# C# Classes

# Topics Covered

- What Are Classes?
- Class Examples
- Encapsulation
- Class Members
- Declaring a Class
- Access Modifiers
- Fields
- Constructors
  - Multiple Constructors
  - Chained Constructors
- this
- Methods
- Parameter Lists
- Properties
  - Lambda Syntax

# C# Classes

Classes are the foundation of any working C# application.  All C# code runs within the context of a class.  You may recall from our first look at a Console application that the Main method, the entry-point for the executable, was written within the Program class.

Well-designed classes represent re-usable units of work.  The String class gives you all the capabilities you need to work with string data.  The Console class gives you full control of a computer Console.  A Window class lets you create an application window and capture a user’s interactions with that window.

## Composing Functionality

When you create an application, you will bring multiple classes together to accomplish the functional objectives of that application.

You will write some of these classes yourself.

Many others will come from libraries – either the Framework Class Library, .NET Core, or the many public libraries that the developer community has made available.

You will discover that some of the classes you create have utility in several applications, so you will create your own class libraries.

# What Are Classes?

Classes model the real world or, for games and other exercises in imagination, the unreal world.

Medical records software will utilize classes to represent patients, pathologies and therapies.

A flight simulator will use classes to represent wind, weather and a jet.

That jet class will certainly be composed of other classes that represent its many components – wings, landing gear, hydraulics and cockpit controls.

When we define a class, we provide **fields** to hold the data, **properties** to provide access to the data, and **methods** to work with the data.  We might also provide **events** to notify consumers of the class of changes made to the data.  We may also provide **constructors** to initialize instances of the class when they are created.

# Class Examples: String

Strings represent an ordered collection of unicode characters, and the string class exposes properties and methods that are specialized for working with these characters.

| Method or Property Name | Purpose |
|---|---|
| Length | This property returns an integer corresponding to the # of characters in the string. |
| IndexOf(char c) | This method finds the index of the first occurrence of character c in the string. |
| Split(char[] delimiters) | This method takes an array of delimiter characters and splits the string into substrings using those delimiters. |
| ToUpper() | This method returns a string with identical characters but all in upper case. |
| this[int index] | This property (an indexer) returns a character at the given index in the string. |
| Substring(int startIndex, int length) | This method returns just a portion of the original string. |

## Class Examples: FileStream

The **FileStream** class represents a stream of bytes originating to or from a file.  This class allows us to perform common file operations, such as reading from the stream, writing to the stream, and asking how long the stream is.

| Method or Property Name | Purpose |
|---|---|
| long Length | This property returns the length of the file in bytes. |
| string Name | This property returns the name of the file. |
| long Position | This property gets or sets the current position in the file. |
| int Read (byte[] array, int offset, int count) | This method reads a series of bytes out of the file. |
| void Write (byte[] array, int offset, int count) | This method writes bytes into the file. |
| long Seek (long offset, SeekOrigin origin) | This method sets the current position in the file. |

platform
By Per Scholas

# Class Examples: Console

The Console class represents a console window and has numerous properties and methods specialized for that specific purpose.

| Method, Event or Property Name | Purpose |
|---|---|
| ConsoleColor BackgroundColor | This property lets you get or set the console background color. |
| int WindowWidth | This property lets you get or set the width of the console window. |
| string Title | This property lets you get or set the title of the console window. |
| WriteLine(string value) | This method lets you write a message to the console window. |
| SetWindowSize(int width, int height) | This method lets you set the width and height of the console window. |
| CancelKeyPress | This event informs you when the Ctrl-C key combination is pressed. |

## Encapsulation

Each of these classes specializes in just one thing and does that one thing very well.

A well-designed class exposes just enough information to make itself most useful. Exposing too much information (“implementation details”) is both a distraction and it introduces the possibility of a consumer of the class “hacking” and interfering with the class responsibilities.

This concept is called **Encapsulation**.

The C# language provides numerous keywords to support encapsulation.

# Class Members

| Member Name | Description |
|---|---|
| Constructors | Constructors are methods with the same name as the class.  They are invoked when objects of the class are created using the new keyword.  They contain code to initialize objects of the class. |
| Fields | Fields are typed variables defined within the class.  They are accessible throughout the class definition and, together with the automatic properties, represent the state of the class instances. |
| Properties | Properties are used to get and/or set the state of an instance of the class. |
| Methods | Methods are subroutines defined by the class that perform calculations relevant to the class.  They can take any number of arguments and can return a value. |
| Events | Events are delegates exposed by the class which allow other objects to be notified when something changes. |
| Indexers | Indexers are similar to Properties, but they allow a value to be retrieved based on an integer, string, or enum using array-like syntax.  They are typically implemented by classes that represent collections of other objects. |
| Operators | Classes may define custom operators like +, -, =, ==, etc.  They can also define custom casting operators. |
| Finalizer | A class may define one finalizer.  This is a special method that will be called by the Garbage Collector prior to reclaiming the heap memory used by an object. |

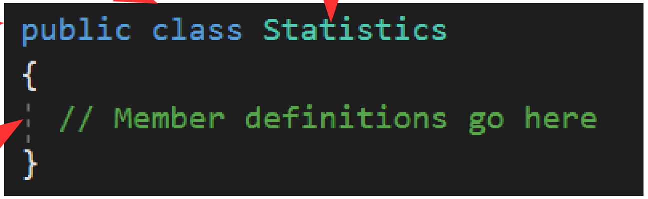
# Declaring a Class

Class names are alphanumeric, but cannot start with a number.  The best-practice is to use Pascal casing, though this is not enforced by the compiler.

Classes are declared using the class keyword.

Classes can optionally have an access modifier.  Classes declared without an access modifier are internal.

The body of a class is contained within a pair of curly braces.



```
public class Statistics
{
    // Member definitions go here
}
```

## C# Access Modifiers

| Name | Description |
|---|---|
| public | Class/member is visible and accessible everywhere. |
| private | Class/member is visible and accessible only to the declaring class or nested classes. |
| internal | Class/member is visible to all classes in the same assembly (project). |
| protected | Member is visible only to the class and to inheriting classes. |
| protected internal | Member is visible to all classes in the same assembly and to inheriting classes. |

Access modifiers are essential to **encapsulation**. Access modifiers allow us to hide the inner workings ("implementation details") of the class and expose only what consumers of the class need to know.

# Declaring Fields

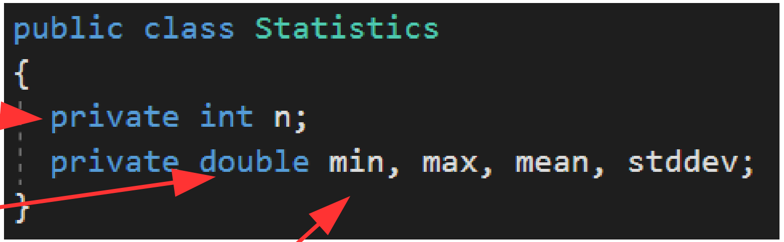
Format:  [access modifier] [readonly] typename fieldname [initialization];

Fields can optionally be given an access modifier.

If no modifier is given, the default is private.

Each field has a type.

These can be built-in types, types from the Framework Class Library, or user-defined types (e.g., other classes).



```
public class Statistics
{
  private int n;
  private double min, max, mean, stddev;
}
```

Field names are alphanumeric but cannot start with a digit.

They must be unique – no two fields can share the same name.

Best-practice is to use camel-casing for field names, though this is not enforced by the compiler.

The readonly keyword means that the field can only be set within a constructor.

# Constructors

Constructors are special methods that are invoked when an instance of a class is created using the new keyword.

Constructors can contain code to initialize the instance.

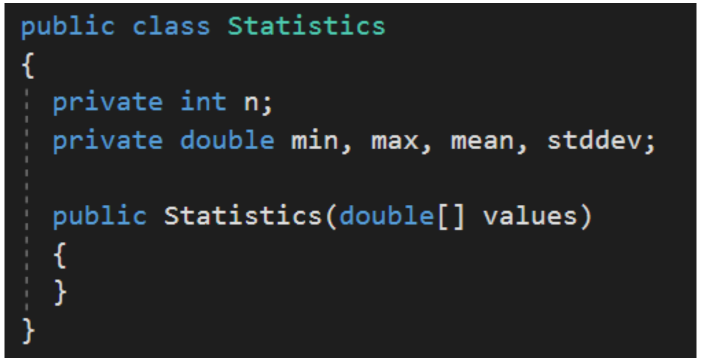
A constructor is called only once during the lifetime of an object.

Format:  [access modifier] ClassName([arguments]) { … }

If the access modifier is omitted, the default is private.

A public constructor that takes no arguments is called a default constructor.

If a class does not define any constructors, the compiler will generate a public default constructor.

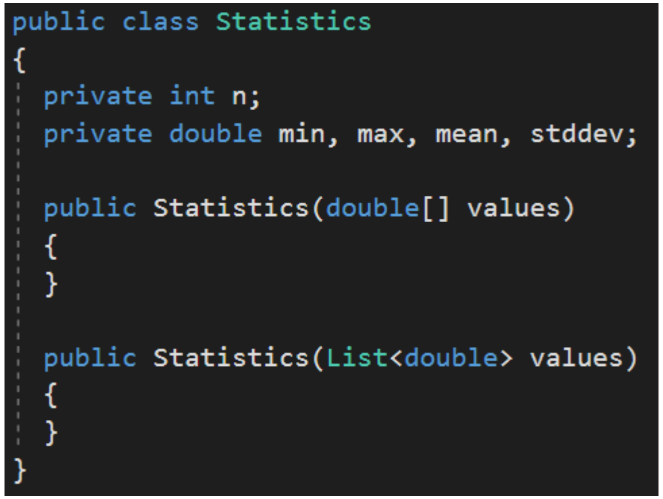


```
public class Statistics
{
    private int n;
    private double min, max, mean, stddev;

    public Statistics(double[] values)
    {
    }
}
```

# Multiple Constructors

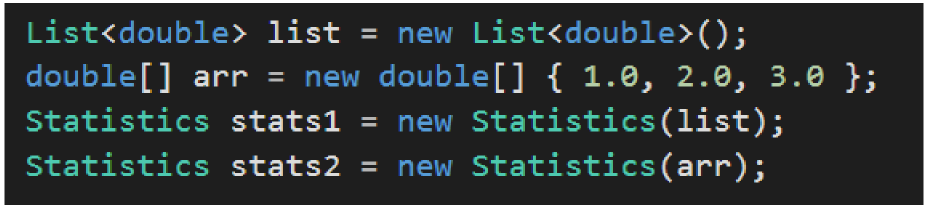
A class can define multiple constructors, but each constructor must have a unique parameter list.  As with methods, this technique is called *overloading*.

Here, our Statistics class has two constructors, one accepting a parameter of type double[], the other taking a parameter of type List<double>.



```
public class Statistics
{
    private int n;
    private double min, max, mean, stddev;

    public Statistics(double[] values)
    {
    }

    public Statistics(List<double> values)
    {
    }
}
```

This gives consumers of our class a choice of how to create a Statistics object:



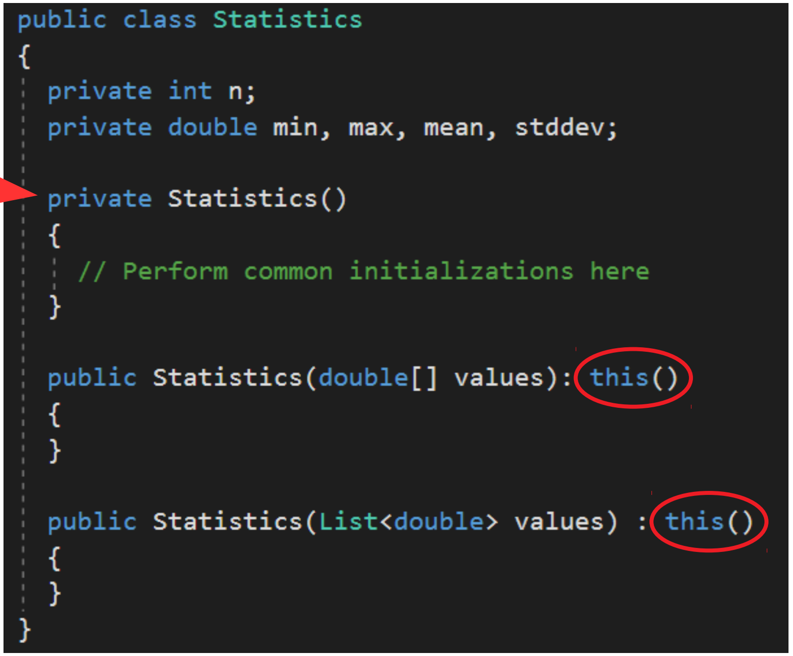
```
List<double> list = new List<double>();
double[] arr = new double[] { 1.0, 2.0, 3.0 };
Statistics stats1 = new Statistics(list);
Statistics stats2 = new Statistics(arr);
```

# Chained Constructors

One constructor can call another constructor using the this keyword as shown below.  This is most useful when one of the constructors performs common initializations needed by all the constructors.

Notice that one of these constructors is private. This means that it can only be accessed by the Statistics class.

Consumers of the Statistics class cannot use the private constructor.



```
public class Statistics
{
    private int n;
    private double min, max, mean, stddev;

    private Statistics()
    {
        // Perform common initializations here
    }

    public Statistics(double[] values): this()
    {
    }

    public Statistics(List<double> values) : this()
    {
    }
}
```

platform
By Per Scholas

# this

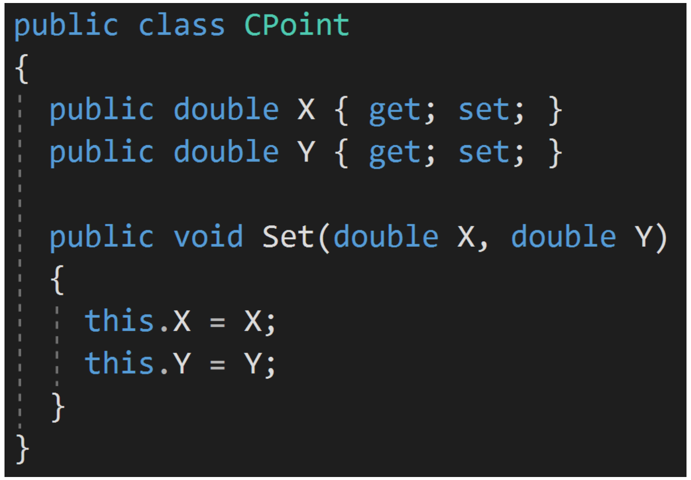
The **this** keyword represents the current instance.

**this** can be used only within instance methods and properties.

It cannot be used in static members because static members are accessed via the class name and there is no “current instance”.

As shown in the previous slide, **this** can be used to access other constructors.

As shown to the right, we can use **this** to differentiate parameters and members which have the same name.



```
public class CPoint
{
    public double X { get; set; }
    public double Y { get; set; }

    public void Set(double X, double Y)
    {
        this.X = X;
        this.Y = Y;
    }
}
```

# Methods

Somewhere, our class needs to perform calculations of the mean and standard deviation. We could put these calculations inside of our constructors, but that would require us to create duplicated code – one routine for double[] and one for List<double>.

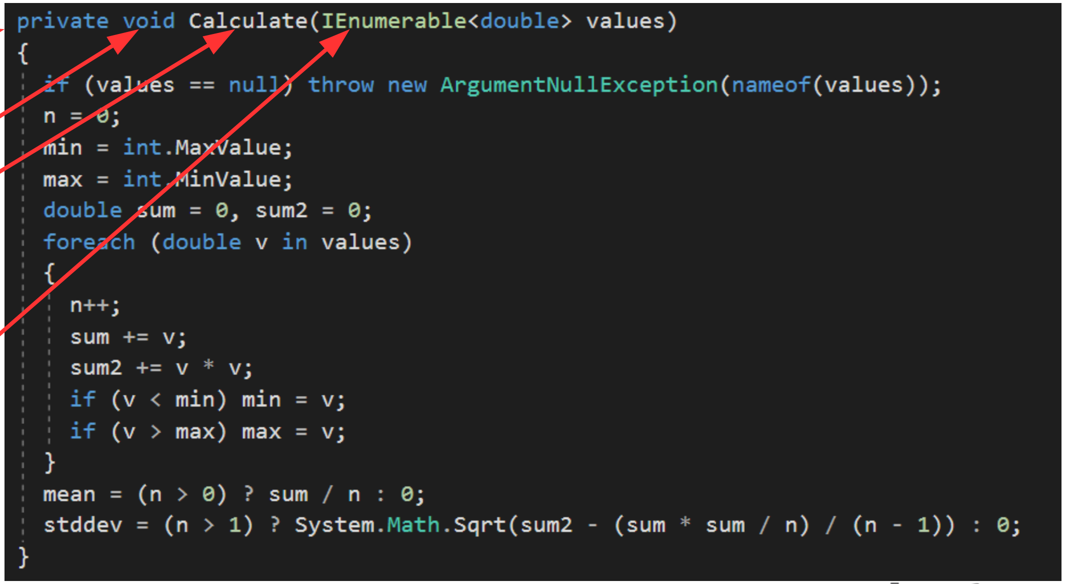
Lets define a method to do the calculations in one place:

This method is private, meaning it can be used only by the Statistics class.

This method does not return a value, thus the void keyword.

This method is named “Calculate”.

This method takes one parameter of type IEnumerable<double>.



```
private void Calculate(IEnumerable<double> values)
{
    if (values == null) throw new ArgumentNullException(nameof(values));
    n = 0;
    min = int.MaxValue;
    max = int.MinValue;
    double sum = 0, sum2 = 0;
    foreach (double v in values)
    {
        n++;
        sum += v;
        sum2 += v * v;
        if (v < min) min = v;
        if (v > max) max = v;
    }
    mean = (n > 0) ? sum / n : 0;
    stddev = (n > 1) ? System.Math.Sqrt(sum2 - (sum * sum / n) / (n - 1)) : 0;
}
```

# Parameter Lists

We find parameter lists not just in methods, but also in constructors, delegates, and indexers.

A _parameter list_ is a list of named parameters that can be passed into a method. Remembering that C# is a *strongly-typed language*, each parameter name is preceded by its type.

Each type and name in the list is separated by a comma.

For methods, constructors and delegates the entired list is enclosed in parentheses. For indexers, it is enclosed in square brackets.

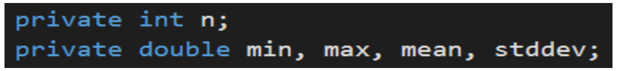
A few examples:

```
MethodName(int count, double incrementAmount)

MethodName(SqlDataReader dataSource, Stream dataSink)

IndexerName[string key]

MethodName()      // an empty parameter list
```

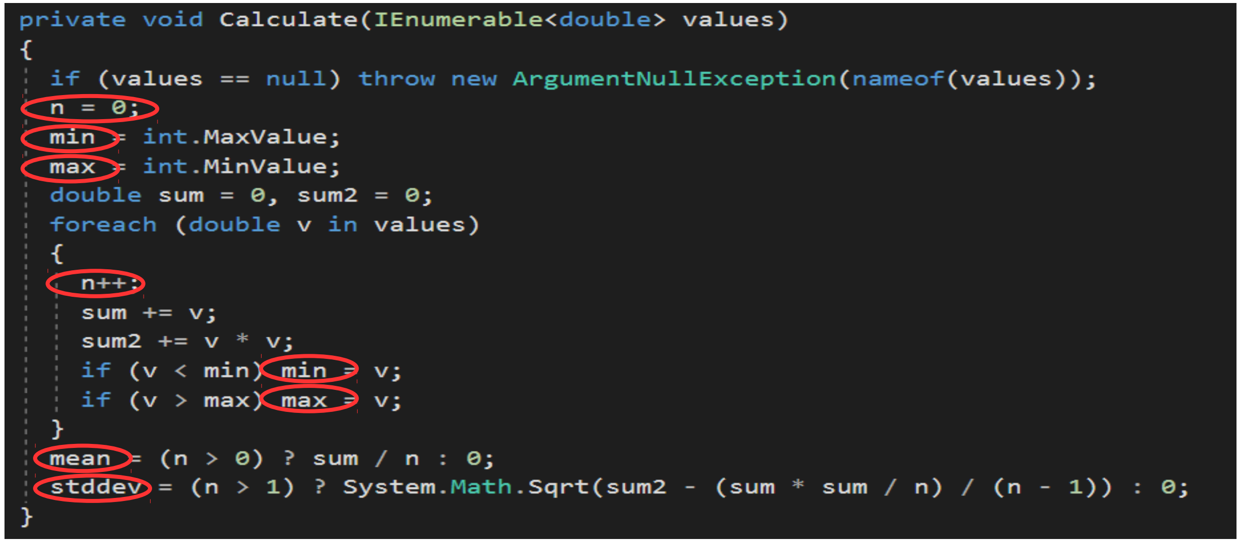
# Managing State

Let’s look a little more closely at the Calculate method.

Notice how we work with our fields.  When the method finishes, each of the fields is set. Our object now has a <u>state</u> that is expected for “statistics”.



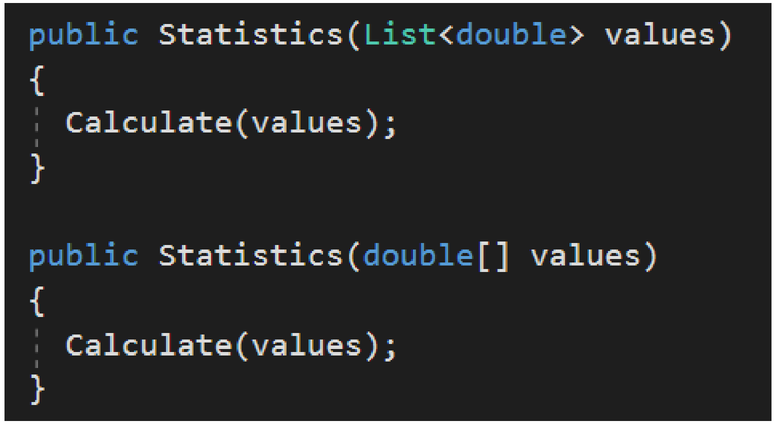
```
private int n;
private double min, max, mean, stddev;
```



```
private void Calculate(IEnumerable<double> values)
{
    if (values == null) throw new ArgumentNullException(nameof(values));
    n = 0;
    min = int.MaxValue;
    max = int.MinValue;
    double sum = 0, sum2 = 0;
    foreach (double v in values)
    {
        n++;
        sum += v;
        sum2 += v * v;
        if (v < min) min = v;
        if (v > max) max = v;
    }
    mean = (n > 0) ? sum / n : 0;
    stddev = (n > 1) ? System.Math.Sqrt(sum2 - (sum * sum / n) / (n - 1)) : 0;
}
```

## Calling Calculate

We can now call this method from each of our constructors:



```
public Statistics(List<double> values)
{
    Calculate(values);
}

public Statistics(double[] values)
{
    Calculate(values);
}
```

Never duplicate code!

Never duplicate code!

Never duplicate code!

Never duplicate code!

Never duplicate code!

Never duplicate code!

Never duplicate code!

Never duplicate code!

Never duplicate code!

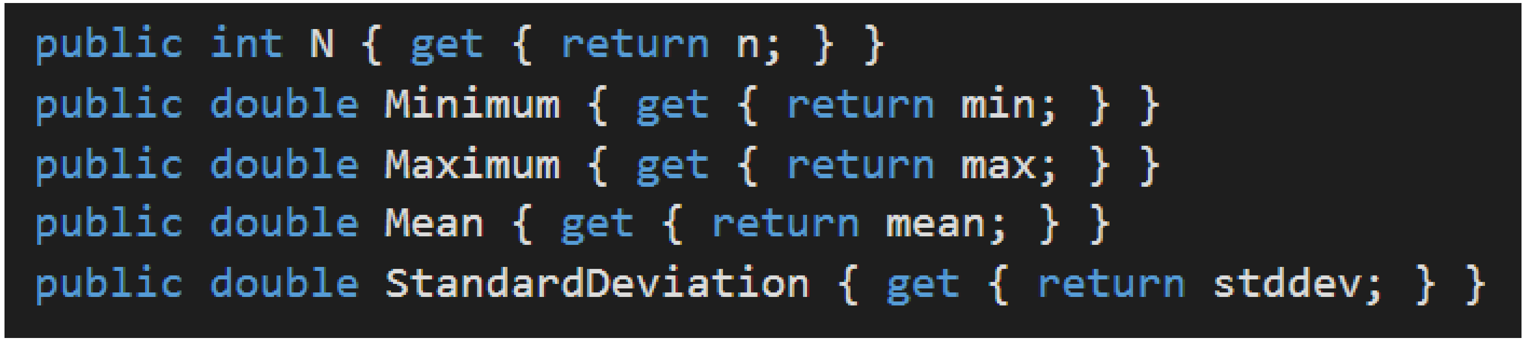
platform By Per Scholas

# Properties

Our Statistics class has these wonderfully initialized fields, but it is useless because nobody can see them – they are all private!  We don't want to make them public because then consumers of our class could change their values and put a Statistics object into an inconsistent state.

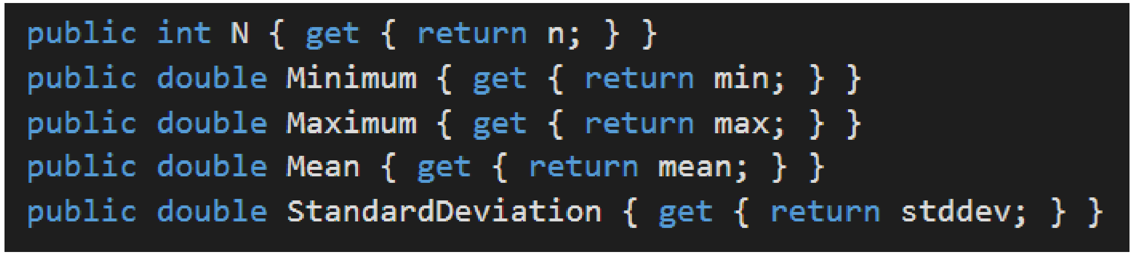
C# provides another type of class member which can be read-only: **Properties**.

Below we have defined five properties which expose the values of our five fields in a read-only manner:



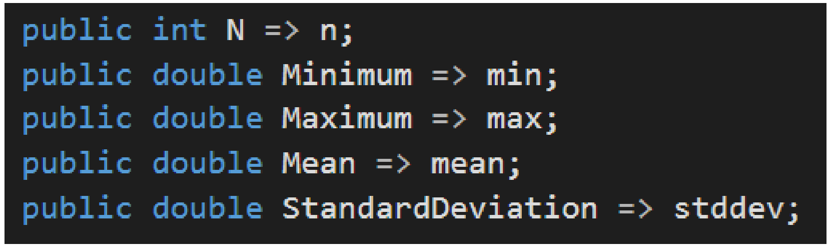
```
public int N { get { return n; } }
public double Minimum { get { return min; } }
public double Maximum { get { return max; } }
public double Mean { get { return mean; } }
public double StandardDeviation { get { return stddev; } }
```

# Properties: Lambda Syntax



```
public int N { get { return n; } }
public double Minimum { get { return min; } }
public double Maximum { get { return max; } }
public double Mean { get { return mean; } }
public double StandardDeviation { get { return stddev; } }
```

The properties shown above use get/return syntax.  The properties shown below using lambda syntax are functionally identical:



```
public int N => n;
public double Minimum => min;
public double Maximum => max;
public double Mean => mean;
public double StandardDeviation => stddev;
```

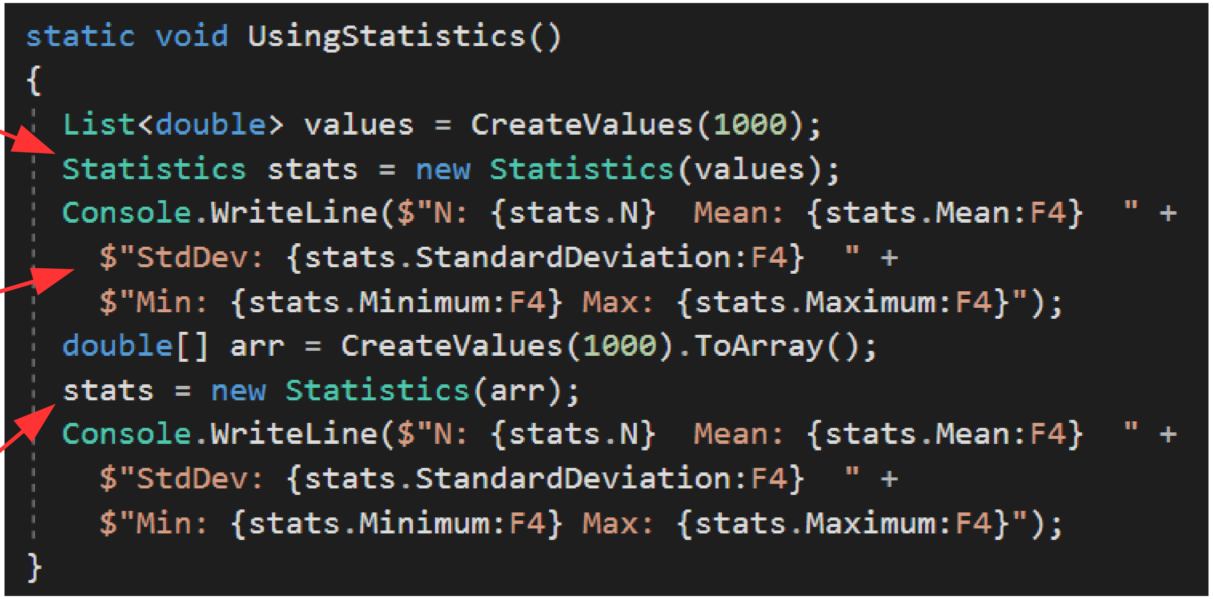
Lambda syntax is attractive for it conciseness.  It cannot be used for settable properties, and is not well-suited for read-only properties that require more than one code statement.

# Using Statistics

Here we create an object of type Statistics using the constructor that accepts List<double>.

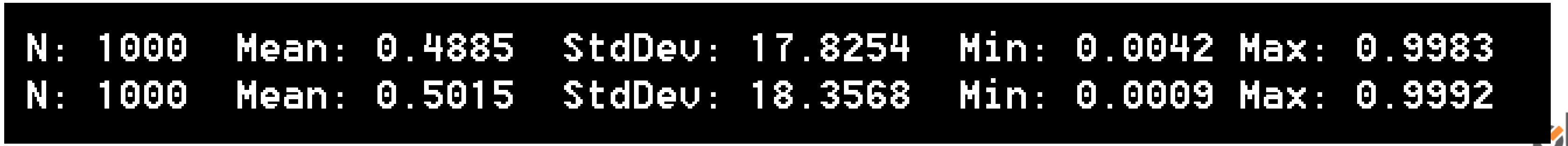
Here use our stats object's properties to write them to the Console.

Here we create an object of type Statistics using the constructor that accepts double[].



```
static void UsingStatistics()
{
    List<double> values = CreateValues(1000);
    Statistics stats = new Statistics(values);
    Console.WriteLine($"N: {stats.N}  Mean: {stats.Mean:F4}  " +
      $"StdDev: {stats.StandardDeviation:F4}  " +
      $"Min: {stats.Minimum:F4} Max: {stats.Maximum:F4}");
    double[] arr = CreateValues(1000).ToArray();
    stats = new Statistics(arr);
    Console.WriteLine($"N: {stats.N}  Mean: {stats.Mean:F4}  " +
      $"StdDev: {stats.StandardDeviation:F4}  " +
      $"Min: {stats.Minimum:F4} Max: {stats.Maximum:F4}");
}
```

This code generates the following output:



```
N: 1000  Mean: 0.4885  StdDev: 17.8254  Min: 0.0042 Max: 0.9983
N: 1000  Mean: 0.5015  StdDev: 18.3568  Min: 0.0009 Max: 0.9992
```

# Topics Covered

- What Are Classes?
- Class Examples
- Encapsulation
- Class Members
- Declaring a Class
- Access Modifiers
- Fields
- Constructors
  - Multiple Constructors
  - Chained Constructors
- this
- Methods
- Parameter Lists
- Properties
  - Lambda Syntax

## Questions

- Classes represent re-usable units of work.  What does that mean?
- What does it mean to compose functionality from classes?
- When you write an application, where will the classes you use come from?
- What is encapsulation?
- What are the 8 different kinds of class members?
- What are *access modifiers*?  What does the **protected** access modifier do?
- What are *fields*?  What are *constructors*?
- What does **this** represent?
- What is a parameter list, and where are they used?
- What is an object's *state*?

# Questions II

The Calculate method of the Statistics class has the signature:

```
Calculate(IEnumerable<double> values);
```

We can call Calculate with arguments of type double[] and List<double>.

Why?

Knowing this, how can we simplify our Statistics class' constructors?