

LinksPlatform's Platform.Collections Class Library

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

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

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Arrays
7 {
8     public class ArrayFiller<TElement>
9     {
10         protected readonly TElement[] _array;
11         protected long _position;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ArrayFiller(TElement[] array, long offset)
15         {
16             _array = array;
17             _position = offset;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ArrayFiller(TElement[] array) : this(array, 0) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _array[_position++] = element;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
28             ↪ _position, element, true);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
32             ↪ _array.AddFirstAndReturnConstant(ref _position, elements, true);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
36             ↪ _array.AddAllAndReturnConstant(ref _position, elements, true);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
40             ↪ _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
41     }
42 }
```

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static class ArrayPool
8      {
9          public static readonly int DefaultSizesAmount = 512;
10         public static readonly int DefaultMaxArraysPerSize = 32;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
17     }
18 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Stacks;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Arrays
10 {
11     /// <remarks>
12     /// Original idea from
13     ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
14     /// </remarks>
15     public class ArrayPool<T>
16     {
17         // May be use Default class for that later.
18         [ThreadStatic]
19         private static ArrayPool<T> _threadInstance;
20         internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
           ↪ ArrayPool<T>());
21
22         private readonly int _maxArraysPerSize;
23         private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
           ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
36         {
37             var destination = AllocateDisposable(size);
38             T[] sourceArray = source;
39             if (!sourceArray.IsNullOrEmpty())
40             {
41                 T[] destinationArray = destination;
42                 Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
                   ↪ sourceArray.LongLength);
43                 source.Dispose();
44             }
45             return destination;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public virtual void Clear() => _pool.Clear();
50     }
51 }

```

```

50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
    ↪     _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public virtual void Free(T[] array)
55     {
56         if (array.IsNullOrEmpty())
57         {
58             return;
59         }
60         var stack = _pool.GetOrAdd(array.LongLength, size => new
    ↪         Stack<T[]>(_maxArraysPerSize));
61         if (stack.Count == _maxArraysPerSize) // Stack is full
62         {
63             return;
64         }
65         stack.Push(array);
66     }
67 }
68 }

```

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

```

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

```

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

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static unsafe class CharArrayExtensions
8      {
9          /// <remarks>
10         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11         ↪     ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12         /// </remarks>
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static int GenerateHashCode(this char[] array, int offset, int length)
15         {
16             var hashSeed = 5381;
17             var hashAccumulator = hashSeed;
18             fixed (char* arrayPointer = &array[offset])
19             {
20                 for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
21                     ↪     ↪ < last; charPointer++)
22                 {
23                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
24                 }
25             }
26             return hashAccumulator + (hashSeed * 1566083941);
27         }
28
29         /// <remarks>
30         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
31         ↪     ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
32         /// </remarks>
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
35             ↪     ↪ right, int rightOffset)

```

```

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

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Arrays
8 {
9     public static class GenericArrayExtensions
10     {
11         /// <summary>
12         /// <para>Checks if an array exists, if so, checks the array length using the index
13         ↪ variable type int, and if the array length is greater than the index - return
14         ↪ array[index], otherwise - default value.</para>
15         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
16         ↪ помощью переменной index, и если длина массива больше индекса - возвращает
17         ↪ array[index], иначе - значение по умолчанию.</para>

```

```

14  /// </summary>
15  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
16  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
17  /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
    → сравнения.</para></param>
18  /// <returns><para>Array element or default value.</para><para>Элемент массива или же
    → значение по умолчанию.</para></returns>
19  [MethodImpl(MethodImplOptions.AggressiveInlining)]
20  public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
    → array.Length > index ? array[index] : default;
21
22  /// <summary>
23  /// <para>Checks whether the array exists, if so, checks the array length using the
    → index variable type long, and if the array length is greater than the index - return
    → array[index], otherwise - default value.</para>
24  /// <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>
25  /// </summary>
26  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
27  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
28  /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
    → для сравнения.</para></param>
29  /// <returns><para>Array element or default value.</para><para>Элемент массива или же
    → значение по умолчанию.</para></returns>
30  [MethodImpl(MethodImplOptions.AggressiveInlining)]
31  public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
    → array.LongLength > index ? array[index] : default;
32
33  /// <summary>
34  /// <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>
35  /// <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>
36  /// </summary>
37  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
38  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
39  /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
    → сравнения.</para></param>
40  /// <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>
41  /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
    → в противном случае false.</para></returns>
42  [MethodImpl(MethodImplOptions.AggressiveInlining)]
43  public static bool TryGetElement<T>(this T[] array, int index, out T element)
44  {
45      if (array != null && array.Length > index)
46      {
47          element = array[index];
48          return true;
49      }
50      else
51      {
52          element = default;
53          return false;
54      }
55  }
56
57  /// <summary>
58  /// <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>

```

```

59  /// <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>
60  /// </summary>
61  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
62  /// <param name="array"><para>Array that will participate in
    → verification.</para><para>Массив который будет участвовать в
    → проверке.</para></param>
63  /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
    → для сравнения.</para></param>
64  /// <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>
65  /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
    → в противном случае false</para></returns>
66  [MethodImpl(MethodImplOptions.AggressiveInlining)]
67  public static bool TryGetElement<T>(this T[] array, long index, out T element)
68  {
69      if (array != null && array.LongLength > index)
70      {
71          element = array[index];
72          return true;
73      }
74      else
75      {
76          element = default;
77          return false;
78      }
79  }
80
81  /// <summary>
82  /// <para>Copying of elements from one array to another array.</para>
83  /// <para>Копирует элементы из одного массива в другой массив.</para>
84  /// </summary>
85  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
86  /// <param name="array"><para>The array to copy.</para><para>Массив который необходимо
    → 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  [MethodImpl(MethodImplOptions.AggressiveInlining)]
97  public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
98
99  /// <summary>
100  /// <para>Extending the array boundaries to shift elements and then copying it, but with
    → the condition that shift > 0. If shift == 0, the extension will not occur, but
    → cloning will still be applied. If shift < 0, a NotImplementedException is
    → thrown.</para>
101  /// <para>Расширение границ массива на shift элементов и последующее его копирование, но
    → с условием что shift > 0. Если же shift == 0 - расширение не произойдет, но
    → клонирование все равно применится. Если shift < 0, выбросится исключение
    → NotImplementedException.</para>
102  /// </summary>
103  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
104  /// <param name="array"><para>Array to expand Elements.</para><para>Массив для
    → расширения элементов.</para></param>
105  /// <param name="shift"><para>The number to expand the array</para><para>Число на
    → которое необходимо расширить массив.</para></param>
106  /// <returns>
107  /// <para>If the value of the shift variable is < 0, it returns a
    → NotImplementedException exception. If shift == 0, the array is cloned, but the
    → extension will not be applied. Otherwise, if the value shift > 0, the length of the
    → array is increased by the shift elements and the array is cloned.</para>
108  /// <para>Если значение переменной shift < 0, возвращается исключение
    → NotImplementedException. Если shift == 0, то массив клонируется, но расширение не
    → применяется. В противном случае, если значение shift > 0, длина массива
    → увеличивается на shift элементов и массив клонируется.</para>

```

```

109 /// </returns>
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public static IList<T> ShiftRight<T>(this T[] array, long shift)
112 {
113     if (shift < 0)
114     {
115         throw new NotImplementedException();
116     }
117     if (shift == 0)
118     {
119         return array.Clone<T>();
120     }
121     else
122     {
123         var restrictions = new T[array.LongLength + shift];
124         Array.Copy(array, 0L, restrictions, shift, array.LongLength);
125         return restrictions;
126     }
127 }
128
129 /// <summary>
130 /// <para>One of the array values with index on variable position++ type int is passed
131   → to the element variable.</para>
132 /// <para>Одно из значений массива с индексом переменной position++ типа int назначается
133   → в переменную element.</para>
134 /// </summary>
135 /// <param name="array"><para>An array whose specific value will be assigned to the
136   → element variable.</para><para>Массив, определенное значений которого присваивается
137   → переменной element</para></param>
138 /// <param name="position"><para>Reference to a position in an array of int
139   → type.</para><para>Ссылка на позицию в массиве типа int.</para></param>
140 /// <param name="element"><para>The variable which needs to be assigned a specific value
141   → from the array.</para><para>Переменная, которой нужно присвоить определенное
142   → значение из массива.</para></param>
143 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
144   → массива.</typeparam>
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public static void Add<T>(this T[] array, ref int position, T element) =>
147   → array[position++] = element;
148
149 /// <summary>
150 /// <para>One of the array values with index on variable position++ type long is passed
151   → to the element variable.</para>
152 /// <para>Одно из значений массива с индексом переменной position++ типа long
153   → назначается в переменную element.</para>
154 /// </summary>
155 /// <param name="array"><para>An array whose specific value will be assigned to the
156   → element variable.</para><para>Массив, определенное значений которого присваивается
157   → переменной element</para></param>
158 /// <param name="position"><para>Reference to a position in an array of long
159   → type.</para><para>Ссылка на позицию в массиве типа long.</para></param>
160 /// <param name="element"><para>The variable which needs to be assigned a specific value
161   → from the array.</para><para>Переменная, которой нужно присвоить определенное
162   → значение из массива.</para></param>
163 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
164   → массива.</typeparam>
165 [MethodImpl(MethodImplOptions.AggressiveInlining)]
166 public static void Add<T>(this T[] array, ref long position, T element) =>
167   → array[position++] = element;
168
169 [MethodImpl(MethodImplOptions.AggressiveInlining)]
170 public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
171   → TElement[] array, ref long position, TElement element, TReturnConstant
172   → returnConstant)
173 {
174     array.Add(ref position, element);
175     return returnConstant;
176 }
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
180   → array[position++] = elements[0];
181
182 [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
184   → TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
185   → returnConstant)

```

```

163 {
164     array.AddFirst(ref position, elements);
165     return returnConstant;
166 }
167
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
    → returnConstant)
170 {
171     array.AddAll(ref position, elements);
172     return returnConstant;
173 }
174
175 /// <summary>
176 /// <para>Adding a collection of elements starting from a specific position.</para>
177 /// <para>Добавляет коллекции элементов начиная с определенной позиции.</para>
178 /// </summary>
179 /// <param name="array"><para>An array to which the collection of elements will be
    → added.</para><para>Массив в который будет добавлена коллекция
    → элементов.</para></param>
180 /// <param name="position"><para>The position from which to start adding
    → elements.</para><para>Позиция с которой начнется добавление элементов.</para></param>
181 /// <param name="elements"><para>Added all collection of elements to
    → array.</para><para>Добавляется вся коллекция элементов в массив. </para></param>
182 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
183 [MethodImpl(MethodImplOptions.AggressiveInlining)]
184 public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
185 {
186     for (var i = 0; i < elements.Count; i++)
187     {
188         array.Add(ref position, elements[i]);
189     }
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
    → TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
    → TReturnConstant returnConstant)
194 {
195     array.AddSkipFirst(ref position, elements);
196     return returnConstant;
197 }
198
199 /// <summary>
200 /// <para>Adds all elements except the first.</para>
201 /// <para>Добавляет все элементы, кроме первого.</para>
202 /// </summary>
203 /// <param name="array"><para>An array to which the collection of elements will be
    → added.</para><para>Массив в который будет добавлена коллекция
    → элементов.</para></param>
204 /// <param name="position"><para>The position from which to start adding
    → elements.</para><para>Позиция, с которой начинается добавление
    → элементов.</para></param>
205 /// <param name="elements"><para>List of added elements.</para><para>Список добавляемых
    → элементов.</para></param>
206 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
    → => array.AddSkipFirst(ref position, elements, 1);
209
210 /// <summary>
211 /// <para>Adds all but the first element, skipping a specified number of elements.</para>
212 /// <para>Добавляет все элементы, кроме первого, пропуская определенное количество
    → элементов.</para>
213 /// </summary>
214 /// <param name="array"><para>An array to which the collection of elements will be
    → added.</para><para>Массив в который будет добавлена коллекция
    → элементов.</para></param>
215 /// <param name="position"><para>The position from which to start adding
    → elements.</para><para>Позиция, с которой начинается добавление
    → элементов.</para></param>
216 /// <param name="elements"><para>List of added elements.</para><para>Список добавляемых
    → элементов.</para></param>
217 /// <param name="skip"></param>

```



```

218     /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>
219     [MethodImpl(MethodImplOptions.AggressiveInlining)]
220     public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
    ↪ int skip)
221     {
222         for (var i = skip; i < elements.Count; i++)
223         {
224             array.Add(ref position, elements[i]);
225         }
226     }
227 }
228 }

```

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

```

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

```

```

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 }
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public BitString VectorNot()
183 {
184     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
185     {
186         return Not();
187     }
188     var step = Vector<long>.Count;
189     if (_array.Length < step)
190     {
191         return Not();
192     }
193     VectorNotLoop(_array, step, 0, _array.Length);
194     MarkBordersAsAllBitsSet();
195     TryShrinkBorders();
196     return this;
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public BitString ParallelVectorNot()
201 {
202     var threads = Environment.ProcessorCount / 2;
203     if (threads <= 1)
204     {
205         return VectorNot();
206     }
207     if (!Vector.IsHardwareAccelerated)
208     {
209         return ParallelNot();
210     }
211     var step = Vector<long>.Count;
212     if (_array.Length < (step * threads))
213     {
214         return VectorNot();
215     }
216     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;
257         if (threads <= 1)
258         {
259             return And(other);
260         }
261         EnsureBitStringHasTheSameSize(other, nameof(other));
262         GetCommonOuterBorders(this, other, out long from, out long to);
263         var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
264         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
265             ↪ MaxDegreeOfParallelism = threads }, range =>
266         {
267             var maximum = range.Item2;
268             for (var i = range.Item1; i < maximum; i++)
269             {
270                 _array[i] &= other._array[i];
271             }
272         });
273         MarkBordersAsAllBitsSet();
274         TryShrinkBorders();
275         return this;
276     }
277
278     [MethodImpl(MethodImplOptions.AggressiveInlining)]
279     public BitString VectorAnd(BitString other)
280     {
281         if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
282         {
283             return And(other);
284         }
285         var step = Vector<long>.Count;
286         if (_array.Length < step)
287         {
288             return And(other);
289         }
290         EnsureBitStringHasTheSameSize(other, nameof(other));
291         GetCommonOuterBorders(this, other, out int from, out int to);
292         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 {
316         ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
317         ↪ step, range.Item1, range.Item2));
318     MarkBordersAsAllBitsSet();
319     TryShrinkBorders();
320     return this;
321 }
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
325     ↪ int maximum)
326 {
327     var i = start;
328     var range = maximum - start - 1;
329     var stop = range - (range % step);
330     for (; i < stop; i += step)
331     {
332         (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
333     }
334     for (; i < maximum; i++)
335     {
336         array[i] &= otherArray[i];
337     }
338 }
339
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 public BitString Or(BitString other)
342 {
343     EnsureBitStringHasTheSameSize(other, nameof(other));
344     GetCommonOuterBorders(this, other, out long from, out long to);
345     for (var i = from; i <= to; i++)
346     {
347         _array[i] |= other._array[i];
348         RefreshBordersByWord(i);
349     }
350     return this;
351 }
352
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public BitString ParallelOr(BitString other)
355 {
356     var threads = Environment.ProcessorCount / 2;
357     if (threads <= 1)
358     {
359         return Or(other);
360     }
361     EnsureBitStringHasTheSameSize(other, nameof(other));
362     GetCommonOuterBorders(this, other, out long from, out long to);
363     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
364     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
365         ↪ MaxDegreeOfParallelism = threads }, range =>
366     {
367         var maximum = range.Item2;

```

```

364         for (var i = range.Item1; i < maximum; i++)
365         {
366             _array[i] |= other._array[i];
367         }
368     });
369     MarkBordersAsAllBitsSet();
370     TryShrinkBorders();
371     return this;
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 public BitString VectorOr(BitString other)
376 {
377     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
378     {
379         return Or(other);
380     }
381     var step = Vector<long>.Count;
382     if (_array.Length < step)
383     {
384         return Or(other);
385     }
386     EnsureBitStringHasTheSameSize(other, nameof(other));
387     GetCommonOuterBorders(this, other, out int from, out int to);
388     VectorOrLoop(_array, other._array, step, from, to + 1);
389     MarkBordersAsAllBitsSet();
390     TryShrinkBorders();
391     return this;
392 }
393
394 [MethodImpl(MethodImplOptions.AggressiveInlining)]
395 public BitString ParallelVectorOr(BitString other)
396 {
397     var threads = Environment.ProcessorCount / 2;
398     if (threads <= 1)
399     {
400         return VectorOr(other);
401     }
402     if (!Vector.IsHardwareAccelerated)
403     {
404         return ParallelOr(other);
405     }
406     var step = Vector<long>.Count;
407     if (_array.Length < (step * threads))
408     {
409         return VectorOr(other);
410     }
411     EnsureBitStringHasTheSameSize(other, nameof(other));
412     GetCommonOuterBorders(this, other, out int from, out int to);
413     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
414     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
415         ↪ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
416         ↪ step, range.Item1, range.Item2));
417     MarkBordersAsAllBitsSet();
418     TryShrinkBorders();
419     return this;
420 }
421
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
424     ↪ int maximum)
425 {
426     var i = start;
427     var range = maximum - start - 1;
428     var stop = range - (range % step);
429     for (; i < stop; i += step)
430     {
431         (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
432     }
433     for (; i < maximum; i++)
434     {
435         array[i] |= otherArray[i];
436     }
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 public BitString Xor(BitString other)
441 {

```

```

439     EnsureBitStringHasTheSameSize(other, nameof(other));
440     GetCommonOuterBorders(this, other, out long from, out long to);
441     for (var i = from; i <= to; i++)
442     {
443         _array[i] ^= other._array[i];
444         RefreshBordersByWord(i);
445     }
446     return this;
447 }
448
449 [MethodImpl(MethodImplOptions.AggressiveInlining)]
450 public BitString ParallelXor(BitString other)
451 {
452     var threads = Environment.ProcessorCount / 2;
453     if (threads <= 1)
454     {
455         return Xor(other);
456     }
457     EnsureBitStringHasTheSameSize(other, nameof(other));
458     GetCommonOuterBorders(this, other, out long from, out long to);
459     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
460     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
461         ↪ MaxDegreeOfParallelism = threads }, range =>
462     {
463         var maximum = range.Item2;
464         for (var i = range.Item1; i < maximum; i++)
465         {
466             _array[i] ^= other._array[i];
467         }
468     });
469     MarkBordersAsAllBitsSet();
470     TryShrinkBorders();
471     return this;
472 }
473
474 [MethodImpl(MethodImplOptions.AggressiveInlining)]
475 public BitString VectorXor(BitString other)
476 {
477     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
478     {
479         return Xor(other);
480     }
481     var step = Vector<long>.Count;
482     if (_array.Length < step)
483     {
484         return Xor(other);
485     }
486     EnsureBitStringHasTheSameSize(other, nameof(other));
487     GetCommonOuterBorders(this, other, out int from, out int to);
488     VectorXorLoop(_array, other._array, step, from, to + 1);
489     MarkBordersAsAllBitsSet();
490     TryShrinkBorders();
491     return this;
492 }
493
494 [MethodImpl(MethodImplOptions.AggressiveInlining)]
495 public BitString ParallelVectorXor(BitString other)
496 {
497     var threads = Environment.ProcessorCount / 2;
498     if (threads <= 1)
499     {
500         return VectorXor(other);
501     }
502     if (!Vector.IsHardwareAccelerated)
503     {
504         return ParallelXor(other);
505     }
506     var step = Vector<long>.Count;
507     if (_array.Length < (step * threads))
508     {
509         return VectorXor(other);
510     }
511     EnsureBitStringHasTheSameSize(other, nameof(other));
512     GetCommonOuterBorders(this, other, out int from, out int to);
513     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
514     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
515         ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
516         ↪ step, range.Item1, range.Item2));

```

```

514     MarkBordersAsAllBitsSet();
515     TryShrinkBorders();
516     return this;
517 }
518
519 [MethodImpl(MethodImplOptions.AggressiveInlining)]
520 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
    ↪ int maximum)
521 {
522     var i = start;
523     var range = maximum - start - 1;
524     var stop = range - (range % step);
525     for (; i < stop; i += step)
526     {
527         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
528     }
529     for (; i < maximum; i++)
530     {
531         array[i] ^= otherArray[i];
532     }
533 }
534
535 [MethodImpl(MethodImplOptions.AggressiveInlining)]
536 private void RefreshBordersByWord(long wordIndex)
537 {
538     if (_array[wordIndex] == 0)
539     {
540         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
541         {
542             _minPositiveWord++;
543         }
544         if (wordIndex == _maxPositiveWord && wordIndex != 0)
545         {
546             _maxPositiveWord--;
547         }
548     }
549     else
550     {
551         if (wordIndex < _minPositiveWord)
552         {
553             _minPositiveWord = wordIndex;
554         }
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) :
876     ↪ false;
877
878 [MethodImpl(MethodImplOptions.AggressiveInlining)]
879 public bool Equals(BitString other)
880 {
881     if (_length != other._length)
882     {
883         return false;
884     }
885     var otherArray = other._array;
886     if (_array.Length != otherArray.Length)
887     {
888         return false;
889     }
890     if (_minPositiveWord != other._minPositiveWord)
891     {
892         return false;
893     }
894     if (_maxPositiveWord != other._maxPositiveWord)
895     {
896         return false;
897     }
898     GetCommonBorders(this, other, out ulong from, out ulong to);
899     for (var i = from; i <= to; i++)
900     {
901         if (_array[i] != otherArray[i])
902         {
903             return false;
904         }
905     }
906     return true;
907 }

```

```

907 [MethodImpl(MethodImplOptions.AggressiveInlining)]
908 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
909 {
910     Ensure.Always.ArgumentNotNull(other, argumentName);
911     if (_length != other._length)
912     {
913         throw new ArgumentException("Bit string must be the same size.", argumentName);
914     }
915 }
916
917 [MethodImpl(MethodImplOptions.AggressiveInlining)]
918 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
919
920 [MethodImpl(MethodImplOptions.AggressiveInlining)]
921 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
922
923 [MethodImpl(MethodImplOptions.AggressiveInlining)]
924 private void GetBorders(out long from, out long to)
925 {
926     from = _minPositiveWord;
927     to = _maxPositiveWord;
928 }
929
930 [MethodImpl(MethodImplOptions.AggressiveInlining)]
931 private void GetBorders(out ulong from, out ulong to)
932 {
933     from = (ulong)_minPositiveWord;
934     to = (ulong)_maxPositiveWord;
935 }
936
937 [MethodImpl(MethodImplOptions.AggressiveInlining)]
938 private void SetBorders(long from, long to)
939 {
940     _minPositiveWord = from;
941     _maxPositiveWord = to;
942 }
943
944 [MethodImpl(MethodImplOptions.AggressiveInlining)]
945 private Range<long> GetValidIndexRange() => (0, _length - 1);
946
947 [MethodImpl(MethodImplOptions.AggressiveInlining)]
948 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
949
950 [MethodImpl(MethodImplOptions.AggressiveInlining)]
951 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
952     ↪ wordValue)
953 {
954     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
955     ↪ bits32to47, out byte[] bits48to63);
956     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
957     ↪ bits48to63);
958 }
959
960 [MethodImpl(MethodImplOptions.AggressiveInlining)]
961 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
962     ↪ wordValue)
963 {
964     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
965     ↪ bits32to47, out byte[] bits48to63);
966     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
967     ↪ bits48to63);
968 }
969
970 [MethodImpl(MethodImplOptions.AggressiveInlining)]
971 private static long CountSetBitsForWord(long word)
972 {
973     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
974     ↪ out byte[] bits48to63);
975     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
976     ↪ bits48to63.LongLength;
977 }
978
979 [MethodImpl(MethodImplOptions.AggressiveInlining)]
980 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
981 {
982     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
983     ↪ 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[]
983             ↪ bits32to47, out byte[] bits48to63);
984         return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
985     }
986
987     [MethodImpl(MethodImplOptions.AggressiveInlining)]
988     private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
989     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
990     {
991         for (var j = 0; j < bits00to15.Length; j++)
992         {
993             result.Add(bits00to15[j] + (i * 64));
994         }
995         for (var j = 0; j < bits16to31.Length; j++)
996         {
997             result.Add(bits16to31[j] + 16 + (i * 64));
998         }
999         for (var j = 0; j < bits32to47.Length; j++)
1000         {
1001             result.Add(bits32to47[j] + 32 + (i * 64));
1002         }
1003         for (var j = 0; j < bits48to63.Length; j++)
1004         {
1005             result.Add(bits48to63[j] + 48 + (i * 64));
1006         }
1007     }
1008
1009     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010     private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1011     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1012     {
1013         for (var j = 0; j < bits00to15.Length; j++)
1014         {
1015             result.Add(bits00to15[j] + (i * 64));
1016         }
1017         for (var j = 0; j < bits16to31.Length; j++)
1018         {
1019             result.Add(bits16to31[j] + 16UL + (i * 64));
1020         }
1021         for (var j = 0; j < bits32to47.Length; j++)
1022         {
1023             result.Add(bits32to47[j] + 32UL + (i * 64));
1024         }
1025         for (var j = 0; j < bits48to63.Length; j++)
1026         {
1027             result.Add(bits48to63[j] + 48UL + (i * 64));
1028         }
1029     }
1030
1031     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1032     private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1033     ↪ bits32to47, byte[] bits48to63)
1034     {
1035         if (bits00to15.Length > 0)
1036         {
1037             return bits00to15[0] + (i * 64);
1038         }
1039         if (bits16to31.Length > 0)
1040         {
1041             return bits16to31[0] + 16 + (i * 64);
1042         }
1043         if (bits32to47.Length > 0)
1044         {
1045             return bits32to47[0] + 32 + (i * 64);
1046         }
1047         return bits48to63[0] + 48 + (i * 64);
1048     }
1049
1050     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1051     private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1052     ↪ 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 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
28     ↳ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
29     ↳ argumentName, null);
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
32     ↳ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
35     ↳ string argument, string argumentName, string message)
36 {
37     if (string.IsNullOrEmpty(argument))
38     {
39         throw new ArgumentException(message, argumentName);
40     }
41 }
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
44     ↳ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
45     ↳ argument, argumentName, null);
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
48     ↳ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
49 #endregion
50 #region OnDebug
51 [Conditional("DEBUG")]
52 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
53     ↳ ICollection<T> argument, string argumentName, string message) =>
54     ↳ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
55 [Conditional("DEBUG")]
56 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
57     ↳ ICollection<T> argument, string argumentName) =>
58     ↳ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
59 [Conditional("DEBUG")]
60 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
61     ↳ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
62 [Conditional("DEBUG")]
63 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
64     ↳ root, string argument, string argumentName, string message) =>
65     ↳ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
66 [Conditional("DEBUG")]
67 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
68     ↳ root, string argument, string argumentName) =>
69     ↳ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
70 [Conditional("DEBUG")]
71 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
72     ↳ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
73     ↳ null, null);
74 #endregion
75 }
76 }

```

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 GetOrDefault<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 GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
            ↳ TKey key, Func<TKey, TValue> valueFactory)
20        {
21            if (!dictionary.TryGetValue(key, out TValue value))
22            {
23                value = valueFactory(key);
24                dictionary.Add(key, value);
25                return value;
26            }
27            return value;
28        }
29    }
30 }

```

1.15 ./csharp/Platform.Collections/Lists/CharListExtensions.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.Lists
7 {
8     public static class CharListExtensions
9     {
10        /// <remarks>
11        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
            ↳ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12        /// </remarks>
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public static int GenerateHashCode(this IList<char> list)
15        {
16            var hashSeed = 5381;
17            var hashAccumulator = hashSeed;
18            for (var i = 0; i < list.Count; i++)
19            {
20                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
21            }
22            return hashAccumulator + (hashSeed * 1566083941);
23        }
24    }
25 }

```

```

24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public static bool EqualTo(this IList<char> left, IList<char> right) =>
26         ↪ left.EqualTo(right, ContentEqualTo);
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public static bool ContentEqualTo(this IList<char> left, IList<char> right)
30     {
31         for (var i = left.Count - 1; i >= 0; --i)
32         {
33             if (left[i] != right[i])
34             {
35                 return false;
36             }
37         }
38         return true;
39     }
40 }
41 }

```

1.16 ./csharp/Platform.Collections/Lists/IListComparer.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.Lists
7  {
8      public class IListComparer<T> : IComparer<IList<T>>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
12     }
13 }

```

1.17 ./csharp/Platform.Collections/Lists/IListEqualityComparer.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.Lists
7  {
8      public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public int GetHashCode(IList<T> list) => list.GenerateHashCode();
15     }
16 }

```

1.18 ./csharp/Platform.Collections/Lists/IListExtensions.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.Lists
8  {
9      public static class IListExtensions
10     {
11         /// <summary>
12         /// <para>Gets the element from specified index if the list is not null and the index is
13         ↪ within the list's boundaries, otherwise it returns default value of type T.</para>
14         /// <para>Получает элемент из указанного индекса, если список не является null и индекс
15         ↪ находится в границах списка, в противном случае он возвращает значение по умолчанию
16         ↪ типа T.</para>
17         /// </summary>
18         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
19         ↪ списка.</typeparam>
20         /// <param name="list"><para>The checked list.</para><para>Проверяемый
21         ↪ список.</param></param>
22         /// <param name="index"><para>The index of element.</para><para>Индекс
23         ↪ элемента.</param></param>
24         /// <returns>

```

```

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

```

```

89  /// <para>Значение true в любом случае.</para>
90  /// </returns>
91  [MethodImpl(MethodImplOptions.AggressiveInlining)]
92  public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
93  {
94      list.AddFirst(elements);
95      return true;
96  }
97
98  /// <summary>
99  /// <para>Adds a value to the list at the first index.</para>
100  /// <para>Добавляет значение в список по первому индексу.</para>
101  /// </summary>
102  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
103  → списка.</para></typeparam>
104  /// <param name="list"><para>The list to add the value to.</para><para>Список в который
105  → нужно добавить значение.</para></param>
106  /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
107  → который нужно добавить в список</para></param>
108  [MethodImpl(MethodImplOptions.AggressiveInlining)]
109  public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
110  → list.Add(elements[0]);
111
112  /// <summary>
113  /// <para>Adds all elements from other list to this list and returns true.</para>
114  /// <para>Добавляет все элементы из другого списка в этот список и возвращает
115  → true.</para>
116  /// </summary>
117  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
118  → списка.</para></typeparam>
119  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
120  → нужно добавить значения.</para></param>
121  /// <param name="elements"><para>List of values to add.</para><para>Список значений
122  → которые необходимо добавить.</para></param>
123  /// <returns>
124  /// <para>True value in any case.</para>
125  /// <para>Значение true в любом случае.</para>
126  /// </returns>
127  [MethodImpl(MethodImplOptions.AggressiveInlining)]
128  public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
129  {
130      list.AddAll(elements);
131      return true;
132  }
133
134  /// <summary>
135  /// <para>Adds all elements from other list to this list.</para>
136  /// <para>Добавляет все элементы из другого списка в этот список.</para>
137  /// </summary>
138  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
139  → списка.</para></typeparam>
140  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
141  → нужно добавить значения.</para></param>
142  /// <param name="elements"><para>The list of values to add.</para><para>Список значений
143  → которые необходимо добавить.</para></param>
144  [MethodImpl(MethodImplOptions.AggressiveInlining)]
145  public static void AddAll<T>(this IList<T> list, IList<T> elements)
146  {
147      for (var i = 0; i < elements.Count; i++)
148      {
149          list.Add(elements[i]);
150      }
151  }
152
153  /// <summary>
154  /// <para>Adds values to the list skipping the first element.</para>
155  /// <para>Добавляет значения в список пропуская первый элемент.</para>
156  /// </summary>
157  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
158  → списка.</para></typeparam>
159  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
160  → нужно добавить значения.</para></param>
161  /// <param name="elements"><para>The list of values to add.</para><para>Список значений
162  → которые необходимо добавить.</para></param>
163  /// <returns>
164  /// <para>True value in any case.</para>
165  /// <para>Значение true в любом случае.</para>

```

```

142     /// </returns>
143     [MethodImpl(MethodImplOptions.AggressiveInlining)]
144     public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
145     {
146         list.AddSkipFirst(elements);
147         return true;
148     }
149
150     /// <summary>
151     /// <para>Adds values to the list skipping the first element.</para>
152     /// <para>Добавляет значения в список пропуская первый элемент.</para>
153     /// </summary>
154     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
155     <para>→ списка.</para></typeparam>
156     /// <param name="list"><para>The list to add the values to.</para><para>Список в который
157     <para>→ нужно добавить значения.</para></param>
158     /// <param name="elements"><para>List of values to add.</para><para>Список значений
159     <para>→ которые необходимо добавить.</para></param>
160     [MethodImpl(MethodImplOptions.AggressiveInlining)]
161     public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
162     <para>→ list.AddSkipFirst(elements, 1);
163
164     /// <summary>
165     /// <para>Adds values to the list skipping a specified number of first elements.</para>
166     /// <para>Добавляет в список значения пропуская определенное количество первых
167     <para>→ элементов.</para>
168     /// </summary>
169     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
170     <para>→ списка.</para></typeparam>
171     /// <param name="list"><para>The list to add the values to.</para><para>Список в который
172     <para>→ нужно добавить значения.</para></param>
173     /// <param name="elements"><para>The number of elements to skip.</para><para>Количество
174     <para>→ пропускаемых элементов.</para></param>
175     /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
176     <para>→ пропускаемых элементов.</para></param>
177     [MethodImpl(MethodImplOptions.AggressiveInlining)]
178     public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
179     {
180         for (var i = skip; i < elements.Count; i++)
181         {
182             list.Add(elements[i]);
183         }
184     }
185
186     /// <summary>
187     /// <para>Reads the number of elements in the list.</para>
188     /// <para>Считывает число элементов списка.</para>
189     /// </summary>
190     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
191     <para>→ списка.</para></typeparam>
192     /// <param name="list"><para>The checked list.</para><para>Список для
193     <para>→ проверки.</para></param>
194     /// <returns>
195     /// <para>The number of items contained in the list or 0.</para>
196     /// <para>Число элементов содержащихся в списке или же 0.</para>
197     /// </returns>
198     [MethodImpl(MethodImplOptions.AggressiveInlining)]
199     public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
200
201     /// <summary>
202     /// <para>Compares two lists for equality.</para>
203     /// <para>Сравнивает два списка на равенство.</para>
204     /// </summary>
205     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
206     <para>→ списка.</para></typeparam>
207     /// <param name="left"><para>The first compared list.</para><para>Первый список для
208     <para>→ сравнения.</para></param>
209     /// <param name="right"><para>The second compared list.</para><para>Второй список для
210     <para>→ сравнения.</para></param>
211     /// <returns>
212     /// <para>If the passed lists are equal to each other, true is returned, otherwise
213     <para>→ false.</para>
214     /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
215     <para>→ false.</para>
216     /// </returns>
217     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

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

```

```

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

```



```

328    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↪ списка.</para></typeparam>
329    /// <param name="list"><para>Hash list.</para><para>Список для
    ↪ хеширования.</para></param>
330    /// <returns>
331    /// <para>The hash code of the list.</para>
332    /// <para>Хэш-код списка.</para>
333    /// </returns>
334    /// <remarks>
335    /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
    ↪ -overridden-system-object-gethashcode
336    /// </remarks>
337    [MethodImpl(MethodImplOptions.AggressiveInlining)]
338    public static int GenerateHashCode<T>(this IList<T> list)
339    {
340        var hashAccumulator = 17;
341        for (var i = 0; i < list.Count; i++)
342        {
343            hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
344        }
345        return hashAccumulator;
346    }
347
348    [MethodImpl(MethodImplOptions.AggressiveInlining)]
349    public static int CompareTo<T>(this IList<T> left, IList<T> right)
350    {
351        var comparer = Comparer<T>.Default;
352        var leftCount = left.GetCountOrZero();
353        var rightCount = right.GetCountOrZero();
354        var intermediateResult = leftCount.CompareTo(rightCount);
355        for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
356        {
357            intermediateResult = comparer.Compare(left[i], right[i]);
358        }
359        return intermediateResult;
360    }
361
362    [MethodImpl(MethodImplOptions.AggressiveInlining)]
363    public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
364
365    /// <summary>
366    /// <para>Skips the specified number of elements in the list and builds an array from
    ↪ the remaining elements.</para>
367    /// <para>Пропускает указанное количество элементов списка и составляет из оставшихся
    ↪ элементов массив.</para>
368    /// </summary>
369    /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↪ списка.</para></typeparam>
370    /// <param name="list"><para>The list to copy from.</para><para>Список для
    ↪ копирования.</para></param>
371    /// <param name="skip"><para>The number of items to skip.</para><para>Количество
    ↪ пропускаемых элементов.</para></param>
372    /// <returns>
373    /// <para>If the list is empty, or the number of skipped elements is greater than the
    ↪ list, it returns an empty array, otherwise - an array with the specified number of
    ↪ missing elements.</para>
374    /// <para>Если список пуст, или количество пропускаемых элементов больше списка -
    ↪ возвращает пустой массив, иначе - массив с указанным количеством пропущенных
    ↪ элементов.</para>
375    /// </returns>
376    [MethodImpl(MethodImplOptions.AggressiveInlining)]
377    public static T[] SkipFirst<T>(this IList<T> list, int skip)
378    {
379        if (list.IsNullOrEmpty() || list.Count <= skip)
380        {
381            return Array.Empty<T>();
382        }
383        var result = new T[list.Count - skip];
384        for (int r = skip, w = 0; r < list.Count; r++, w++)
385        {
386            result[w] = list[r];
387        }
388        return result;
389    }
390
391    [MethodImpl(MethodImplOptions.AggressiveInlining)]
392    public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
393

```

```

394 /// <summary>
395 /// <para>Shifts all elements of the list to the right by the specified number of
    → elements and returns an array.</para>
396 /// <para>Сдвигает вправо все элементы списка на указанное количество элементов и
    → возвращает массив.</para>
397 /// </summary>
398 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
399 /// <param name="list"><para>The list to copy from.</para><para>Список для
    → копирования.</para></param>
400 /// <param name="skip"><para>The number of items to shift.</para><para>Количество
    → сдвигаемых элементов.</para></param>
401 /// <returns>
402 /// <para>If the variable shift is less than zero - an <see
    → cref="NotImplementedException"/> exception is thrown, but if the variable shift is 0
    → an exact copy of the array is returned. Otherwise, an array is returned with the
    → shift of the elements.</para>
403 /// <para>Если переменная shift меньше нуля - выбрасывается исключение <see
    → cref="NotImplementedException"/>, если же переменная shift равна 0 возвращается
    → точная копия массива. Иначе возвращается массив со сдвигом элементов.</para>
404 /// </returns>
405 [MethodImpl(MethodImplOptions.AggressiveInlining)]
406 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
407 {
408     if (shift < 0)
409     {
410         throw new NotImplementedException();
411     }
412     if (shift == 0)
413     {
414         return list.ToArray();
415     }
416     else
417     {
418         var result = new T[list.Count + shift];
419         for (int r = 0, w = shift; r < list.Count; r++, w++)
420         {
421             result[w] = list[r];
422         }
423         return result;
424     }
425 }
426 }
427 }

```

1.19 ./csharp/Platform.Collections/Lists/ListFiller.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.Lists
7 {
8     public class ListFiller<TElement, TReturnConstant>
9     {
10         protected readonly List<TElement> _list;
11         protected readonly TReturnConstant _returnConstant;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
15         {
16             _list = list;
17             _returnConstant = returnConstant;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ListFiller(List<TElement> list) : this(list, default) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _list.Add(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
            → _list.AddFirstAndReturnTrue(elements);
31

```

```

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

```

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

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class CharSegment : Segment<char>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
            → length) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public override int GetHashCode()
18         {
19             // Base can be not an array, but still IList<char>
20             if (Base is char[] baseArray)
21             {
22                 return baseArray.GenerateHashCode(Offset, Length);
23             }
24             else
25             {
26                 return this.GenerateHashCode();
27             }
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public override bool Equals(Segment<char> other)
32         {
33             bool contentEqualityComparer(IList<char> left, IList<char> right)
34             {
35                 // Base can be not an array, but still IList<char>
36                 if (Base is char[] baseArray && other.Base is char[] otherArray)
37                 {
38                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
39                 }

```

```

40         else
41         {
42             return left.ContentEqualTo(right);
43         }
44     }
45     return this.EqualTo(other, contentEqualityComparer);
46 }
47
48 public override bool Equals(object obj) => obj is Segment<char> charSegment ?
    ↳ Equals(charSegment) : false;
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public static implicit operator string(CharSegment segment)
52 {
53     if (!(segment.Base is char[] array))
54     {
55         array = segment.Base.ToArray();
56     }
57     return new string(array, segment.Offset, segment.Length);
58 }
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public override string ToString() => this;
62 }
63 }

```

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

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Segments
11 {
12     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
13     {
14         public IList<T> Base
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19         public int Offset
20         {
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             get;
23         }
24         public int Length
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public Segment(IList<T> @base, int offset, int length)
32         {
33             Base = @base;
34             Offset = offset;
35             Length = length;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public override int GetHashCode() => this.GenerateHashCode();
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
            ↳ false;
46
47         #region IList
48
49         public T this[int i]
50         {
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             get => Base[Offset + i];
53         }
54     }
55 }

```

```

53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     set => Base[Offset + i] = value;
55 }
56
57 public int Count
58 {
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     get => Length;
61 }
62
63 public bool IsReadOnly
64 {
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     get => true;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public int IndexOf(T item)
71 {
72     var index = Base.IndexOf(item);
73     if (index >= Offset)
74     {
75         var actualIndex = index - Offset;
76         if (actualIndex < Length)
77         {
78             return actualIndex;
79         }
80     }
81     return -1;
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public void Insert(int index, T item) => throw new NotSupportedException();
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 public void RemoveAt(int index) => throw new NotSupportedException();
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 public void Add(T item) => throw new NotSupportedException();
92
93 [MethodImpl(MethodImplOptions.AggressiveInlining)]
94 public void Clear() => throw new NotSupportedException();
95
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 public bool Contains(T item) => IndexOf(item) >= 0;
98
99 [MethodImpl(MethodImplOptions.AggressiveInlining)]
100 public void CopyTo(T[] array, int arrayIndex)
101 {
102     for (var i = 0; i < Length; i++)
103     {
104         array.Add(ref arrayIndex, this[i]);
105     }
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public bool Remove(T item) => throw new NotSupportedException();
110
111 [MethodImpl(MethodImplOptions.AggressiveInlining)]
112 public IEnumerator<T> GetEnumerator()
113 {
114     for (var i = 0; i < Length; i++)
115     {
116         yield return this[i];
117     }
118 }
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
122
123 #endregion
124 }
125 }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      public abstract class AllSegmentsWalkerBase

```

```

6     {
7         public static readonly int DefaultMinimumStringSegmentLength = 2;
8     }
9 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
9         where TSegment : Segment<T>
10     {
11         private readonly int _minimumStringSegmentLength;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
15             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
19
20         [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);
35
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             protected abstract void Iteration(TSegment segment);
38         }
39     }
40 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected override Segment<T> CreateSegment(ICollection<T> elements, int offset, int length)
12             ↪ => new Segment<T>(elements, offset, length);
13     }
14 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Segments.Walkers
6 {
7     public static class AllSegmentsWalkerExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
11             ↪ walker.WalkAll(@string.ToCharArray());
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
15             ↪ string @string) where TSegment : Segment<char> =>
16             ↪ walker.WalkAll(@string.ToCharArray());
17     }
18 }

```

```

14     }
15 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Segments.Walkers
8  {
9      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
39             ↳ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
40             ↳ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
41             ↳ { }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
45             ↳ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
49             ↳ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public override void WalkAll(ICollection<T> elements)
53         {
54             if (_resetDictionaryOnEachWalk)
55             {
56                 var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
57                 Dictionary = new Dictionary<TSegment, long>((int)capacity);
58             }
59             base.WalkAll(elements);
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override long GetSegmentFrequency(TSegment segment) =>
64             ↳ Dictionary.GetOrDefault(segment);
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
68             ↳ Dictionary[segment] = frequency;
69     }
70 }

```

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

```

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

```

```

4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
9          DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
13             dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
14             base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
18             dictionary, int minimumStringSegmentLength) : base(dictionary,
19             minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
23             dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
24             DefaultResetDictionaryOnEachWalk) { }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
28             bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
29             resetDictionaryOnEachWalk) { }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
33             base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
37             base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
38     }
39 }

```

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

```

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

```


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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
        ↳ 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(ICollection<TElement> elements) =>
            ↪ _set.AddFirstAndReturnTrue(elements);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
            ↪ _set.AddAllAndReturnTrue(elements);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
            ↪ _set.AddSkipFirstAndReturnTrue(elements);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public TReturnConstant AddAndReturnConstant(TElement element)
40         {
41             _set.Add(element);
42             return _returnConstant;
43         }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
47         {
48             _set.AddFirst(elements);
49             return _returnConstant;
50         }
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
54         {
55             _set.AddAll(elements);
56             return _returnConstant;
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements)
61         {
62             _set.AddSkipFirst(elements);
63             return _returnConstant;
64         }
65     }
66 }

```

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

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

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

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public interface IStack<TElement>
8     {
9         bool IsEmpty
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         void Push(TElement element);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         TElement Pop();
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         TElement Peek();
23     }
24 }
```

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

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public static class IStackExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static void Clear<T>(this IStack<T> stack)
11         {
12             while (!stack.IsEmpty)
13             {
14                 _ = stack.Pop();
15             }
16         }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
20             ↪ stack.Pop();
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
24             ↪ stack.Peek();
25     }
26 }
```

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

```
1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
```

```

7     public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
8     {
9     }
10 }

```

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

```

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

```

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

```

1 using System;
2 using System.Globalization;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections
8 {
9     public static class StringExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static string CapitalizeFirstLetter(this string @string)
13        {
14            if (string.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            }
55        }
56    }
57 }

```

```

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         }
57     }
58 }

```

```

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));

```



```

159         Assert.False(w.Equals(v));
160         Assert.True(x.Equals(w));
161         Assert.True(y.Equals(v));
162         test(x, y, w, v);
163         Assert.True(x.Equals(w));
164     }
165 }
166 }

```

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

```

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

```

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

```

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, 46
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 46
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 49
- ./csharp/Platform.Collections.Tests/ListTests.cs, 49
- ./csharp/Platform.Collections.Tests/StringTests.cs, 49
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayPool.cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayString.cs, 3
- ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs, 3
- ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 4
- ./csharp/Platform.Collections/BitString.cs, 9
- ./csharp/Platform.Collections/BitStringExtensions.cs, 23
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 24
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 24
- ./csharp/Platform.Collections/EnsureExtensions.cs, 24
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 25
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 26
- ./csharp/Platform.Collections/Lists/CharIListExtensions.cs, 26
- ./csharp/Platform.Collections/Lists/IListComparer.cs, 27
- ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 27
- ./csharp/Platform.Collections/Lists/IListExtensions.cs, 27
- ./csharp/Platform.Collections/Lists/ListFiller.cs, 34
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 35
- ./csharp/Platform.Collections/Segments/Segment.cs, 36
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 37
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 38
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 38
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 38
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 39
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 39
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 40
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 40
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 41
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 42
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 42
- ./csharp/Platform.Collections/Stacks/IStack.cs, 43
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 43
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 43
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 44
- ./csharp/Platform.Collections/StringExtensions.cs, 44
- ./csharp/Platform.Collections/Trees/Node.cs, 45