The Learning House

427 S. 4th Street #300

Louisville KY 40202

Classes in C#
Part 2

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

Lesson Goals

- Explore additional members of classes (we covered fields and methods)
- Constants
- Statics
- Constructors
- Partials

Types of Class Members
(We will cover the orange ones)

Data Member Store Data

- Fields
- Constants

Function Members Execute Code

- Methods
- Properties
- Constructors
- Operators
- Indexers
- Events
- Destructors

Constants

- Member constants are declared and initialized in line.
- They cannot be changed after being set. In other words… they are constant.
- Declared like other fields, but using the *const* keyword.

```
class Shape
{
    public const double Pi = 3.14;
}
```

Static Members

A static member is shared by all instances of the class. All instances access the same memory location.

Statics can be classes, fields, methods, properties, operators, events, or constructors.

Because statics do not belong to any single instance, they can not be used with indexers, destructors, or types other than classes.

What Happens When We Declare Something Static?

When a class or member is declared static, a single copy is automatically instantiated before it is used.

The single copy is given the same name as the class type.

The key idea behind static is that there is only one instance… ever.

By the way, constants are implicitly static.

Why Should I Make Something Static?

Static members are convenient when:

1. There is no benefit to having multiple instances
2. There is a utility member that does not require object state

Instantiating a new object does have a small cost. So, if something meets those two criteria, a static member could be more efficient.

Static Example

The Math class in C# is a good example of static methods in use.

1. There is no benefit in having multiple copies of a method that calculates powers.
2. We have no need for object state, because all calculations are done in the method and a value is returned (but not stored in the method).

If Power() was not static, we would have to instantiate a new object every method we wanted to use it in… and all of those objects would have the same state and same behavior as each other. So, it would be wasteful.

```
class Program
{
    static void Main()
    {
        Console.WriteLine("2 to the 3rd power is: {0}",
            MyMathUtilities.Power(2, 3));

        Console.ReadLine();
    }
}

public class MyMathUtilities
{
    public static int Power(int num, int power)
    {
        int result=1;

        for (int i = 0; i < power; i++)
        {
            result *= num;
        }

        return result;
    }
}
```

Arguments Against Static

Static classes…

- Cannot be inherited
- Cannot implement interfaces
- Cannot be automatically serialized (converted to XML/JSON)
- Do not get garbage collected

Properties and Encapsulation

As mentioned earlier, encapsulation allows us to hide the internals of a class and force other callers to access internals through methods we define.

This allows us to control access to our object's state and prevent invalid data from being entered.

Justification For Properties Part 1

Let's assume we have a class definition that represents a period of time. The class should be able to easily convert between hours, minutes, and seconds.

We want to make sure that negative numbers are not entered as seconds. So, we create a private field and only allow access to it through a method.

```
class TimePeriod
{
    private double _seconds;

    public double Get_Seconds()
    {
        return _seconds;
    }

    public void Set_Seconds(double value)
    {
        if (value < 0)
            throw new ArgumentException(
                "Seconds can not be negative", "value");

        _seconds = value;
    }
}
```

Justification For Properties Part 2

If we wanted to allow for minutes, we could add additional private fields.

However, this causes a problem. Now, when we change one value, we have to be sure to change the others.

Notice how the code is going to get more complicated for each additional time period we add.

What if we forgot to update all of the methods when we changed a value?

```
class TimePeriod
{
    private double _seconds;
    private double _minutes;


    public double Get_Seconds()
    {
        return _seconds;
    }

    public void Set_Seconds(double value)
    {
        if (value < 0)
            throw new ArgumentException(
                "Seconds can not be negative", "value");

        _seconds = value;
        _minutes = value*60;
    }

    public double Get_Minutes()
    {
        return _minutes;
    }

    public void Set_Minutes(double value)
    {
        if (value < 0)
            throw new ArgumentException(
                "Hours can not be negative", "value");

        _minutes = value;
        _seconds = value/60;
    }
}
```

Justification For Properties Part 3

A better solution to this issue is to only store seconds, and make minutes and hours calculated members such that, whenever seconds is changed, the other members use that in their calculation to give the appearance of automatic updates.

These can also be referred to as read-only members because you cannot write to minutes or hours, only seconds.

So this makes our code a bit shorter, but it still has some issues.

```
class TimePeriod
{
    private double _seconds;


    public double Get_Seconds()
    {
        return _seconds;
    }

    public void Set_Seconds(double value)
    {
        if (value < 0)
            throw new ArgumentException(
                "Seconds can not be negative", "value");

        _seconds = value;
    }

    public double Get_Minutes()
    {
        return _seconds*60;
    }

    public double Get_Hours()
    {
        return _seconds*60*60;
    }
}
```

Justification For Properties Part 4

The first issue is that method syntax just does not read well. For example: when we want to set the seconds field, we have to do it using method syntax and pass a parameter.

With variables, we are used to using the equals sign to assign values, so method syntax breaks the convention we're used to.

It's not all that nice to look at…

* this is how Java does it

```
static void Main()
{
    TimePeriod t = new TimePeriod();
    t.Set_Seconds(10);

    Console.WriteLine("seconds: {0}", t.Get_Seconds());
    Console.WriteLine("minutes: {0}", t.Get_Minutes());
    Console.WriteLine("hours: {0}", t.Get_Hours());

    Console.ReadLine();
}
```

Justification For Properties Part 5

A second, but arguably more annoying issue, is that we have field data that doesn't have any rules around it. This makes it difficult to encapsulate (remember, that's good practice).

A great example is text data on an object like an employee:

What if this class had a dozen fields? We would need to create a dozen private variables, a dozen Get methods, and a dozen Set methods.

This is really tedious!

```
class Employee
{
    private string _firstName;
    private string _lastName;

    public string Get_FirstName()
    {
        return _firstName;
    }

    public void Set_FirstName(string value)
    {
        _firstName = value;
    }

    public string Get_LastName()
    {
        return _lastName;
    }

    public void Set_LastName(string value)
    {
        _lastName = value;
    }
}
```

Properties

Property syntax solves the issues we just showed by providing shortcut syntax for creating private fields and Get/Set methods when the code is compiled.

- In their usage, they look like a variable assignment, so the syntax is cleaner.
- Unlike a field, they can execute code and have two accessor methods available to them (get and set)

Properties by Example Part 1

Let's start with the simplest scenario. This employee class has access methods for all of its fields and requires no special logic.

Be mindful that the property syntax for FirstName generates the same code as the commented out methods and private backing field.

It's easy to see why C# developers prefer to write 1 line of code instead of 11.

```
class Employee
{
    /*
    private string _firstName;

    public string Get_FirstName()
    {
        return _firstName;
    }

    public void Set_FirstName(string value)
    {
        _firstName = value;
    }
    */

    public string FirstName { get; set; }
}
```

Properties by Example Part 2

What about our data validation scenario (e.g. seconds cannot be negative) to protect private fields?

Property syntax can do that too! However, this time, you do need to create a private backing field.

When looking at this code, remember that get and set compile to methods: get returns a double value, and set receives a double value.

The dark blue value keyword only exists in set methods, and it represents the data being passed in.

We can validate it and then store it in our private backing field.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set
        {
            if (value < 0)
                throw new ArgumentException(
                    "Seconds can not be negative", "value");

            _seconds = value;
        }
    }
}
```

Properties by Example Part 3

Properties can handle calculated members as well. If we omit the set statement from the property definition, the property will be read-only.

Also, if we want to create a property that can only be written to by the class that contains it, we can mark set as private!

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set
        {
            if (value < 0)
                throw new ArgumentException(
                    "Seconds can not be negative", "value");

            _seconds = value;
        }
    }

    public double Minutes { get { return _seconds*60; } }
    public double Hours { get { return _seconds*60*60; } }
}
```

Properties by Example Part 4

Once we have declared properties, we no longer have to use the “ugly” method syntax.

The dot notation for reading and writing data from properties looks exactly like working with any other variable.

This is much nicer to look at and the use of the equal sign when assigning data is more intuitive.

```
static void Main()
{
    TimePeriod t = new TimePeriod();
    t.Set_Seconds(10);

    Console.WriteLine("seconds: {0}", t.Get_Seconds());
    Console.WriteLine("minutes: {0}", t.Get_Minutes());
    Console.WriteLine("hours: {0}", t.Get_Hours());

    Console.ReadLine();
}
```

```
static void Main()
{
    TimePeriod t = new TimePeriod();
    t.Seconds = 10;

    Console.WriteLine("seconds: {0}", t.Seconds);
    Console.WriteLine("minutes: {0}", t.Minutes);
    Console.WriteLine("hours: {0}", t.Hours);

    Console.ReadLine();
}
```

Recap: Property Declarations

The set accessor

- Always has a return type of void
- Has a special parameter called value, which represents the data coming into the property

The get accessor

- Has no parameters
- Will always return the same type as the property
- All paths through the code implementation of the get block must include a return statement that returns a value of the property type

Recap: Naming Conventions

Generally, developers want our property names to be *Pascal Cased*

- First letter capitalized
- MyPropertyName

Generally, backing fields should be *Camel Cased*

- First letter lowercase
- myBackingField

Some old school devs, like me, prefer to use camel case with an underscore for backing fields.

FAQ: If you're not going to write an accessor, why not just use a field?

- Because some modern two-way data binding technology in .NET (WPF, Silverlight, ASP.NET MVC, etc) can only perform their magic on properties
  - Two-way data binding is an automated way to keep your user interface in sync with the underlying class instances.
  - We used to have to write all that code by hand… it sucked.

Constructors

- An instance constructor (constructor for short) is a special method that is executed whenever a new class instance is created.
  - It is used to initialize the state of the class instance
  - They can have accessors, but most of the time they will be public

Declaring Constructors

- Constructors are special methods that are called when the class is instantiated using the new() keyword.
  - There is no return type
  - The name must be the same as the class name

```
class Number
{
    public Number()
    {
    }
}
```

Empty constructor is redundant. The compiler generates the same by default.

Constructor Usage Illustrated

- We use constructors to set up the state of a class instance so it is valid to use

```
private static void Main()
{
    var n = new MyClass();

    Console.WriteLine(n.TimeOfCreation);
}

class MyClass
{
    public DateTime TimeOfCreation { get; set; }

    public MyClass()
    {
        TimeOfCreation = DateTime.Now;
    }
}
```

Fun with constructors

DEMO: OVERLOADING CONSTRUCTORS

Default Constructors

- If no constructor is defined, the compiler creates a default constructor with no parameters.
- If you define a parameterized constructor but not a default one, then the compiler forces the use of the parameterized one.

Static Constructors

- Static constructors initialize items at the class level (not instance).
- Generally, they initialize static fields of a class.
- There can only be one static constructor in a class, and it *cannot* take parameters or have access modifiers.
- A class cannot have both static and instance constructors.
- You cannot directly call static constructors; instead, the system decides when to call them (before they are used in code).

Fun with static constructors

DEMO: STATIC CONSTRUCTORS

Object Initializers

Object initializer syntax is a way to shorthand field and property values.

The fields and properties being initialized must be accessible to the code creating the object.

Initialization occurs *after* the constructor. Be careful if you set a value in the constructor and then also put it in the initializer.

To initialize data, simply put a code block after the new statement and comma-separate the properties and values you wish to fill in.

```
// long hand syntax
Employee e1 = new Employee();
e1.FirstName = "Joe";
e1.LastName = "Schmoe";

// initializer syntax
Employee e2 = new Employee
{
    FirstName = "Jane",
    LastName = "Doe"
};
```

The readonly Modifier

- A field can be declared with a readonly modifier.
  - Unlike a const, a readonly field can have its value set in the constructor.
  - Once set, it cannot be changed.

```
class Rectangle
{
    public readonly int NumberOfSides;

    public Rectangle()
    {
        NumberOfSides = 4;
    }
}
```

The this Keyword

- The this keyword, used in a class, is a reference to the current instance.
  - Since it operates on the current instance, you can't use it with statics!
- We use it to distinguish between class members and local variables or parameters.

```
class Person
{
    public string Name { get; set; }

    public Person(string Name)
    {
        this.Name = Name;
    }
}
```

Partial Classes

- A class declared partial will be recombined by the compiler.
- The declarations can be in the same file or different files.
- Visual Studio uses this to separate generated code from the code you write (ex: in ASP.NET).

Fun with partials

DEMO

In Conclusion

- Constants can handle data that shouldn't change.
- Statics can be used to share members without having to instantiate objects.
- Properties provide short hand syntax for encapsulation.
- Constructors can be used to initialize objects.
- readonly can be used when data is constant after being initialized (usually in the constructor).
- this refers to the current instance.
- Partial classes can be used to separate class definition parts to be recombined when compiled.