doCollections

Multiple Elements

## Template Collections

IEnumerable<T>

ICollection<T>

IDictionary<T>

Forward Traversal

Count

## Enumerators

Enumerators are read-only, forward cursors over a sequence of values. Enumerable objects represent a logical sequence of values and provide forward cursors (enumerators) over themselves. The classic consumer of an enumerable object is the foreach statement. The compiler will take the foreach statement

IEnumerable<double> enumerable = new double[] { 2.0, 4.0, 6.0 };

foreach (double val in enumerable)

Console.WriteLine(val);

And generate the following

using (IEnumerator<double> enumerator = enumerable.GetEnumerator())

{

while (enumerator.MoveNext())

{

double val = enumerator.Current;

Console.WriteLine(val);

}

}

One can instantiate and populate an enumerable object in a single step

IEnumerable<double> enumerable = new double[] { 2.0, 4.0, 6.0 };

In summary enumerators are forward only cursors that implement the iterator design pattern.

## Iterators

At iterator produces an enumerator. The compiler uses code generation to create behind the scenes classes that hold the actual implementation. The following code generates an iterator that returns the Fibonacci sequence.

public static IEnumerable<int> GetFibonacciIterator(int numEntries)

{

for ( int counter = 0, prev = 1, current = 1;

counter <= numEntries; counter++)

{

yield return prev;

int next = prev + current;

prev = current;

current = next;

}

}

The yield return statement returns control to the caller but the actual implementation of the iterator maintains the internal state so the next time the consumer asks the iterator to advance to the next element the iterator executes the correct code. If you want the code to return early without returning any more elements you need to use the yield break statement. Note you cannot use the normal return from an iterator generation method.

## Arrays

### Dimensions, Rank and Length

Consider the following piece of code.

byte[,] a = new byte[3, 2]

{

{0, 1},

{2, 3},

{4, 5}

};

byte[,] b = new byte[2, 3]

{

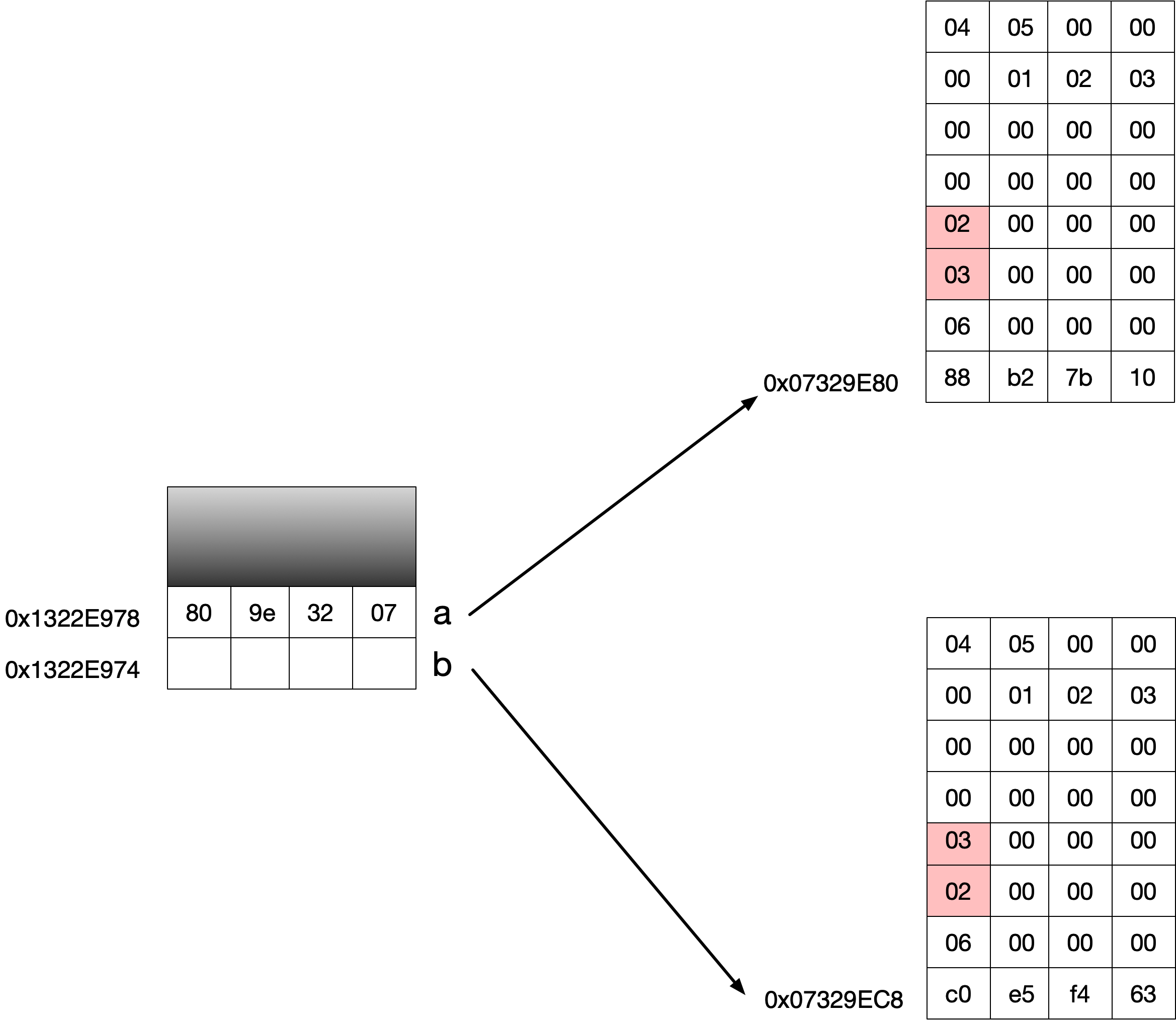
{0, 1,2},

{3, 4,5}

};

After executing this code our memory looks as follows. Notice that a is a array of 3 rows and two columns and b is an array of two rows and 3 columns. The only difference in the memory layout is the order on the dimension sizes which I have shown in red.

Figure 1Array Dimensions



To highlight the difference we can use the following index code.

Console.WriteLine(a[1,0]);

Console.WriteLine(b[1,0]);

The output of this code is then

2  
3

### Indices and ranges

|  |  |  |
| --- | --- | --- |
| Name | Code | Result |
| Last element | WriteLine(new [] {1,2,3}[^1]); | 3 |
| Second last | WriteLine(new [] {1,2,3}[^2]); | 2 |
| First two elements | WriteLine(new [] {1,2,3}[..2]); | [1,2] |
| Slice from index 1 to index 2 (second index is exclusive) | WriteLine(new [] {1,2,3}[1..3]); | [2,3] |
| Last two elements | WriteLine(new [] {1,2,3}[^2..]); | [2,3] |

We can add support for indices and ranges to our own type by adding indexers with the Range and Index types.

### Span<T>,

Defines an interface to a contiguous area of memory as a pointer and a length. When used together with stackalloc they can wrap stack allocated memory. For this reason, Stack<T> is a ref struct meaning it can never be used in places where it would end up on the managed heap.

Ref Struct

Although a struct assigned to a local variable or formal parameter lives on the stack, if a struct value is assigned to a class field it will reside on the heap. Marking a struct definition with the ref keyword tells the compiler this struct can never live on the managed heap. As such the compiler can raise an error if we try to use the struct in a place it would be stored on the heap.

By using this abstraction to access and manipulate stack allocated memory we can reduce heap allocations and garbage collections.

Spans can be sliced, enabling us to work with subsections without the need to make copies. By avoiding the manager heap we can reduce garbage collections.

void Main()

{

Span<int> numbers = stackalloc int[5];

for (int i = 0; i < numbers.Length; i++) numbers[i] = i;

DoubleSpan(numbers);

// 0,2,4,6,7,10

numbers.Dump();

}

public void DoubleSpan(Span<int> s)

{

for (int i = 0; i < s.Length; i++)

s[i] \*= 2;

}

#### Indices and Ranges

Spans support indices and ranges.

#### Uses

Reduce heap allocations and garbage collections

Enable efficient Json parsing in .NET Core.

We can write code that works with manged and unmanaged memory

We can work with stack allocated memory with minimum of pointers and without unsafe.

Efficient I/O networking in ASP.NET Core pipeline.

### ReadonlySpan<T>

Prevent unintentional change and allows the compiler to make low level optimisations.

### Memory<T>

Memory<T> removes such of the restrictions on Span but as such it cannot be used to access stack allocated memory.

## List<T> (Array Backed)

## LinkedList<T>

## Queue<T>

## Stack<T>

## BitArray

## HashSet<T>

## SortedSet<T>

## Dictionaries

## Customizable Collections

### Collection<T>

Collection<T> provides a wrapper around an IList<T> allowing a subclass to intercept insertions and removals.

CollectionBase is the old non-generic version which is largely superceded.

Listing Customizable List

public class SpecialIntList : Collection<int>

{

public SpecialIntList(IList<int> del) : base(del) {}

protected override void InsertItem(int index, int item)

{

// Specific logic

base.InsertItem(index, item);

}

protected override void RemoveItem(int index)

{

// Pug in any extra logic then do the base

base.RemoveItem(index);

}

}

### KeyedCollection<T>

Questions

Enumerators

1. What are enumerators?

Read-only forward cursors over a sequence of values

**What is an enumerable object?**

Logical representation of a sequence that provides forward cursors over itself**What will the**

**compiler do when it sees the following code?**

IEnumerable<double> enumerable = new double[] { 2.0, 4.0, 6.0 };

foreach (double val in enumerable)

Console.WriteLine(val);

Translate it into this

using (IEnumerator<double> enumerator = enumerable.GetEnumerator())

{

while (enumerator.MoveNext())

{

double val = enumerator.Current;

Console.WriteLine(val);

}

}

**What criteria must be met to support instantiating and population in a single step?**

* It must implement IEnumerable
* It must have an add method accepting the appropriate type

**Why is it designed this way?**

Implementing IEnumerable shows that the object is a collection. If it just had an add method this could be meant for something else e.g. arithmetic sub.

**What does the compiler generate when it sees this?**

IEnumerable<double> enumerable = new double[] { 2.0, 4.0, 6.0 };

It generates this

List<double> enumerable = new List<double>();

enumerable.Add(2.0);

enumerable.Add(4.0);

enumerable.Add(6.0);

**What design pattern does IEnumerator implement?**

Iterator

**Since IEnumerable<T> extends the non-generic IEnumerable what is the standard pattern for implementing IEnumerable<T>**

Publicly expose IEnumerable<T> and explicitly implement non-generic IEnumerable

**Why is it done like this?**

So by default we get back a type safe enumerator.

**What do enumerators support?**

1. Collection Initializers
2. Foreach statements

Iterators

**What is an iterator method?**

A producer of an iterator

**What does the compiler do with an iterator method?**

Generates private classes that implement IEnumerable<T> and IEnumerator<T>

**What does the yield return statement do?**

Control returns to the caller but calee state is maintained so the method can continue executing when the code next calls the same method.

**What does the yield break statement do?**

Indicate that the method should return early without returning any ore elements

**What does the normal return statement do in an iterator method?**

Not allowed