Indexers, Ranges, and Indices

Filip Ekberg
Principal Consultant & CEO

@fekberg | fekberg.com

Using an Indexer

```
Order[] orders = new Order[10];

Order first = orders[0];
```

Using an Indexer

```
Order[] orders = new Order[10];

Order first = orders[0];
```

Using an Indexer

```
Order[] orders = new Order[10];

Order first = orders[0];
```

Using an Indexer

```
Order[] orders = new Order[10];

Order first = orders[0];
```

Indexer

Indexers are **useful** when you create **list like data structures**

Creating an Indexer

Creating an Indexer

```
public class OrderList
{
    private readonly Order[] orders;



    public OrderList(IEnumerable<Order> orders)
    {
        this.orders = orders.ToArray();
    }
}
```

Creating an Indexer

```
public class OrderList
{
    private readonly Order[] orders;

    public Order this[int index]

    public OrderList(IEnumerable<Order> orders)
    {
        this.orders = orders.ToArray();
    }
}
```

Creating an Indexer

```
public class OrderList
{
    private readonly Order[] orders;

    public Order this[int index]

    public OrderList(IEnumerable<Order> orders)
    {
        this.orders = orders.ToArray();
    }
}
```

Creating an Indexer

```
public class OrderList
{
    private readonly Order[] orders;

    public Order this[int index] => orders[index];

    public OrderList(IEnumerable<Order> orders)
    {
        this.orders = orders.ToArray();
    }
}
```

Creating an Indexer

```
public class OrderList
{
    private readonly Order[] orders;

    public Order this[int index] => orders[index];

    public OrderList(IEnumerable<Order> orders)
    {
        this.orders = orders.ToArray();
    }
}
```

Creating an Indexer

```
public class OrderList
{
    private readonly Order[] orders;

    public Order this[int index] => orders[index];

    public OrderList(IEnumerable<Order> orders)
    {
        this.orders = orders.ToArray();
    }
}
```

Using the Indexer

```
public class OrderList
{
    public Order this[int index] => orders[index];

    ...
}

var list = new OrderList(orders);
var order = list[0];
```

Using the Indexer

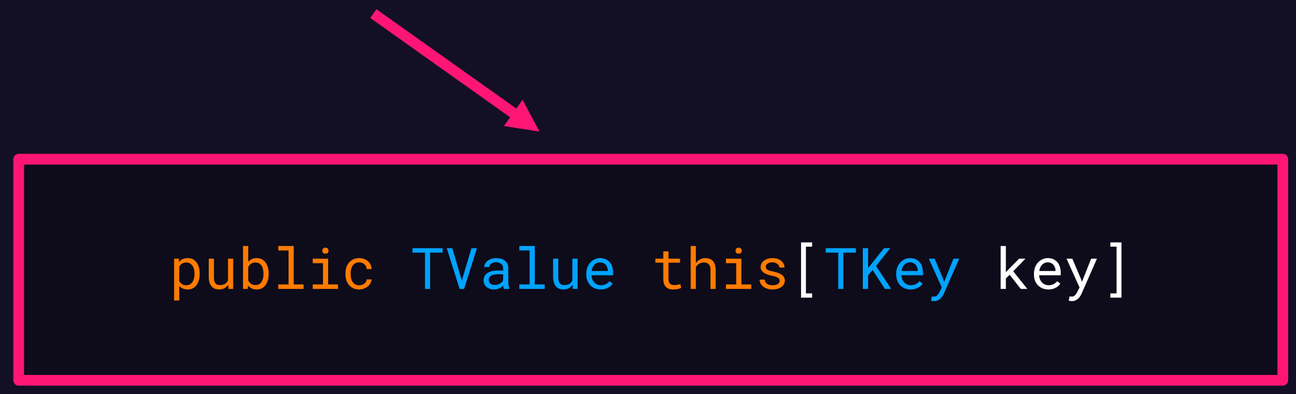
```
public class OrderList
{
    public Order this[int index] => orders[index];

    ...
}

var list = new OrderList(orders);
var order = list[0];
```

Indexers in .NET

```
var dictionary = new Dictionary<Guid, Order>();
```

Indexers in .NET

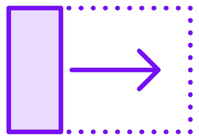
```
var dictionary = new Dictionary<Guid, Order>();

Order order = dictionary[orderNumber];
```



How Would You Solve This?

Only select the first 10 orders

Skip the first order and get the rest of the array



Start from the middle of the array and get the rest of the orders

You can slice an array using a range

Ranges in C#

Used with indices

Specify where to begin and where to end

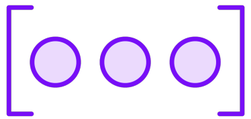
Define a Range

```
Index start    = 0;
Index end      = 10;
Range range    = start..end;
```

Define a Range

```
Index start    = 0;
Index end      = 10;
Range range    = start..end;
```

Defining a Range



All of it

```
Range range    = ..;
```

Start the range from

```
Range range    = 10..;
```

End the range at

```
Range range    = ..2;
```

System.Index can be **implicitly** converted to an **integer**

The Hat Operator

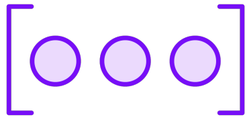
```
var lastElement = orders[^1];
```

The Hat Operator

```
var lastElement = orders[^1];
```

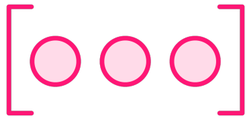
orders.Length - 1

Range Can Only Be Used With



Arrays
You have to call ToArray() on your List or IEnumerable

Span<T>



Memory<T>

Think about the **extra allocations** that are required

Ranges Work with All Arrays

```
var end = orders.Length / 2;

var subset = orders[..end];
```

Substring Using a Range

Substring Using a Range

```
var name = "Filip Ekberg";

var substring = name[..5];
```

The **range** syntax together with the **index** provides a **simple way of working** with **arrays**

How do you **avoid unnecessary allocations**?

Span<T>

“Provides a type- and memory-safe representation of a contiguous region of arbitrary memory.

Span<T> is a ref struct that is allocated on the stack rather than on the managed heap. Ref struct types have a number of restrictions to ensure that they cannot be promoted to the managed heap, including that they can't be boxed, they can't be assigned to variables of type Object, dynamic or to any interface type, they can't be fields in a reference type, and they can't be used across await and yield boundaries.”

docs.microsoft.com/en-us/dotnet/api/system.span-1

Implicit Conversion to Span<T>

```
Span<Order> orders = new Order[100];
```

Efficient implicit conversion!

Doesn’t copy the array and doesn’t consume much power

Creating a Span over an Array

```
Span<Order> ordersSpan = new Span<Order>(ordersArray);
```

Span<T> Internals

Only references the original array

Does not make any copies of elements or require any heavy computation

Span<T> Only References the Array

Span<T> Only References the Array

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };
```

Span<T> Only References the Array

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = arrayOfNumbers;
```

Span<T> Only References the Array

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = arrayOfNumbers;

spanOfNumbers[0] = 1;
```

Span<T> Only References the Array

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = arrayOfNumbers;

spanOfNumbers[0] = 1;
```

```
spanOfNumbers:      { 1, 1, 2, 3, 4, 5, 6 }
arrayOfNumbers:     { 1, 1, 2, 3, 4, 5, 6 }
```

A span can be used to work with unmanaged memory

Span<T> with IEnumerable and Lists

```
List<int> listOfNumbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6 };
```

Span<T> with IEnumerable and Lists

```
List<int> listOfNumbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = listOfNumbers.ToArray();
```

Span<T> with IEnumerable and Lists

```
List<int> listOfNumbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = listOfNumbers.ToArray();
```

Span<T> with IEnumerable and Lists

```
List<int> listOfNumbers = new List<int>() { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = listOfNumbers.ToArray();
```

WARNING: This will cause extra allocations!

Span<T> Is a Ref Struct

“allocated on the stack rather than on the managed heap.

Ref struct types have a number of restrictions to ensure that they cannot be promoted to the managed heap, including that they can't be boxed, they can't be assigned to variables of type Object, dynamic or to any interface type, they can't be fields in a reference type, and they can't be used across await and yield boundaries.”

docs.microsoft.com/en-us/dotnet/api/system.span-1

Spans can be **sliced** in a **very memory efficient** manner!

Slicing a Span

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = arrayOfNumbers;
```

Slicing a Span

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = arrayOfNumbers;

Span<int> slice = spanOfNumbers[^5..];

Span<int> anotherSlice = spanOfNumbers[..^5];
```

Slicing a Span

```
var arrayOfNumbers = new [] { 0, 1, 2, 3, 4, 5, 6 };

Span<int> spanOfNumbers = arrayOfNumbers;

Span<int> slice = spanOfNumbers[^5..];

Span<int> anotherSlice = spanOfNumbers[..^5];
```

Creates a representation that **points to** the **elements**!

No extra allocations than the variable necessary!

Request multiple different slices without any extra allocations

Example: Slicing a Byte Array

Example: Slicing a Byte Array

```
void Process(Span<byte> payload)
{



}
```

Example: Slicing a Byte Array

```
void Process(Span<byte> payload)
{
    var header    = payload[..10];

    var data      = payload[10..^128];

    var signature = payload[^128..];
}
```

You can use a span as a parameter to a method

It cannot be async or use yield!

Array Is Implicitly Converted to the Span

```
void Process(Span<byte> payload)
{
    var header    = payload[..10];

    var data      = payload[10..^128];

    var signature = payload[^128..];
}
```

Array Is Implicitly Converted to the Span

```
void Process(Span<byte> payload)
{
    var header    = payload[..10];

    var data      = payload[10..^128];

    var signature = payload[^128..];
}

var payload = new byte[1024];
```

Array Is Implicitly Converted to the Span

```
void Process(Span<byte> payload)
{
    var header    = payload[..10];

    var data      = payload[10..^128];

    var signature = payload[^128..];
}

var payload = new byte[1024];

Process(payload);
```

Array Is Implicitly Converted to the Span

```
void Process(Span<byte> payload)
{
    var header    = payload[..10];

    var data      = payload[10..^128];

    var signature = payload[^128..];
}

var payload = new byte[1024];

Process(payload);
```

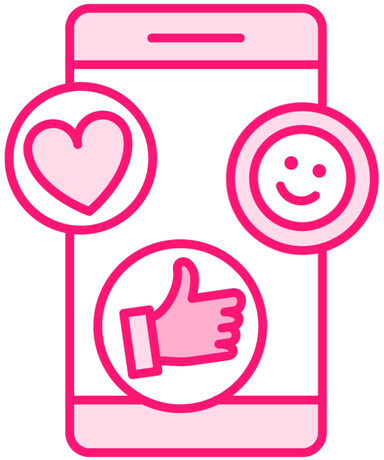
Don’t need to do anything else, it is implicitly converted to a **Span<byte>**

Array Is Implicitly Converted to the Span

```
void Process(Span<byte> payload)
{
    var header    = payload[..10];

    var data      = payload[10..^128];

    var signature = payload[^128..];
}

var payload = new byte[1024];

Process(payload);
```

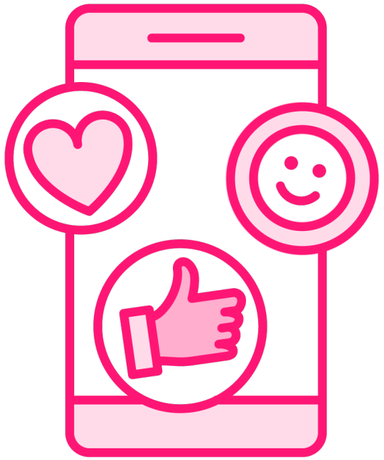
Example: Slicing Bytes

Example: Slicing Bytes



Business Monitoring

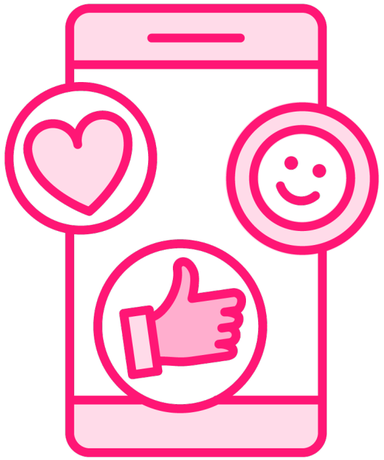
Example: Slicing Bytes



Business Monitoring

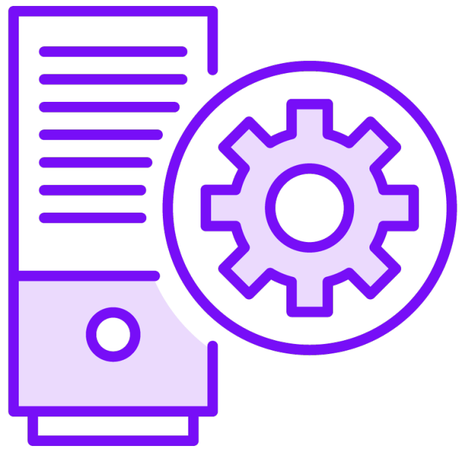
```
Byte[] payload
```

Example: Slicing Bytes



Business Monitoring

```
Byte[] payload
```



Warehouse Management System

Let’s use Span<T> and ranges to get the last 128 bytes!

What happens when you modify the span?

Use the **read-only span**!

You have now promised that the array is not tampered with!

Containing reference types?
Its properties are not made read-only!

Memory<T>

Memory<T>

```
public class PayloadValidator
{
     private ReadOnlyMemory<byte> payload;









}
```

Memory<T>

```
public class PayloadValidator
{
    private ReadOnlyMemory<byte> payload;
    public PayloadValidator(ReadOnlyMemory<byte> payload)
    {
        this.payload = payload;
    }

    public bool Validate()
    {
        var slice = payload[^128..];

        foreach (var item in slice.Span) { ... }

        return true;
    }
}
```

The **assumption** when **accessing** an **index** is that **it is** nearly **instant**!

Different Ranges

```
Span<byte> payload = new byte[1024];

var smallSlice = payload[1..10];
```

Different Ranges

```
Span<byte> payload = new byte[1024];

var smallSlice = payload[1..10];

var everything = payload[..];
```

Different Ranges

```
Span<byte> payload = new byte[1024];

var smallSlice = payload[1..10];

var everything = payload[..];

var signature  = payload[^128..];
```

Different Ranges

```
Span<byte> payload = new byte[1024];

var smallSlice = payload[1..10];

var everything = payload[..];

var signature  = payload[^128..];
```

This is very memory efficient!

Requires **no extra allocations**

Use **ReadOnlySpan** where you want to promise you won’t change the array

Example: ReadOnlySpan<char>

Example: ReadOnlySpan<char>

```
ReadOnlySpan<char> name   = "Filip Ekberg";
```

Example: ReadOnlySpan<char>

```
ReadOnlySpan<char> name   = "Filip Ekberg";
```

Don’t have to do anything else, it is implicitly converted to a **ReadOnlySpan<char>**

Example: ReadOnlySpan<char>

```
ReadOnlySpan<char> name  = "Filip Ekberg";

ReadOnlySpan<char> first = name[..5];
ReadOnlySpan<char> last  = name[6..];
```