Advanced Development Techniques

Delegate – recap
Using Delegates using modern syntax

Delegate

- **A type that is capable of storing multiple methods inside**
  - The **delegate type** determines the signature of the method it can contain

```
delegate double MyDelegate(int param1, string param2);
```

  - The specific **delegate variable** will actually store the methods
    (in the background: class=type and instance+variable+list)

```
double funct(int first, string second)
{
    return first + second.Length;
}
```

```
MyDelegate del = new MyDelegate(funct); //long syntax
MyDelegate del = funct;                 //short syntax
```

  - **The delegate variable has a null value until it has no added methods**

Usage of delegates

- **C# delegates are multicast delegates, it is capable of storing multiple methods – we can add/remove methods:**

```
del += new MyDelegate(Function1);  //long syntax
del += Function1;                  //short syntax
```

```
del -= new MyDelegate(Function1);  //long syntax
del -= Function1;                  //short syntax
```

- **Call the methods inside the delegate:**

```
MyDelegate temp = del;  //The temporary variable is
if (temp != null)       //needed because of thread safety
    temp(0, "alma");    //

del?.Invoke(0, "alma"); //New syntax, ATOMIC operation
```

  - **The calling order is not guaranteed, must not rely on it!** (.NET 4.5: queue order)
  - If there is a return value, then the lastly returned value is used

Self-made vs. built-in delegate types

- **We can define delegate types now:**

```
delegate double MyDelegate(int param1, string param2);
```

  - „It is capable of storing methods where the return value is a double, and the parameters are int + string”

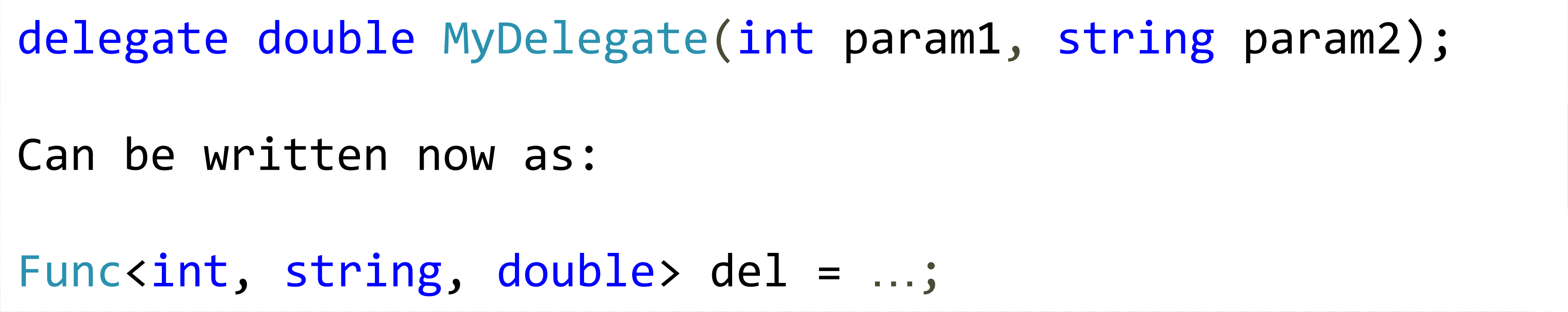
- **Almost never used, as the framework has many built-in delegate types, we always use those!**

- **So the type of the delegate-variable will NOT be MyDelegate, but instead some framework class that fixes that what method signatures can be used with the variable (must define the result + parameter types)**

Built-in delegate types

Predicate<T>	bool(T)	List<T>.Find(), .Exists(), RemoveAll()…
Comparison<T>	int(T1,T2)	List<T>.Sort(), Array.Sort()
MethodInvoker	void()	
EventHandler	void(object,EventArgs)	
EventHandler<T>	void(object,T) (T EventArgs utód)	
Action	**void()**	
Action<T>	**void(T)**	
Action<T1,T2>	**void(T1,T2)**	
Action<T1,T2,...,T16>	**void(T1,T2,...,T16)**	
Func<TRes>	**TRes()**	
Func<T, TRes>	**TRes(T)**	
Func<T1, T2, TRes>	**TRes(T1,T2)**	
Func<T1, T2, ... T16, TRes>	**TRes(T1,T2,...,T16)**	

Example



```
delegate double MyDelegate(int param1, string param2);

Can be written now as:

Func<int, string, double> del = …;
```

Using delegates

- **Many times as parameters!**

```
private bool IsItEven(int i)
{
    return i % 2 == 0;
}
private int EvenNumbersGoFirst(int i1, int i2)
{
    bool i1Even = IsItEven(i1);
    bool i2Even = IsItEven(i2);
    if (i1Even && !i2Even) return -1;
        else if (!i1Even && i2Even) return 1;
        else return 0;
}
```

```
int[] myArray; List<int> myList;
// ...
int firstEven = myList.Find(IsItEven);
List<int> allEvenNumbers = myList.FindAll(IsItEven);
bool isThereAnEven = myList.Exists(IsItEven);
Array.Sort(myArray, EvenNumbersGoFirst);
```

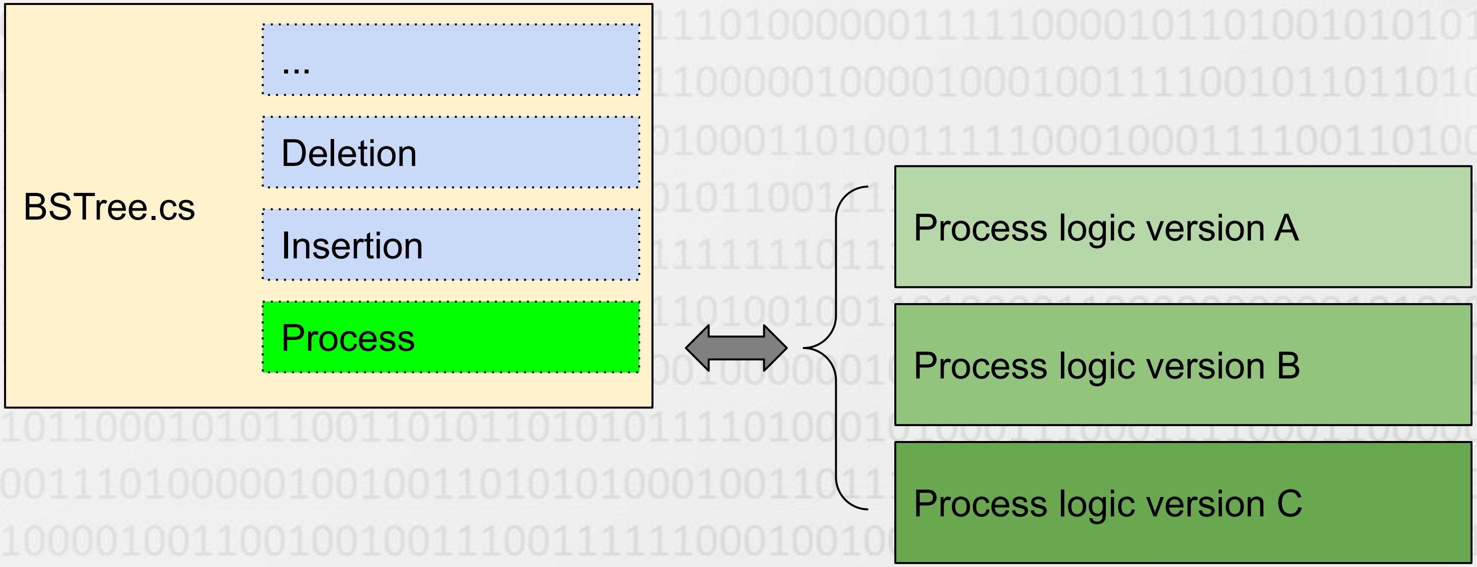
Using delegates

Most important part: I can use ‘operation’ as a parameter!

Example: BSTree with delegate processing @ my GitHub

https://github.com/siposm/oktatas-hft-20211/blob/master/LA-01-delegate/binaris-keresofa-delegalttal/bstree/Program.cs

https://github.com/siposm/oktatas-hft-20211/blob/master/LA-01-delegate/binaris-keresofa-delegalttal/bstree/BST.cs

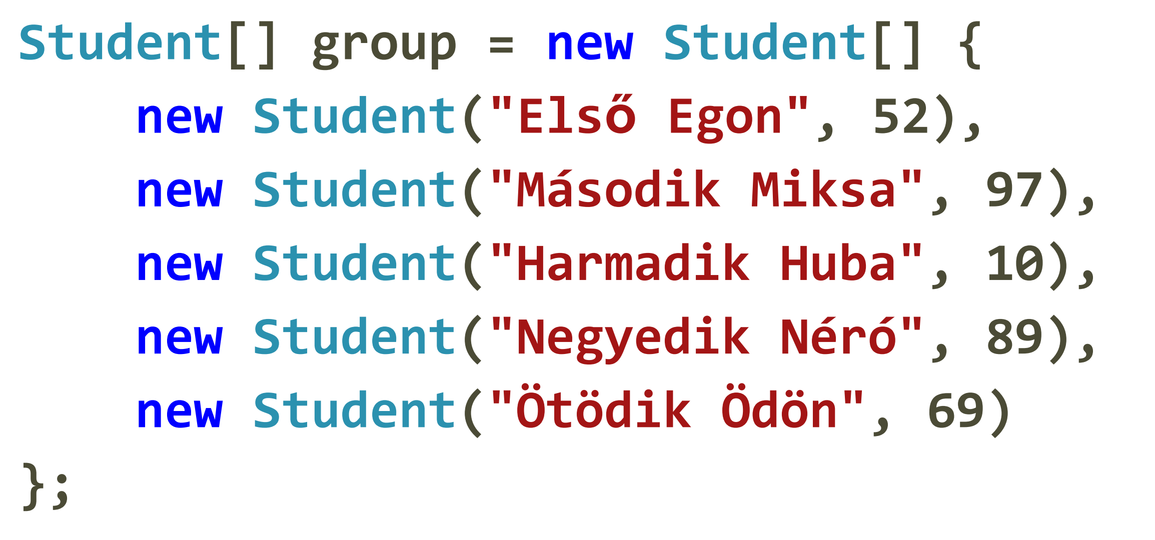


Let’s re-implement Array.Sort

```
delegate bool MyComparer(object left, object right);
class SimpleReplaceSort
{
    public static void Sort(object[] array, MyComparer isLarger)
    public static void Sort(object[] array, Func<object, object, bool>
    isLarger)
    {
        for (int i = 0; i < array.Length; i++)
            for (int j = i + 1; j < array.Length; j++)
                if (isLarger?.Invoke(array[j], array[i]))
                {
                    object temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
    }
}
```

Let’s re-implement Array.Sort

```
class Student
{
    public string Name { get; set; }
    public int Credits { get; set; }
    public Student(string name, int credits)
    {
        this.Name = name; this.Credits = credits;
    }
}
```



```
Student[] group = new Student[] {
    new Student("Első Egon", 52),
    new Student("Második Miksa", 97),
    new Student("Harmadik Huba", 10),
    new Student("Negyedik Néró", 89),
    new Student("Ötödik Ödön", 69)
};
```

Let’s re-implement Array.Sort

```
bool ByCredits(object first, object second)
{
    return ((first as Student).Credits <
      (second as Student).Credits);
}

SimpleReplaceSort.Sort(group, ByCredits);
```

Event vs Delegate

- **Delegate member variable in a class:**

```
DelegateType variableName;
```

- **Event member variable in a class:**

```
event DelegateType eventName;
```

 The event is simply a delegate with the keyword “event”, which is there for protection, as the event is usually public

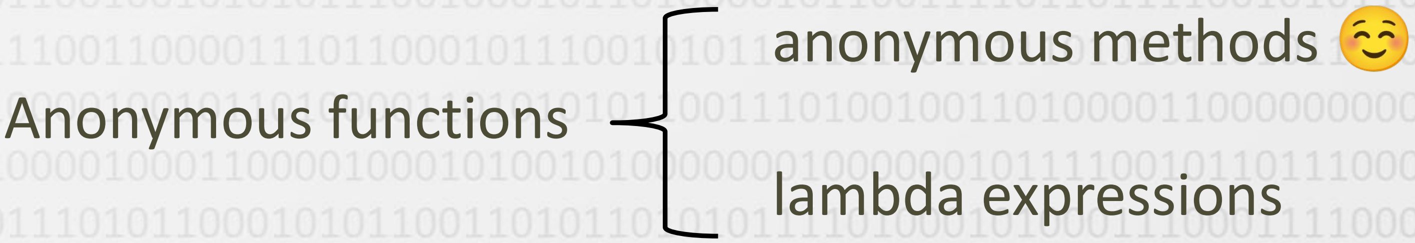
Delegate	Event
Can be called from anywhere	Can only be called from within the containing class
Can be overwritten (with =)	Cannot be overwritten, only the operators += and -= are allowed
Simple property with get and set keywords	Event property with add (+=) and remove (-=) keywords
Cannot be in an interface	Can be in an interface

Event handling – Naming conventions

Role	Name	Location
Event Parameter	...EventArgs (PropertyChangedEventArgs)	In the namespace or event source class
Delegate	...EventHandler (PropertyChangedEventHandler)	In the namespace or event source class
Event variable	... (PropertyChanged)	In the event source class
Method that directly calls the event	On... (OnPropertyChanged)	In the event source class
Handle events	---	In the event handler class

Anonymous functions

- **We use delegates for:**
  - Events
  - Use methods as parameters
- **Problem: the many single-use methods are hard to follow, they make the code hard to read and understand**
- **Solution: anonymous functions, define methods in-place**
- **http://msdn.microsoft.com/en-us/library/bb882516.aspx**
- **Not to mix up with: local/inline functions, which are BAD**



Anonymous methods

```
int firstEvenNumber =
    myList.Find(delegate(int i) { return i % 2 == 0; });
List<int> allEvenNumbers =
    myList.FindAll(delegate(int i) { return i % 2 == 0; });
bool isThereAnEvenNumber =
    myList.Exists(delegate(int i) { return i % 2 == 0; });

Array.Sort(myArray,
    delegate(int i1, int i2)
    {
        bool i1Even = i1 % 2 == 0;
        bool i2Even = i2 % 2 == 0;
        if (i1Even && !i2Even) return -1;
        else if (!i1Even && i2Even) return 1;
        else return 0;
    });
```

- **Not really used (rather: lambda expressions)**

Lambda expressions

- **New operator: => (Lambda operator)**
  - Connects the input and the output
  - “If the input is a number called X, then the output is …”
- **Syntax: parameter[s] => expression to determine the output**
- **Usage:**
  - delegate type (self-made or framework), this is usually a parameter type

```
delegate double SingleParamMathOp(double x)
Func<double, double>
```

  - create a delegate variable, and we can specify the method using lambda expression syntax, then call the method

```
SingleParamMathOp operation = x => x * x;
double j = operation(5);
```

Lambda expressions

```
delegate double TwoParamMathOp(double x, double y);
```

```
TwoParamMathOp myFunc = (x, y) => x + y;
double j = myFunc(5, 10);    //j = 15
```

- **Using built-in delegate types:**

```
Func<int, int> myFunc = (x) => x * x;
int j = myFunc(5); //j = 25
Func<int, int, int> myFunc2 = (x, y) => x + y;
int j2 = myFunc2(5, 10); //j = 15
```

- **If there are multiple parameters, we must use parentheses**
- **Specifying the parameter types is only needed in special circumstances**

Lambda expressions

```
int firstEvenNumber =
    myList.Find(i => i % 2 == 0);
List<int> allEvenNumbers =
    myList.FindAll(i => i % 2 == 0);
bool IsThereAnEvenNumber =
    myList.Exists(i => i % 2 == 0);

Array.Sort(myArray,
      (i1, i2) =>
      {
       bool i1Even = i1 % 2 == 0;
       bool i2Even = i2 % 2 == 0;
       if (i1Even && !i2Even) return -1;
       else if (!i1Even && i2Even) return 1;
       else return 0;
      });
```

Lambda expressions

- **Subtypes:**
  - **Expression Lambda**
    - One statement on the right side to determine the return value

      ```
      x => x * x
      ```
  - **Statement Lambda**
    - Multiple statements on the right side; anything is possible between curly braces

      ```
      x => { Console.WriteLine(x); }
      ```
  - **Difference:**
    - **The expression lambda in some locations (e.g. communication with databases) is not compiled to a delegate, but rather into an Expression Tree**
    - **This Expression Tree can be translated into an SQL statement that can be sent out to the Database**

Lambda expressions

- **Pros:**
  - The method is instantly readable where it’s used
  - Less “trash” methods in the class
- **Only use it with single-use, and possibly short operations:**
  - Long code is hard to read
  - Not re-useable method
- **Do not ember lambda inside lambda inside lambda … or anonymous method inside anonymous method inside anonymous method…**

- **Possible bug: Outer Variable Trap**

Outer Variable Trap

- You can use external variables on the right side of the lambda expression (closure), but this needs special attention

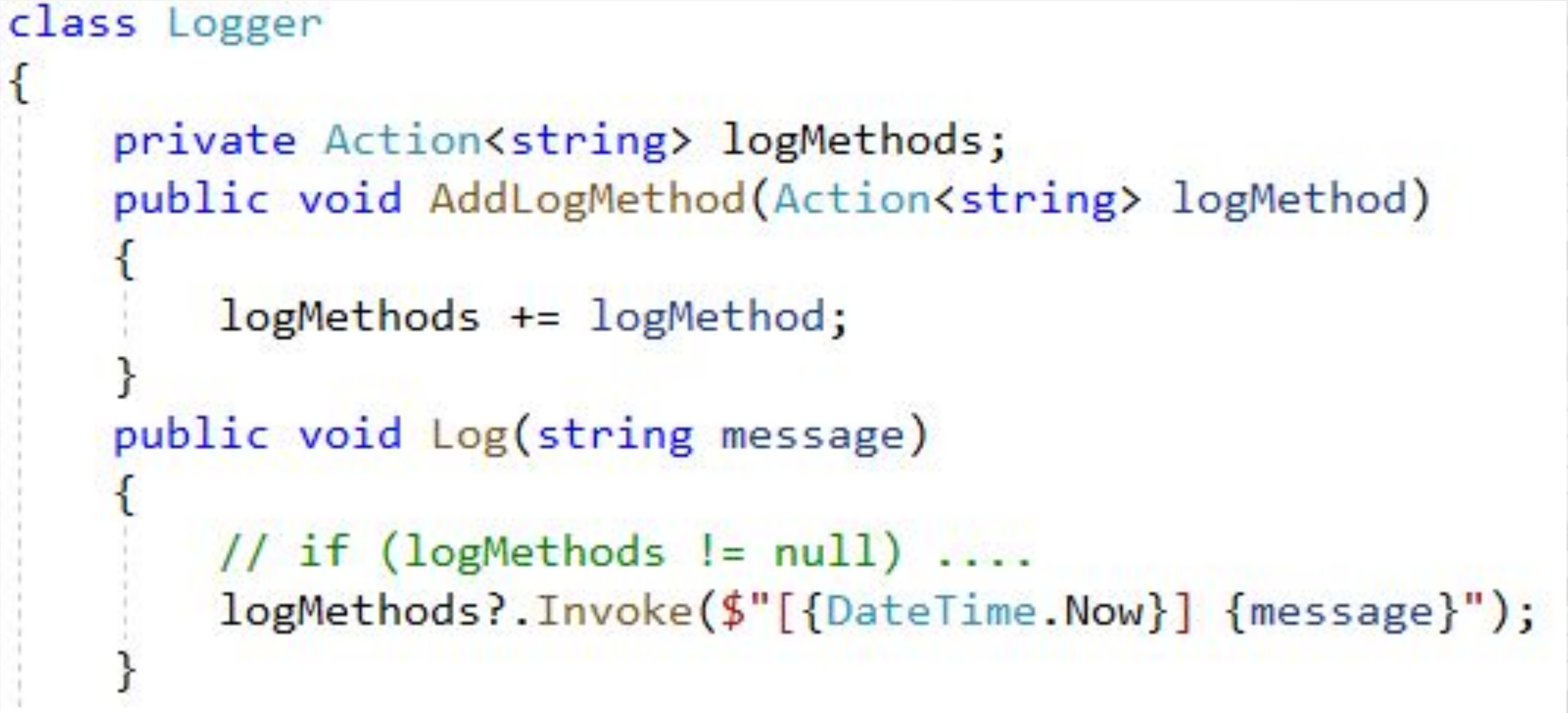
```
Action numberWriter = null;
for (int i = 0; i < 10; i++)
{
      numberWriter += () => { Console.WriteLine(i); };
}
numberWriter();
```

- „Expected” output: 0, 1, 2, 3, 4, 5, …
- BUT all external variables are passed as references – the value typed variables TOO!
- Real output: 10, 10, 10, 10, 10 …
- Solution: must introduce a local temporary variable (that is not changed later)

```
// ...
int f = i;
numberWriter += () => { Console.WriteLine(f); };
// ...
```

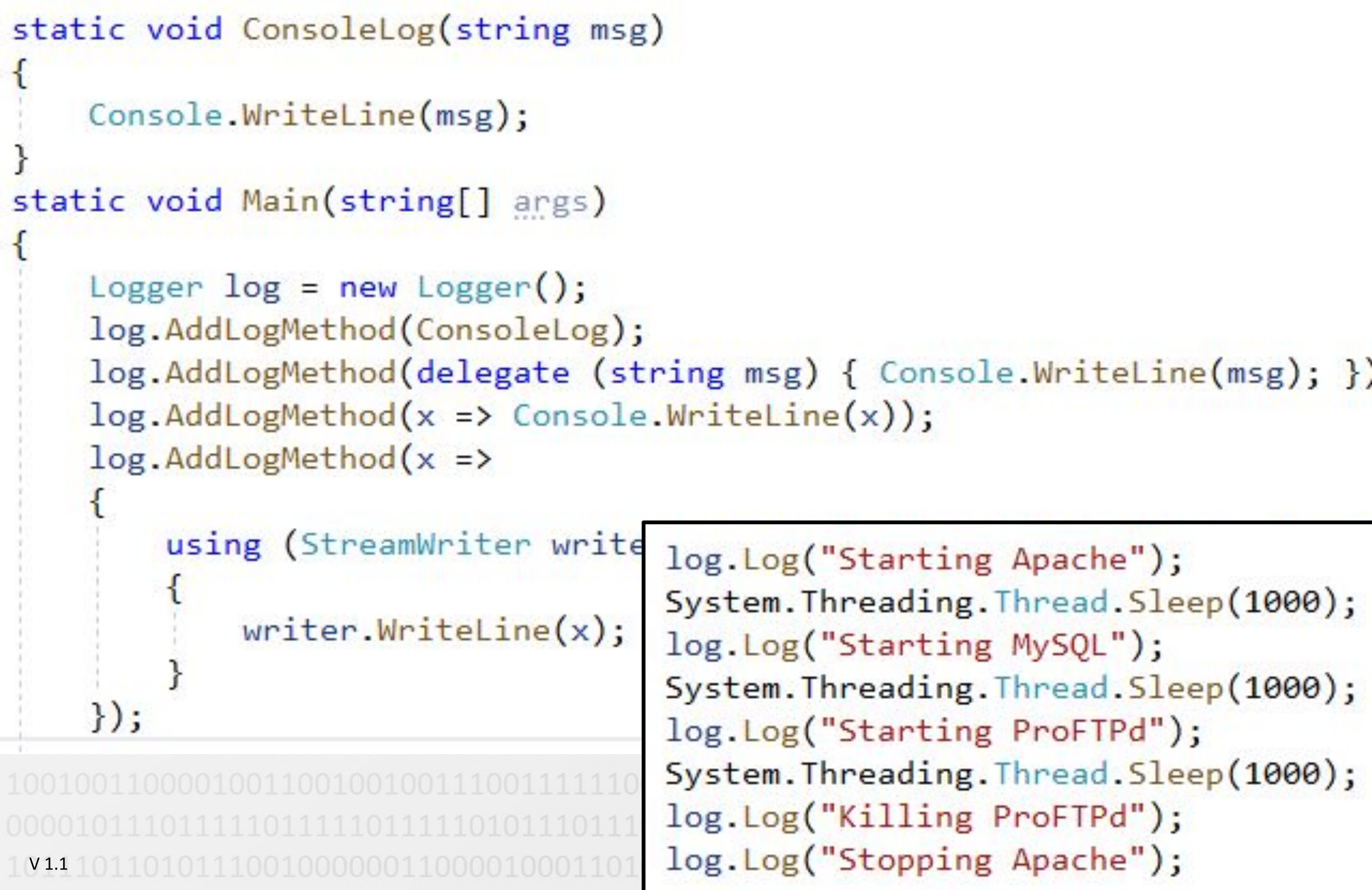
Example

- **We want to write a logger application, where the Logger class does not know, exactly what logger methods we have (email, database, local OS event log, syslog-ng, …)**



```
class Logger
{
    private Action<string> logMethods;
    public void AddLogMethod(Action<string> logMethod)
    {
        logMethods += logMethod;
    }
    public void Log(string message)
    {
        // if (logMethods != null) ....
        logMethods?.Invoke($"[{DateTime.Now}] {message}");
    }
```

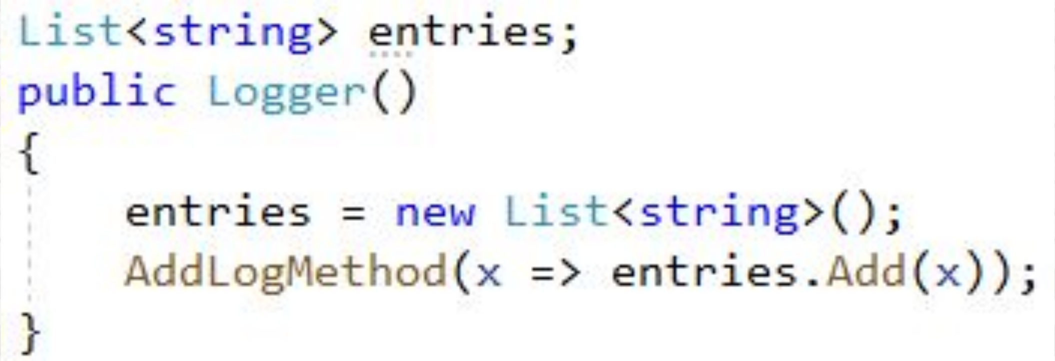
Example – Main method



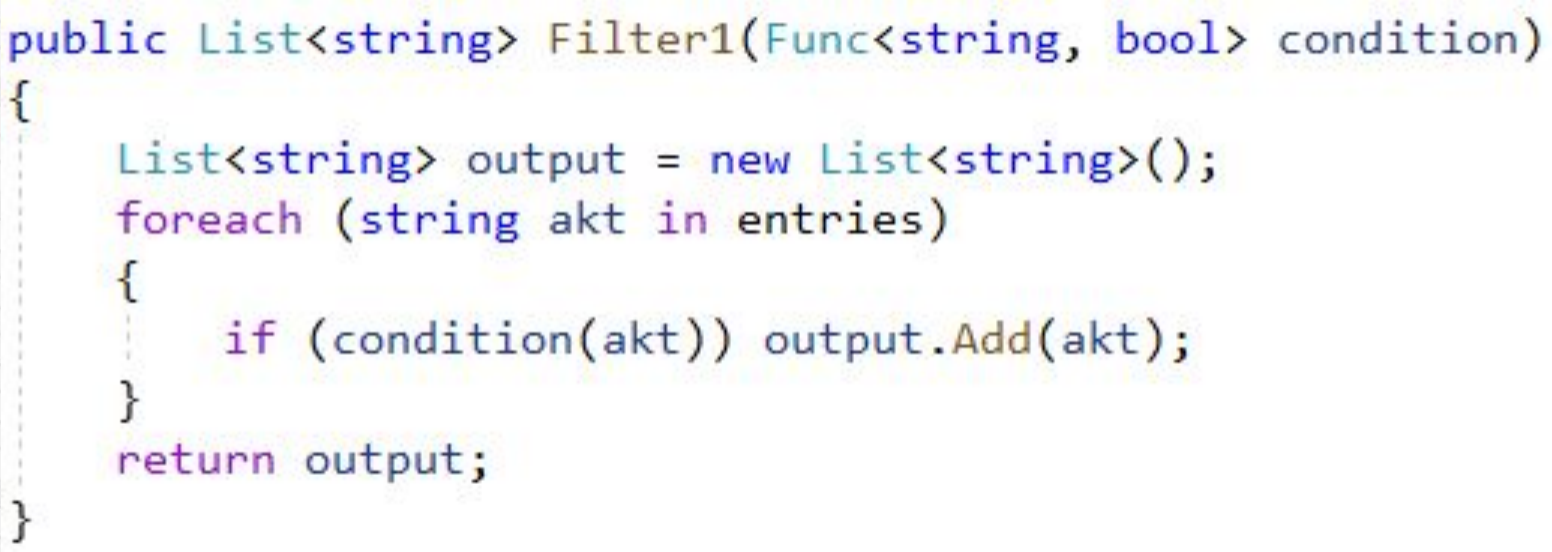
```
static void ConsoleLog(string msg)
{
    Console.WriteLine(msg);
}
static void Main(string[] args)
{
    Logger log = new Logger();
    log.AddLogMethod(ConsoleLog);
    log.AddLogMethod(delegate (string msg) { Console.WriteLine(msg); })
    log.AddLogMethod(x => Console.WriteLine(x));
    log.AddLogMethod(x =>
    {
        using (StreamWriter write
        {
            writer.WriteLine(x);
        }
    });
```

```
log.Log("Starting Apache");
System.Threading.Thread.Sleep(1000);
log.Log("Starting MySQL");
System.Threading.Thread.Sleep(1000);
log.Log("Starting ProFTPd");
System.Threading.Thread.Sleep(1000);
log.Log("Killing ProFTPd");
log.Log("Stopping Apache");
```

Addition: Filter

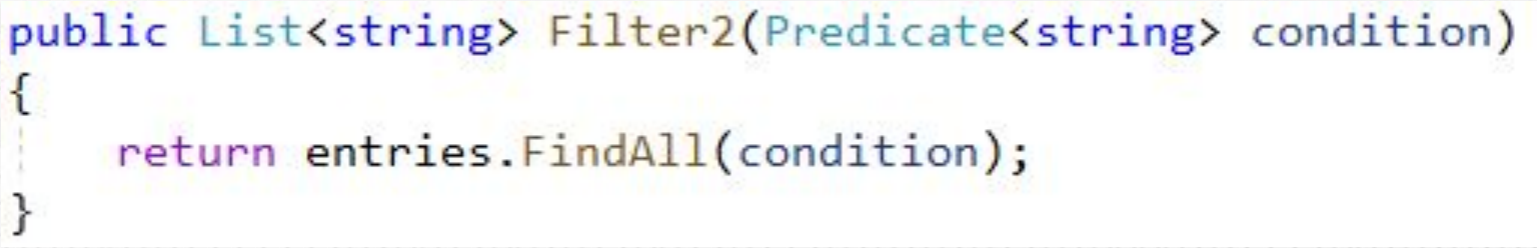


```
List<string> entries;
public Logger()
{
    entries = new List<string>();
    AddLogMethod(x => entries.Add(x));
}
```

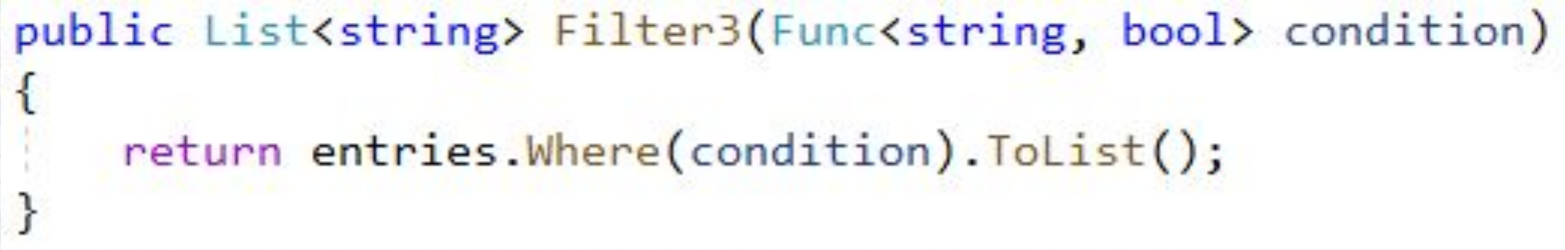


```
public List<string> Filter1(Func<string, bool> condition)
{
    List<string> output = new List<string>();
    foreach (string akt in entries)
    {
        if (condition(akt)) output.Add(akt);
    }
    return output;
}
```

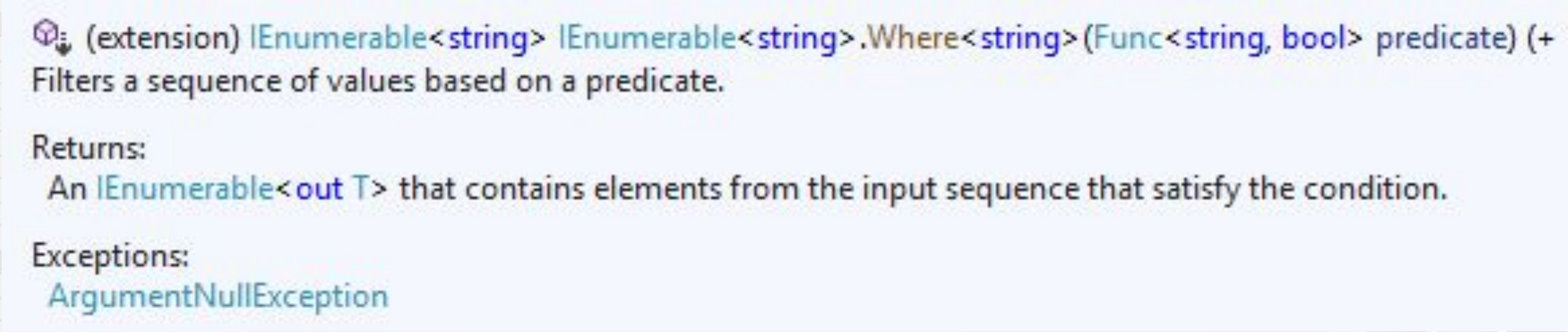
Reinvent the wheel???



```
public List<string> Filter2(Predicate<string> condition)
{
    return entries.FindAll(condition);
}
```



```
public List<string> Filter3(Func<string, bool> condition)
{
    return entries.Where(condition).ToList();
}
```



(extension) IEnumerable<string> IEnumerable<string>.Where<string>(Func<string, bool> predicate) (+

Filters a sequence of values based on a predicate.

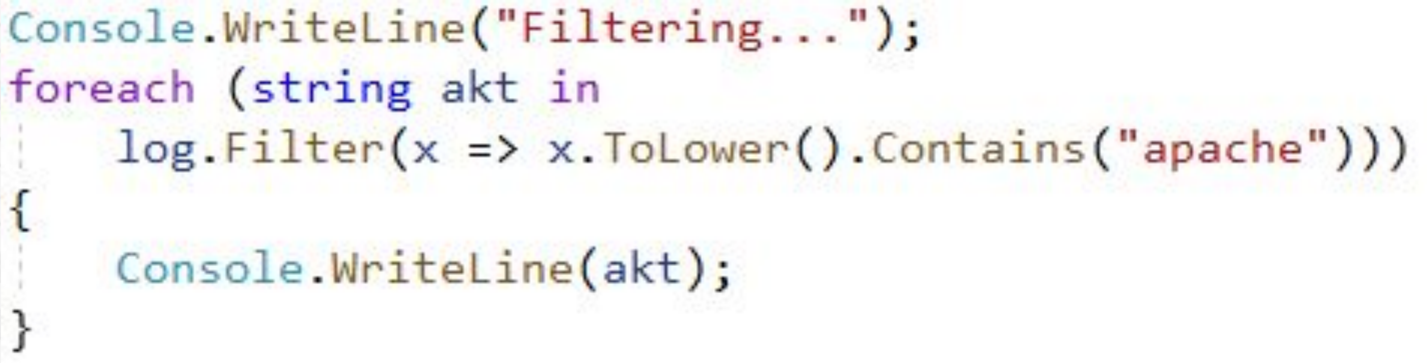
Returns:

An IEnumerable<out T> that contains elements from the input sequence that satisfy the condition.

Exceptions:

ArgumentNullException

Filter – Main method



```
Console.WriteLine("Filtering...");
foreach (string akt in
    log.Filter(x => x.ToLower().Contains("apache")))
{
    Console.WriteLine(akt);
}
```

Practice

- **We want to handle feedbacks (Opinion, Bugreport, FeatureRequest)**
- **All feedback types can have various multiple handler methods**
- **Periodically after every tenth (on the practice: third) feedback, we have to call the handler methods**
- **Let’s store the handler methods in a Dictionary<Category, Action<Feedback>>**