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

```
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[_position++] = element;
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
        {
            _array[_position++] = collection[0];
            return _returnConstant;
        }
    }
}
```

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

```
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[_position++] = element;
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> collection)
        {
            _array[_position++] = collection[0];
            return true;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Arrays
{
    public static class ArrayPool
    {
        public static readonly int DefaultSizesAmount = 512;
        public static readonly int DefaultMaxArraysPerSize = 32;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Collections.Stacks;

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

namespace Platform.Collections.Arrays
{
    /// <remarks>
    /// Original idea from
    ↪  http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
    /// </remarks>
    public class ArrayPool<T>
    {
        public static readonly T[] Empty = Array.Empty<T>();

        // May be use Default class for that later.
        [ThreadStatic]
        internal static ArrayPool<T> _threadInstance;
        internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
        ↪  = new ArrayPool<T>()); }

        private readonly int _maxArraysPerSize;
        private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
        ↪  Stack<T[]>>(ArrayPool.DefaultSizesAmount);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Disposable<T[]> Resize(Disposable<T[]> source, long size)
        {
            var destination = AllocateDisposable(size);
            T[] sourceArray = source;
            T[] destinationArray = destination;
            Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
            ↪  sourceArray.Length);
            source.Dispose();
            return destination;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Clear() => _pool.Clear();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual T[] Allocate(long size)
        {
            Ensure.Always.ArgumentInRange(size, (0, int.MaxValue));
```

```
            return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new T[size];
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual void Free(T[] array)
        {
            Ensure.Always.ArgumentNotNull(array, nameof(array));
            if (array.Length == 0)
            {
                return;
            }
            var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
            if (stack.Count == _maxArraysPerSize) // Stack is full
            {
                return;
            }
            stack.Push(array);
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;
using Platform.Collections.Segments;

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

namespace Platform.Collections.Arrays
{
    public class ArrayString<T> : Segment<T>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(int length) : base(new T[length], 0, length) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(T[] array) : base(array, 0, array.Length) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArrayString(T[] array, int length) : base(array, 0, length) { }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Arrays
{
    public static unsafe class CharArrayExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GenerateHashCode(this char[] array, int offset, int length)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            fixed (char* pointer = &array[offset])
            {
                for (char* s = pointer, last = s + length; s < last; s++)
                {
                    hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
                }
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L364
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[] right, int rightOffset)
        {
```

```
            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 ./Platform.Collections/Arrays/GenericArrayExtensions.cs

```
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
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] Clone<T>(this T[] array)
        {
            var copy = new T[array.Length];
            Array.Copy(array, 0, copy, 0, array.Length);
            return copy;
        }

```

```
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1);
21
22          [MethodImpl(MethodImplOptions.AggressiveInlining)]
23          public static IList<T> ShiftRight<T>(this T[] array, int shift)
24          {
25              var restrictions = new T[array.Length + shift];
26              Array.Copy(array, 0, restrictions, shift, array.Length);
27              return restrictions;
28          }
29      }
30  }
```

1.8 ./Platform.Collections/BitString.cs

```
1   using System;
2   using System.Collections.Concurrent;
3   using System.Collections.Generic;
4   using System.Numerics;
5   using System.Runtime.CompilerServices;
6   using System.Threading.Tasks;
7   using Platform.Exceptions;
8   using Platform.Ranges;
9
10  // ReSharper disable ForCanBeConvertedToForeach
11  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13  namespace Platform.Collections
14  {
15      /// <remarks>
16      /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
        ↪  64 бит в массиве значений.
17      /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
        ↪  байт в 8 байт.
18      /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
        ↪  помощью которой можно быстро
19      /// проверять есть ли значения непосредственно далее (ниже по уровню).
20      /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
21      /// </remarks>
22      public class BitString : IEquatable<BitString>
23      {
24          private static readonly byte[][] _bitsSetIn16Bits;
25          private long[] _array;
26          private long _length;
27          private long _minPositiveWord;
28          private long _maxPositiveWord;
29
30          public bool this[long index]
31          {
32              [MethodImpl(MethodImplOptions.AggressiveInlining)]
33              get => Get(index);
34              [MethodImpl(MethodImplOptions.AggressiveInlining)]
35              set => Set(index, value);
36          }
37
38          public long Length
39          {
40              [MethodImpl(MethodImplOptions.AggressiveInlining)]
41              get => _length;
42              [MethodImpl(MethodImplOptions.AggressiveInlining)]
43              set
44              {
45                  if (_length == value)
46                  {
47                      return;
48                  }
49                  Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
50                  // Currently we never shrink the array
51                  if (value > _length)
52                  {
53                      var words = GetWordsCountFromIndex(value);
54                      var oldWords = GetWordsCountFromIndex(_length);
55                      if (words > _array.LongLength)
56                      {
57                          var copy = new long[words];
58                          Array.Copy(_array, copy, _array.LongLength);
59                          _array = copy;
60                      }
61                      else
62                      {
63                          // What is going on here?
```

```
                    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();
        }
    }
```

```
143
144         #endregion
145
146         [MethodImpl(MethodImplOptions.AggressiveInlining)]
147         public BitString Not()
148         {
149             for (var i = 0; i < _array.Length; 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 processorCount = Environment.ProcessorCount;
161             if (processorCount <= 1)
162             {
163                 return Not();
164             }
165             var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
                ↪  processorCount);
166             Parallel.ForEach(partitioner.GetDynamicPartitions(), 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)
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 processorCount = Environment.ProcessorCount;
201             if (processorCount <= 1 && Vector.IsHardwareAccelerated)
202             {
203                 return VectorNot();
204             }
205             if (!Vector.IsHardwareAccelerated)
206             {
207                 return Not();
208             }
209             var step = Vector<long>.Count;
210             if (_array.Length < (step * Environment.ProcessorCount))
211             {
212                 return VectorNot();
213             }
214             var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
                ↪  processorCount);
215             Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorNotLoop(_array,
                ↪  step, range.Item1, range.Item2));
216             MarkBordersAsAllBitsSet();
217             TryShrinkBorders();
```

```
            return this;
        }

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
297         public BitString ParallelVectorAnd(BitString other)
298         {
299             var processorCount = Environment.ProcessorCount;
300             if (processorCount <= 1 && Vector.IsHardwareAccelerated)
301             {
302                 return VectorAnd(other);
303             }
304             if (!Vector.IsHardwareAccelerated)
305             {
306                 return And(other);
307             }
308             var step = Vector<long>.Count;
309             if (_array.Length < (step * Environment.ProcessorCount))
310             {
311                 return VectorAnd(other);
312             }
313             EnsureBitStringHasTheSameSize(other, nameof(other));
314             GetCommonOuterBorders(this, other, out long from, out long to);
315             var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
316             Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorAndLoop(_array,
                ↪  other._array, step, (int)range.Item1, (int)range.Item2));
317             MarkBordersAsAllBitsSet();
318             TryShrinkBorders();
319             return this;
320         }
321 
322         [MethodImpl(MethodImplOptions.AggressiveInlining)]
323         static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
                ↪  int maximum)
324         {
325             var i = start;
326             var range = maximum - start - 1;
327             var stop = range - (range % step);
328             for (; i < stop; i += step)
329             {
330                 var thisVector = new Vector<long>(array, i);
331                 var otherVector = new Vector<long>(otherArray, i);
332                 (thisVector & otherVector).CopyTo(array, i);
333             }
334             for (; i < maximum; i++)
335             {
336                 array[i] &= otherArray[i];
337             }
338         }
339 
340         [MethodImpl(MethodImplOptions.AggressiveInlining)]
341         public BitString Or(BitString other)
342         {
343             EnsureBitStringHasTheSameSize(other, nameof(other));
344             GetCommonOuterBorders(this, other, out long from, out long to);
345             for (var i = from; i <= to; i++)
346             {
347                 _array[i] |= other._array[i];
348                 RefreshBordersByWord(i);
349             }
350             return this;
351         }
352 
353         [MethodImpl(MethodImplOptions.AggressiveInlining)]
354         public BitString ParallelOr(BitString other)
355         {
356             var processorCount = Environment.ProcessorCount;
357             if (processorCount <= 1)
358             {
359                 return Or(other);
360             }
361             EnsureBitStringHasTheSameSize(other, nameof(other));
362             GetCommonOuterBorders(this, other, out long from, out long to);
363             var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
364             Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
365             {
366                 var maximum = range.Item2;
367                 for (var i = range.Item1; i < maximum; i++)
368                 {
369                     _array[i] |= other._array[i];
370                 }
371             });
372             MarkBordersAsAllBitsSet();
```

```
            TryShrinkBorders();
            return this;
        }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString ParallelVectorOr(BitString other)
        {
            var processorCount = Environment.ProcessorCount;
            if (processorCount <= 1 && Vector.IsHardwareAccelerated)
            {
                return VectorOr(other);
            }
            if (!Vector.IsHardwareAccelerated)
            {
                return Or(other);
            }
            var step = Vector<long>.Count;
            if (_array.Length < (step * Environment.ProcessorCount))
            {
                return VectorOr(other);
            }
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
            Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorOrLoop(_array,
            ↪  other._array, step, (int)range.Item1, (int)range.Item2));
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static private void VectorOrLoop(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)
            {
                var thisVector = new Vector<long>(array, i);
                var otherVector = new Vector<long>(otherArray, i);
                (thisVector | otherVector).CopyTo(array, i);
            }
            for (; i < maximum; i++)
            {
                array[i] |= otherArray[i];
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BitString Xor(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];
```

```
449                 RefreshBordersByWord(i);
450             }
451             return this;
452         }
453 
454         [MethodImpl(MethodImplOptions.AggressiveInlining)]
455         public BitString ParallelXor(BitString other)
456         {
457             var processorCount = Environment.ProcessorCount;
458             if (processorCount <= 1)
459             {
460                 return Xor(other);
461             }
462             EnsureBitStringHasTheSameSize(other, nameof(other));
463             GetCommonOuterBorders(this, other, out long from, out long to);
464             var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
465             Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
466             {
467                 var maximum = range.Item2;
468                 for (var i = range.Item1; i < maximum; i++)
469                 {
470                     _array[i] ^= other._array[i];
471                 }
472             });
473             MarkBordersAsAllBitsSet();
474             TryShrinkBorders();
475             return this;
476         }
477 
478         [MethodImpl(MethodImplOptions.AggressiveInlining)]
479         public BitString VectorXor(BitString other)
480         {
481             if (!Vector.IsHardwareAccelerated)
482             {
483                 return Xor(other);
484             }
485             var step = Vector<long>.Count;
486             if (_array.Length < step)
487             {
488                 return Xor(other);
489             }
490             EnsureBitStringHasTheSameSize(other, nameof(other));
491             GetCommonOuterBorders(this, other, out long from, out long to);
492             VectorXorLoop(_array, other._array, step, (int)from, (int)(to + 1));
493             MarkBordersAsAllBitsSet();
494             TryShrinkBorders();
495             return this;
496         }
497 
498         [MethodImpl(MethodImplOptions.AggressiveInlining)]
499         public BitString ParallelVectorXor(BitString other)
500         {
501             var processorCount = Environment.ProcessorCount;
502             if (processorCount <= 1 && Vector.IsHardwareAccelerated)
503             {
504                 return VectorXor(other);
505             }
506             if (!Vector.IsHardwareAccelerated)
507             {
508                 return Xor(other);
509             }
510             var step = Vector<long>.Count;
511             if (_array.Length < (step * Environment.ProcessorCount))
512             {
513                 return VectorXor(other);
514             }
515             EnsureBitStringHasTheSameSize(other, nameof(other));
516             GetCommonOuterBorders(this, other, out long from, out long to);
517             var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
518             Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorXorLoop(_array,
                ↪  other._array, step, (int)range.Item1, (int)range.Item2));
519             MarkBordersAsAllBitsSet();
520             TryShrinkBorders();
521             return this;
522         }
523 
524         [MethodImpl(MethodImplOptions.AggressiveInlining)]
525         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)
            {
                var thisVector = new Vector<long>(array, i);
                var otherVector = new Vector<long>(otherArray, i);
                (thisVector ^ otherVector).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;
        }

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

```
605
606         [MethodImpl(MethodImplOptions.AggressiveInlining)]
607         public void Set(long index, bool value)
608         {
609             if (value)
610             {
611                 Set(index);
612             }
613             else
614             {
615                 Reset(index);
616             }
617         }
618
619         [MethodImpl(MethodImplOptions.AggressiveInlining)]
620         public void Set(long index)
621         {
622             Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
623             var wordIndex = GetWordIndexFromIndex(index);
624             var mask = GetBitMaskFromIndex(index);
625             _array[wordIndex] |= mask;
626             RefreshBordersByWord(wordIndex);
627         }
628
629         [MethodImpl(MethodImplOptions.AggressiveInlining)]
630         public void Reset(long index)
631         {
632             Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
633             var wordIndex = GetWordIndexFromIndex(index);
634             var mask = GetBitMaskFromIndex(index);
635             _array[wordIndex] &= ~mask;
636             RefreshBordersByWord(wordIndex);
637         }
638
639         [MethodImpl(MethodImplOptions.AggressiveInlining)]
640         public bool Add(long index)
641         {
642             var wordIndex = GetWordIndexFromIndex(index);
643             var mask = GetBitMaskFromIndex(index);
644             if ((_array[wordIndex] & mask) == 0)
645             {
646                 _array[wordIndex] |= mask;
647                 RefreshBordersByWord(wordIndex);
648                 return true;
649             }
650             else
651             {
652                 return false;
653             }
654         }
655
656         [MethodImpl(MethodImplOptions.AggressiveInlining)]
657         public void SetAll(bool value)
658         {
659             if (value)
660             {
661                 SetAll();
662             }
663             else
664             {
665                 ResetAll();
666             }
667         }
668
669         [MethodImpl(MethodImplOptions.AggressiveInlining)]
670         public void SetAll()
671         {
672             const long fillValue = unchecked((long)0xffffffffffffffff);
673             var words = GetWordsCountFromIndex(_length);
674             for (var i = 0; i < words; i++)
675             {
676                 _array[i] = fillValue;
677             }
678             MarkBordersAsAllBitsSet();
679         }
680
681         [MethodImpl(MethodImplOptions.AggressiveInlining)]
682         public void ResetAll()
683         {
```

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

```
763         }
764
765         [MethodImpl(MethodImplOptions.AggressiveInlining)]
766         public bool HaveCommonBits(BitString other)
767         {
768             EnsureBitStringHasTheSameSize(other, nameof(other));
769             GetCommonInnerBorders(this, other, out long from, out long to);
770             var otherArray = other._array;
771             for (var i = from; i <= to; i++)
772             {
773                 var left = _array[i];
774                 var right = otherArray[i];
775                 if (left != 0 && right != 0 && (left & right) != 0)
776                 {
777                     return true;
778                 }
779             }
780             return false;
781         }
782
783         [MethodImpl(MethodImplOptions.AggressiveInlining)]
784         public long CountCommonBits(BitString other)
785         {
786             EnsureBitStringHasTheSameSize(other, nameof(other));
787             GetCommonInnerBorders(this, other, out long from, out long to);
788             var total = 0L;
789             var otherArray = other._array;
790             for (var i = from; i <= to; i++)
791             {
792                 var left = _array[i];
793                 var right = otherArray[i];
794                 var combined = left & right;
795                 if (combined != 0)
796                 {
797                     total += CountSetBitsForWord(combined);
798                 }
799             }
800             return total;
801         }
802
803         [MethodImpl(MethodImplOptions.AggressiveInlining)]
804         public List<long> GetCommonIndices(BitString other)
805         {
806             EnsureBitStringHasTheSameSize(other, nameof(other));
807             GetCommonInnerBorders(this, other, out long from, out long to);
808             var result = new List<long>();
809             var otherArray = other._array;
810             for (var i = from; i <= to; i++)
811             {
812                 var left = _array[i];
813                 var right = otherArray[i];
814                 var combined = left & right;
815                 if (combined != 0)
816                 {
817                     AppendAllSetBitIndices(result, i, combined);
818                 }
819             }
820             return result;
821         }
822
823         [MethodImpl(MethodImplOptions.AggressiveInlining)]
824         public List<ulong> GetCommonUInt64Indices(BitString other)
825         {
826             EnsureBitStringHasTheSameSize(other, nameof(other));
827             GetCommonBorders(this, other, out ulong from, out ulong to);
828             var result = new List<ulong>();
829             var otherArray = other._array;
830             for (var i = from; i <= to; i++)
831             {
832                 var left = _array[i];
833                 var right = otherArray[i];
834                 var combined = left & right;
835                 if (combined != 0)
836                 {
837                     AppendAllSetBitIndices(result, i, combined);
838                 }
839             }
840             return result;
841         }
```

```
842
843         [MethodImpl(MethodImplOptions.AggressiveInlining)]
844         public long GetFirstCommonBitIndex(BitString other)
845         {
846             EnsureBitStringHasTheSameSize(other, nameof(other));
847             GetCommonInnerBorders(this, other, out long from, out long to);
848             var otherArray = other._array;
849             for (var i = from; i <= to; i++)
850             {
851                 var left = _array[i];
852                 var right = otherArray[i];
853                 var combined = left & right;
854                 if (combined != 0)
855                 {
856                     return GetFirstSetBitForWord(i, combined);
857                 }
858             }
859             return -1;
860         }
861
862         [MethodImpl(MethodImplOptions.AggressiveInlining)]
863         public long GetLastCommonBitIndex(BitString other)
864         {
865             EnsureBitStringHasTheSameSize(other, nameof(other));
866             GetCommonInnerBorders(this, other, out long from, out long to);
867             var otherArray = other._array;
868             for (var i = to; i >= from; i--)
869             {
870                 var left = _array[i];
871                 var right = otherArray[i];
872                 var combined = left & right;
873                 if (combined != 0)
874                 {
875                     return GetLastSetBitForWord(i, combined);
876                 }
877             }
878             return -1;
879         }
880
881         [MethodImpl(MethodImplOptions.AggressiveInlining)]
882         public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
            ↪ false;
883
884         [MethodImpl(MethodImplOptions.AggressiveInlining)]
885         public bool Equals(BitString other)
886         {
887             if (_length != other._length)
888             {
889                 return false;
890             }
891             var otherArray = other._array;
892             if (_array.Length != otherArray.Length)
893             {
894                 return false;
895             }
896             if (_minPositiveWord != other._minPositiveWord)
897             {
898                 return false;
899             }
900             if (_maxPositiveWord != other._maxPositiveWord)
901             {
902                 return false;
903             }
904             GetCommonBorders(this, other, out ulong from, out ulong to);
905             for (var i = from; i <= to; i++)
906             {
907                 if (_array[i] != otherArray[i])
908                 {
909                     return false;
910                 }
911             }
912             return true;
913         }
914
915         [MethodImpl(MethodImplOptions.AggressiveInlining)]
916         private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
917         {
918             Ensure.Always.ArgumentNotNull(other, argumentName);
919             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);
            AppendAllSetIndices(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);
            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 AppendAllSetBitIndices(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)
        {
            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 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 ./Platform.Collections/BitStringExtensions.cs

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

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

namespace Platform.Collections
{
    public static class BitStringExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void SetRandomBits(this BitString @string)
        {
            for (var i = 0; i < @string.Length; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                @string.Set(i, value);
            }
```

```
        }
    }
}
```

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

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

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

namespace Platform.Collections.Concurrent
{
    public static class ConcurrentQueueExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
        {
            while (queue.TryDequeue(out T item))
            {
                yield return item;
            }
        }
    }
}
```

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

```
using System.Collections.Concurrent;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Concurrent
{
    public static class ConcurrentStackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
        ↪  value) ? value : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
        ↪  value) ? value : default;
    }
}
```

1.12 ./Platform.Collections/EnsureExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections
{
    public static class EnsureExtensions
    {
        #region Always

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName, string message)
        {
            if (argument.IsNullOrEmpty())
            {
                throw new ArgumentException(message, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
        ↪  ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
        ↪  argumentName, null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

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

1.13 ./Platform.Collections/ICollectionExtensions.cs

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

```
            return collection.All(item => equalityComparer.Equals(item, default));
        }
    }
}
```

1.14 ./Platform.Collections/IDictionaryExtensions.cs

```
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
{
    public static class IDictionaryExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
        ↪  dictionary, TKey key)
        {
            dictionary.TryGetValue(key, out TValue value);
            return value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
        ↪  TKey key, Func<TKey, TValue> valueFactory)
        {
            if (!dictionary.TryGetValue(key, out TValue value))
            {
                value = valueFactory(key);
                dictionary.Add(key, value);
                return value;
            }
            return value;
        }
    }
}
```

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

```
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 static class CharIListExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
        ↪  a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static unsafe int GenerateHashCode(this IList<char> list)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            for (var i = 0; i < list.Count; i++)
            {
                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo(this IList<char> left, IList<char> right) =>
        ↪  left.EqualTo(right, ContentEqualTo);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool ContentEqualTo(this IList<char> left, IList<char> right)
        {
            for (var i = left.Count - 1; i >= 0; --i)
            {
                if (left[i] != right[i])
                {
                    return false;
                }
            }
```

```
            return true;
        }
    }
}
```

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

```
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 IListComparer<T> : IComparer<IList<T>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
    }
}
```

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

```
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 IListEqualityComparer<T> : IEqualityComparer<IList<T>>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(IList<T> list) => list.GenerateHashCode();
    }
}
```

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

```
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.Lists
{
    public static class IListExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
        {
            list.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
        ↪  right, ContentEqualTo);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
        ↪  IList<T>, bool> contentEqualityComparer)
        {
            if (ReferenceEquals(left, right))
            {
                return true;
            }
            var leftCount = left.GetCountOrZero();
            var rightCount = right.GetCountOrZero();
            if (leftCount == 0 && rightCount == 0)
            {
                return true;
            }
            if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
            {
```

```
                return false;
            }
            return contentEqualityComparer(left, right);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
        {
            var equalityComparer = EqualityComparer<T>.Default;
            for (var i = left.Count - 1; i >= 0; --i)
            {
                if (!equalityComparer.Equals(left[i], right[i]))
                {
                    return false;
                }
            }
            return true;
        }

        [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();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] ToArray<T>(this IList<T> list)
        {
            var array = new T[list.Count];
            list.CopyTo(array, 0);
            return array;
        }

        [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]);
            }
        }

        /// <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 result = 17;
            for (var i = 0; i < list.Count; i++)
            {
                result = unchecked((result * 23) + list[i].GetHashCode());
            }
            return result;
        }

        [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);

        [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);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
        {
            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 ./Platform.Collections/Lists/ListFiller.cs

```
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.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> list)
        {
            _list.Add(list[0]);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element)
```

```
        {
            _list.Add(element);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> list)
        {
            _list.Add(list[0]);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAllValuesAndReturnConstant(IList<TElement> list)
        {
            for (int i = 1; i < list.Count; i++)
            {
                _list.Add(list[i]);
            }
            return _returnConstant;
        }
    }
}
```

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

```
using System.Linq;
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 CharSegment : Segment<char>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
        ↪  length) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode()
        {
            // Base can be not an array, but still IList<char>
            if (Base is char[] baseArray)
            {
                return baseArray.GenerateHashCode(Offset, Length);
            }
            else
            {
                return this.GenerateHashCode();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(Segment<char> other)
        {
            bool contentEqualityComparer(IList<char> left, IList<char> right)
            {
                // Base can be not an array, but still IList<char>
                if (Base is char[] baseArray && other.Base is char[] otherArray)
                {
                    return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
                }
                else
                {
                    return left.ContentEqualTo(right);
                }
            }
            return this.EqualTo(other, contentEqualityComparer);
        }

        [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 ./Platform.Collections/Segments/Segment.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
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];
            [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[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 ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs

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

namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase
    {
        public static readonly int DefaultMinimumStringSegmentLength = 2;
    }
}
```

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

```
using System.Collections.Generic;
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 AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
        where TSegment : Segment<T>
    {
        private readonly int _minimumStringSegmentLength;

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

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Collections.Segments.Walkers
8   {
9       public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
        ↪  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 ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs

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

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

```
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 ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs

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

namespace Platform.Collections.Segments.Walkers
{
    public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
    ↪ Segment<T>>
    {
    }
}
```

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

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

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

namespace Platform.Collections.Sets
```

```
{
    public static class ISetExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
        ↪  set.Remove(element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
        ↪  !set.Contains(element);
    }
}
```

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

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

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

namespace Platform.Collections.Sets
{
    public class SetFiller<TElement, TReturnConstant>
    {
        protected readonly ISet<TElement> _set;
        protected readonly TReturnConstant _returnConstant;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
        {
            _set = set;
            _returnConstant = returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SetFiller(ISet<TElement> set) : this(set, default) { }

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element)
        {
            _set.Add(element);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> list)
        {
            _set.Add(list[0]);
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element)
        {
            _set.Add(element);
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> list)
        {
            _set.Add(list[0]);
            return _returnConstant;
        }
    }
}
```

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

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

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

namespace Platform.Collections.Stacks
{
```

```
    public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
    {
        public bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Count <= 0;
        }
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public interface IStack<TElement>
    {
        bool IsEmpty
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        void Push(TElement element);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Pop();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TElement Peek();
    }
}
```

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

```
using System.Runtime.CompilerServices;

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

namespace Platform.Collections.Stacks
{
    public static class IStackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Clear<T>(this IStack<T> stack)
        {
            while (!stack.IsEmpty)
            {
                _ = stack.Pop();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪  stack.Pop();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
        ↪  stack.Peek();
    }
}
```

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

```
using Platform.Interfaces;

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

namespace Platform.Collections.Stacks
{
    public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
    {
    }
}
```

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

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

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

namespace Platform.Collections.Stacks
{
    public static class StackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
        ↪  default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
        ↪  : default;
    }
}
```

1.37 ./Platform.Collections/StringExtensions.cs

```
using System;
using System.Globalization;
using System.Runtime.CompilerServices;

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

namespace Platform.Collections
{
    public static class StringExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string CapitalizeFirstLetter(this string @string)
        {
            if (string.IsNullOrWhiteSpace(@string))
            {
                return @string;
            }
            var chars = @string.ToCharArray();
            for (var i = 0; i < chars.Length; i++)
            {
                var category = char.GetUnicodeCategory(chars[i]);
                if (category == UnicodeCategory.UppercaseLetter)
                {
                    return @string;
                }
                if (category == UnicodeCategory.LowercaseLetter)
                {
                    chars[i] = char.ToUpper(chars[i]);
                    return new string(chars);
                }
            }
            return @string;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string Truncate(this string @string, int maxLength) =>
        ↪  string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
        ↪  Math.Min(@string.Length, maxLength));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string TrimSingle(this string @string, char charToTrim)
        {
            if (!string.IsNullOrEmpty(@string))
            {
                if (@string.Length == 1)
                {
                    if (@string[0] == charToTrim)
                    {
                        return "";
                    }
                    else
                    {
                        return @string;
                    }
                }
                else
                {
                    var left = 0;
```

```
                    var right = @string.Length - 1;
                    if (@string[left] == charToTrim)
                    {
                        left++;
                    }
                    if (@string[right] == charToTrim)
                    {
                        right--;
                    }
                    return @string.Substring(left, right - left + 1);
                }
            }
            else
            {
                return @string;
            }
        }
    }
}
```

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

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

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

namespace Platform.Collections.Trees
{
    public class Node
    {
        private Dictionary<object, Node> _childNodes;

        public object Value
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        public Dictionary<object, Node> ChildNodes
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
        }

        public Node this[object key]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                var child = GetChild(key);
                if (child == null)
                {
                    child = AddChild(key);
                }
                return child;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => SetChildValue(value, key);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node(object value) => Value = value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node() : this(null) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool ContainsChild(params object[] keys) => GetChild(keys) != null;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node GetChild(params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node.ChildNodes.TryGetValue(keys[i], out node);
                if (node == null)
```

```
                {
                    return null;
                }
            }
            return node;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key) => AddChild(key, new Node(null));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, object value) => AddChild(key, new Node(value));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node AddChild(object key, Node child)
        {
            ChildNodes.Add(key, child);
            return child;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(params object[] keys) => SetChildValue(null, keys);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChild(object key) => SetChildValue(null, key);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChildValue(object value, params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node = SetChildValue(value, keys[i]);
            }
            node.Value = value;
            return node;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Node SetChildValue(object value, object key)
        {
            if (!ChildNodes.TryGetValue(key, out Node child))
            {
                child = AddChild(key, value);
            }
            child.Value = value;
            return child;
        }
    }
}
```

1.39 ./Platform.Collections.Tests/BitStringTests.cs

```
using System;
using System.Collections;
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);
                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));
            Assert.False(w.Equals(v));
            Assert.True(x.Equals(w));
            Assert.True(y.Equals(v));
            test(x, y, w, v);
            Assert.True(x.Equals(w));
        }
    }
}
```

1.40 ./Platform.Collections.Tests/CharsSegmentTests.cs

```
using Xunit;
using Platform.Collections.Segments;

namespace Platform.Collections.Tests
{
    public static class CharsSegmentTests
    {
        [Fact]
        public static void GetHashCodeEqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var first = new CharSegment(testArray, 0, 4);
```

```
            var firstHashCode = first.GetHashCode();
            var second = new CharSegment(testArray, 5, 4);
            var secondHashCode = second.GetHashCode();
            Assert.Equal(firstHashCode, secondHashCode);
        }

        [Fact]
        public static void EqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var first = new CharSegment(testArray, 0, 4);
            var second = new CharSegment(testArray, 5, 4);
            Assert.True(first.Equals(second));
        }
    }
}
```

1.41 ./Platform.Collections.Tests/StringTests.cs

```
using Xunit;

namespace Platform.Collections.Tests
{
    public static class StringTests
    {
        [Fact]
        public static void CapitalizeFirstLetterTest()
        {
            var source1 = "hello";
            var result1 = source1.CapitalizeFirstLetter();
            Assert.Equal("Hello", result1);
            var source2 = "Hello";
            var result2 = source2.CapitalizeFirstLetter();
            Assert.Equal("Hello", result2);
            var source3 = "  hello";
            var result3 = source3.CapitalizeFirstLetter();
            Assert.Equal("  Hello", result3);
        }

        [Fact]
        public static void TrimSingleTest()
        {
            var source1 = "''";
            var result1 = source1.TrimSingle('\'');
            Assert.Equal("", result1);
            var source2 = "''";
            var result2 = source2.TrimSingle('\'');
            Assert.Equal("", result2);
            var source3 = "'hello'";
            var result3 = source3.TrimSingle('\'');
            Assert.Equal("hello", result3);
            var source4 = "hello'";
            var result4 = source4.TrimSingle('\'');
            Assert.Equal("hello", result4);
            var source5 = "'hello";
            var result5 = source5.TrimSingle('\'');
            Assert.Equal("hello", result5);
        }
    }
}
```

Index