LinksPlatform's Platform.Collections Class Library

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

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

```csharp
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

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

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

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

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

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

```
1   using System.Runtime.CompilerServices;
2
3   #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          /// </remarks>
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          public static int GenerateHashCode(this char[] array, int offset, int length)
14          {
15              var hashSeed = 5381;
16              var hashAccumulator = hashSeed;
17              fixed (char* arrayPointer = &array[offset])
18              {
19                  for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
     ↪    < last; charPointer++)
20                  {
21                      hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
22                  }
23              }
24              return hashAccumulator + (hashSeed * 1566083941);
25          }
26
27          /// <remarks>
28          /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783⌋
     ↪    a3eda37d3d4cd10/mscorlib/system/string.cs#L364
29          /// </remarks>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
     ↪    right, int rightOffset)
```

```csharp
            {
                fixed (char* leftPointer = &left[leftOffset])
                {
                    fixed (char* rightPointer = &right[rightOffset])
                    {
                        char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
                        if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
                        ↪  rightPointerCopy, ref length))
                        {
                            return false;
                        }
                        CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
                        ↪  ref length);
                        return length <= 0;
                    }
                }
            }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
        ↪  int length)
        {
            while (length >= 10)
            {
                if ((*(int*)left != *(int*)right)
                 || (*(int*)(left + 2) != *(int*)(right + 2))
                 || (*(int*)(left + 4) != *(int*)(right + 4))
                 || (*(int*)(left + 6) != *(int*)(right + 6))
                 || (*(int*)(left + 8) != *(int*)(right + 8)))
                {
                    return false;
                }
                left += 10;
                right += 10;
                length -= 10;
            }
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
        ↪  int length)
        {
            // This depends on the fact that the String objects are
            // always zero terminated and that the terminating zero is not included
            // in the length. For odd string sizes, the last compare will include
            // the zero terminator.
            while (length > 0)
            {
                if (*(int*)left != *(int*)right)
                {
                    break;
                }
                left += 2;
                right += 2;
                length -= 2;
            }
        }
    }
}
```

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

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

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

namespace Platform.Collections.Arrays
{
    public static class GenericArrayExtensions
    {
        /// <summary>
        /// <para>Checks if an array exists, if so,  checks the array length using the  index
        ↪  variable type int, and if the array length is greater than the index - return
        ↪  array[index], otherwise - default value.</para>
        /// <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>
```

```csharp
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 учавствовать 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 учавствовать 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
   ↪   varible 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 учавствовать 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>Передает
   ↪   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 varible type long, and if the array length is greater than the index,  set the
   ↪   element variable to array[index] and return true.</para>
```

```csharp
        /// <para>Проверяет, существует ли массив, если да, то идет проверка длины массива с
        ///     помощью переменной index  типа long, и если длина массива больше значения index,
        ///     устанавливает значение переменной element - array[index] и возвращает true.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///     массива.</para></typeparam>
        /// <param name="array"><para>Array that will participate in
        ///     verification.</para><para>Массив который будет учавствовать в
        ///     проверке.</para></param>
        /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
        ///     для сравнения.</para></param>
        /// <param name="element"><para>Passing the argument by reference, if successful, it
        ///     will take the value array[index] otherwise default value.</para><para>Передает
        ///     аргумент по ссылке, в случае успеха он примет значение array[index] в противном
        ///     случае значение по умолчанию.</para></param>
        /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
        ///     в противном случае false</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool TryGetElement<T>(this T[] array, long index, out T element)
        {
            if (array != null && array.LongLength > index)
            {
                element = array[index];
                return true;
            }
            else
            {
                element = default;
                return false;
            }
        }

        /// <summary>
        /// <para>Copying  of elements from one array to another array.</para>
        /// <para>Копирует  элементы из одного массива в другой массив.</para>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///     массива.</para></typeparam>
        /// <param name="array"><para>The array  to copy.</para><para>Массив который необходимо
        ///     скопировать.</para></param>
        /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] Clone<T>(this T[] array)
        {
            var copy = new T[array.LongLength];
            Array.Copy(array, 0L, copy, 0L, array.LongLength);
            return copy;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);

        /// <summary>
        /// <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>
        /// <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>
        /// </summary>
        /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
        ///     массива.</para></typeparam>
        /// <param name="array"><para>Array to expand  Elements.</para><para>Массив для
        ///     расширения элементов.</para></param>
        /// <param name="shift"><para>The number to expand the array</para><para>Число на
        ///     которое необходимо расширить массив.</para></param>
        /// <returns>
        /// <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>
        /// <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>
```

```csharp
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
            ↪  to the element variable.</para>
131         /// <para>Одно из значений массива с индексом переменной position++ типа int назначается
            ↪  в переменную element.</para>
132         /// </summary>
133         /// <param name="array"><para>An array whose specific value will be assigned to the
            ↪  element variable.</para><para>Массив, определенное значений которого присваивается
            ↪  переменной element</para></param>
134         /// <param name="position"><para>Reference to a position in an array of int
            ↪  type.</para><para>Ссылка на позицию в массиве типа int.</para></param>
135         /// <param name="element"><para>The variable which needs to be assigned a specific value
            ↪  from the array.</para><para>Переменная, которой нужно присвоить определенное
            ↪  значение из массива.</para></param>
136         /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
            ↪  массива.<para></typeparam>
137         [MethodImpl(MethodImplOptions.AggressiveInlining)]
138         public static void Add<T>(this T[] array, ref int position, T element) =>
            ↪  array[position++] = element;
139
140         /// <summary>
141         /// <para>One of the array values with index on variable position++ type long is passed
            ↪  to the element variable.</para>
142         /// <para>Одно из значений массива с индексом переменной possition++ типа long
            ↪  назначается в переменную element.</para>
143         /// </summary>
144         /// <param name="array"><para>An array whose specific value will be assigned to the
            ↪  element variable.</para><para>Массив, определенное значений которого присваивается
            ↪  переменной element</para></param>
145         /// <param name="position"><para>Reference to a position in an array of long
            ↪  type.</para><para>Ссылка на позицию в массиве типа long.</para></param>
146         /// <param name="element"><para>The variable which needs to be assigned a specific value
            ↪  from the array.</para><para>Переменная, которой нужно присвоить определенное
            ↪  значение из массива.</para></param>
147         /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
            ↪  массива.<para></typeparam>
148         [MethodImpl(MethodImplOptions.AggressiveInlining)]
149         public static void Add<T>(this T[] array, ref long position, T element) =>
            ↪  array[position++] = element;
150
151         [MethodImpl(MethodImplOptions.AggressiveInlining)]
152         public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, TElement element, TReturnConstant
            ↪  returnConstant)
153         {
154             array.Add(ref position, element);
155             return returnConstant;
156         }
157
158         [MethodImpl(MethodImplOptions.AggressiveInlining)]
159         public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
            ↪  array[position++] = elements[0];
160
161         [MethodImpl(MethodImplOptions.AggressiveInlining)]
162         public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
            ↪  TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
            ↪  returnConstant)
```

```csharp
            {
                array.AddFirst(ref position, elements);
                return returnConstant;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
                TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
                returnConstant)
            {
                array.AddAll(ref position, elements);
                return returnConstant;
            }

            /// <summary>
            /// <para>Adding a collection of elements starting from a specific position.</para>
            /// <para>Добавляет коллекции элементов начиная с определенной позиции.</para>
            /// </summary>
            /// <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>
            /// <param name="position"><para>The position from which to start adding
                elements.</para><para>Позиция с которой начнется добавление элементов.</para></param>
            /// <param name="elements"><para>Added all collection of elements to
                array.</para><para>Добавляется вся коллекция элементов в массив. </para></param>
            /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
                массива.</para></typeparam>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
            {
                for (var i = 0; i < elements.Count; i++)
                {
                    array.Add(ref position, elements[i]);
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
                TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
                TReturnConstant returnConstant)
            {
                array.AddSkipFirst(ref position, elements);
                return returnConstant;
            }

            /// <summary>
            /// <para>Adds all elements except the first.</para>
            /// <para>Добавляет все элементы, кроме первого.</para>
            /// </summary>
            /// <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>
            /// <param name="position"><para>The position from which to start adding
                elements.</para><para>Позиция, с которой начинается добавление
                элементов.</para></param>
            /// <param name="elements"><para>List of added elements.</para><para>Список добавляемых
                элементов.</para></param>
            /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
                массива.</para></typeparam>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
                => array.AddSkipFirst(ref position, elements, 1);

            /// <summary>
            /// <para>Adds all but the first element, skipping a specified number of elements.</para>
            /// <para>Добавляет все элементы, кроме первого, пропуская определенное количество
                элементов.</para>
            /// </summary>
            /// <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>
            /// <param name="position"><para>The position from which to start adding
                elements.</para><para>Позиция, с которой начинается добавление
                элементов.</para></param>
            /// <param name="elements"><para>List of added elements.</para><para>Список добавляемых
                элементов.</para></param>
            /// <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
```

```csharp
                    {
                        // What is going on here?
                        Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
                    }
                    // What is going on here?
                    var mask = (int)(_length % 64);
                    if (mask > 0)
                    {
                        _array[oldWords - 1] &= (1L << mask) - 1;
                    }
                }
                else
                {
                    // Looks like minimum and maximum positive words are not updated
                    throw new NotImplementedException();
                }
                _length = value;
            }
        }

        #region Constructors

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static BitString()
        {
            _bitsSetIn16Bits = new byte[65536][];
            int i, c, k;
            byte bitIndex;
            for (i = 0; i < 65536; i++)
            {
                // Calculating size of array (number of positive bits)
                for (c = 0, k = 1; k <= 65536; k <<= 1)
                {
                    if ((i & k) == k)
                    {
                        c++;
                    }
                }
                var array = new byte[c];
                // Adding positive bits indices into array
                for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <<= 1)
                {
                    if ((i & k) == k)
                    {
                        array[c++] = bitIndex;
                    }
                    bitIndex++;
                }
                _bitsSetIn16Bits[i] = array;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString(BitString other)
        {
            Ensure.Always.ArgumentNotNull(other, nameof(other));
            _length = other._length;
            _array = new long[GetWordsCountFromIndex(_length)];
            _minPositiveWord = other._minPositiveWord;
            _maxPositiveWord = other._maxPositiveWord;
            Array.Copy(other._array, _array, _array.LongLength);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString(long length)
        {
            Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
            _length = length;
            _array = new long[GetWordsCountFromIndex(_length)];
            MarkBordersAsAllBitsReset();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString(long length, bool defaultValue)
            : this(length)
        {
            if (defaultValue)
            {
                SetAll();
```

```csharp
141                }
142            }
143
144        #endregion
145
146        [MethodImpl(MethodImplOptions.AggressiveInlining)]
147        public BitString Not()
148        {
149            for (var i = 0L; i < _array.LongLength; i++)
150            {
151                _array[i] = ~_array[i];
152                RefreshBordersByWord(i);
153            }
154            return this;
155        }
156
157        [MethodImpl(MethodImplOptions.AggressiveInlining)]
158        public BitString ParallelNot()
159        {
160            var threads = Environment.ProcessorCount / 2;
161            if (threads <= 1)
162            {
163                return Not();
164            }
165            var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
    ↪  threads);
166            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    ↪  MaxDegreeOfParallelism = threads }, range =>
167            {
168                var maximum = range.Item2;
169                for (var i = range.Item1; i < maximum; i++)
170                {
171                    _array[i] = ~_array[i];
172                }
173            });
174            MarkBordersAsAllBitsSet();
175            TryShrinkBorders();
176            return this;
177        }
178
179        [MethodImpl(MethodImplOptions.AggressiveInlining)]
180        public BitString VectorNot()
181        {
182            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
183            {
184                return Not();
185            }
186            var step = Vector<long>.Count;
187            if (_array.Length < step)
188            {
189                return Not();
190            }
191            VectorNotLoop(_array, step, 0, _array.Length);
192            MarkBordersAsAllBitsSet();
193            TryShrinkBorders();
194            return this;
195        }
196
197        [MethodImpl(MethodImplOptions.AggressiveInlining)]
198        public BitString ParallelVectorNot()
199        {
200            var threads = Environment.ProcessorCount / 2;
201            if (threads <= 1)
202            {
203                return VectorNot();
204            }
205            if (!Vector.IsHardwareAccelerated)
206            {
207                return ParallelNot();
208            }
209            var step = Vector<long>.Count;
210            if (_array.Length < (step * threads))
211            {
212                return VectorNot();
213            }
214            var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
```

```csharp
215            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
     ↪   MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
     ↪   range.Item1, range.Item2));
216            MarkBordersAsAllBitsSet();
217            TryShrinkBorders();
218            return this;
219        }
220
221        [MethodImpl(MethodImplOptions.AggressiveInlining)]
222        static private void VectorNotLoop(long[] array, int step, int start, int maximum)
223        {
224            var i = start;
225            var range = maximum - start - 1;
226            var stop = range - (range % step);
227            for (; i < stop; i += step)
228            {
229                (~new Vector<long>(array, i)).CopyTo(array, i);
230            }
231            for (; i < maximum; i++)
232            {
233                array[i] = ~array[i];
234            }
235        }
236
237        [MethodImpl(MethodImplOptions.AggressiveInlining)]
238        public BitString And(BitString other)
239        {
240            EnsureBitStringHasTheSameSize(other, nameof(other));
241            GetCommonOuterBorders(this, other, out long from, out long to);
242            var otherArray = other._array;
243            for (var i = from; i <= to; i++)
244            {
245                _array[i] &= otherArray[i];
246                RefreshBordersByWord(i);
247            }
248            return this;
249        }
250
251        [MethodImpl(MethodImplOptions.AggressiveInlining)]
252        public BitString ParallelAnd(BitString other)
253        {
254            var threads = Environment.ProcessorCount / 2;
255            if (threads <= 1)
256            {
257                return And(other);
258            }
259            EnsureBitStringHasTheSameSize(other, nameof(other));
260            GetCommonOuterBorders(this, other, out long from, out long to);
261            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
262            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
     ↪   MaxDegreeOfParallelism = threads }, range =>
263            {
264                var maximum = range.Item2;
265                for (var i = range.Item1; i < maximum; i++)
266                {
267                    _array[i] &= other._array[i];
268                }
269            });
270            MarkBordersAsAllBitsSet();
271            TryShrinkBorders();
272            return this;
273        }
274
275        [MethodImpl(MethodImplOptions.AggressiveInlining)]
276        public BitString VectorAnd(BitString other)
277        {
278            if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
279            {
280                return And(other);
281            }
282            var step = Vector<long>.Count;
283            if (_array.Length < step)
284            {
285                return And(other);
286            }
287            EnsureBitStringHasTheSameSize(other, nameof(other));
288            GetCommonOuterBorders(this, other, out int from, out int to);
289            VectorAndLoop(_array, other._array, step, from, to + 1);
```

```csharp
                    MarkBordersAsAllBitsSet();
                    TryShrinkBorders();
                    return this;
                }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorAnd(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return VectorAnd(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return ParallelAnd(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * threads))
            {
                return VectorAnd(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out int from, out int to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
                step, range.Item1, range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
            int maximum)
        {
            var i = start;
            var range = maximum - start - 1;
            var stop = range - (range % step);
            for (; i < stop; i += step)
            {
                (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] &= otherArray[i];
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Or(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] |= other._array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelOr(BitString other)
        {
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                return Or(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                MaxDegreeOfParallelism = threads }, range =>
            {
                var maximum = range.Item2;
```

```
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 {
      ↪  MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
      ↪  step, range.Item1, range.Item2));
415        MarkBordersAsAllBitsSet();
416        TryShrinkBorders();
417        return this;
418    }
419
420    [MethodImpl(MethodImplOptions.AggressiveInlining)]
421    static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
      ↪  int maximum)
422    {
423        var i = start;
424        var range = maximum - start - 1;
425        var stop = range - (range % step);
426        for (; i < stop; i += step)
427        {
428            (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
429        }
430        for (; i < maximum; i++)
431        {
432            array[i] |= otherArray[i];
433        }
434    }
435
436    [MethodImpl(MethodImplOptions.AggressiveInlining)]
437    public BitString Xor(BitString other)
438    {
```

```csharp
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonOuterBorders(this, other, out long from, out long to);
                for (var i = from; i <= to; i++)
                {
                    _array[i] ^= other._array[i];
                    RefreshBordersByWord(i);
                }
                return this;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public BitString ParallelXor(BitString other)
            {
                var threads = Environment.ProcessorCount / 2;
                if (threads <= 1)
                {
                    return Xor(other);
                }
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonOuterBorders(this, other, out long from, out long to);
                var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
                Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                →  MaxDegreeOfParallelism = threads }, range =>
                {
                    var maximum = range.Item2;
                    for (var i = range.Item1; i < maximum; i++)
                    {
                        _array[i] ^= other._array[i];
                    }
                });
                MarkBordersAsAllBitsSet();
                TryShrinkBorders();
                return this;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public BitString VectorXor(BitString other)
            {
                if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
                {
                    return Xor(other);
                }
                var step = Vector<long>.Count;
                if (_array.Length < step)
                {
                    return Xor(other);
                }
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonOuterBorders(this, other, out int from, out int to);
                VectorXorLoop(_array, other._array, step, from, to + 1);
                MarkBordersAsAllBitsSet();
                TryShrinkBorders();
                return this;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public BitString ParallelVectorXor(BitString other)
            {
                var threads = Environment.ProcessorCount / 2;
                if (threads <= 1)
                {
                    return VectorXor(other);
                }
                if (!Vector.IsHardwareAccelerated)
                {
                    return ParallelXor(other);
                }
                var step = Vector<long>.Count;
                if (_array.Length < (step * threads))
                {
                    return VectorXor(other);
                }
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonOuterBorders(this, other, out int from, out int to);
                var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
                Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                →  MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
                →  step, range.Item1, range.Item2));
```

```csharp
                    MarkBordersAsAllBitsSet();
                    TryShrinkBorders();
                    return this;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
            ↪   int maximum)
            {
                var i = start;
                var range = maximum - start - 1;
                var stop = range - (range % step);
                for (; i < stop; i += step)
                {
                    (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
                }
                for (; i < maximum; i++)
                {
                    array[i] ^= otherArray[i];
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void RefreshBordersByWord(long wordIndex)
            {
                if (_array[wordIndex] == 0)
                {
                    if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
                    {
                        _minPositiveWord++;
                    }
                    if (wordIndex == _maxPositiveWord && wordIndex != 0)
                    {
                        _maxPositiveWord--;
                    }
                }
                else
                {
                    if (wordIndex < _minPositiveWord)
                    {
                        _minPositiveWord = wordIndex;
                    }
                    if (wordIndex > _maxPositiveWord)
                    {
                        _maxPositiveWord = wordIndex;
                    }
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool TryShrinkBorders()
            {
                GetBorders(out long from, out long to);
                while (from <= to && _array[from] == 0)
                {
                    from++;
                }
                if (from > to)
                {
                    MarkBordersAsAllBitsReset();
                    return true;
                }
                while (to >= from && _array[to] == 0)
                {
                    to--;
                }
                if (to < from)
                {
                    MarkBordersAsAllBitsReset();
                    return true;
                }
                var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
                if (bordersUpdated)
                {
                    SetBorders(from, to);
                }
                return bordersUpdated;
            }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Get(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(long index, bool value)
        {
            if (value)
            {
                Set(index);
            }
            else
            {
                Reset(index);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            _array[wordIndex] |= mask;
            RefreshBordersByWord(wordIndex);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            _array[wordIndex] &= ~mask;
            RefreshBordersByWord(wordIndex);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(long index)
        {
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            if ((_array[wordIndex] & mask) == 0)
            {
                _array[wordIndex] |= mask;
                RefreshBordersByWord(wordIndex);
                return true;
            }
            else
            {
                return false;
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetAll(bool value)
        {
            if (value)
            {
                SetAll();
            }
            else
            {
                ResetAll();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void SetAll()
        {
            const long fillValue = unchecked((long)0xffffffffffffffff);
            var words = GetWordsCountFromIndex(_length);
            for (var i = 0; i < words; i++)
            {
                _array[i] = fillValue;
            }
```

```csharp
                    MarkBordersAsAllBitsSet();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void ResetAll()
            {
                const long fillValue = 0;
                GetBorders(out long from, out long to);
                for (var i = from; i <= to; i++)
                {
                    _array[i] = fillValue;
                }
                MarkBordersAsAllBitsReset();
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public List<long> GetSetIndices()
            {
                var result = new List<long>();
                GetBorders(out long from, out long to);
                for (var i = from; i <= to; i++)
                {
                    var word = _array[i];
                    if (word != 0)
                    {
                        AppendAllSetBitIndices(result, i, word);
                    }
                }
                return result;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public List<ulong> GetSetUInt64Indices()
            {
                var result = new List<ulong>();
                GetBorders(out ulong from, out ulong to);
                for (var i = from; i <= to; i++)
                {
                    var word = _array[i];
                    if (word != 0)
                    {
                        AppendAllSetBitIndices(result, i, word);
                    }
                }
                return result;
            }

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

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public long CountSetBits()
            {
                var total = 0L;
                GetBorders(out long from, out long to);
                for (var i = from; i <= to; i++)
                {
                    var word = _array[i];
```

```csharp
                    if (word != 0)
                    {
                        total += CountSetBitsForWord(word);
                    }
                }
                return total;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool HaveCommonBits(BitString other)
            {
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonInnerBorders(this, other, out long from, out long to);
                var otherArray = other._array;
                for (var i = from; i <= to; i++)
                {
                    var left = _array[i];
                    var right = otherArray[i];
                    if (left != 0 && right != 0 && (left & right) != 0)
                    {
                        return true;
                    }
                }
                return false;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public long CountCommonBits(BitString other)
            {
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonInnerBorders(this, other, out long from, out long to);
                var total = 0L;
                var otherArray = other._array;
                for (var i = from; i <= to; i++)
                {
                    var left = _array[i];
                    var right = otherArray[i];
                    var combined = left & right;
                    if (combined != 0)
                    {
                        total += CountSetBitsForWord(combined);
                    }
                }
                return total;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public List<long> GetCommonIndices(BitString other)
            {
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonInnerBorders(this, other, out long from, out long to);
                var result = new List<long>();
                var otherArray = other._array;
                for (var i = from; i <= to; i++)
                {
                    var left = _array[i];
                    var right = otherArray[i];
                    var combined = left & right;
                    if (combined != 0)
                    {
                        AppendAllSetBitIndices(result, i, combined);
                    }
                }
                return result;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public List<ulong> GetCommonUInt64Indices(BitString other)
            {
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonBorders(this, other, out ulong from, out ulong to);
                var result = new List<ulong>();
                var otherArray = other._array;
                for (var i = from; i <= to; i++)
                {
                    var left = _array[i];
                    var right = otherArray[i];
                    var combined = left & right;
                    if (combined != 0)
```

```csharp
                {
                    AppendAllSetBitIndices(result, i, combined);
                }
            }
            return result;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetFirstCommonBitIndex(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    return GetFirstSetBitForWord(i, combined);
                }
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public long GetLastCommonBitIndex(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = to; i >= from; i--)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    return GetLastSetBitForWord(i, combined);
                }
            }
            return -1;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(BitString other)
        {
            if (_length != other._length)
            {
                return false;
            }
            var otherArray = other._array;
            if (_array.Length != otherArray.Length)
            {
                return false;
            }
            if (_minPositiveWord != other._minPositiveWord)
            {
                return false;
            }
            if (_maxPositiveWord != other._maxPositiveWord)
            {
                return false;
            }
            GetCommonBorders(this, other, out ulong from, out ulong to);
            for (var i = from; i <= to; i++)
            {
                if (_array[i] != otherArray[i])
                {
                    return false;
                }
            }
            return true;
        }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
        {
            Ensure.Always.ArgumentNotNull(other, argumentName);
            if (_length != other._length)
            {
                throw new ArgumentException("Bit string must be the same size.", argumentName);
            }
        }

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

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Range<long> GetValidLengthRange() => (0, long.MaxValue);

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
        {
            GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
            ↪  bits32to47, out byte[] bits48to63);
```

```csharp
                return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
            }

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

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
            ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
            {
                for (var j = 0; j < bits00to15.Length; j++)
                {
                    result.Add(bits00to15[j] + (i * 64));
                }
                for (var j = 0; j < bits16to31.Length; j++)
                {
                    result.Add(bits16to31[j] + 16 + (i * 64));
                }
                for (var j = 0; j < bits32to47.Length; j++)
                {
                    result.Add(bits32to47[j] + 32 + (i * 64));
                }
                for (var j = 0; j < bits48to63.Length; j++)
                {
                    result.Add(bits48to63[j] + 48 + (i * 64));
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
            ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
            {
                for (var j = 0; j < bits00to15.Length; j++)
                {
                    result.Add(bits00to15[j] + (i * 64));
                }
                for (var j = 0; j < bits16to31.Length; j++)
                {
                    result.Add(bits16to31[j] + 16UL + (i * 64));
                }
                for (var j = 0; j < bits32to47.Length; j++)
                {
                    result.Add(bits32to47[j] + 32UL + (i * 64));
                }
                for (var j = 0; j < bits48to63.Length; j++)
                {
                    result.Add(bits48to63[j] + 48UL + (i * 64));
                }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
            ↪ bits32to47, byte[] bits48to63)
            {
                if (bits00to15.Length > 0)
                {
                    return bits00to15[0] + (i * 64);
                }
                if (bits16to31.Length > 0)
                {
                    return bits16to31[0] + 16 + (i * 64);
                }
                if (bits32to47.Length > 0)
                {
                    return bits32to47[0] + 32 + (i * 64);
                }
                return bits48to63[0] + 48 + (i * 64);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
            ↪ bits32to47, byte[] bits48to63)
```

```csharp
        {
            if (bits48to63.Length > 0)
            {
                return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
            }
            if (bits32to47.Length > 0)
            {
                return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
            }
            if (bits16to31.Length > 0)
            {
                return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
            }
            return bits00to15[bits00to15.Length - 1] + (i * 64);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
        ↪ byte[] bits32to47, out byte[] bits48to63)
        {
            bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
            bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
            bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
            bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
        ↪ out long to)
        {
            from = Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
        ↪ out long to)
        {
            from = Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
        ↪ out int to)
        {
            from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
        ↪ ulong to)
        {
            from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordIndexFromIndex(long index) => index >> 6;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);

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

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

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

```csharp
using System.Runtime.CompilerServices;
using Platform.Random;
```

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

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

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

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

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

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

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Runtime.CompilerServices;
5  using Platform.Exceptions;
6  using Platform.Exceptions.ExtensionRoots;
7
8  #pragma warning disable IDE0060 // Remove unused parameter
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     public static class EnsureExtensions
14     {
15         #region Always
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
                public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
18
                    ICollection<T> argument, string argumentName, string message)
19
                {
20
                    if (argument.IsNullOrEmpty())
21
                    {
22
                        throw new ArgumentException(message, argumentName);
23
                    }
24
                }
25
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
26
                public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
27
                    ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
                    argumentName, null);
28
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
                public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
30
                    ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
31
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
32
                public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
33
                    string argument, string argumentName, string message)
34
                {
35
                    if (string.IsNullOrWhiteSpace(argument))
36
                    {
37
                        throw new ArgumentException(message, argumentName);
38
                    }
39
                }
40
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
41
                public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
42
                    string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
                    argument, argumentName, null);
43
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
44
                public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
45
                    string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
46
                #endregion
47
48
                #region OnDebug
49
50
                [Conditional("DEBUG")]
51
                public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
52
                    ICollection<T> argument, string argumentName, string message) =>
                    Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
53
                [Conditional("DEBUG")]
54
                public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
55
                    ICollection<T> argument, string argumentName) =>
                    Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
56
                [Conditional("DEBUG")]
57
                public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
58
                    ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
59
                [Conditional("DEBUG")]
60
                public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
61
                    root, string argument, string argumentName, string message) =>
                    Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
62
                [Conditional("DEBUG")]
63
                public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
64
                    root, string argument, string argumentName) =>
                    Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
65
                [Conditional("DEBUG")]
66
                public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
67
                    root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
                    null, null);
68
                #endregion
69
            }
70
        }
71
```

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

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

```csharp
 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

```csharp
 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/CharIListExtensions.cs

```csharp
 1   using System.Collections.Generic;
 2   using System.Runtime.CompilerServices;
 3
 4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 5
 6   namespace Platform.Collections.Lists
 7   {
 8       public static class CharIListExtensions
 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           }
```

```csharp
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

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

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Collections.Lists
8   {
9       public static class IListExtensions
10      {
11          /// <summary>
12          /// <para>Gets the element from specified index if the list is not null and the index is
    ↪  within the list's boundaries, otherwise it returns default value of type T.</para>
13          /// <para>Получает элемент из указанного индекса, если список не является null и индекс
    ↪  находится в границах списка, в противном случае он возвращает значение по умолчанию
    ↪  типа T.</para>
14          /// </summary>
15          /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↪  списка.</typeparam>
16          /// <param name="list"><para>The checked list.</para><para>Проверяемый
    ↪  список.</para></param>
17          /// <param name="index"><para>The index of element.</para><para>Индекс
    ↪  элемента.</para></param>
18          /// <returns>
```

```csharp
19          /// <para>If the specified index is within list's boundaries, then - list[index],
            ↪   otherwise the default value.</para>
20          /// <para>Если указанный индекс находится в пределах границ списка, тогда - list[index],
            ↪   иначе же значение по умолчанию.</para>
21          /// </returns>
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
            ↪   list.Count > index ? list[index] : default;

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

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

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

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

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

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

```csharp
142         /// </returns>
143         [MethodImpl(MethodImplOptions.AggressiveInlining)]
144         public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
145         {
146             list.AddSkipFirst(elements);
147             return true;
148         }
149
150         /// <summary>
151         /// <para>Adds values to the list skipping the first element.</para>
152         /// <para>Добавляет значения в список пропуская первый элемент.</para>
153         /// </summary>
154         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
         ↪   списка.</para></typeparam>
155         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
         ↪   нужно добавить значения.</para></param>
156         /// <param name="elements"><para>List of values to add.</para><para>Список значений
         ↪   которые необходимо добавить.</para></param>
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 values to the list skipping a specified number of first elements.</para>
162         /// <para>Добавляет в список значения пропуская определенное количество первых
         ↪   элементов.</para>
163         /// </summary>
164         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
         ↪   списка.</para></typeparam>
165         /// <param name="list"><para>The list to add the values to.</para><para>Список в который
         ↪   нужно добавить значения.</para></param>
166         /// <param name="elements"><para>The number of elements to skip.</para><para>Количество
         ↪   пропускаемых элементов.</para></param>
167         /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
         ↪   пропускаемых элементов.</para></param>
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         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
         ↪   списка.</para></typeparam>
182         /// <param name="list"><para>The checked list.</para><para>Список для
         ↪   проверки.</para></param>
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 equality.</para>
192         /// <para>Сравнивает два списка на равенство.</para>
193         /// </summary>
194         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
         ↪   списка.</para></typeparam>
195         /// <param name="left"><para>The first compared list.</para><para>Первый список для
         ↪   сравнения.</para></param>
196         /// <param name="right"><para>The second compared list.</para><para>Второй список для
         ↪   сравнения.</para></param>
197         /// <returns>
198         /// <para>If the passed lists are equal to each other, true is returned, otherwise
         ↪   false.</para>
199         /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
         ↪   false.</para>
200         /// </returns>
201         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

```csharp
        /// <para>Создаёт массив, копируя из списка все элементы которые удовлетворяют
        ///    предикату. Если список не передан, возвращается null.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ///    списка.</para></typeparam>
        /// <param name="list">The list to copy from.<para>Список для копирования.</para></param>
        /// <param name="predicate"><para>A function that determines whether an element should
        ///    be copied.</para><para>Функция определяющая должен ли копироваться
        ///    элемент.</para></param>
        /// <returns>
        /// <para>An array with copied elements from the list.</para>
        /// <para>Массив с скопированными элементами из списка.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
        {
            if (list == null)
            {
                return null;
            }
            var result = new List<T>(list.Count);
            for (var i = 0; i < list.Count; i++)
            {
                if (predicate(list[i]))
                {
                    result.Add(list[i]);
                }
            }
            return result.ToArray();
        }

        /// <summary>
        /// <para>Copies all the elements of the list into an array and returns it.</para>
        /// <para>Копирует все элементы списка в массив и возвращает его.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ///    списка.</para></typeparam>
        /// <param name="list"><para>The list to copy from.</para><para>Список для
        ///    копирования.</para></param>
        /// <returns>
        /// <para>An array with all the elements of the passed list.</para>
        /// <para>Массив со всеми элементами переданного списка.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] ToArray<T>(this IList<T> list)
        {
            var array = new T[list.Count];
            list.CopyTo(array, 0);
            return array;
        }

        /// <summary>
        /// <para>Executes the passed action for each item in the list.</para>
        /// <para>Выполняет переданное действие для каждого элемента в списке.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ///    списка.</para></typeparam>
        /// <param name="list"><para>The list of elements for which the action will be
        ///    executed.</para><para>Список элементов для которых будет выполняться
        ///    действие.</para></param>
        /// <param name="action"><para>A function that will be called for each element of the
        ///    list.</para><para>Функция которая будет вызываться для каждого элемента
        ///    списка.</para></param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ForEach<T>(this IList<T> list, Action<T> action)
        {
            for (var i = 0; i < list.Count; i++)
            {
                action(list[i]);
            }
        }

        /// <summary>
        /// <para>Generates a hash code for the entire list based on the values of its
        ///    elements.</para>
        /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
        /// </summary>
```

```csharp
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪ списка.</para></typeparam>
        /// <param name="list"><para>Hash list.</para><para>Список для
        ↪ хеширования.</para></param>
        /// <returns>
        /// <para>The hash code of the list.</para>
        /// <para>Хэш-код списка.</para>
        /// </returns>
        /// <remarks>
        /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
        ↪ -overridden-system-object-gethashcode
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GenerateHashCode<T>(this IList<T> list)
        {
            var hashAccumulator = 17;
            for (var i = 0; i < list.Count; i++)
            {
                hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
            }
            return hashAccumulator;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
```

```csharp
        /// <summary>
        /// <para>Shifts all elements of the list to the right by the specified number of
        ↪  elements and returns an array.</para>
        /// <para>Сдвигает вправо все элементы списка на указанное количество элементов и
        ↪  возвращает массив.</para>
        /// </summary>
        /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
        ↪  списка.</para></typeparam>
        /// <param name="list"><para>The list to copy from.</para><para>Список для
        ↪  копирования.</para></param>
        /// <param name="skip"><para>The number of items to shift.</para><para>Количество
        ↪  сдвигаемых элементов.</para></param>
        /// <returns>
        /// <para>If the variable shift is less than zero - an <see
        ↪  cref="NotImplementedException"/> exception is thrown, but if the variable shift is 0
        ↪  an exact copy of the array is returned. Otherwise, an array is returned with the
        ↪  shift of the elements.</para>
        /// <para>Если переменная shift меньше нуля - выбрасывается исключение <see
        ↪  cref="NotImplementedException"/>, если же переменная shift равена 0 возвращается
        ↪  точная копия массива. Иначе возвращается массив со сдвигом элементов.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
        {
            if (shift < 0)
            {
                throw new NotImplementedException();
            }
            if (shift == 0)
            {
                return list.ToArray();
            }
            else
            {
                var result = new T[list.Count + shift];
                for (int r = 0, w = shift; r < list.Count; r++, w++)
                {
                    result[w] = list[r];
                }
                return result;
            }
        }
    }
}
```

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

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

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

namespace Platform.Collections.Lists
{
    public class ListFiller<TElement, TReturnConstant>
    {
        protected readonly List<TElement> _list;
        protected readonly TReturnConstant _returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ListFiller(List<TElement> list, TReturnConstant returnConstant)
        {
            _list = list;
            _returnConstant = returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ListFiller(List<TElement> list) : this(list, default) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
        ↪  _list.AddFirstAndReturnTrue(elements);
```

```csharp
32        [MethodImpl(MethodImplOptions.AggressiveInlining)]
33        public bool AddAllAndReturnTrue(IList<TElement> elements) =>
   ↪        _list.AddAllAndReturnTrue(elements);

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

38        [MethodImpl(MethodImplOptions.AggressiveInlining)]
39        public TReturnConstant AddAndReturnConstant(TElement element)
40        {
41            _list.Add(element);
42            return _returnConstant;
43        }

45        [MethodImpl(MethodImplOptions.AggressiveInlining)]
46        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
47        {
48            _list.AddFirst(elements);
49            return _returnConstant;
50        }

52        [MethodImpl(MethodImplOptions.AggressiveInlining)]
53        public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
54        {
55            _list.AddAll(elements);
56            return _returnConstant;
57        }

59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
61        {
62            _list.AddSkipFirst(elements);
63            return _returnConstant;
64        }
65    }
66 }
```

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

```csharp
1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class CharSegment : Segment<char>
12     {
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
   ↪        length) { }

16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17        public override int GetHashCode()
18        {
19            // Base can be not an array, but still IList<char>
20            if (Base is char[] baseArray)
21            {
22                return baseArray.GenerateHashCode(Offset, Length);
23            }
24            else
25            {
26                return this.GenerateHashCode();
27            }
28        }

30        [MethodImpl(MethodImplOptions.AggressiveInlining)]
31        public override bool Equals(Segment<char> other)
32        {
33            bool contentEqualityComparer(IList<char> left, IList<char> right)
34            {
35                // Base can be not an array, but still IList<char>
36                if (Base is char[] baseArray && other.Base is char[] otherArray)
37                {
38                    return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
39                }
```

```csharp
                    else
                    {
                        return left.ContentEqualTo(right);
                    }
                }
                return this.EqualTo(other, contentEqualityComparer);
            }

            public override bool Equals(object obj) => obj is Segment<char> charSegment ?
            ↪  Equals(charSegment) : false;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator string(CharSegment segment)
            {
                if (!(segment.Base is char[] array))
                {
                    array = segment.Base.ToArray();
                }
                return new string(array, segment.Offset, segment.Length);
            }

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

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

```csharp
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;

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

namespace Platform.Collections.Segments
{
    public class Segment<T> : IEquatable<Segment<T>>, IList<T>
    {
        public IList<T> Base
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        public int Offset
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }
        public int Length
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

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

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

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

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

        #region IList

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

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => Base[Offset + i] = value;
        }

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int IndexOf(T item)
        {
            var index = Base.IndexOf(item);
            if (index >= Offset)
            {
                var actualIndex = index - Offset;
                if (actualIndex < Length)
                {
                    return actualIndex;
                }
            }
            return -1;
        }

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Contains(T item) => IndexOf(item) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void CopyTo(T[] array, int arrayIndex)
        {
            for (var i = 0; i < Length; i++)
            {
                array.Add(ref arrayIndex, this[i]);
            }
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerator<T> GetEnumerator()
        {
            for (var i = 0; i < Length; i++)
            {
                yield return this[i];
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        #endregion
    }
}
```

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

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

namespace Platform.Collections.Segments.Walkers
{
    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) =>
       ↪   _minimumStringSegmentLength = minimumStringSegmentLength;
15
16           [MethodImpl(MethodImplOptions.AggressiveInlining)]
17           protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
18
19           [MethodImpl(MethodImplOptions.AggressiveInlining)]
20           public virtual void WalkAll(IList<T> elements)
21           {
22               for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
                   ↪   offset <= maxOffset; offset++)
23               {
24                   for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                       ↪   offset; length <= maxLength; length++)
25                   {
26                       Iteration(CreateSegment(elements, offset, length));
27                   }
28               }
29           }
30
31           [MethodImpl(MethodImplOptions.AggressiveInlining)]
32           protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
33
34           [MethodImpl(MethodImplOptions.AggressiveInlining)]
35           protected abstract void Iteration(TSegment segment);
36       }
37   }
```

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

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

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

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

```
14          }
15      }
```

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

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

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

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

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

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

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

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

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

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

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

```csharp
using System.Runtime.CompilerServices;

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

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

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

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

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

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

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

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

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

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

```csharp
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Sets
7  {
8      public static class ISetExtensions
9      {
10          [MethodImpl(MethodImplOptions.AggressiveInlining)]
11          public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
12
13          [MethodImpl(MethodImplOptions.AggressiveInlining)]
14          public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
   ↪   set.Remove(element);
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
18          {
19              set.Add(element);
20              return true;
21          }
22
23          [MethodImpl(MethodImplOptions.AggressiveInlining)]
24          public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
25          {
26              AddFirst(set, elements);
27              return true;
28          }
29
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
   ↪   set.Add(elements[0]);
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
35          {
36              set.AddAll(elements);
37              return true;
38          }
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public static void AddAll<T>(this ISet<T> set, IList<T> elements)
42          {
43              for (var i = 0; i < elements.Count; i++)
44              {
45                  set.Add(elements[i]);
46              }
47          }
48
49          [MethodImpl(MethodImplOptions.AggressiveInlining)]
50          public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
51          {
52              set.AddSkipFirst(elements);
53              return true;
54          }
55
56          [MethodImpl(MethodImplOptions.AggressiveInlining)]
57          public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
   ↪   set.AddSkipFirst(elements, 1);
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
61          {
62              for (var i = skip; i < elements.Count; i++)
63              {
64                  set.Add(elements[i]);
```

```
 65            }
 66        }

 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

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

```csharp
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

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

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

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

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

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

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

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

```
1   using System;
2   using System.Globalization;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Collections
8   {
9       public static class StringExtensions
10      {
11          [MethodImpl(MethodImplOptions.AggressiveInlining)]
12          public static string CapitalizeFirstLetter(this string @string)
13          {
14              if (string.IsNullOrWhiteSpace(@string))
15              {
16                  return @string;
17              }
18              var chars = @string.ToCharArray();
19              for (var i = 0; i < chars.Length; i++)
20              {
21                  var category = char.GetUnicodeCategory(chars[i]);
22                  if (category == UnicodeCategory.UppercaseLetter)
23                  {
24                      return @string;
25                  }
26                  if (category == UnicodeCategory.LowercaseLetter)
27                  {
28                      chars[i] = char.ToUpper(chars[i]);
29                      return new string(chars);
30                  }
31              }
32              return @string;
33          }
34
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public static string Truncate(this string @string, int maxLength) =>
            ↪   string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
            ↪   Math.Min(@string.Length, maxLength));
37
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public static string TrimSingle(this string @string, char charToTrim)
40          {
41              if (!string.IsNullOrEmpty(@string))
42              {
43                  if (@string.Length == 1)
44                  {
45                      if (@string[0] == charToTrim)
46                      {
47                          return "";
48                      }
49                      else
50                      {
51                          return @string;
```

```
52                           }
53                     }
54                     else
55                     {
56                         var left = 0;
57                         var right = @string.Length - 1;
58                         if (@string[left] == charToTrim)
59                         {
60                             left++;
61                         }
62                         if (@string[right] == charToTrim)
63                         {
64                             right--;
65                         }
66                         return @string.Substring(left, right - left + 1);
67                     }
68                 }
69                 else
70                 {
71                     return @string;
72                 }
73             }
74         }
75     }
```

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

```csharp
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                  }
```

```csharp
55            }
56            return node;
57        }
58
59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
61
62        [MethodImpl(MethodImplOptions.AggressiveInlining)]
63        public Node AddChild(object key) => AddChild(key, new Node(null));
64
65        [MethodImpl(MethodImplOptions.AggressiveInlining)]
66        public Node AddChild(object key, object value) => AddChild(key, new Node(value));
67
68        [MethodImpl(MethodImplOptions.AggressiveInlining)]
69        public Node AddChild(object key, Node child)
70        {
71            ChildNodes.Add(key, child);
72            return child;
73        }
74
75        [MethodImpl(MethodImplOptions.AggressiveInlining)]
76        public Node SetChild(params object[] keys) => SetChildValue(null, keys);
77
78        [MethodImpl(MethodImplOptions.AggressiveInlining)]
79        public Node SetChild(object key) => SetChildValue(null, key);
80
81        [MethodImpl(MethodImplOptions.AggressiveInlining)]
82        public Node SetChildValue(object value, params object[] keys)
83        {
84            var node = this;
85            for (var i = 0; i < keys.Length; i++)
86            {
87                node = SetChildValue(value, keys[i]);
88            }
89            node.Value = value;
90            return node;
91        }
92
93        [MethodImpl(MethodImplOptions.AggressiveInlining)]
94        public Node SetChildValue(object value, object key)
95        {
96            if (!ChildNodes.TryGetValue(key, out Node child))
97            {
98                child = AddChild(key, value);
99            }
100            child.Value = value;
101            return child;
102        }
103    }
104 }
```

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

```csharp
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

```csharp
1  using System;
2  using System.Collections;
```

```csharp
using Xunit;
using Platform.Random;

namespace Platform.Collections.Tests
{
    public static class BitStringTests
    {
        [Fact]
        public static void BitGetSetTest()
        {
            const int n = 250;
            var bitArray = new BitArray(n);
            var bitString = new BitString(n);
            for (var i = 0; i < n; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                bitArray.Set(i, value);
                bitString.Set(i, value);
                Assert.Equal(value, bitArray.Get(i));
                Assert.Equal(value, bitString.Get(i));
            }
        }

        [Fact]
        public static void BitVectorNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitParallelNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitParallelVectorNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelVectorNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitVectorAndTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorAnd(y);
                w.And(v);
            });
        }

        [Fact]
        public static void BitParallelAndTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelAnd(y);
                w.And(v);
            });
        }

        [Fact]
        public static void BitParallelVectorAndTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelVectorAnd(y);
```

```csharp
                    w.And(v);
                });
        }

        [Fact]
        public static void BitVectorOrTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorOr(y);
                w.Or(v);
            });
        }

        [Fact]
        public static void BitParallelOrTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelOr(y);
                w.Or(v);
            });
        }

        [Fact]
        public static void BitParallelVectorOrTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelVectorOr(y);
                w.Or(v);
            });
        }

        [Fact]
        public static void BitVectorXorTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorXor(y);
                w.Xor(v);
            });
        }

        [Fact]
        public static void BitParallelXorTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelXor(y);
                w.Xor(v);
            });
        }

        [Fact]
        public static void BitParallelVectorXorTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.ParallelVectorXor(y);
                w.Xor(v);
            });
        }

        private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
        ↪  BitString, BitString> test)
        {
            const int n = 5654;
            var x = new BitString(n);
            var y = new BitString(n);
            while (x.Equals(y))
            {
                x.SetRandomBits();
                y.SetRandomBits();
            }
            var w = new BitString(x);
            var v = new BitString(y);
            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

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

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

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

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

```csharp
1    using Xunit;
2
3    namespace Platform.Collections.Tests
4    {
5        public static class StringTests
6        {
7            [Fact]
8            public static void CapitalizeFirstLetterTest()
9            {
10               Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
11               Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
```

```csharp
            Assert.Equal("  Hello", "  hello".CapitalizeFirstLetter());
        }

        [Fact]
        public static void TrimSingleTest()
        {
            Assert.Equal("", "'".TrimSingle('\''));
            Assert.Equal("", "''".TrimSingle('\''));
            Assert.Equal("hello", "'hello'".TrimSingle('\''));
            Assert.Equal("hello", "hello'".TrimSingle('\''));
            Assert.Equal("hello", "'hello".TrimSingle('\''));
        }
    }
}
```

Index