

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.</para></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.</para></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,
521     ↪ int maximum)
522     {
523         var i = start;
524         var range = maximum - start - 1;
525         var stop = range - (range % step);
526         for (; i < stop; i += step)
527         {
528             (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
529         }
530         for (; i < maximum; i++)
531         {
532             array[i] ^= otherArray[i];
533         }
534     }
535
536     [MethodImpl(MethodImplOptions.AggressiveInlining)]
537     private void RefreshBordersByWord(long wordIndex)
538     {
539         if (_array[wordIndex] == 0)
540         {
541             if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
542             {
543                 _minPositiveWord++;
544             }
545             if (wordIndex == _maxPositiveWord && wordIndex != 0)
546             {
547                 _maxPositiveWord--;
548             }
549         }
550         else
551         {
552             if (wordIndex < _minPositiveWord)
553             {
554                 _minPositiveWord = wordIndex;
555             }
556             if (wordIndex > _maxPositiveWord)
557             {
558                 _maxPositiveWord = wordIndex;
559             }
560         }
561     }
562
563     [MethodImpl(MethodImplOptions.AggressiveInlining)]
564     public bool TryShrinkBorders()
565     {
566         GetBorders(out long from, out long to);
567         while (from <= to && _array[from] == 0)
568         {
569             from++;
570         }
571         if (from > to)
572         {
573             MarkBordersAsAllBitsReset();
574             return true;
575         }
576         while (to >= from && _array[to] == 0)
577         {
578             to--;
579         }
580         if (to < from)
581         {
582             MarkBordersAsAllBitsReset();
583             return true;
584         }
585         var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
586         if (bordersUpdated)
587         {
588             SetBorders(from, to);
589         }
590         return bordersUpdated;
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
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public static bool EqualTo(this IList<char> left, IList<char> right) =>
        ↪ 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>Checks if a list is passed and checks its length.</para>
13         /// <para>Проверяет, передан ли список и сверяет его длину.</para>
14         /// </summary>
15         /// <param name="list"><para>Checklist.</para><para>Проверяемый список.</para></param>
16         /// <param name="index"><para>Index to check.</para><para>Индекс для
        ↪ проверки.</para></param>
17         /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
        ↪ списка.</typeparam>
18         /// <returns>
19         /// <para>If the condition is true - list[index], otherwise default.</para>
20         /// <para>Если значение верно - 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 &&
24     ↳ list.Count > index ? list[index] : default;
25
26 /// <summary>
27 /// <para>Checks if a list is passed, checks its length, and if successful, copies the
28     ↳ value of list [index] into the variable element. Otherwise, the element variable has
29     ↳ a default value.</para>
30 /// <para>Проверяет, передан ли список, сверяет его длину и в случае успеха копирует
31     ↳ значение list[index] в переменную element. Иначе переменная element имеет значение
32     ↳ по умолчанию.</para>
33 /// </summary>
34 /// <param name="list"><para>Checklist.</para><para>Список для проверки.</para></param>
35 /// <param name="index"><para>Index to check</para><para>Индекс для
36     ↳ проверки.</para></param>
37 /// <param name="element"><para></para><para>Переменная для передачи значения
38     ↳ индекса.</para></param>
39 /// <typeparam name="T"><para>List item types</para><para>Тип элементов
40     ↳ списка.</para></typeparam>
41 /// <returns>
42 /// <para>True on success, false otherwise.</para>
43 /// <para>True в случае успеха, иначе false.</para>
44 /// </returns>
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
47 {
48     if (list != null && list.Count > index)
49     {
50         element = list[index];
51         return true;
52     }
53     else
54     {
55         element = default;
56         return false;
57     }
58 }
59
60 /// <summary>
61 /// <para>Adds a value to the list.</para>
62 /// <para>Добавляет значение в список.</para>
63 /// </summary>
64 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
65     ↳ нужно добавить значение.</para></param>
66 /// <param name="element"><para>The item to add to the list.</para><para>Элемент который
67     ↳ нужно добавить в список</para></param>
68 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
69     ↳ списка.</para></typeparam>
70 /// <returns>
71 /// <para>Returns true anyway.</para>
72 /// <para>В любом случае возвращает true.</para>
73 /// </returns>
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
76 {
77     list.Add(element);
78     return true;
79 }
80
81 /// <summary>
82 /// <para>Adds a value to the list at the first index.</para>
83 /// <para>Добавляет значение в список по первому индексу.</para>
84 /// </summary>
85 /// <param name="list"><para>The list to add the value to.</para><para>Список в который
86     ↳ нужно добавить значение.</para></param>
87 /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
88     ↳ который нужно добавить в список</para></param>
89 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
90     ↳ списка.</para></typeparam>
91 /// <returns>
92 /// <para>Returns true anyway.</para>
93 /// <para>В любом случае возвращает true.</para>
94 /// </returns>
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
97 {
98     list.AddFirst(elements);
99     return true;
100 }

```

```

86     }
87
88     /// <summary>
89     /// <para>Adds a value to the list at the first index.</para>
90     /// <para>Добавляет значение в список по первому индексу.</para>
91     /// </summary>
92     /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
93     /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
    → который нужно добавить в список</para></param>
94     /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    → списка.</para></typeparam>
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
    → list.Add(elements[0]);
97
98     /// <summary>
99     /// <para>Adds a list of values to the variable list.</para>
100    /// <para>Добавляет список значений в переменную list.</para>
101    /// </summary>
102    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
103    /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
104    /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    → списка.</para></typeparam>
105    /// <returns>
106    /// <para>Returns true anyway.</para>
107    /// <para>В любом случае возвращает true.</para>
108    /// </returns>
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
111    {
112        list.AddAll(elements);
113        return true;
114    }
115
116    /// <summary>
117    /// <para>Adds a list of values to the variable list.</para>
118    /// <para>Добавляет список значений в переменную list.</para>
119    /// </summary>
120    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
121    /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
122    /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    → списка.</para></typeparam>
123    [MethodImpl(MethodImplOptions.AggressiveInlining)]
124    public static void AddAll<T>(this IList<T> list, IList<T> elements)
125    {
126        for (var i = 0; i < elements.Count; i++)
127        {
128            list.Add(elements[i]);
129        }
130    }
131
132    /// <summary>
133    /// <para>Adds a list of values, skipping the first index.</para>
134    /// <para>Добавляет список значений пропуская первый индекс.</para>
135    /// </summary>
136    /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
137    /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
138    /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    → списка.</para></typeparam>
139    /// <returns>
140    /// <para>Returns true anyway.</para>
141    /// <para>В любом случае возвращает true.</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 a list of values, skipping the first index.</para>
152 /// <para>Добавляет список значений пропуская первый индекс.</para>
153 /// </summary>
154 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    ↳ нужно добавить значения.</para></param>
155 /// <param name="elements"><para>List of values to add.</para><para>Список значений
    ↳ которые необходимо добавить.</para></param>
156 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
    ↳ list.AddSkipFirst(elements, 1);
159
160 /// <summary>
161 /// <para>Adds a list of values skipping a specific index.</para>
162 /// <para>Добавляет список значений пропуская определенный индекс</para>
163 /// </summary>
164 /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    ↳ нужно добавить значения.</para></param>
165 /// <param name="elements"><para>List of values to add.</para><para>Список значений
    ↳ которые необходимо добавить.</para></param>
166 /// <param name="skip"><para>Number of indexes to skip.</para><para>Количество
    ↳ пропускаемых индексов.</para></param>
167 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
170 {
171     for (var i = skip; i < elements.Count; i++)
172     {
173         list.Add(elements[i]);
174     }
175 }
176
177 /// <summary>
178 /// <para>Reads the number of elements in the list.</para>
179 /// <para>Считывает число элементов списка.</para>
180 /// </summary>
181 /// <param name="list"><para>Checklist.</para><para>Список для проверки.</para></param>
182 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
183 /// <returns>
184 /// <para>The number of items contained in the list, or 0.</para>
185 /// <para>Число элементов содержащихся в списке, или же 0.</para>
186 /// </returns>
187 [MethodImpl(MethodImplOptions.AggressiveInlining)]
188 public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
189
190 /// <summary>
191 /// <para>Compares two lists for their identity.</para>
192 /// <para>Сравниваются два списка на идентичность.</para>
193 /// <para></para>
194 /// </summary>
195 /// <param name="left"><para>Checklist.</para><para>Список для проверки.</para></param>
196 /// <param name="right"><para>Checklist.</para><para>Список для проверки.</para></param>
197 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    ↳ списка.</para></typeparam>
198 /// <returns>
199 /// <para>If the passed lists are identical to each other, true is returned, Otherwise
    ↳ false.</para>
200 /// <para>Если передаваемые списки идентичны друг другу, возвращается true, иначе же
    ↳ false.</para>
201 /// </returns>
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
    ↳ right, ContentEqualTo);
204
205 /// <summary>
206 /// <para>Compares two lists for their identity.</para>
207 /// <para>Сравниваются два списка на идентичность.</para>
208 /// </summary>
209 /// <param name="left"><para>Checklist.</para><para>Список для проверки.</para></param>
210 /// <param name="right"><para>Checklist.</para><para>Список для проверки.</para></param>
211 /// <param name="contentEqualityComparer"></param>
212 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    ↳ списка.</para></typeparam>

```

```

213 /// <returns>
214 /// <para>If the passed lists are identical to each other, true is returned, otherwise
    → false.</para>
215 /// <para>Если передаваемые списки идентичны друг другу, возвращается true, иначе же
    → false.</para>
216 /// </returns>
217 [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
    → IList<T>, bool> contentEqualityComparer)
219 {
220     if (ReferenceEquals(left, right))
221     {
222         return true;
223     }
224     var leftCount = left.GetCountOrZero();
225     var rightCount = right.GetCountOrZero();
226     if (leftCount == 0 && rightCount == 0)
227     {
228         return true;
229     }
230     if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
231     {
232         return false;
233     }
234     return contentEqualityComparer(left, right);
235 }
236
237 /// <summary>
238 /// <para>Compares each element in the list for identity.</para>
239 /// <para>Сравнивает на идентичность каждый элемент списка.</para>
240 /// </summary>
241 /// <param name="left"><para>Checklist.</para><para>Список для проверки.</para></param>
242 /// <param name="right"><para>Checklist.</para><para>Список для проверки.</para></param>
243 /// <typeparam name="T"><para>List item types.</para><para>Тип элементов
    → списка.</para></typeparam>
244 /// <returns>
245 /// <para>If every element of one list is not equal to every element of another list -
    → return false, otherwise - true.</para>
246 /// <para>Если каждый элемент одного списка не равен каждому элемента другого списка -
    → return false, иначе - true. </para>
247 /// </returns>
248 [MethodImpl(MethodImplOptions.AggressiveInlining)]
249 public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
250 {
251     var equalityComparer = EqualityComparer<T>.Default;
252     for (var i = left.Count - 1; i >= 0; --i)
253     {
254         if (!equalityComparer.Equals(left[i], right[i]))
255         {
256             return false;
257         }
258     }
259     return true;
260 }
261
262 [MethodImpl(MethodImplOptions.AggressiveInlining)]
263 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
264 {
265     if (list == null)
266     {
267         return null;
268     }
269     var result = new List<T>(list.Count);
270     for (var i = 0; i < list.Count; i++)
271     {
272         if (predicate(list[i]))
273         {
274             result.Add(list[i]);
275         }
276     }
277     return result.ToArray();
278 }
279
280 [MethodImpl(MethodImplOptions.AggressiveInlining)]
281 public static T[] ToArray<T>(this IList<T> list)
282 {
283     var array = new T[list.Count];
284     list.CopyTo(array, 0);

```

```

285     return array;
286 }
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]
289 public static void ForEach<T>(this IList<T> list, Action<T> action)
290 {
291     for (var i = 0; i < list.Count; i++)
292     {
293         action(list[i]);
294     }
295 }
296
297 /// <remarks>
298 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
299 /// ↳ -overridden-system-object-gethashcode
300 /// </remarks>
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 public static int GenerateHashCode<T>(this IList<T> list)
303 {
304     var hashAccumulator = 17;
305     for (var i = 0; i < list.Count; i++)
306     {
307         hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
308     }
309     return hashAccumulator;
310 }
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 public static int CompareTo<T>(this IList<T> left, IList<T> right)
314 {
315     var comparer = Comparer<T>.Default;
316     var leftCount = left.GetCountOrZero();
317     var rightCount = right.GetCountOrZero();
318     var intermediateResult = leftCount.CompareTo(rightCount);
319     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
320     {
321         intermediateResult = comparer.Compare(left[i], right[i]);
322     }
323     return intermediateResult;
324 }
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 public static T[] SkipFirst<T>(this IList<T> list, int skip)
331 {
332     if (list.IsNullOrEmpty() || list.Count <= skip)
333     {
334         return Array.Empty<T>();
335     }
336     var result = new T[list.Count - skip];
337     for (int r = skip, w = 0; r < list.Count; r++, w++)
338     {
339         result[w] = list[r];
340     }
341     return result;
342 }
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
346
347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
348 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
349 {
350     if (shift < 0)
351     {
352         throw new NotImplementedException();
353     }
354     if (shift == 0)
355     {
356         return list.ToArray();
357     }
358     else
359     {
360         var result = new T[list.Count + shift];
361         for (int r = 0, w = shift; r < list.Count; r++, w++)
362         {
363             result[w] = list[r];
364         }
365     }
366 }

```



```

363     }
364     return result;
365 }
366 }
367 }
368 }

```

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(ICollection<TElement> elements) =>
31             => _list.AddFirstAndReturnTrue(elements);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
35             => _list.AddAllAndReturnTrue(elements);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public bool AddSkipFirstAndReturnTrue(ICollection<TElement> elements) =>
39             => _list.AddSkipFirstAndReturnTrue(elements);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public TReturnConstant AddAndReturnConstant(TElement element)
43         {
44             _list.Add(element);
45             return _returnConstant;
46         }
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements)
50         {
51             _list.AddFirst(elements);
52             return _returnConstant;
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements)
57         {
58             _list.AddAll(elements);
59             return _returnConstant;
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements)
64         {
65             _list.AddSkipFirst(elements);
66             return _returnConstant;
67         }
68     }
69 }

```

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

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class CharSegment : Segment<char>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
15             ↪ length) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override int GetHashCode()
19         {
20             // Base can be not an array, but still IList<char>
21             if (Base is char[] baseArray)
22             {
23                 return baseArray.GenerateHashCode(Offset, Length);
24             }
25             else
26             {
27                 return this.GenerateHashCode();
28             }
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override bool Equals(Segment<char> other)
33         {
34             bool contentEqualityComparer(IList<char> left, IList<char> right)
35             {
36                 // Base can be not an array, but still IList<char>
37                 if (Base is char[] baseArray && other.Base is char[] otherArray)
38                 {
39                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
40                 }
41                 else
42                 {
43                     return left.ContentEqualTo(right);
44                 }
45             }
46             return this.EqualTo(other, contentEqualityComparer);
47         }
48
49         public override bool Equals(object obj) => obj is Segment<char> charSegment ?
50             ↪ Equals(charSegment) : false;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public static implicit operator string(CharSegment segment)
54         {
55             if (!(segment.Base is char[] array))
56             {
57                 array = segment.Base.ToArray();
58             }
59             return new string(array, segment.Offset, segment.Length);
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public override string ToString() => this;
64     }
65 }

```

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

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Segments
11 {
12     public class Segment<T> : IEquatable<Segment<T>>, IList<T>

```

```

13 {
14     public IList<T> Base
15     {
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         get;
18     }
19     public int Offset
20     {
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         get;
23     }
24     public int Length
25     {
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         get;
28     }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public Segment(IList<T> @base, int offset, int length)
32     {
33         Base = @base;
34         Offset = offset;
35         Length = length;
36     }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public override int GetHashCode() => this.GenerateHashCode();
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
46         ↪ false;
47
48     #region IList
49     public T this[int i]
50     {
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         get => Base[Offset + i];
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         set => Base[Offset + i] = value;
55     }
56
57     public int Count
58     {
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         get => Length;
61     }
62
63     public bool IsReadOnly
64     {
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         get => true;
67     }
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     public int IndexOf(T item)
71     {
72         var index = Base.IndexOf(item);
73         if (index >= Offset)
74         {
75             var actualIndex = index - Offset;
76             if (actualIndex < Length)
77             {
78                 return actualIndex;
79             }
80         }
81         return -1;
82     }
83
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     public void Insert(int index, T item) => throw new NotSupportedException();
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public void RemoveAt(int index) => throw new NotSupportedException();
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public void Add(T item) => throw new NotSupportedException();

```

```

92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public void Clear() => throw new NotSupportedException();
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public bool Contains(T item) => IndexOf(item) >= 0;
97
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public void CopyTo(T[] array, int arrayIndex)
100     {
101         for (var i = 0; i < Length; i++)
102         {
103             array.Add(ref arrayIndex, this[i]);
104         }
105     }
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     public bool Remove(T item) => throw new NotSupportedException();
109
110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
111     public IEnumerator<T> GetEnumerator()
112     {
113         for (var i = 0; i < Length; i++)
114         {
115             yield return this[i];
116         }
117     }
118
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
121
122     #endregion
123 }
124
125 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
9          where TSegment : Segment<T>
10     {
11         private readonly int _minimumStringSegmentLength;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
15             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public virtual void WalkAll(ICollection<T> elements)
22         {
23             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
24                 ↪ offset <= maxOffset; offset++)
25             {
26                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
27                     ↪ offset; length <= maxLength; length++)
28                 {
29                     Iteration(CreateSegment(elements, offset, length));
30                 }
31             }
32         }
33     }
34 }

```

```

30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected abstract void Iteration(TSegment segment);
36 }
37 }

```

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

```

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

```

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

```

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

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Segments.Walkers
8 {
9     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
10        DuplicateSegmentsWalkerBase<T, TSegment>
11        where TSegment : Segment<T>
12     {
13         public static readonly bool DefaultResetDictionaryOnEachWalk;
14
15         private readonly bool _resetDictionaryOnEachWalk;
16         protected IDictionary<TSegment, long> Dictionary;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
20             dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
21             : base(minimumStringSegmentLength)
22         {
23             Dictionary = dictionary;
24             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
29             dictionary, int minimumStringSegmentLength) : this(dictionary,
30             minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     }
34 }

```

```

29     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
        ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
        ↪ DefaultResetDictionaryOnEachWalk) { }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
        ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
        ↪ { }
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
        ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
        ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public override void WalkAll(ICollection<T> elements)
42     {
43         if (_resetDictionaryOnEachWalk)
44         {
45             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
46             Dictionary = new Dictionary<TSegment, long>((int)capacity);
47         }
48         base.WalkAll(elements);
49     }
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override long GetSegmentFrequency(TSegment segment) =>
        ↪ Dictionary.GetOrDefault(segment);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
        ↪ Dictionary[segment] = frequency;
56 }
57 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
        ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
            ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
            ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
            ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
            ↪ DefaultResetDictionaryOnEachWalk) { }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
            ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
            ↪ resetDictionaryOnEachWalk) { }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
            ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27     }

```

```
28 }
```

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

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

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

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

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Sets
7 {
8     public static class ISetExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
15             ↪ set.Remove(element);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
19         {
20             set.Add(element);
21             return true;
22         }
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
```

```

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

```

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

```

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

```



```

29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
        => _set.AddFirstAndReturnTrue(elements);
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     public bool AddAllAndReturnTrue(IList<TElement> elements) =>
        => _set.AddAllAndReturnTrue(elements);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     public bool AddSkipFirstAndReturnTrue(IList<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(IList<TElement> elements)
47     {
48         _set.AddFirst(elements);
49         return _returnConstant;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
54     {
55         _set.AddAll(elements);
56         return _returnConstant;
57     }
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     public TReturnConstant AddSkipFirstAndReturnConstant(IList<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() :
12            ↪ default;
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
16            ↪ : default;
17    }
18 }

```

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

```

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

```

```

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

```

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

```

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

```

```

12
13 public object Value
14 {
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     get;
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     set;
19 }
20
21 public Dictionary<object, Node> ChildNodes
22 {
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
25 }
26
27 public Node this[object key]
28 {
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     get => GetChild(key) ?? AddChild(key);
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     set => SetChildValue(value, key);
33 }
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public Node(object value) => Value = value;
37
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 public Node() : this(null) { }
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 public Node GetChild(params object[] keys)
46 {
47     var node = this;
48     for (var i = 0; i < keys.Length; i++)
49     {
50         node.ChildNodes.TryGetValue(keys[i], out node);
51         if (node == null)
52         {
53             return null;
54         }
55     }
56     return node;
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 public Node AddChild(object key) => AddChild(key, new Node(null));
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public Node AddChild(object key, object value) => AddChild(key, new Node(value));
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public Node AddChild(object key, Node child)
70 {
71     ChildNodes.Add(key, child);
72     return child;
73 }
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public Node SetChild(object key) => SetChildValue(null, key);
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public Node SetChildValue(object value, params object[] keys)
83 {
84     var node = this;
85     for (var i = 0; i < keys.Length; i++)
86     {
87         node = SetChildValue(value, keys[i]);
88     }
89     node.Value = value;
90     return node;
91 }

```

```

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

```

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

```

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

```

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

```

1  using System;
2  using System.Collections;
3  using Xunit;
4  using Platform.Random;
5
6  namespace Platform.Collections.Tests
7  {
8      public static class BitStringTests
9      {
10         [Fact]
11         public static void BitGetSetTest()
12         {
13             const int n = 250;
14             var bitArray = new BitArray(n);
15             var bitString = new BitString(n);
16             for (var i = 0; i < n; i++)
17             {
18                 var value = RandomHelpers.Default.NextBoolean();
19                 bitArray.Set(i, value);
20                 bitString.Set(i, value);
21                 Assert.Equal(value, bitArray.Get(i));
22                 Assert.Equal(value, bitString.Get(i));
23             }
24         }
25
26         [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,
    ↪ BitString, BitString> test)
147 {
148     const int n = 5654;
149     var x = new BitString(n);
150     var y = new BitString(n);
151     while (x.Equals(y))
152     {
153         x.SetRandomBits();
154         y.SetRandomBits();
155     }
156     var w = new BitString(x);
157     var v = new BitString(y);
158     Assert.False(x.Equals(y));
159     Assert.False(w.Equals(v));
160     Assert.True(x.Equals(w));
161     Assert.True(y.Equals(v));
162     test(x, y, w, v);
163     Assert.True(x.Equals(w));
164 }
165 }
166 }

```

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

```

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

```

```
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, 45
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 45
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 47
- ./csharp/Platform.Collections.Tests/ListTests.cs, 48
- ./csharp/Platform.Collections.Tests/StringTests.cs, 48
- ./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, 33
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 33
- ./csharp/Platform.Collections/Segments/Segment.cs, 34
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 36
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 36
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 37
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 37
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 37
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 38
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 39
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 39
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 39
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 40
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 41
- ./csharp/Platform.Collections/Stacks/IStack.cs, 41
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 42
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 42
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 42
- ./csharp/Platform.Collections/StringExtensions.cs, 42
- ./csharp/Platform.Collections/Trees/Node.cs, 43