C# Methods

Topics Covered

- What is a Method?
- Re-using Code
- Method Declarations
  - Access Modifiers
  - Return Type
  - Name
  - Parameter List
- Methods and Classes
  - State Management
- Method Signatures
  - Uniqueness
  - Overloading
- Lambda Expressions
- Local Methods
- Default Parameters
- Named Parameters
- Open-ended Argument Lists
- ref and out Parameters
- Extension Methods

What Is a Method?

A method is a named code block that contains a series of statements.

In C#, every method is defined in the context of a class.

Methods can be called (or *invoked*), which makes them re-usable computational units.

Methods can optionally return a value to the caller.

Each method has a unique *signature* consisting of its name and its parameter list.

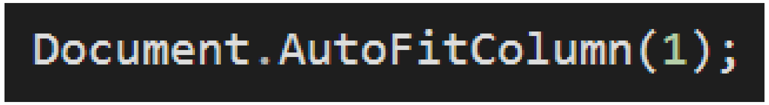
Methods are one of 8 types of class *members*. The other class member types are:

Fields Constructors Properties Events Indexers Operators Finalizers

Method Reuseability

Suppose you have an application that creates a spreadsheet document. You want to be able to auto-size a column so that it is wide enough to visually accommodate all of its content. This involves examining each row of the column that has content, assessing the width needed for that content, and then keeping the maximum value found to use as the final column width.

That's a lot of code! And once you've written the code, it is not something that you want to be copy/pasting all over your application. It is ideal for placing in a method named AutoFitColumn. The method could take just 1 parameter – the integer index of the column to auto-fit. Now you can re-use it whenever you need:



```
Document.AutoFitColumn(1);
```

Method Declarations

Here is the declaration of the method we just created:

```
public void AutoFitColumn(int ColumnIndex) { … }
```

Access modifier:
- public
- private
- protected
- internal

Return Type:
Any valid type, including user-defined types, or void if the method has no return value.

Method Name:
Can be any valid C# identifier.

Parameter List:
This is an ordered list of comma-separated types/names of parameters that are passed to the method. It can be an empty list.

The body of the method – its executable code, its *implementation* – is not shown, but would be within the pair of curly braces.

Method Declaration Examples

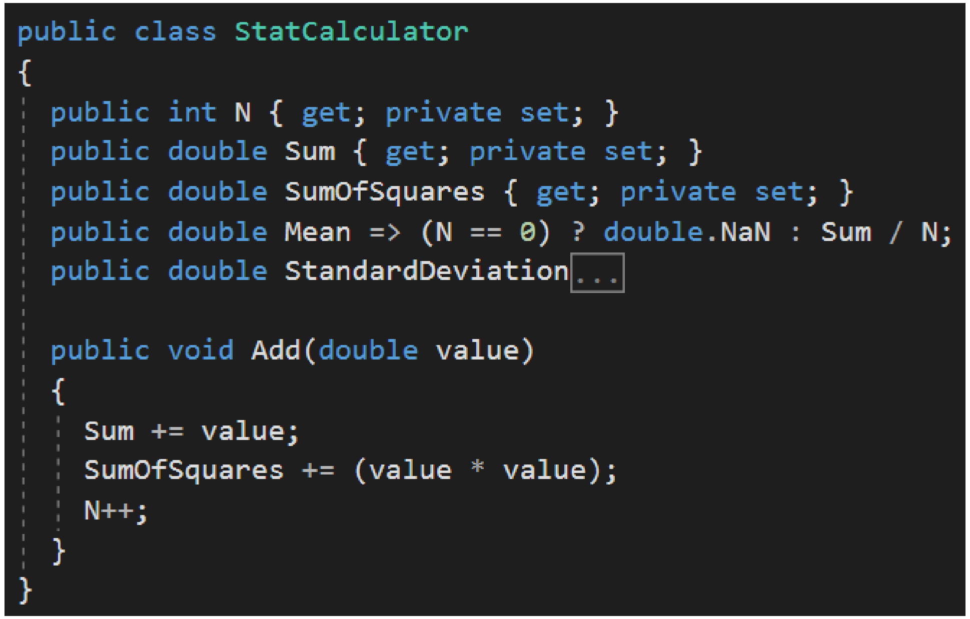
```
public bool AddWorksheet(string worksheetName);
public void AutoFitRow(int startRowIndex, int endRowIndex);
public DateTime GetCellValueAsDateTime(int rowIndex, int columnIndex);
public double GetCellValueAsDouble(string cellReference);
public double GetColumnWidth(int columnIndex);
public List<string> GetSheetNames();
public bool IsColumnHidden(string columnName);
```

Here are some additional methods that our spreadsheet application might declare.

Notice that some return types are built-in primitive types. Others (like List<string>) are classes themselves.

Methods, Classes, and State

As stated earlier, methods are always defined in the context of a class. Classes represent real things (like a spreadsheet), and very often the methods of a class modify that object and make orderly changes to its state. For example, suppose we have a class that calculates a mean and standard deviation of values that we give it. It might look like this:

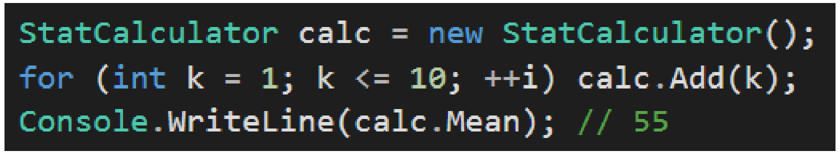


```
public class StatCalculator
{
    public int N { get; private set; }
    public double Sum { get; private set; }
    public double SumOfSquares { get; private set; }
    public double Mean => (N == 0) ? double.NaN : Sum / N;
    public double StandardDeviation...

    public void Add(double value)
    {
        Sum += value;
        SumOfSquares += (value * value);
        N++;
    }
}
```

Notice how the Add method makes well-defined changes to the state of the object so that the calculated Mean is correct.

We can create an *instance* of StatCalculator, then call its Add method:



```
StatCalculator calc = new StatCalculator();
for (int k = 1; k <= 10; ++i) calc.Add(k);
Console.WriteLine(calc.Mean); // 55
```

platform
By Per Scholas

More about Methods

Methods do not execute unless called. Typically, the Main method will create an object (maybe a Window object) and call one of its methods to make an application run.

Methods can make calls to other methods. Methods can call themselves, a technique called *recursion*.

Values returned from methods can be captured and stored in a variable and/or passed to other methods.

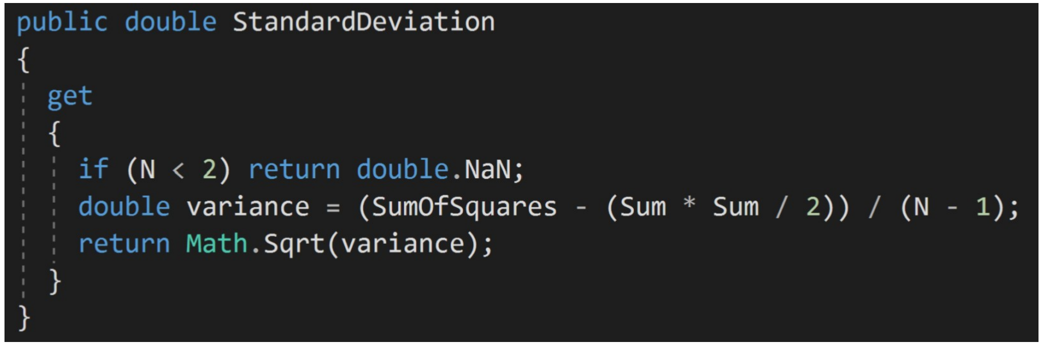
Methods declared with the static keyword do not require an object instance to be called. For example, we can call Console.WriteLine() without first instantiating a Console object because WriteLine is a static method.

Methods declared without the static keyword are instance methods and can only be called using an instance of the class.

Exercises

1. Open Visual Studio and create a new project named Numerics.Lib. The project type should be Class Library (.NET Framework). Name the solution “Analytics”.
2. Implement the StatCalculator class shown in the previous slide.
3. Create another project, a Console Application (.NET Framework) named Verifications. We will us it to verify that our code is correctly implemented.
4. In the Main method of our Verifications class, implement the code from the previous slide where we create a StatCalculator, call its methods, and write results to the Console.

This is the implementation of StandardDeviation.



```
public double StandardDeviation
{
    get
    {
        if (N < 2) return double.NaN;
        double variance = (SumOfSquares - (Sum * Sum / 2)) / (N - 1);
        return Math.Sqrt(variance);
    }
}
```

Method Signatures and Overloading

The signature of a method consists of its name and its parameter list. The access modifiers and return value are <u>not</u> part of the method signature.

The signatures of all methods declared on a class must be unique. They can have the same name, but if they do, then they must have distinct parameter lists.

Why is this? When you call a method, the compiler must know exactly which method to call, without ambiguity. If two methods have the same signature, they cannot be distinguished.

This is why a return type is not part of the signature – it cannot serve to unambiguously distinguish two methods with the same name and parameters.

Creating two or more methods with the same names but distinct parameter lists is called *overloading*.

Overloading Examples

These could all be valid overloads:

1. void Run()
2. int Run(int n)
3. void Run(double d)
4. void Run(int n, double d)
5. void Run(double d, int n)
6. void Run(Window w)

Whether these methods and parameters are well-named is another matter.

Method names and parameter names should be used to give a clear indication of what the method and parameter do.

Notice that 4 and 5 have the same argument types but in different order. This makes them distinct to the compiler.

This would not be a valid overload: double Run(int n)

Methods and Lambda Expressions

Methods with a concise implementation can be written using a lambda expression instead of a body. For example, we could overload our StatCalculator's Add method with a method accepting a List<double> like this:

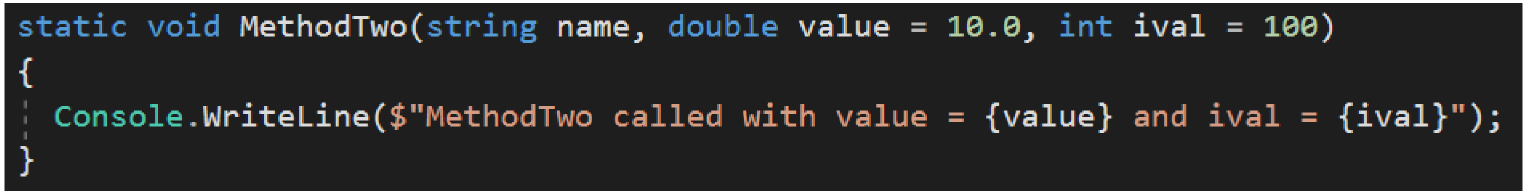
```
public void Add(List<double> values) => values.ForEach(v => Add(v));
```

There is no performance advantage to using a lambda expression, so the technique should be used only when it improves clarity of the code.

The **=>** operator is called the lambda operator.

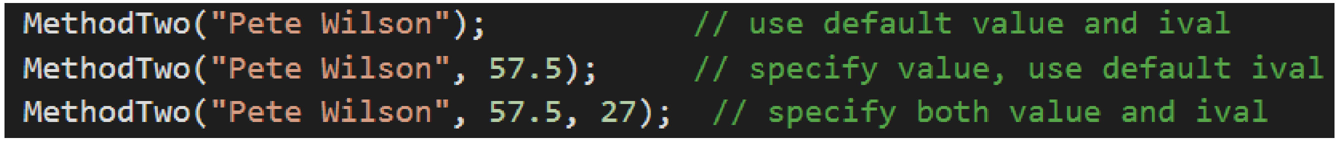
Default Parameters

Method declarations can define default values for parameters using the following syntax:



```
static void MethodTwo(string name, double value = 10.0, int ival = 100)
{
    Console.WriteLine($"MethodTwo called with value = {value} and ival = {ival}");
}
```

This method declares two default parameters, and can be invoked with any of the following statements:



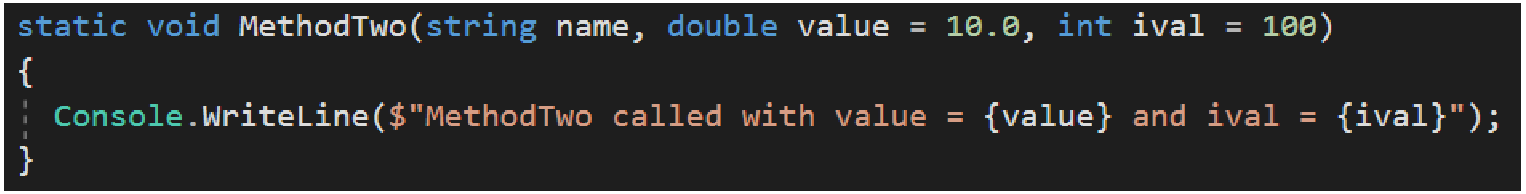
```
MethodTwo("Pete Wilson");              // use default value and ival
MethodTwo("Pete Wilson", 57.5);        // specify value, use default ival
MethodTwo("Pete Wilson", 57.5, 27);    // specify both value and ival
```

Default parameters must be declared at the end of the parameter list, after all required parameters.

platform
By Per Scholas

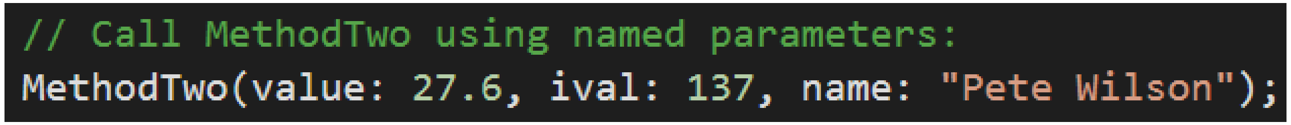
Using Named Parameters

Most often we call methods with parameters passed by order. C# also gives us the option of passing parameters by name. Given the same MethodTwo:



```
static void MethodTwo(string name, double value = 10.0, int ival = 100)
{
    Console.WriteLine($"MethodTwo called with value = {value} and ival = {ival}");
}
```

We can invoke MethodTwo with arguments in any order by naming them:



```
// Call MethodTwo using named parameters:
MethodTwo(value: 27.6, ival: 137, name: "Pete Wilson");
```

Open-ended Argument Lists

A method can offer the option of passing a variable number of arguments using the params keyword. For example:

```
static void MethodThree(string name, params double[] values)
{
    Console.WriteLine($"MethodThree called with values: {String.Join(",", values)}");
}
```

This allows us to call MethodThree with any number of values:

```
// Call MethodThree with 0 or more values:
MethodThree("Pete Wilson");
MethodThree("Pete Wilson", 1.0);
MethodThree("Pete Wilson", 1.0, 2.0, 10.0, 500.0);
```

We can also call MethodThree with an actual array of double:

```
// Call MethodThree with an actual array of double:
double[] values = new double[] { 1.0, 2.0, 3.0 };
MethodThree("Pete Wilson", values);
```

Exercises

Create a new class in Numerics.Lib named Numerics. Add the following methods to it:

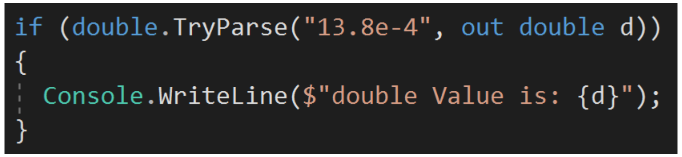
1. Write a method named MaxOf. It takes int two parameters and returns an int. The return value is the greater of the two parameters.
2. Write a method named MinOf that also takes two parameters and returns an int.
3. Write another method named MaxOf that takes two double parameters.
4. Write another method named MinOf that takes two double parameters.
5. Write another method named MaxOf that takes an open-ended list of integers.
6. Write another method named MaxOf that takes an open-ended list of doubles.
7. Write another method named MinOf that takes an open-ended list of integers.
8. Write another method named MinOf that takes an open-ended list of doubles.

out Parameters

Any method can return a value to the caller, but occasionally it is useful to also declare a value which can be modified by the method in tandem with the return value. Consider the method double.TryParse:

```
public static bool TryParse (string s, out double result);
```

We can invoke TryParse like this:



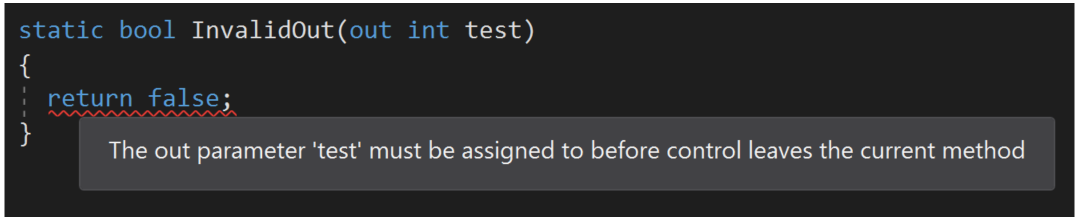
```
if (double.TryParse("13.8e-4", out double d))
{
    Console.WriteLine($"double Value is: {d}");
}
```

The method's return value is a boolean indicating success or failure. In case of success, the out parameter will contain the result of the parse. In case of failure, it will have a default value (0).

Using out Parameters

Usually when we pass parameters to a method, the method receives a copy of the value. With out parameters, the method receives a reference to the original value so that changes made to the reference are visible to the caller.

When a method declares an out parameter it makes a promise to the caller: the value of the out parameter will be set before the method returns. The compiler enforces this, as shown below.



```
static bool InvalidOut(out int test)
{
    return false;
}
```

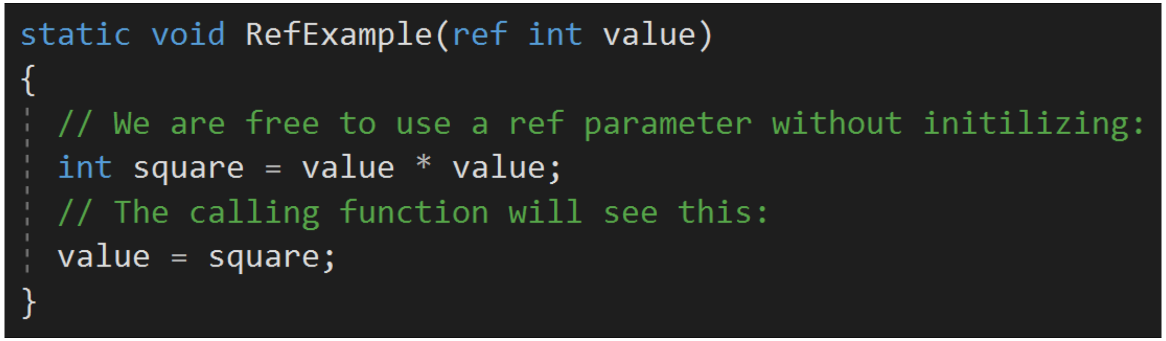
The out parameter 'test' must be assigned to before control leaves the current method

Two consequences of this promise are that uninitialized variables can be passed as out parameters, and a method cannot use an out parameter until it is initialized.

ref Parameters

Like out parameters, ref parameters are passed by reference, and changes made to a ref parameter are available to a caller after the method returns. They differ from out parameters in that

- Variables must be initialized before being passed as ref parameters.
- The method declaring the ref parameter makes no promise to set it before returning.
- The method declaring the ref parameter is free to use the parameter before assigning it a value.

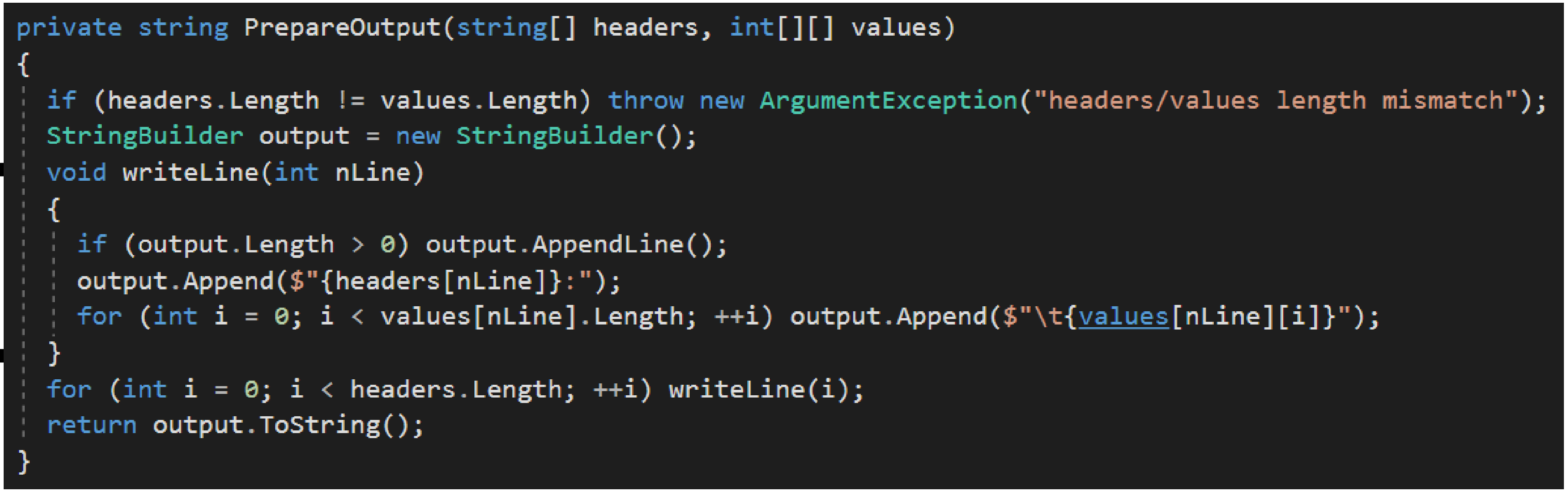


```
static void RefExample(ref int value)
{
    // We are free to use a ref parameter without initilizing:
    int square = value * value;
    // The calling function will see this:
    value = square;
}
```

Local Methods

Sometimes within a method we need to do something repetitive using locally-defined variables. C# allows us to define *local methods* which, like local variables, are accessible only within the method in which they are defined:

Local method

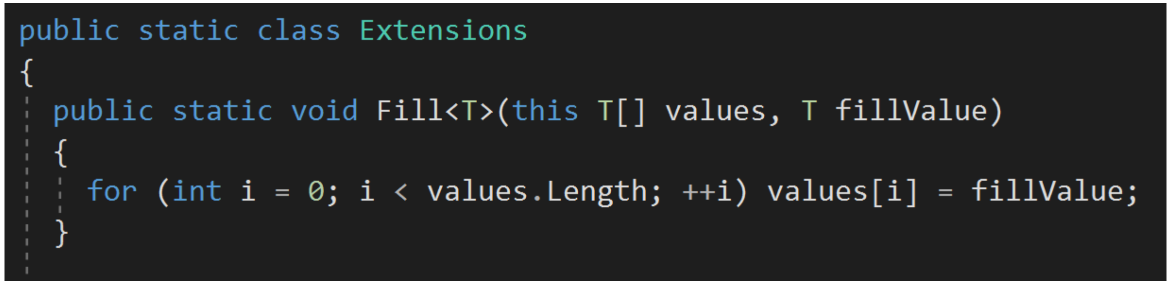


```
private string PrepareOutput(string[] headers, int[][] values)
{
  if (headers.Length != values.Length) throw new ArgumentException("headers/values length mismatch");
  StringBuilder output = new StringBuilder();
  void writeLine(int nLine)
  {
    if (output.Length > 0) output.AppendLine();
    output.Append($"{headers[nLine]}:");
    for (int i = 0; i < values[nLine].Length; ++i) output.Append($"\t{values[nLine][i]}");
  }
  for (int i = 0; i < headers.Length; ++i) writeLine(i);
  return output.ToString();
}
```

Notice how within the local method we can access the *output* variable defined in the outer scope.

Extension Methods

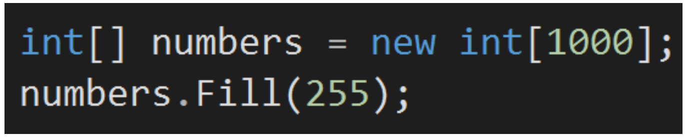
The C# language provides syntax for extending classes with new methods without modify any source code or recompiling. We could, for example, define an extension method to fill an array with a given value:



```
public static class Extensions
{
    public static void Fill<T>(this T[] values, T fillValue)
    {
        for (int i = 0; i < values.Length; ++i) values[i] = fillValue;
    }
```

Notice the **this** keyword before the first parameter.

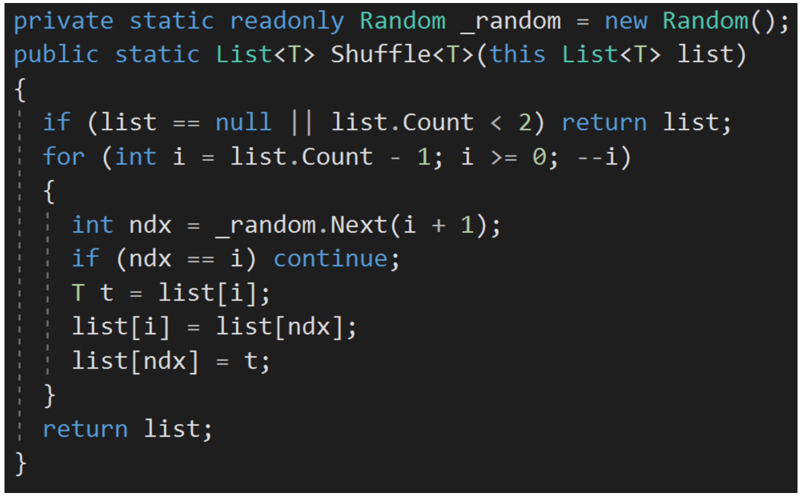
We can then invoke this method as if it were an instance member of the array class:



```
int[] numbers = new int[1000];
numbers.Fill(255);
```

Defining Extension Methods

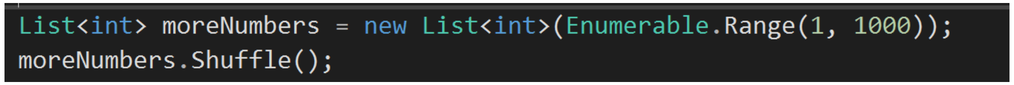
- The first parameter is preceded by ***this*** and determines the type to which the extension method applies.
- Extension methods are always static methods defined within a non-nested static class.
- Extension methods are visible anywhere that the enclosing class is visible.



```
private static readonly Random _random = new Random();
public static List<T> Shuffle<T>(this List<T> list)
{
  if (list == null || list.Count < 2) return list;
  for (int i = list.Count - 1; i >= 0; --i)
  {
    int ndx = _random.Next(i + 1);
    if (ndx == i) continue;
    T t = list[i];
    list[i] = list[ndx];
    list[ndx] = t;
  }
  return list;
}
```

This is an efficient shuffling algorithm. By defining it in a library, we can apply it to any List<T> just by referencing that library.

Like any extension method, we call it using the same syntax we use for an intance method.



```
List<int> moreNumbers = new List<int>(Enumerable.Range(1, 1000));
moreNumbers.Shuffle();
```

Topics Covered

- What is a Method?
- Re-using Code
- Method Declarations
  - Access Modifiers
  - Return Type
  - Name
  - Parameter List
- Methods and Classes
  - State Management
- Method Signatures
  - Uniqueness
  - Overloading
- Lambda Expressions
- Local Methods
- Default Parameters
- Named Parameters
- Open-ended Argument Lists
- ref and out Parameters
- Extension Methods

Exercises