IAP C# 2011 Lecture 2: Delegates, Lambdas, LINQ

Geza Kovacs

Two ways of thinking about operations on collections

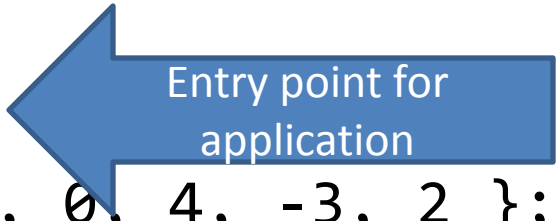
- Task: I have an array of integers. I want a new array containing just the positive integers
- **Imperative style: Use a loop**
- With a Language Integrated Query (LINQ): Define a query (a request for information) that'll request the positive integers, and execute it

```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```

```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```



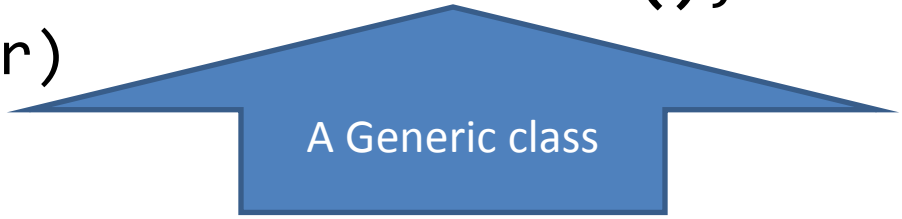
```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```

Array initialization

```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```



```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```

A using statement, saves us the need to type out **System.Collections.Generic.LinkedList**

A Generic class

```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```

Type is inferred

```
using System;
using System.Collections.Generic;

static class MyMainClass
{
    static void Main(string[] args) {
        int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
        var positiveList = new LinkedList<int>();
        foreach (int x in arr)
        {
            if (x > 0)
                positiveList.AddLast(x);
        }
        int[] positiveArr = positiveList.ToArray();
        // contains { 1, 4, 2 }
    }
}
```

Instance method (defined in the LinkedList class)

Two ways of thinking about operations on collections

- Task: I have an array of integers. I want a new array containing just the positive integers
- Imperative style: Use a loop
- With a **Language Integrated Query (LINQ)**: Define a query (a request for information) that'll request the positive integers, and execute it

LINQ

- Compact way to define a query
- To understand how it's implemented, we'll need to know about **delegates** and **lambdas**

```
int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
var query =
    from x in arr
    where x > 0
    select x;
int[] positiveArr = query.ToArray();
// contains { 1, 4, 2 }
```

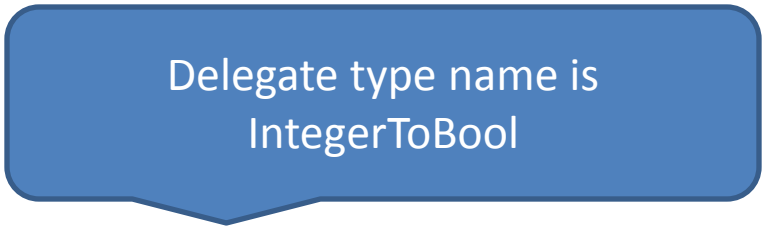
Delegates

- A type that can reference a method
- Here, we declare a delegate type IntegerToBool which can reference a method that has one int argument and returns a bool

```
delegate bool IntegerToBool(int x);
```

Delegates

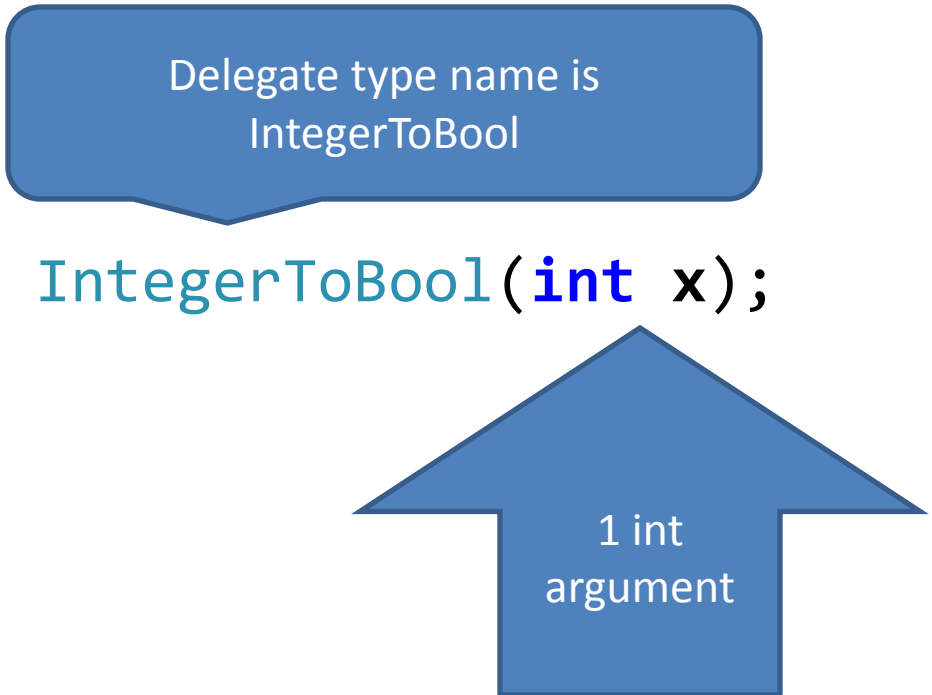
- A type that can reference a method
- Here, we declare a delegate type IntegerToBool which can reference a method that has one int argument and returns a bool




```
delegate bool IntegerToBool(int x);
```

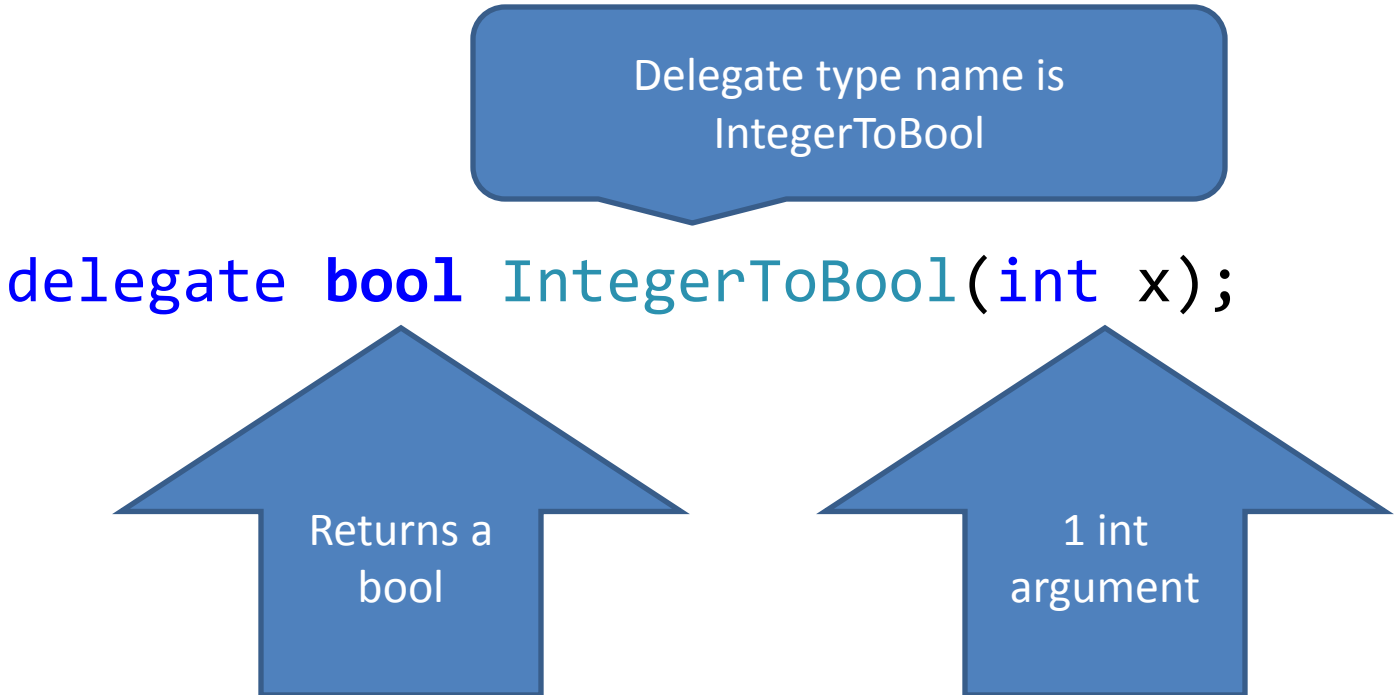
Delegates

- A type that can reference a method
- Here, we declare a delegate type IntegerToBool which can reference a method that has one int argument and returns a bool



Delegates

- A type that can reference a method
- Here, we declare a delegate type IntegerToBool which can reference a method that has one int argument and returns a bool

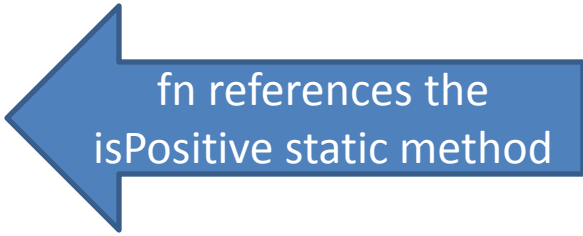


- A delegate can reference a static method

```
delegate bool IntegerToBool(int x);

static class MyMainClass
{
    static bool IsPositive(int v) {
        return v > 0;
    }

    static void Main(string[] args) {
        IntegerToBool fn = IsPositive;
        bool isFivePositive = fn(5);
    }
}
```

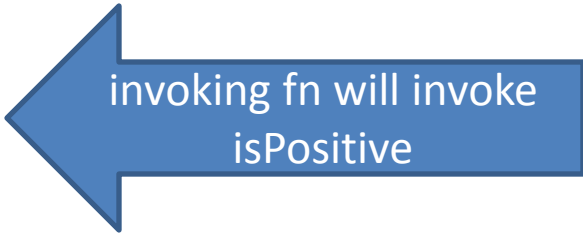


- A delegate can reference a static method

```
delegate bool IntegerToBool(int x);

static class MyMainClass
{
    static bool IsPositive(int v) {
        return v > 0;
    }

    static void Main(string[] args) {
        IntegerToBool fn = IsPositive;
        bool isFivePositive = fn(5);
    }
}
```

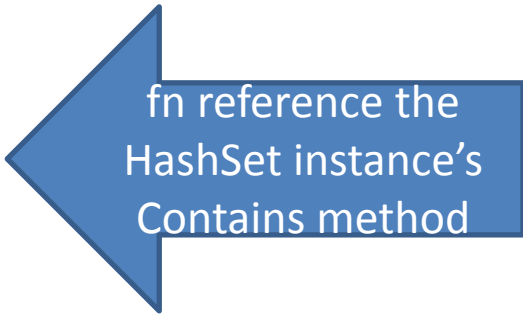


- A delegate can also reference an instance method

```
using System.Collections.Generic;

delegate bool IntegerToBool(int x);

static class MyMainClass
{
    static void Main(string[] args) {
        var set = new HashSet<int>();
        set.Add(5);
        IntegerToBool fn = set.Contains;
        bool setContainsFive = fn(5);
        bool setContainsThree = fn(3);
    }
}
```

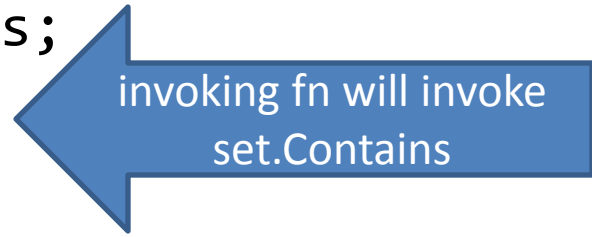


- A delegate can also reference an instance method

```
using System.Collections.Generic;

delegate bool IntegerToBool(int x);

static class MyMainClass
{
    static void Main(string[] args) {
        var set = new HashSet<int>();
        set.Add(5);
        IntegerToBool fn = set.Contains;
        bool setContainsFive = fn(5);
        bool setContainsThree = fn(3);
    }
}
```



- Methods can accept delegate types as arguments

filterIntegers is a method that takes an IntegerToBool delegate as an argument

```
delegate bool IntegerToBool(int x);

static int[] filterIntegers(int[] orig, IntegerToBool fn) {
    var list = new LinkedList<int>();
    foreach (int x in orig) {
        if (fn(x))
            list.AddLast(x);
    }
    return list.ToArray();
}
```

- Can pass any method matching the delegate's signature (same return value and arguments) to a function that has a delegate as an argument

```
static class MyMainClass
{
    delegate bool IntegerToBool(int x);

    static int[] filterIntegers(int[] orig, IntegerToBool fn){
        …
    }

    static bool IsPositive(int v) { return v > 0; }

    static void Main(string[] args)
    {
        int[] orig = new int[] { 1, -1, 0, 4, -3, 2};
        int[] arr = filterIntegers(orig, IsPositive);
    }
}
```

IsPositive matches IntegerToBool's signature, so it can be passed without explicit conversion

- Observe the definition of our IntegerToBool delegate type:

```
delegate bool IntegerToBool(int x);
```

- It can reference only methods with 1 int argument, and a bool return value.

- Observe the definition of our IntegerToBool delegate type:

```
delegate bool IntegerToBool(int x);
```

- It can reference only methods with 1 int argument, and a bool return value.
- If we want to reference a method with an int return value, we'd need to declare another type:

```
delegate int IntegerToInteger(int x);
```

- Observe the definition of our IntegerToBool delegate type:

```
delegate bool IntegerToBool(int x);
```

- It can reference only methods with 1 int argument, and a bool return value.
- If we want to reference a method with an int return value, we'd need to declare another type:

```
delegate int IntegerToInteger(int x);
```

- If we want to reference a method with a string argument, we'd need to declare another type:

```
delegate bool StringToBool(string x);
```

The System.Func delegate

- A generic delegate type defined as:

Used for referencing methods with 1 argument and 1 return value

```
delegate TResult Func<T1, TResult>(T arg1);
```

The System.Func delegate

- A generic delegate type defined as:

Used for referencing methods with 1 argument and 1 return value

```
delegate TResult Func<T1, TResult>(T arg1);
```

Used for referencing methods with 2 arguments and 1 return value

```
delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2);
```

The System.Func delegate

- A generic delegate type defined as:

Used for referencing methods with 1 argument and 1 return value

```
delegate TResult Func<T1, TResult>(T arg1);
```

Used for referencing methods with 2 arguments and 1 return value

```
delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2);
```

Used for referencing methods with 3 arguments and 1 return value

```
delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3);
```

… (etc)

- We can use **System.Func<int, bool>** instead of defining our own IntegerToBool delegate type:

Func<int, bool>: references method with one int argument which returns bool; matches IsPositive's signature

```
using System;

static class MyMainClass
{
    static int[] filterIntegers(int[] orig, Func<int, bool> fn)
    {
        …
    }

    static bool IsPositive(int v) { return v > 0; }

    static void Main(string[] args)
    {
        int[] orig = new int[] { 1, -1, 0, 4, -3, 2};
        int[] arr = filterIntegers(orig, IsPositive);
    }
}
```

Lambdas

- Notice that in our previous example, we had an IsPositive method which did very little:

```
static bool IsPositive(int v) { return v > 0; }
```

- Lambdas are a shorthand for declaring short methods, which allow them to be declared in a single expression:

```
Func<int, bool> IsPositive = (int x) => { return x > 0; };
```

```
Func<int, bool> IsPositive = (int v) => { return v > 0; };
```

- Argument types can be excluded:

```
Func<int, bool> IsPositive = (v) => { return v > 0; };
```

- If it's just a single statement, you can also omit the return statement and braces

```
Func<int, bool> IsPositive = (v) => v > 0;
```

- If the lambda has just a single argument, the parentheses can be omitted:

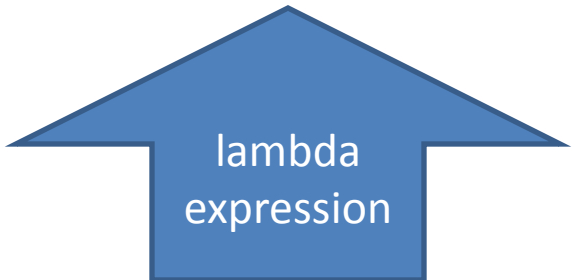
```
Func<int, bool> IsPositive = v => v > 0;
```

- Because lambdas are expressions, we can declare lambdas directly in function invocations

```
using System;

static class MyMainClass
{
    static int[] filterIntegers(int[] orig, Func<int, bool> fn)
    {
        …
    }

    static void Main(string[] args)
    {
        int[] orig = new int[] { 1, -1, 0, 4, -3, 2};
        int[] arr = filterIntegers(orig, v => v > 0);
    }
}
```



Returning to LINQ

- LINQ defines an method called **Where**, which is similar to our filterIntegers example method:
  - Takes a collection of values and a Func that outputs a bool for each element; returns those elements for which the Func returns true

LINQ: Where

- LINQ's Where method, however, works with more general collections (as opposed to filterIntegers, which works only with int[])

```
static int[] filterIntegers(int[] orig, Func<int, bool> fn);


static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
);
```

LINQ: Where

- **IEnumerable**: An interface implemented by collections (including LinkedList, HashSet, arrays, etc)

```
static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
);
```

LINQ: Where

- **TSource**: a generic type parameter (so it'll work with collections of int, string, custom classes, etc)

```
static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
);
```

LINQ: Where

- **this**: It's an extension method for IEnumerable

```
static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
);
```

LINQ: Where

```
using System.Collections.Generic;
using System.Linq;

static void Main(string[] args)
{
    int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
    IEnumerable<int> query =
        arr.Where(v => v > 0);
    int[] positiveArr = arr.ToArray();
    // contains { 1, 4, 2 }
}
```

LINQ: Where

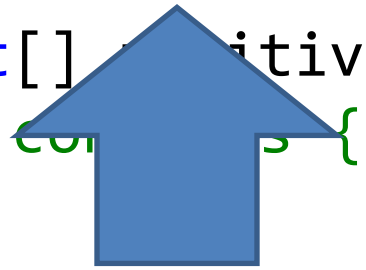
```
using System.Collections.Generic;
using System.Linq;

static void Main(string[] args)
{
    int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
    IEnumerable<int> query =
        arr.Where(v => v > 0);
    int[] positiveArr = arr.ToArray();
    // contains { 1, 4, 2 }
}
```

LINQ: Where

```
using System.Collections.Generic;
using System.Linq;

static void Main(string[] args)
{
    int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
    IEnumerable<int> query =
        arr.Where(v => v > 0);
    int[] positiveArr = arr.ToArray();
    // contains { 1, 4, 2 }
}
```



- Extension methods (like Where, for IEnumerable) can be invoked with same syntax as instance methods

LINQ: Where

```
using System.Collections.Generic;
using System.Linq;

static void Main(string[] args)
{
    int[] arr = new int[] { 1, -1, 0, 4, -3, 2 };
    IEnumerable<int> query =
        arr.Where(v => v > 0);
    int[] positiveArr = arr.ToArray();
    // contains { 1, 4, 2 }
}
```

Need this, otherwise Where and ToArray won't be defined

- **ToArray**, like **Where**, is also an extension method for IEnumerable. Both are defined in **System.Linq**

LINQ: Select

- Suppose you have an array of product objects, each of which have a name and price.

```
class Product
{
    public string name;
    public int price;
}
```

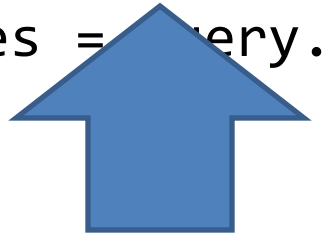
- You want an array of strings, with all the product names. We can use **Select** for this

LINQ: Select

```
Product[] products = new Product[] {…};
IEnumerable<string> query =
    products.Select(v => v.name);
string[] productNames = query.ToArray();
```

LINQ: Select

```
Product[] products = new Product[] {…};
IEnumerable<string> query =
    products.Select(v => v.name);
string[] productNames = query.ToArray();
```



- Argument to Select is a selector function which take one element (a Product instance), and returns something else (which may have a different type, as in this example)

Alternative LINQ Syntax

```
Product[] products = new Product[] {…};
IEnumerable<string> query =
    products.Select(v => v.name);
string[] productNames = query.ToArray();
```


```
Product[] products = new Product[] {…};
IEnumerable<string> query =
    from v in products
    select v.name;
string[] productNames = query.ToArray();
```

Using Where and Select Together

- LINQ allows query operators like Where and Select to be combined
- For example, what are the names of all products with price less than 4?

```
Product[] products = new Product[] {…};
IEnumerable<string> query =
    products.Where(v => v.price < 4)
            .Select(v => v.name);
string[] productNames = query.ToArray();
```

Using Where and Select Together

- LINQ allows query operators like Where and Select to be combined
- For example, what are the names of all products with price less than 4?

```
Product[] products = new Product[] {…};
IEnumerable<string> query =
    from v in products
    where v.price < 4
    select v.name;
string[] productNames = query.ToArray();
```

Aggregate

- Known in other languages as “reduce” or “fold”
- Begins with a seed value (ie first element in the sequence), then applies a function from left to right in the sequence, keeping some running value.
- Ex: Finding a sum: keep a running total of the sum so far, initialize it to the leftmost element in the sequence, and for each new element, add it to the running total

Aggregate – Implementing Sum

- Ex: Finding a sum: keep a running total of the sum so far, initialize it to the leftmost element in the sequence, and for each new element, add it to the running total

```
static class MyMainClass
{
    static void Main(string[] args)
    {
        double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };
        double sum =
            doubles.Aggregate((runningSum, nextItem) =>
                runningSum + nextItem);
        // 12.9
    }
}
```

Aggregate – Implementing Product

- Ex: Finding a product: keep a running product, initialize it to the leftmost element in the sequence, and for each new element, multiply the running product by it

```
static class MyMainClass
{
    static void Main(string[] args)
    {
        double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };
        double product =
            doubles.Aggregate((runningProduct, nextItem) =>
                runningProduct * nextItem);
        // 88.33081
    }
}
```

More Operators

- Max, Min, Reverse, OrderBy, etc
- See http://msdn.microsoft.com/en-us/vcsharp/aa336746 for examples