
1049     if (bits48to63.Length > 0)
1050     {
1051         return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1052     }
1053     if (bits32to47.Length > 0)
1054     {
1055         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1056     }
1057     if (bits16to31.Length > 0)
1058     {
1059         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1060     }
1061     return bits00to15[bits00to15.Length - 1] + (i * 64);
1062 }
1063
1064 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1065 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↪ byte[] bits32to47, out byte[] bits48to63)
1066 {
1067     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1068     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,
    ↪ out long to)
1075 {
1076     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1077     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1078 }
1079
1080 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081 public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1082 {
1083     from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1084     to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1085 }
1086

```

```

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

```

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

```

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             if (string.IsNullOrEmpty(argument))
41             {
42                 throw new ArgumentException(message, argumentName);
43             }
44         }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
48             ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
49             ↪ argument, argumentName, null);
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
53             ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
54
55         #endregion
56
57         #region OnDebug
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
```

```

51     [Conditional("DEBUG")]
52     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
    ↪     ICollection<T> argument, string argumentName, string message) =>
    ↪     Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
53
54     [Conditional("DEBUG")]
55     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
    ↪     ICollection<T> argument, string argumentName) =>
    ↪     Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
56
57     [Conditional("DEBUG")]
58     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
    ↪     ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
59
60     [Conditional("DEBUG")]
61     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
    ↪     root, string argument, string argumentName, string message) =>
    ↪     Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
62
63     [Conditional("DEBUG")]
64     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
    ↪     root, string argument, string argumentName) =>
    ↪     Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
65
66     [Conditional("DEBUG")]
67     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
    ↪     root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
    ↪     null, null);
68
69     #endregion
70 }
71 }

```

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 ==
    ↪     null || collection.Count == 0;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
16         {
17             var equalityComparer = EqualityComparer<T>.Default;
18             return collection.All(item => equalityComparer.Equals(item, default));
19         }
20     }
21 }

```

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 GetOrCreateDefault<TKey, TValue>(this IDictionary<TKey, TValue>
    ↪     dictionary, TKey key)
13         {
14             dictionary.TryGetValue(key, out TValue value);
15             return value;
16         }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public static TValue GetOrCreateAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
    ↪     TKey key, Func<TKey, TValue> valueFactory)

```

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public static class CharIListExtensions
7      {
8          /// <summary>
9          /// <para>Generates a hash code for the entire list based on the values of its
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>
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public static bool EqualTo(this IList<char> left, IList<char> right) =>
49             ↪ left.EqualTo(right, ContentEqualTo);
50
51          /// <summary>
52          /// <para>Compares each element in the list for equality.</para>
53          /// <para>Сравнивает на равенство каждый элемент списка.</para>
54          /// </summary>
55          /// <param name="left"><para>The first compared list.</para><para>Первый список для
56             ↪ сравнения.</para></param>
57          /// <param name="right"><para>The second compared list.</para><para>Второй список для
58             ↪ сравнения.</para></param>
59          /// <returns>
60          /// <para>If at least one element of one list is not equal to the corresponding element
61             ↪ from another list returns false, otherwise - true.</para>
62          /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
63             ↪ из другого списка возвращает false, иначе - true.</para>
64          /// </returns>
65          [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
13         → списка.</para></typeparam>
14         /// <param name="left"><para>The first compared list.</para><para>Первый список для
15         → сравнения.</para></param>
16         /// <param name="right"><para>The second compared list.</para><para>Второй список для
17         → сравнения.</para></param>
18         /// <returns>
19         /// <para>
20         ///     A signed integer that indicates the relative values of <paramref name="left" />
21         → and <paramref name="right" /> lists' elements, as shown in the following table.
22         ///     <list type="table">
23         ///         <listheader>
24         ///             <term>Value</term>
25         ///             <description>Meaning</description>
26         ///         </listheader>
27         ///         <item>
28         ///             <term>Is less than zero</term>
29         ///             <description>First non equal element of <paramref name="left" /> list is
30         → less than first not equal element of <paramref name="right" /> list.</description>
31         ///         </item>
32         ///         <item>
33         ///             <term>Zero</term>
34         ///             <description>All elements of <paramref name="left" /> list equals to all
35         → elements of <paramref name="right" /> list.</description>
36         ///         </item>
37         ///         <item>
38         ///             <term>Is greater than zero</term>
39         ///             <description>First non equal element of <paramref name="left" /> list is
40         → greater than first not equal element of <paramref name="right" /> list.</description>
41         ///         </item>
42         ///     </list>
43         /// <para>
44         /// <para>
45         ///     Целое число со знаком, которое указывает относительные значения элементов
46         → списков <paramref name="left" /> и <paramref name="right" /> как показано в
47         → следующей таблице.
48         ///     <list type="table">
49         ///         <listheader>
50         ///             <term>Значение</term>
51         ///             <description>Смысл</description>
52         ///         </listheader>
53         ///         <item>
54         ///             <term>Меньше нуля</term>
55         ///             <description>Первый не равный элемент <paramref name="left" /> списка
56         → меньше первого неравного элемента <paramref name="right" /> списка.</description>
57         ///         </item>
58         ///         <item>
59         ///             <term>Ноль</term>
60         ///             <description>Все элементы <paramref name="left" /> списка равны всем
61         → элементам <paramref name="right" /> списка.</description>
62         ///         </item>
63         ///     </list>
64         /// </para>
65     }
66 }

```

```

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

```

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

```

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

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Collections.Lists
6  {
7      public static class IListExtensions
8      {
9          /// <summary>
10         /// <para>Gets the element from specified index if the list is not null and the index is
    → within the list's boundaries, otherwise it returns default value of type T.</para>
11         /// <para>Получает элемент из указанного индекса, если список не является null и индекс
    → находится в границах списка, в противном случае он возвращает значение по умолчанию
    → типа T.</para>
12         /// </summary>
13         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</typeparam>
14         /// <param name="list"><para>The checked list.</para><para>Проверяемый
    → список.</param>
15         /// <param name="index"><para>The index of element.</para><para>Индекс
    → элемента.</param>
16         /// <returns>

```

```

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

```



```

77     /// <para>Значение true в любом случае.</para>
78     /// </returns>
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
81     {
82         list.AddFirst(elements);
83         return true;
84     }
85
86     /// <summary>
87     /// <para>Adds a value to the list at the first index.</para>
88     /// <para>Добавляет значение в список по первому индексу.</para>
89     /// </summary>
90     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
91     /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
92     /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
    → который нужно добавить в список</para></param>
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
    → list.Add(elements[0]);
95
96     /// <summary>
97     /// <para>Adds all elements from other list to this list and returns true.</para>
98     /// <para>Добавляет все элементы из другого списка в этот список и возвращает
    → true.</para>
99     /// </summary>
100    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
101    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
102    /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
103    /// <returns>
104    /// <para>True value in any case.</para>
105    /// <para>Значение true в любом случае.</para>
106    /// </returns>
107    [MethodImpl(MethodImplOptions.AggressiveInlining)]
108    public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
109    {
110        list.AddAll(elements);
111        return true;
112    }
113
114    /// <summary>
115    /// <para>Adds all elements from other list to this list.</para>
116    /// <para>Добавляет все элементы из другого списка в этот список.</para>
117    /// </summary>
118    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
119    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
120    /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
121    [MethodImpl(MethodImplOptions.AggressiveInlining)]
122    public static void AddAll<T>(this IList<T> list, IList<T> elements)
123    {
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
153     ↳ списка.</para></typeparam>
154 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
155     ↳ нужно добавить значения.</para></param>
156 /// <param name="elements"><para>List of values to add.</para><para>Список значений
157     ↳ которые необходимо добавить.</para></param>
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
160     ↳ list.AddSkipFirst(elements, 1);
161
162 /// <summary>
163 /// <para>Adds values to the list skipping a specified number of first elements.</para>
164 /// <para>Добавляет в список значения пропуская определенное количество первых
165     ↳ элементов.</para>
166 /// </summary>
167 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
168     ↳ списка.</para></typeparam>
169 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
170     ↳ нужно добавить значения.</para></param>
171 /// <param name="elements"><para>List of values to add.</para><para>Список значений
172     ↳ которые необходимо добавить.</para></param>
173 /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
174     ↳ пропускаемых элементов.</para></param>
175 [MethodImpl(MethodImplOptions.AggressiveInlining)]
176 public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
177 {
178     for (var i = skip; i < elements.Count; i++)
179     {
180         list.Add(elements[i]);
181     }
182 }
183
184 /// <summary>
185 /// <para>Reads the number of elements in the list.</para>
186 /// <para>Считывает число элементов списка.</para>
187 /// </summary>
188 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
189     ↳ списка.</para></typeparam>
190 /// <param name="list"><para>The checked list.</para><para>Список для
191     ↳ проверки.</para></param>
192 /// <returns>
193 /// <para>The number of items contained in the list or 0.</para>
194 /// <para>Число элементов содержащихся в списке или же 0.</para>
195 /// </returns>
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
198
199 /// <summary>
200 /// <para>Compares two lists for equality.</para>
201 /// <para>Сравнивает два списка на равенство.</para>
202 /// </summary>
203 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
204     ↳ списка.</para></typeparam>
205 /// <param name="left"><para>The first compared list.</para><para>Первый список для
206     ↳ сравнения.</para></param>
207 /// <param name="right"><para>The second compared list.</para><para>Второй список для
208     ↳ сравнения.</para></param>
209 /// <returns>
210 /// <para>If the passed lists are equal to each other, true is returned, otherwise
211     ↳ false.</para>
212 /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
213     ↳ false.</para>
214 /// </returns>
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

200 public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
201     ↳ right, ContentEqualTo);
202
203 /// <summary>
204 /// <para>Compares two lists for equality.</para>
205 /// <para>Сравнивает два списка на равенство.</para>
206 /// </summary>
207 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
208     ↳ списка.</para></typeparam>
209 /// <param name="left"><para>The first compared list.</para><para>Первый список для
210     ↳ проверки.</para></param>
211 /// <param name="right"><para>The second compared list.</para><para>Второй список для
212     ↳ сравнения.</para></param>
213 /// <param name="contentEqualityComparer"><para>Function to test two lists for their
214     ↳ content equality.</para><para>Функция для проверки двух списков на равенство их
215     ↳ содержимого.</para></param>
216 /// <returns>
217 /// <para>If the passed lists are equal to each other, true is returned, otherwise
218     ↳ false.</para>
219 /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
220     ↳ false.</para>
221 /// </returns>
222 [MethodImpl(MethodImplOptions.AggressiveInlining)]
223 public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
224     ↳ IList<T>, bool> contentEqualityComparer)
225 {
226     if (ReferenceEquals(left, right))
227     {
228         return true;
229     }
230     var leftCount = left.GetCountOrZero();
231     var rightCount = right.GetCountOrZero();
232     if (leftCount == 0 && rightCount == 0)
233     {
234         return true;
235     }
236     if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
237     {
238         return false;
239     }
240     return contentEqualityComparer(left, right);
241 }
242
243 /// <summary>
244 /// <para>Compares each element in the list for identity.</para>
245 /// <para>Сравнивает на равенство каждый элемент списка.</para>
246 /// </summary>
247 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
248     ↳ списка.</para></typeparam>
249 /// <param name="left"><para>The first compared list.</para><para>Первый список для
250     ↳ сравнения.</para></param>
251 /// <param name="right"><para>The second compared list.</para><para>Второй список для
252     ↳ сравнения.</para></param>
253 /// <returns>
254 /// <para>If at least one element of one list is not equal to the corresponding element
255     ↳ from another list returns false, otherwise - true.</para>
256 /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
257     ↳ из другого списка возвращает false, иначе - true.</para>
258 /// </returns>
259 [MethodImpl(MethodImplOptions.AggressiveInlining)]
260 public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
261 {
262     var equalityComparer = EqualityComparer<T>.Default;
263     for (var i = left.Count - 1; i >= 0; --i)
264     {
265         if (!equalityComparer.Equals(left[i], right[i]))
266         {
267             return false;
268         }
269     }
270     return true;
271 }
272
273 /// <summary>
274 /// <para>Creates an array by copying all elements from the list that satisfy the
275     ↳ predicate. If no list is passed, null is returned.</para>

```

```

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

```

```

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

```

```

389     </item>
390     <item>
391         <term>Больше нуля</term>
392         <description>Первый не равный элемент <paramref name="left" /> списка
→ больше первого неравного элемента <paramref name="right" /> списка.</description>
393     </item>
394 </list>
395 </para>
396 </returns>
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 public static int CompareTo<T>(this IList<T> left, IList<T> right)
399 {
400     var comparer = Comparer<T>.Default;
401     var leftCount = left.GetCountOrZero();
402     var rightCount = right.GetCountOrZero();
403     var intermediateResult = leftCount.CompareTo(rightCount);
404     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
405     {
406         intermediateResult = comparer.Compare(left[i], right[i]);
407     }
408     return intermediateResult;
409 }
410
411 <summary>
412 <para>Skips one element in the list and builds an array from the remaining
→ elements.</para>
413 <para>Пропускает один элемент списка и составляет из оставшихся элементов
→ массив.</para>
414 </summary>
415 <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
→ списка.</para></typeparam>
416 <param name="list"><para>The list to copy from.</para><para>Список для
→ копирования.</para></param>
417 <returns>
418 <para>If the list is empty, returns an empty array, otherwise - an array with a
→ missing first element.</para>
419 <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
→ первым элементом.</para>
420 </returns>
421 [MethodImpl(MethodImplOptions.AggressiveInlining)]
422 public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
423
424 <summary>
425 <para>Skips the specified number of elements in the list and builds an array from
→ the remaining elements.</para>
426 <para>Пропускает указанное количество элементов списка и составляет из оставшихся
→ элементов массив.</para>
427 </summary>
428 <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
→ списка.</para></typeparam>
429 <param name="list"><para>The list to copy from.</para><para>Список для
→ копирования.</para></param>
430 <param name="skip"><para>The number of items to skip.</para><para>Количество
→ пропускаемых элементов.</para></param>
431 <returns>
432 <para>If the list is empty, or the number of skipped elements is greater than the
→ list, returns an empty array, otherwise - an array with the specified number of
→ missing elements.</para>
433 <para>Если список пуст, или количество пропускаемых элементов больше списка -
→ возвращает пустой массив, иначе - массив с указанным количеством пропущенных
→ элементов.</para>
434 </returns>
435 [MethodImpl(MethodImplOptions.AggressiveInlining)]
436 public static T[] SkipFirst<T>(this IList<T> list, int skip)
437 {
438     if (list.IsNullOrEmpty() || list.Count <= skip)
439     {
440         return Array.Empty<T>();
441     }
442     var result = new T[list.Count - skip];
443     for (int r = skip, w = 0; r < list.Count; r++, w++)
444     {
445         result[w] = list[r];
446     }
447     return result;
448 }
449

```

```

450 /// <summary>
451 /// <para>Shifts all the elements of the list by one position to the right.</para>
452 /// <para>Сдвигает вправо все элементы списка на одну позицию.</para>
453 /// </summary>
454 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
455 /// <param name="list"><para>The list to copy from.</para><para>Список для
    ↳ копирования.</para></param>
456 /// <returns>
457 /// <para>Array with a shift of elements by one position.</para>
458 /// <para>Массив со сдвигом элементов на одну позицию.</para>
459 /// </returns>
460 [MethodImpl(MethodImplOptions.AggressiveInlining)]
461 public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
462
463 /// <summary>
464 /// <para>Shifts all elements of the list to the right by the specified number of
    ↳ elements.</para>
465 /// <para>Сдвигает вправо все элементы списка на указанное количество элементов.</para>
466 /// </summary>
467 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
468 /// <param name="list"><para>The list to copy from.</para><para>Список для
    ↳ копирования.</para></param>
469 /// <param name="skip"><para>The number of items to shift.</para><para>Количество
    ↳ сдвигаемых элементов.</para></param>
470 /// <returns>
471 /// <para>If the value of the shift variable is less than zero - an <see
    ↳ cref="NotImplementedException"/> exception is thrown, but if the value of the shift
    ↳ variable is 0 - an exact copy of the array is returned. Otherwise, an array is
    ↳ returned with the shift of the elements.</para>
472 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
    ↳ cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
    ↳ возвращается точная копия массива. Иначе возвращается массив со сдвигом
    ↳ элементов.</para>
473 /// </returns>
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
476 {
477     if (shift < 0)
478     {
479         throw new NotImplementedException();
480     }
481     if (shift == 0)
482     {
483         return list.ToArray();
484     }
485     else
486     {
487         var result = new T[list.Count + shift];
488         for (int r = 0, w = shift; r < list.Count; r++, w++)
489         {
490             result[w] = list[r];
491         }
492         return result;
493     }
494 }
495 }
496 }

```

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

```

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

```

```

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

```



```

82
83     /// <summary>
84     /// <para>Adds an item to the end of the list and return constant.</para>
85     /// <para>Добавляет элемент в конец списка и возвращает константу.</para>
86     /// </summary>
87     /// <param name="element"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
88     /// <returns>
89     /// <para>Constant value in any case.</para>
90     /// <para>Значение константы в любом случае.</para>
91     /// </returns>
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public TReturnConstant AddAndReturnConstant(TElement element)
94     {
95         _list.Add(element);
96         return _returnConstant;
97     }
98
99     /// <summary>
100    /// <para>Adds a value to the list at the first index and return constant.</para>
101    /// <para>Добавляет значение в список по первому индексу и возвращает константу.</para>
102    /// </summary>
103    /// <param name="element"><para>Element to add.</para><para>Добавляемый
    → элемент.</para></param>
104    /// <returns>
105    /// <para>Constant value in any case.</para>
106    /// <para>Значение константы в любом случае.</para>
107    /// <returns></returns>
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    public TReturnConstant AddFirstAndReturnConstant(ICollection<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(ICollection<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(ICollection<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,
15             ↪ length) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override int GetHashCode()
19         {
20             // Base can be not an array, but still IList<char>
21             if (Base is char[] baseArray)
22             {
23                 return baseArray.GenerateHashCode(Offset, Length);
24             }
25             else
26             {
27                 return this.GenerateHashCode();
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 ?
49                 ↪ Equals(charSegment) : false;
50
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             public static implicit operator string(CharSegment segment)
53             {
54                 if (!(segment.Base is char[] array))
55                 {
56                     array = segment.Base.ToArray();
57                 }
58                 return new string(array, segment.Offset, segment.Length);
59             }
60
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             public override string ToString() => this;
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<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     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public bool Contains(T item) => IndexOf(item) >= 0;
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public void CopyTo(T[] array, int arrayIndex)
100     {
101         for (var i = 0; i < Length; i++)
102         {
103             array.Add(ref arrayIndex, this[i]);
104         }
105     }
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     public bool Remove(T item) => throw new NotSupportedException();
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public IEnumerator<T> GetEnumerator()
112     {
113         for (var i = 0; i < Length; i++)
114         {
115             yield return this[i];
116         }
117     }
118
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
121
122     #endregion
123 }
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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public virtual void WalkAll(ICollection<T> elements)
22         {
23             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
24                 ↪ offset <= maxOffset; offset++)
25             {
26                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
27                     ↪ offset; length <= maxLength; length++)
28                 {
29                     Iteration(CreateSegment(elements, offset, length));
30                 }
31             }
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);

```

```

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

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
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, TSegment> walker,
15            string @string) where TSegment : Segment<char> =>
16            walker.WalkAll(@string.ToCharArray());
17    }
18 }

```

1.26 ./csharp/Platform.Collections.Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment<T>].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.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)]
19         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
20            dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
21             : base(minimumStringSegmentLength)
22         {
23             Dictionary = dictionary;
24             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
29            dictionary, int minimumStringSegmentLength) : this(dictionary,
30            minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
34            dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
35            DefaultResetDictionaryOnEachWalk) { }
36     }
37 }

```

```

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

33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪     this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
    ↪     this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public override void WalkAll(ICollection<T> elements)
42     {
43         if (_resetDictionaryOnEachWalk)
44         {
45             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
46             Dictionary = new Dictionary<TSegment, long>((int)capacity);
47         }
48         base.WalkAll(elements);
49     }

50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override long GetSegmentFrequency(TSegment segment) =>
    ↪     Dictionary.GetOrDefault(segment);

53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
    ↪     Dictionary[segment] = frequency;

56 }
57 }

```

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> :
    ↪     DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪         dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
    ↪         base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }

12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪         dictionary, int minimumStringSegmentLength) : base(dictionary,
    ↪         minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪         dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪         DefaultResetDictionaryOnEachWalk) { }

18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪         bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪         resetDictionaryOnEachWalk) { }

21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪         base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
    ↪         base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

27     }
28 }

```

```

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,
      ↪ TSegment>
8          where TSegment : Segment<T>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            ↪ base(minimumStringSegmentLength) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected override void Iteration(TSegment segment)
18         {
19             var frequency = GetSegmentFrequency(segment);
20             if (frequency == 1)
21             {
22                 OnDuplicateFound(segment);
23             }
24             SetSegmentFrequency(segment, frequency + 1);
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected abstract void OnDuplicateFound(TSegment segment);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract long GetSegmentFrequency(TSegment segment);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
35     }
36 }

```

```

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,
      ↪ Segment<T>>
6      {
7      }
8  }

```

```

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) =>
            ↪ set.Remove(element);
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
18         {
19             set.Add(element);
20             return true;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
25         {
26             AddFirst(set, elements);

```

```

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

```

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

```

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

```



```

30     public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
31         ↪ _set.AddFirstAndReturnTrue(elements);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public bool AddAllAndReturnTrue(IList<TElement> elements) =>
35         ↪ _set.AddAllAndReturnTrue(elements);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public bool AddSkipFirstAndReturnTrue(IList<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(IList<TElement> elements)
50     {
51         _set.AddFirst(elements);
52         return _returnConstant;
53     }
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
57     {
58         _set.AddAll(elements);
59         return _returnConstant;
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public TReturnConstant AddSkipFirstAndReturnConstant(IList<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 }

```

```

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 }

```

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

```

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

```

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

```

```

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
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public Node SetChild(object key) => SetChildValue(null, key);
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public Node SetChildValue(object value, params object[] keys)
83 {
84     var node = this;
85     for (var i = 0; i < keys.Length; i++)
86     {
87         node = SetChildValue(value, keys[i]);
88     }
89     node.Value = value;
90     return node;
91 }
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

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

```

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

```

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

```

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

```

```

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, 53
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 53
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 55
- ./csharp/Platform.Collections.Tests/ListTests.cs, 56
- ./csharp/Platform.Collections.Tests/StringTests.cs, 56
- ./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, 5
- ./csharp/Platform.Collections/BitString.cs, 11
- ./csharp/Platform.Collections/BitStringExtensions.cs, 26
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 26
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 26
- ./csharp/Platform.Collections/EnsureExtensions.cs, 27
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 28
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 28
- ./csharp/Platform.Collections/Lists/CharIListExtensions.cs, 29
- ./csharp/Platform.Collections/Lists/IListComparer.cs, 30
- ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 31
- ./csharp/Platform.Collections/Lists/IListExtensions.cs, 31
- ./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, 44
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 44
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 45
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 45
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 45
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 46
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 46
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 47
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 47
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 48
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 49
- ./csharp/Platform.Collections/Stacks/IStack.cs, 49
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 50
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 50
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 50
- ./csharp/Platform.Collections/StringExtensions.cs, 50
- ./csharp/Platform.Collections/Trees/Node.cs, 51