VIA University College

DNP-A18

Advanced C#

An overview of advanced C# features

● **Customizing functionality**

...with extension methods and operator overloading

● **Generics**

Maximize code reuse, type safety and performance.

● **Lambda Expressions**

Less is more!

● **Delegates**

Turn methods into variables!

● **Exercises**

Get familiar with advanced concepts in C#

Extension Methods

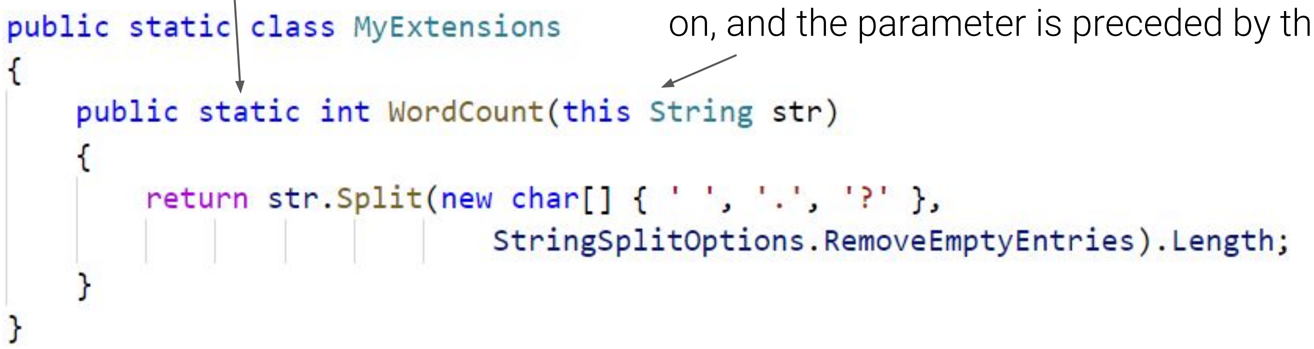
Customizing functionality

Allow us to add methods to an existing class without changing its source code or creating a new class that inherits from it

Can’t extend the String class because it is sealed? No problem!

Extension methods are defined as static methods but are called by using instance method syntax.

Their first parameter specifies which type the method operates on, and the parameter is preceded by the this modifier.



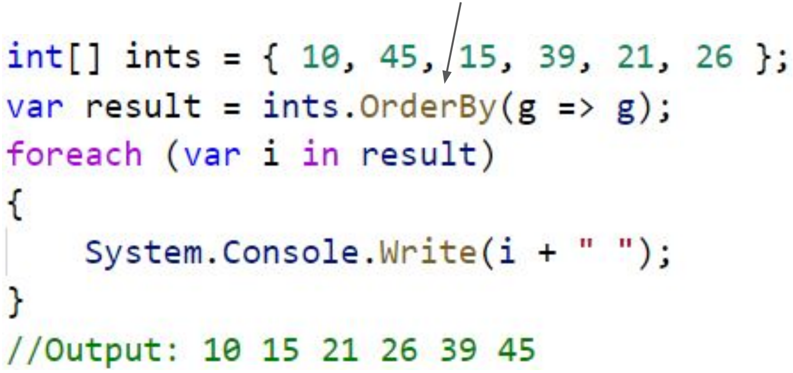
```
public static class MyExtensions
{
    public static int WordCount(this String str)
    {
        return str.Split(new char[] { ' ', '.', '?' },
                            StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Extension Methods

Customizing functionality

Extension methods are used more than created
(e.g. LINQ, adding query functionality to the existing System.Collections.IEnumerable)

Extension method

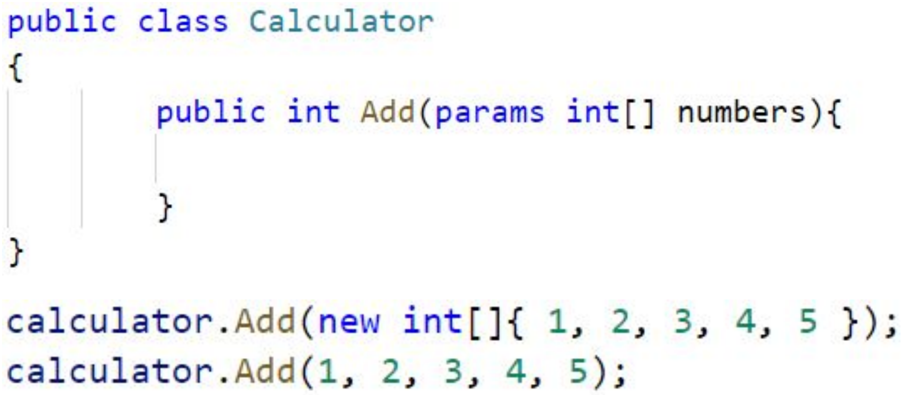


```
int[] ints = { 10, 45, 15, 39, 21, 26 };
var result = ints.OrderBy(g => g);
foreach (var i in result)
{
    System.Console.Write(i + " ");
}
//Output: 10 15 21 26 39 45
```

Method Parameters

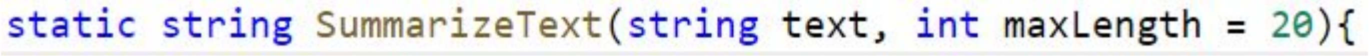
Customizing functionality

Params Modifier



```
public class Calculator
{
        public int Add(params int[] numbers){

        }
}

calculator.Add(new int[]{ 1, 2, 3, 4, 5 });
calculator.Add(1, 2, 3, 4, 5);
```

Default Parameter



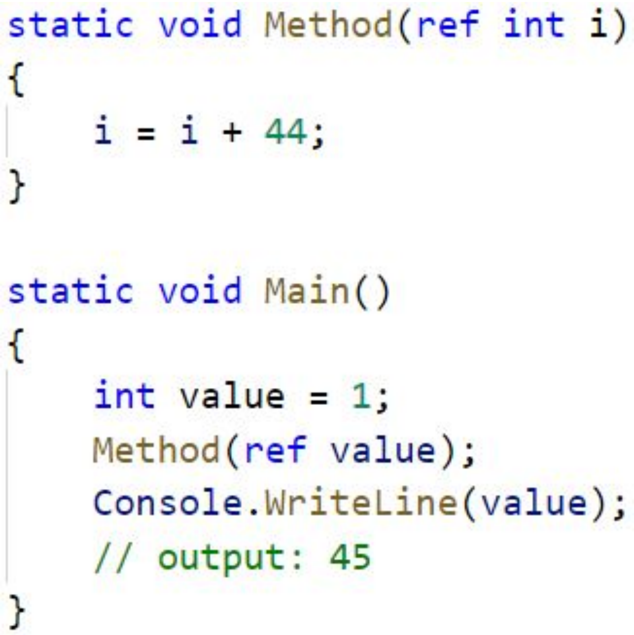
```
static string SummarizeText(string text, int maxLength = 20){
```

Makes second argument optional

Params Modifier

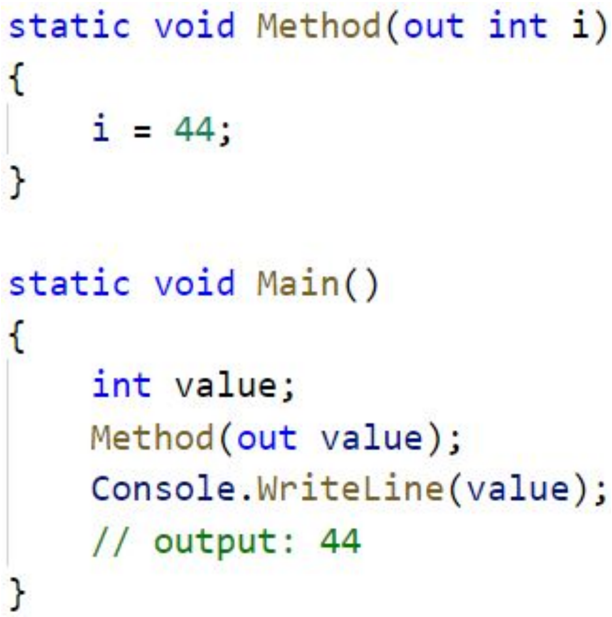
Method Parameters

Ref Modifier



```
static void Method(ref int i)
{
    i = i + 44;
}

static void Main()
{
    int value = 1;
    Method(ref value);
    Console.WriteLine(value);
    // output: 45
}
```

Out Modifier



```
static void Method(out int i)
{
    i = 44;
}

static void Main()
{
    int value;
    Method(out value);
    Console.WriteLine(value);
    // output: 44
}
```

[Ref Modifier]

[Out Modifier]

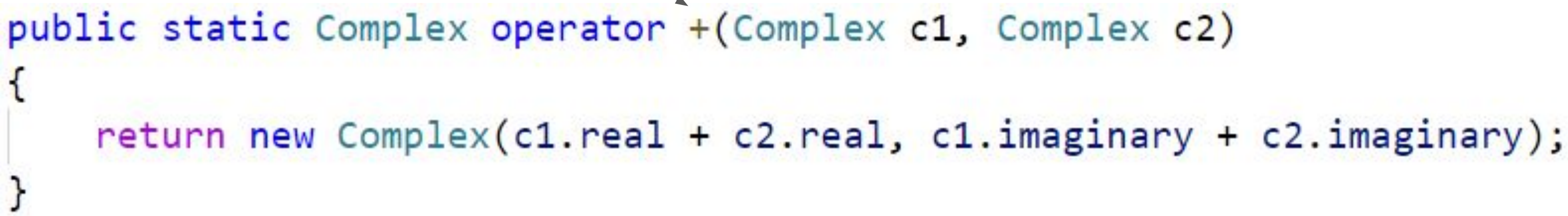
Operator Overloading

Customizing functionality

C# allows user-defined types to overload operators by defining static member functions using the operator keyword

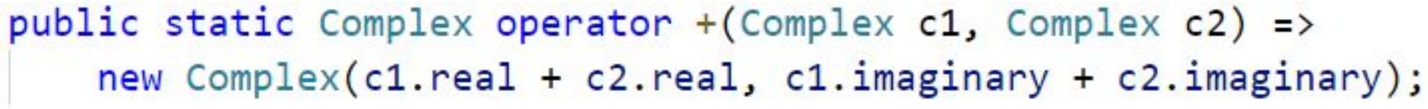
Symbol of the operator being overloaded

One parameter must be the same type as the class or struct that declares the operator



```
public static Complex operator +(Complex c1, Complex c2)
{
    return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
}
```

Shorthand for definitions that simply return an expression

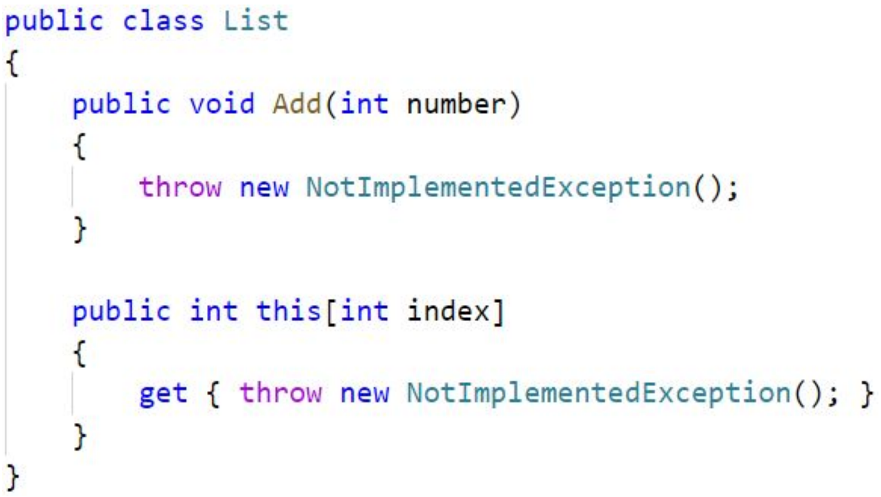


```
public static Complex operator +(Complex c1, Complex c2) =>
    new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
```

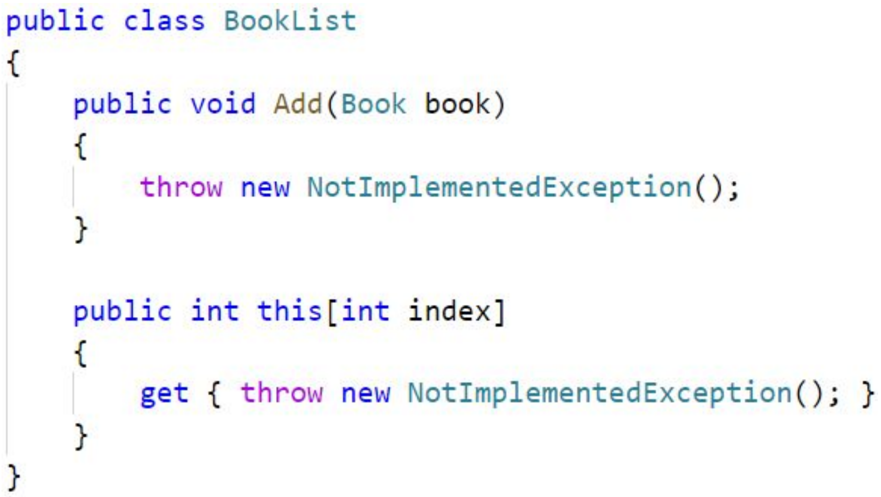
Overloadable Operators

Generics

Before generics, for every list type we had to create a separate list class => code duplication; bugs have to fixed in multiple places



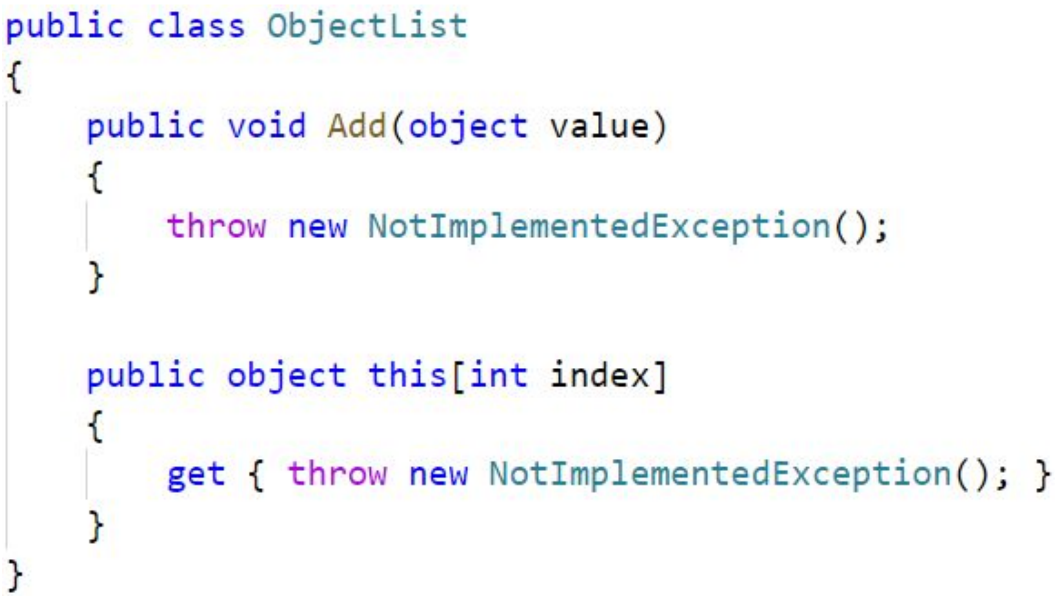
```
public class List
{
    public void Add(int number)
    {
        throw new NotImplementedException();
    }

    public int this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```



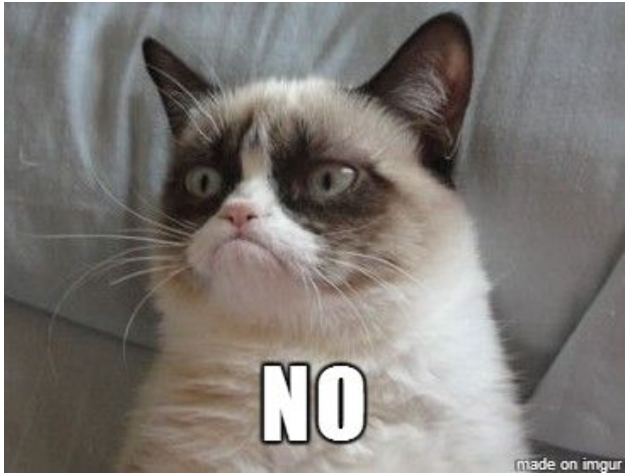
```
public class BookList
{
    public void Add(Book book)
    {
        throw new NotImplementedException();
    }

    public int this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

Generics

The solution?

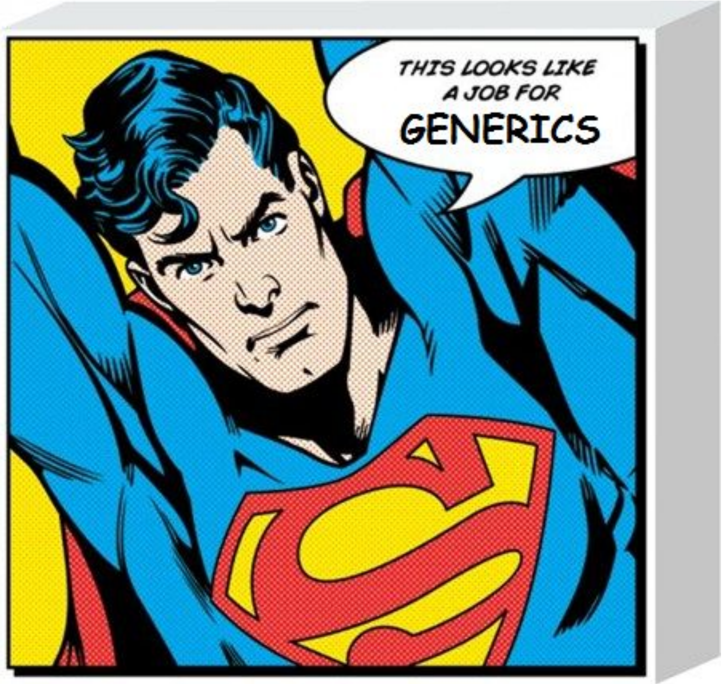


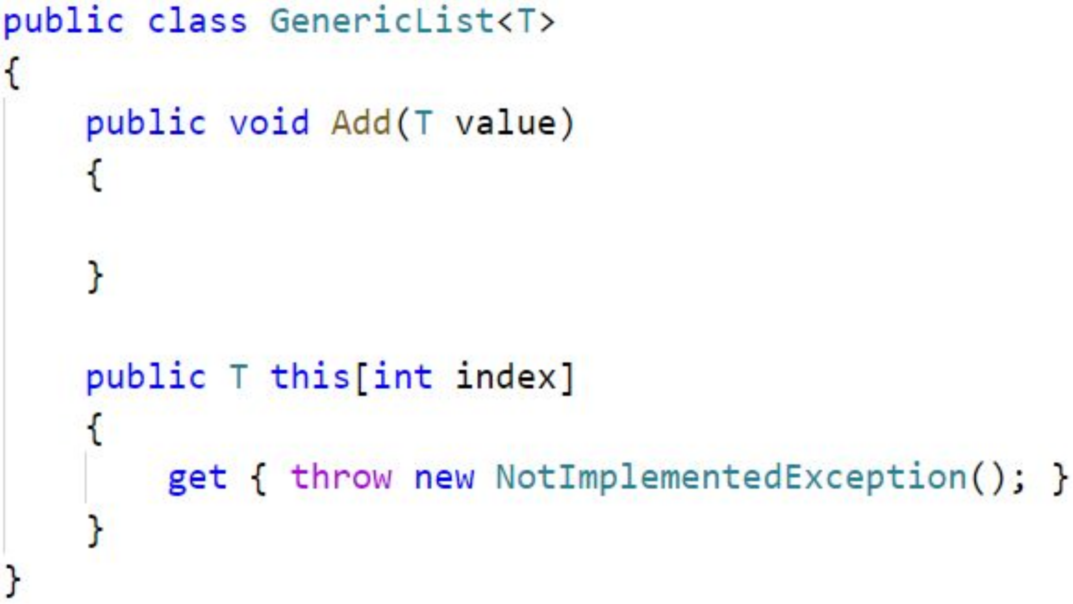
```
public class ObjectList
{
    public void Add(object value)
    {
        throw new NotImplementedException();
    }

    public object this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```



Generics

Generics introduce the concept of type parameters which makes it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code

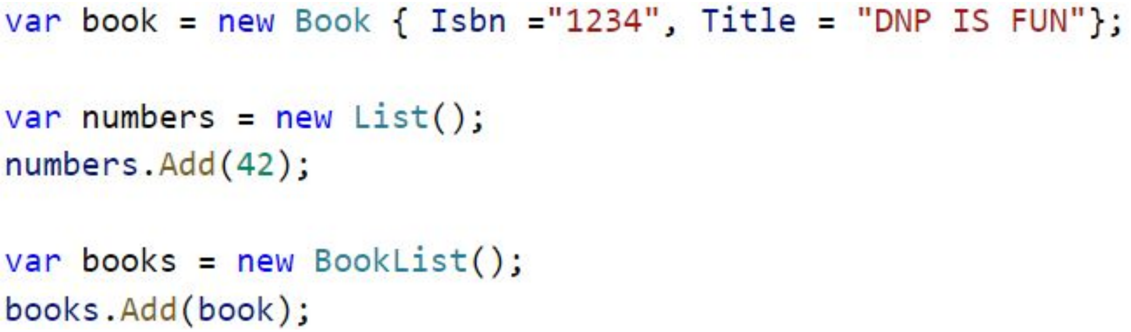




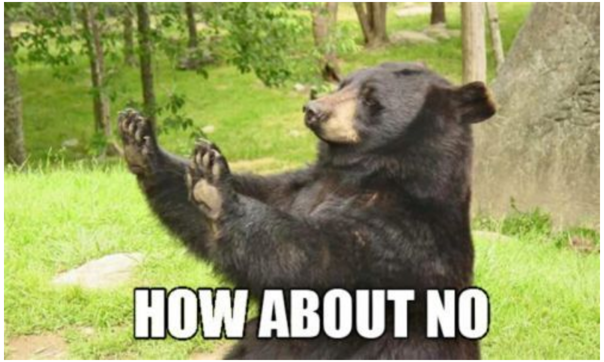
```
public class GenericList<T>
{
    public void Add(T value)
    {

    }

    public T this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

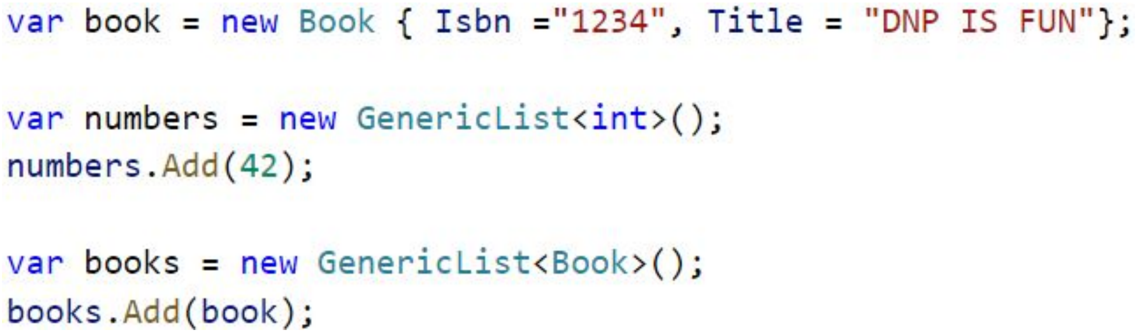
No runtime penalty! No boxing or casting!

Generics



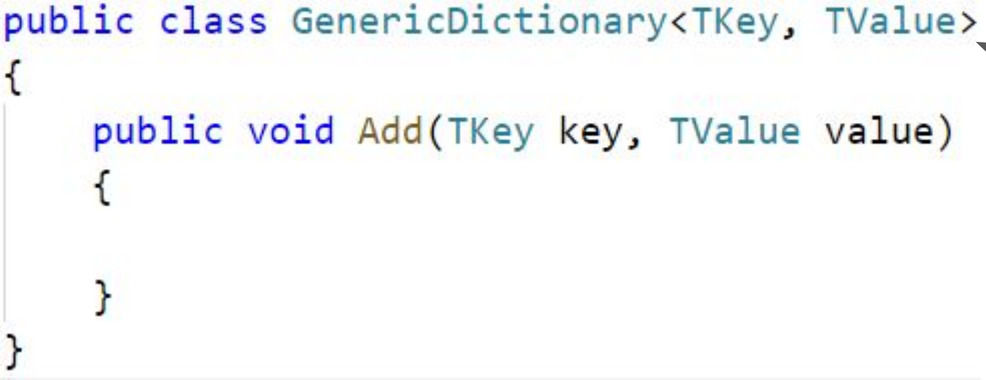
```
var book = new Book { Isbn ="1234", Title = "DNP IS FUN"};

var numbers = new List();
numbers.Add(42);

var books = new BookList();
books.Add(book);
```





```
var book = new Book { Isbn ="1234", Title = "DNP IS FUN"};

var numbers = new GenericList<int>();
numbers.Add(42);

var books = new GenericList<Book>();
books.Add(book);
```

Generics

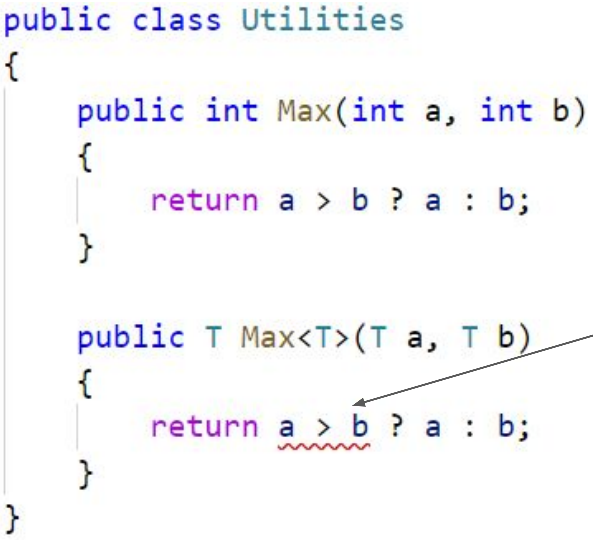


```
public class GenericDictionary<TKey, TValue>
{
    public void Add(TKey key, TValue value)
    {

    }
}
```

Multiple type parameters!

As a developer, you use generics more than you create them, e.g. System.Collections.Generic

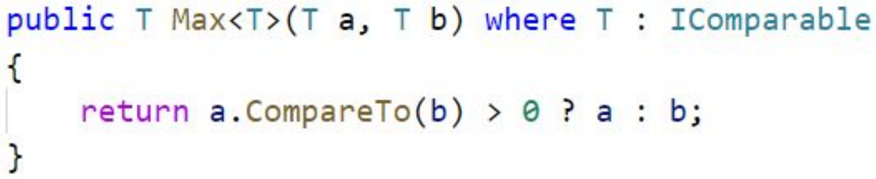
Generics



```
public class Utilities
{
    public int Max(int a, int b)
    {
        return a > b ? a : b;
    }

    public T Max<T>(T a, T b)
    {
        return a > b ? a : b;
    }
}
```

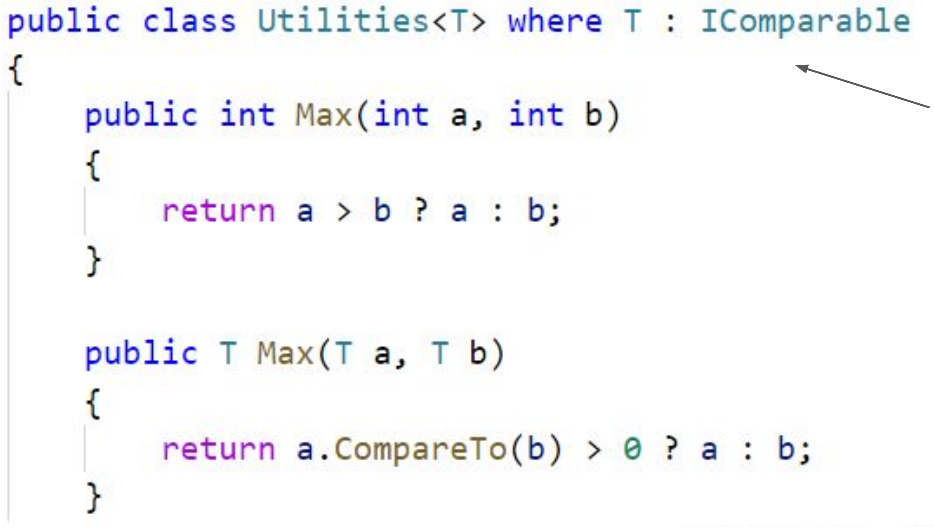
Compiler thinks that a and b are both objects

Solution! Applying a constraint



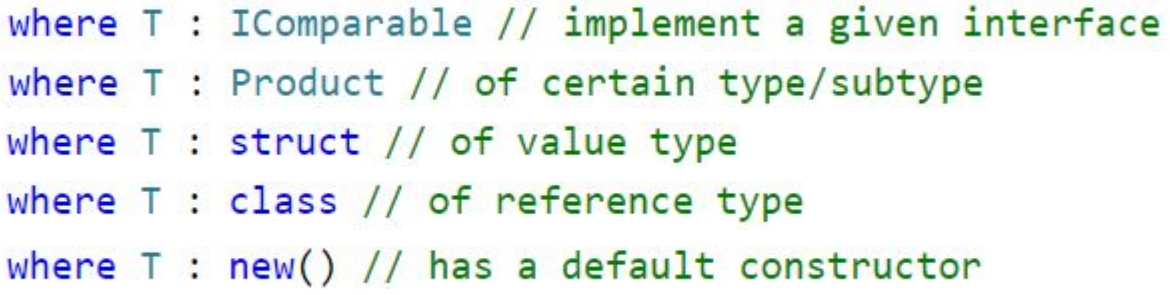
```
public T Max<T>(T a, T b) where T : IComparable
{
    return a.CompareTo(b) > 0 ? a : b;
}
```

Generics - constraints



```
public class Utilities<T> where T : IComparable
{
    public int Max(int a, int b)
    {
        return a > b ? a : b;
    }

    public T Max(T a, T b)
    {
        return a.CompareTo(b) > 0 ? a : b;
    }
}
```

Constraint can be moved to class level if needed



```
where T : IComparable // implement a given interface
where T : Product // of certain type/subtype
where T : struct // of value type
where T : class // of reference type
where T : new() // has a default constructor
```

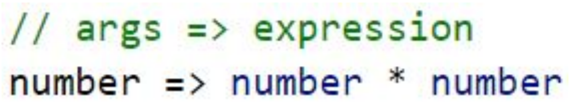
Multiple constraints can be applied

Lambda Expressions

An anonymous method!

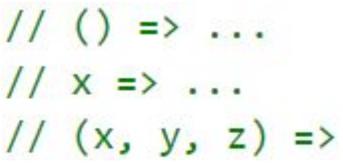
- No access modifier
- No name
- No return statement

Why? For ultimate convenience!

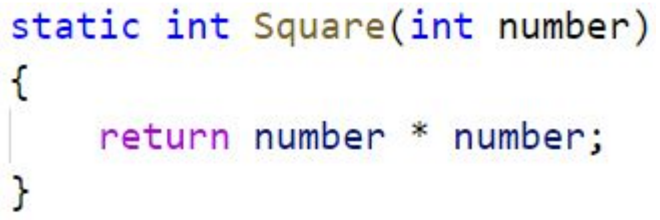


```
// args => expression
number => number * number
```

Syntax

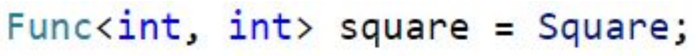


```
// () => ...
// x => ...
// (x, y, z) =>
```



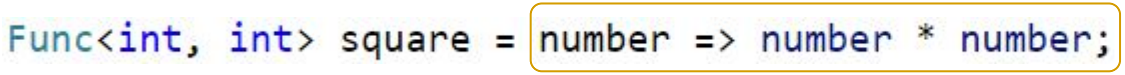
```
static int Square(int number)
{
    return number * number;
}
```

Assign method to delegate



```
Func<int, int> square = Square;
```

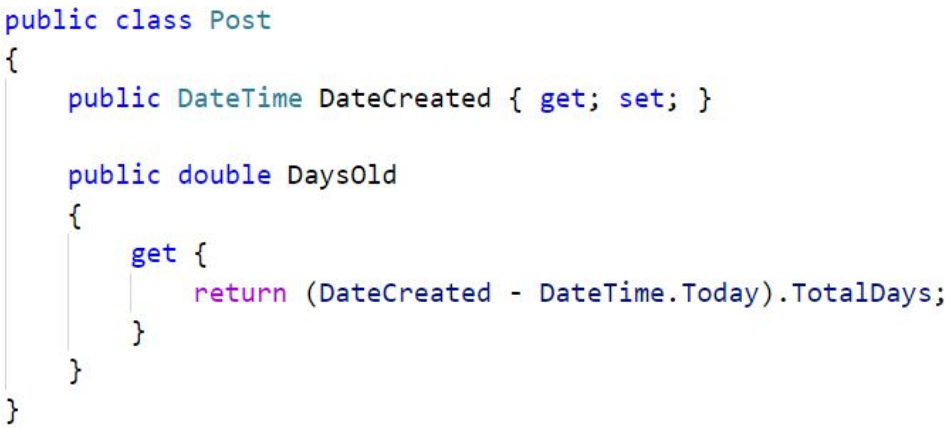
We don't even need the Square method!



```
Func<int, int> square = number => number * number;
```

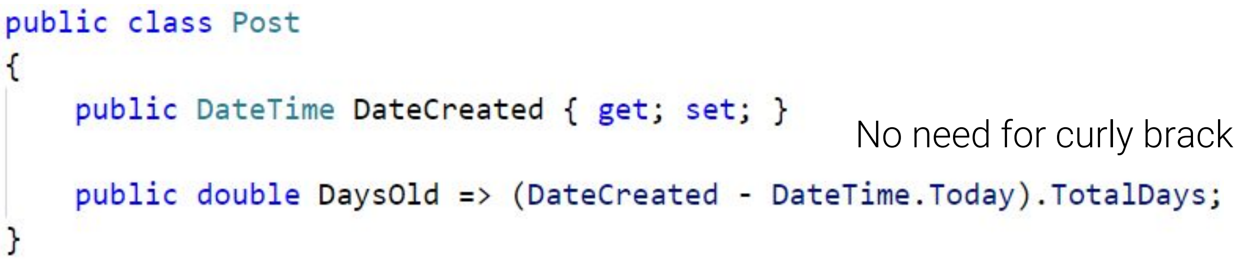
Expression Bodied Members

Cleaner syntax for implementing properties or methods



```
public class Post
{
    public DateTime DateCreated { get; set; }

    public double DaysOld
    {
        get {
            return (DateCreated - DateTime.Today).TotalDays;
        }
    }
}
```

Immediately implementing the getter

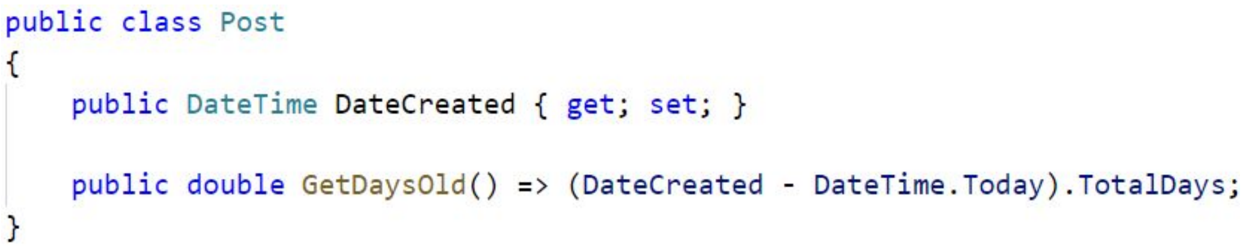


```
public class Post
{
    public DateTime DateCreated { get; set; }

    public double DaysOld => (DateCreated - DateTime.Today).TotalDays;
}
```

No need for curly brackets or return keyword

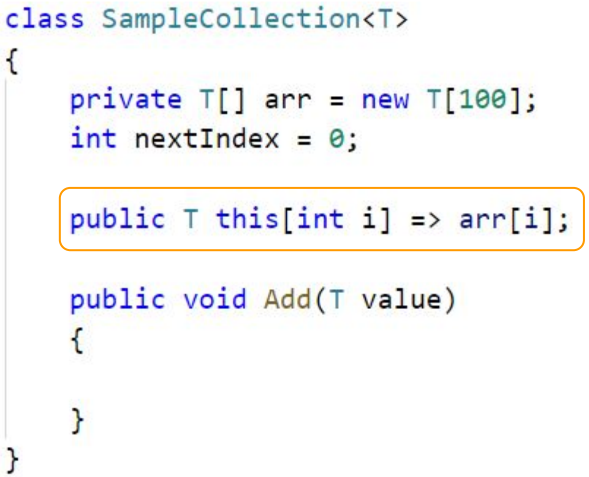
Expression Bodied Members

You can use the same syntax for methods as well!



```
public class Post
{
    public DateTime DateCreated { get; set; }

    public double GetDaysOld() => (DateCreated - DateTime.Today).TotalDays;
}
```

Or read-only indexers!



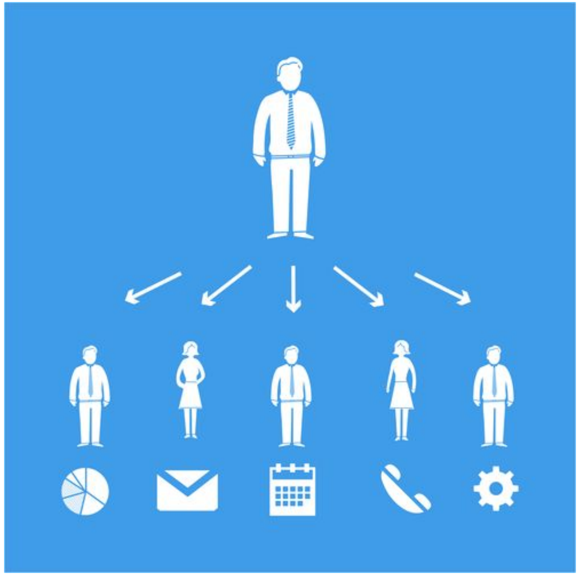
```
class SampleCollection<T>
{
    private T[] arr = new T[100];
    int nextIndex = 0;

    public T this[int i] => arr[i];

    public void Add(T value)
    {

    }
}
```

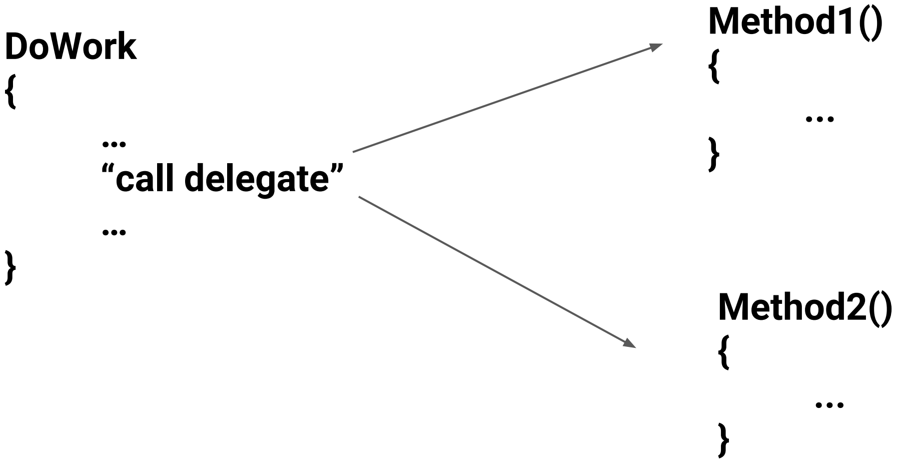
Expression Body Definitions

Delegates

A delegate is a type that represents **references to methods** with a particular parameter list and return type

When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type



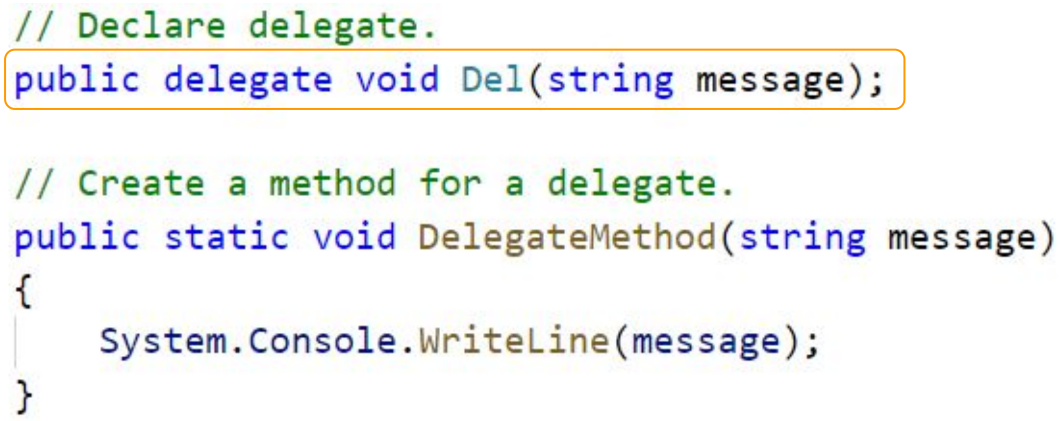


```
DoWork
{
        ...
        "call delegate"
        ...
}
```

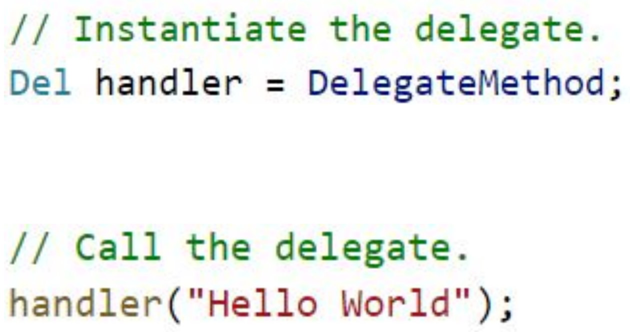
```
Method1()
{
        ...
}
```

```
Method2()
{
        ...
}
```

Delegates



```
// Declare delegate.
public delegate void Del(string message);

// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```



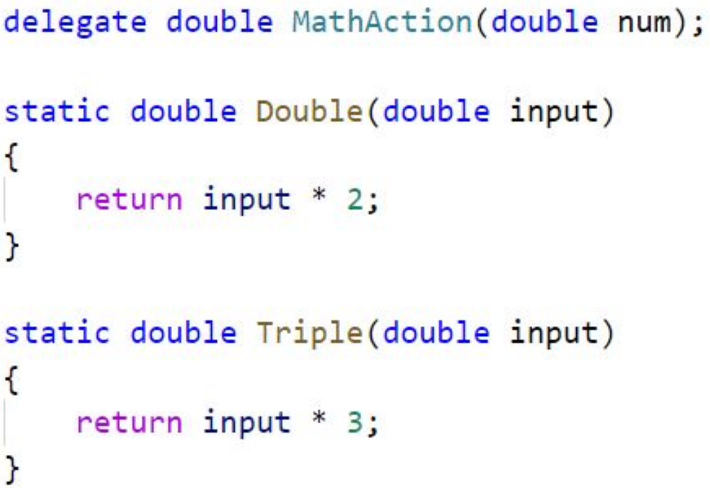
```
// Instantiate the delegate.
Del handler = DelegateMethod;


// Call the delegate.
handler("Hello World");
```

A delegate variable stores a method and its receiver, but no parameters

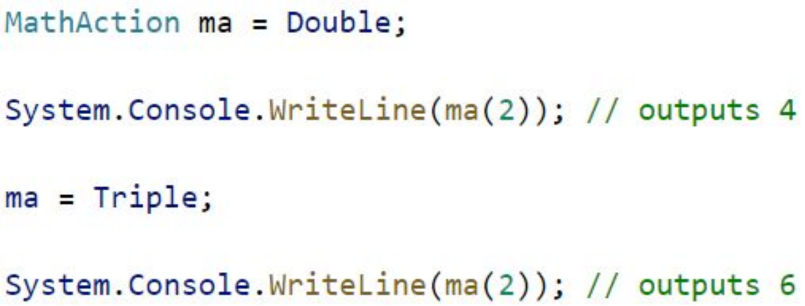
Delegates

Delegates

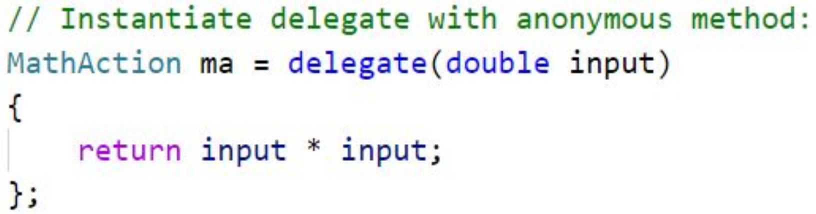
Assigning different methods



```
delegate double MathAction(double num);

static double Double(double input)
{
    return input * 2;
}

static double Triple(double input)
{
    return input * 3;
}
```



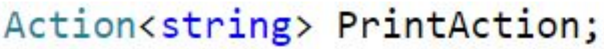
```
MathAction ma = Double;

System.Console.WriteLine(ma(2)); // outputs 4

ma = Triple;

System.Console.WriteLine(ma(2)); // outputs 6
```



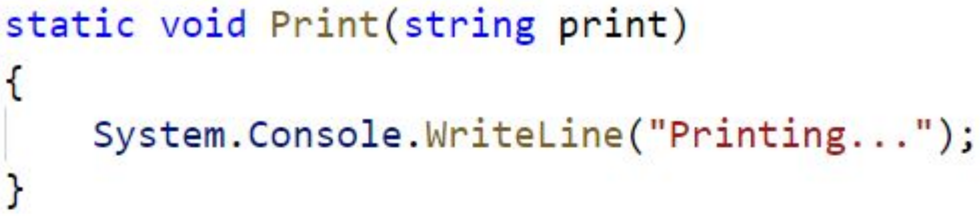
```
// Instantiate delegate with anonymous method:
MathAction ma = delegate(double input)
{
    return input * input;
};
```

Often you can use Action<> and Func<> delegates instead of defining your own

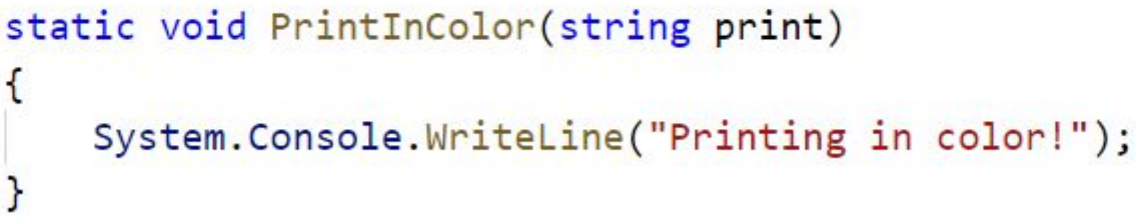
Multicast Delegates



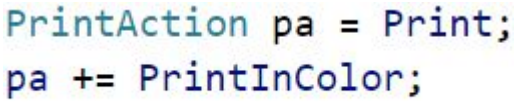
```
Action<string> PrintAction;
```



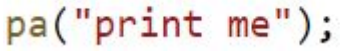
```
static void Print(string print)
{
    System.Console.WriteLine("Printing...");
}
```



```
static void PrintInColor(string print)
{
    System.Console.WriteLine("Printing in color!");
}
```

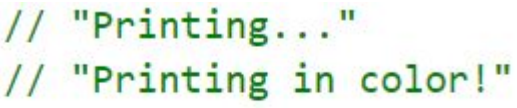


```
PrintAction pa = Print;
pa += PrintInColor;
```



```
pa("print me");
```

Output

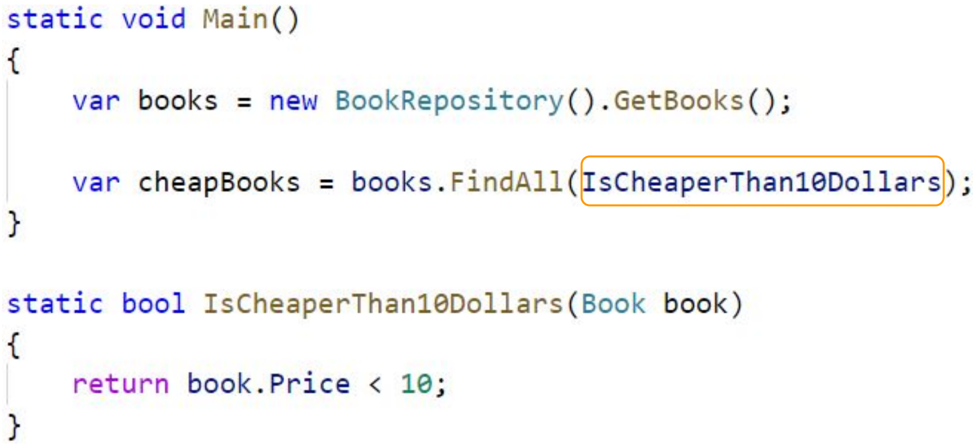


```
// "Printing..."
// "Printing in color!"
```

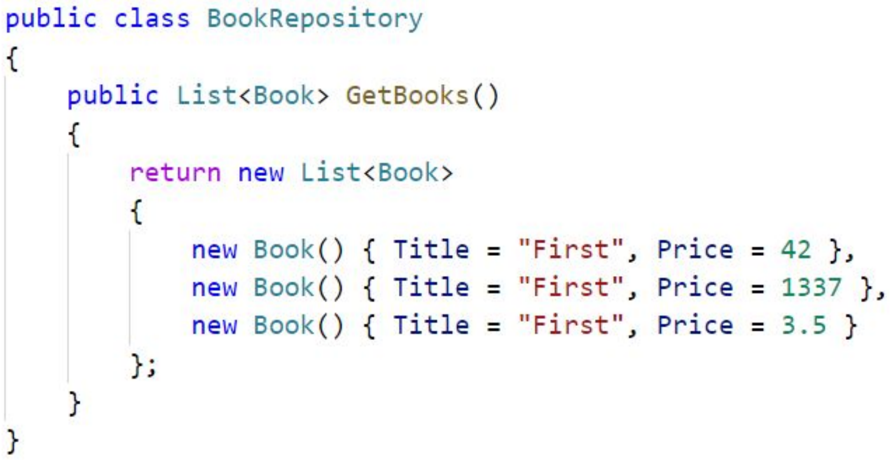
Predicate<T> Delegates

Represents the method that defines a set of criteria and determines whether the specified object meets those criteria.

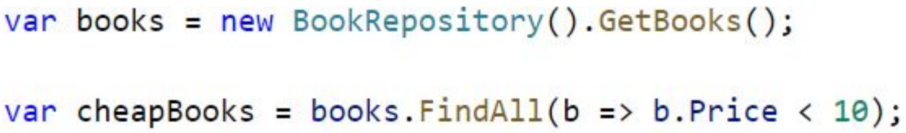
Signature: `public delegate bool Predicate<in T>(T obj);`



```
static void Main()
{
    var books = new BookRepository().GetBooks();

    var cheapBooks = books.FindAll(IsCheaperThan10Dollars);
}

static bool IsCheaperThan10Dollars(Book book)
{
    return book.Price < 10;
}
```



```
public class BookRepository
{
    public List<Book> GetBooks()
    {
        return new List<Book>
        {
            new Book() { Title = "First", Price = 42 },
            new Book() { Title = "First", Price = 1337 },
            new Book() { Title = "First", Price = 3.5 }
        };
    }
}
```



```
var books = new BookRepository().GetBooks();

var cheapBooks = books.FindAll(b => b.Price < 10);
```

Predicate delegate methods must take one input parameter and return a boolean.

Predicate<T> Delegates