

# Core C# Programming: Encapsulation and Classes

CSIS 3540

Client Server Systems

Class 02

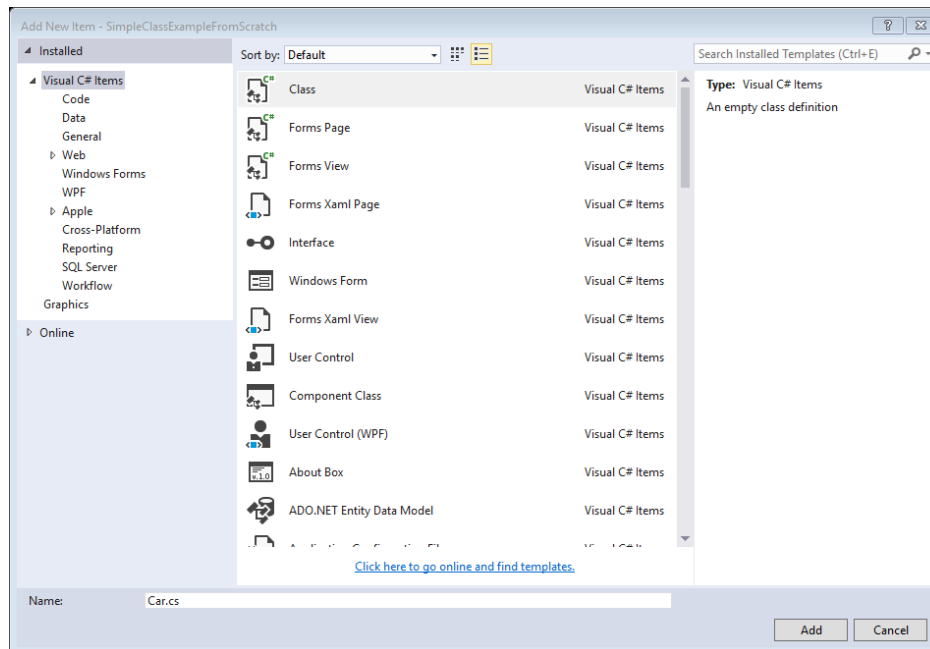
©Michael Hrybyk and others  
NOT TO BE REDISTRIBUTED

# Topics

- Class Type
- Constructors
- This keyword
- Static keyword
- Access modifiers
- Encapsulation
- Automatic properties
- Object initialization
- Constants
- Partial Classes

# Classes

- A class is a type definition
  - Consists of
    - Field data (or member variables)
    - Methods – functions/procedures that operate on the data
- Create in VS using Project->Add New Item, brings up window below
  - Name your class, it will be placed in a .cs file within the project
  - Can also simply add it to a file



# Example

- See **SimpleClassExampleFromScratch**
- Field data
- Methods
- Declaration

```
using System;
namespace SimpleClassExampleFromScratch
{
    class Program
    {
        static void Main(string[] args)
        {
            Car myCar = new Car();
            myCar.petName = "Corvette";
            for (int i = 0; i < 10; i++)
            {
                myCar.SpeedUp(i);
                myCar.PrintState();
            }
            Console.ReadLine();
        }
    }
}
```

```
using System;
namespace SimpleClassExampleFromScratch
{
    class Car
    {
        // The 'state' of the Car.
        public string carName;
        public int currentSpeed;

        // The functionality of the Car.
        public void PrintState()
        {
            Console.WriteLine("{0} is going {1} MPH.", carName,
                currentSpeed);
        }

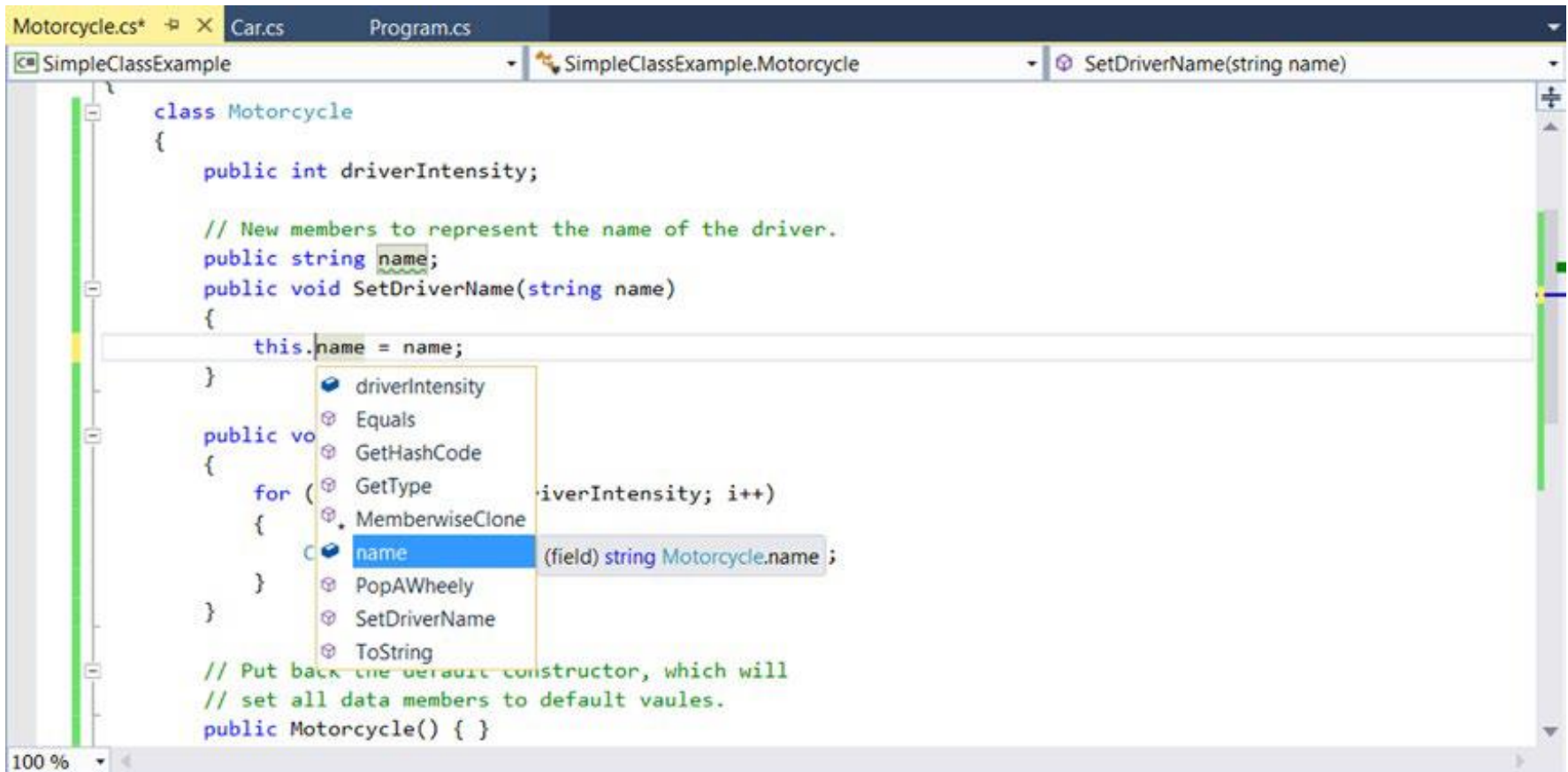
        public void SpeedUp(int delta)
        {
            currentSpeed += delta;
        }
    }
}
```

# Constructors

- Objects are “created” in memory when the new operator is invoked.
- Classes can define constructors to initialize
- Default constructor is called with the default new invocation
  - `Car myCar = new Car();`
    - Would call the default constructor method `Car()` defined within the class.
    - If there is none, fields are simply given general default values
- Example
  - `SimpleClassExample`

```
public Car()  
{  
    carName = "Chuck";  
    currentSpeed = 10;  
}
```

# Use of **this** keyword



The screenshot shows a Visual Studio editor window with the following tabs: Motorcycle.cs\*, Car.cs, and Program.cs. The active file is Motorcycle.cs, and the current view is the SimpleClassExample.Motorcycle class. The method SetDriverName(string name) is selected. The code in the class is as follows:

```
class Motorcycle
{
    public int driverIntensity;

    // New members to represent the name of the driver.
    public string name;
    public void SetDriverName(string name)
    {
        this.name = name;
    }

    public void ...
    {
        for (
        {
            driverIntensity; i++)
            {
                name (field) string Motorcycle.name ;
            }
        }
    }

    // Put back the default constructor, which will
    // set all data members to default vaules.
    public Motorcycle() { }
```

A context menu is open over the `this.name` property access in the `SetDriverName` method. The menu items are:

- driverIntensity
- Equals
- GetHashCode
- GetType
- MemberwiseClone
- name (field) string Motorcycle.name ;
- PopAWheely
- SetDriverName
- ToString

# Constructors with arguments

- Define unique invocations of constructors
- Use of chaining – notice use of `this` keyword to keep references clean

```
// Single constructor using optional  
args.  
public Motorcycle(int intensity = 0,  
string name = "")  
{  
    if (intensity > 10)  
    {  
        intensity = 10;  
    }  
    driverIntensity = intensity;  
    driverName = name;  
}
```

```
static void MakeSomeBikes()  
{  
    // driverName = "", driverIntensity = 0  
    Motorcycle m1 = new Motorcycle();  
    Console.WriteLine("Name= {0}, Intensity= {1}",  
        m1.driverName, m1.driverIntensity);  
  
    // driverName = "Tiny", driverIntensity = 0  
    Motorcycle m2 = new Motorcycle(name: "Tiny");  
    Console.WriteLine("Name= {0}, Intensity= {1}",  
        m2.driverName, m2.driverIntensity);  
  
    // driverName = "", driverIntensity = 7  
    Motorcycle m3 = new Motorcycle(7);  
    Console.WriteLine("Name= {0}, Intensity= {1}",  
        m3.driverName, m3.driverIntensity);  
}
```

# static keyword

- static members
  - deemed (by the class designer) to be so commonplace that there is no need to create an instance of the class before invoking the member.
- Called a **utility class**
  - does not maintain any object-level state and is not created with the new keyword.
  - exposes all functionality as class-level (aka static) members.
  - Example
    - `Console c = new Console(); c.WriteLine(); // ERROR!`
- Console, Math are two examples
- See SimpleUtilityClass

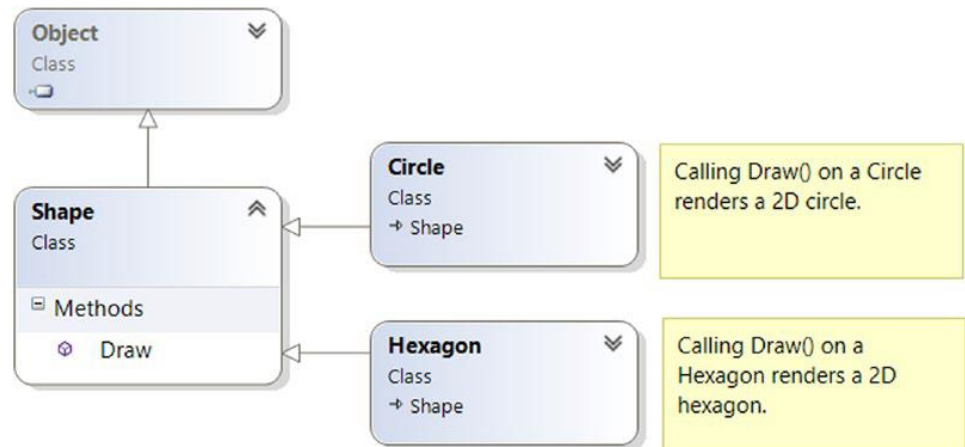
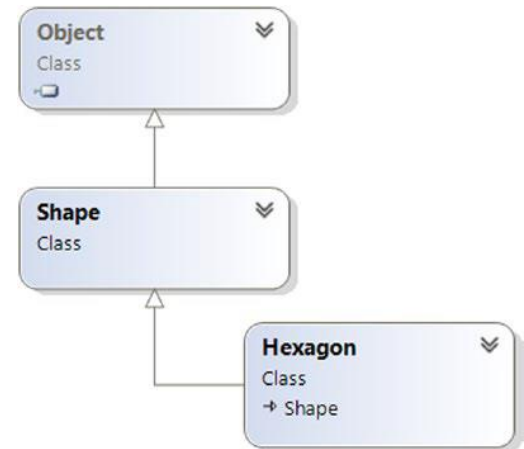


# static can be applied to ...

- Data of a class (fields)
  - Field that can carry across all objects of that class
  - Example: interest rate, set once for the first object, everyone else uses it unless updated.
- Methods of a class
  - Get/Set static
  - Operation associated with a class but carries across all instances – a utility method
- Properties of a class
  - Combination of static data and static methods
- A constructor
  - Initializes values of static data
  - No parameters
  - Invoked once when first object is created
- The entire class definition
  - Covers all members and methods
  - Creates a true utility class like Console
- In conjunction with the C# using keyword
  - Allows one to avoid the class name for utility functions
  - Example
  - using static System.Console;
  - Means that you only need to use WriteLine() without a prefix!
- Example
  - See **StaticDataAndMembers**

# Pillars of Object Oriented Programming and Design

- Encapsulation
  - Hides data
  - Uses access mechanisms (accessor, mutator) -> (get,set)
- Inheritance
  - Build classes from classes
  - Focus on relationships
- Polymorphism
  - A set of member (data/methods) available to all descendants
  - Descendants can override virtual/abstract methods
- See **Shapes** and included Class Diagram



# Access modifiers (for encapsulation)

**Table 5-1.** *C# Access Modifiers*

C# Access Modifier	May Be Applied To	Meaning in Life
public	Types or type members	Public items have no access restrictions. A public member can be accessed from an object, as well as any derived class. A public type can be accessed from other external assemblies.
private	Type members or nested types	Private items can be accessed only by the class (or structure) that defines the item.
protected	Type members or nested types	Protected items can be used by the class that defines it and any child class. However, protected items cannot be accessed from the outside world using the C# dot operator.
internal	Types or type members	Internal items are accessible only within the current assembly. Therefore, if you define a set of internal types within a .NET class library, other assemblies are not able to use them.
protected internal	Type members or nested types	When the protected and internal keywords are combined on an item, the item is accessible within the defining assembly, within the defining class, and by derived classes.

default

# Properties

- A way to encode rules into accessor/mutators, yet allow for internal data to be manipulated by intrinsic operators
- Form
  - Automatic: just using a backing field, no logic
    - Type PropertyName { get; set; }
  - Normal
    - Type PropertyName { get {return data;} set { data = value;}}
    - Where data is a private field.
- Example
  - EmployeeApp
  - AutoProps

```
// Automatic properties!  
public string PetName { get; set; }  
public int Speed { get; set; }  
public string Color { get; set; }
```

# Properties - example

- Get, set used to enforce rules
- Encapsulates private data
  - Generates invisible backing field
- Note use of **value** keyword as operator argument on invocation

```
// Properties!
public string Name
{
    get { return empName; }
    set
    {
        if (value.Length > 15)
            Console.WriteLine("Error! Name length exceeds 15 characters");
        else
            empName = value;
    }
}

// We could add additional business rules to the sets of these properties,
// however there is no need to do so for this example.
public int ID
{
    get { return empID; }
    set { empID = value; }
}

public float Pay
{
    get { return currPay; }
    set { currPay = value; }
}

public int Age
{
    get { return empAge; }
    set { empAge = value; }
}

public string SocialSecurityNumber
{
    get { return empSSN; }
}
```

# More on Properties

- Can be set to initial value
  - `public int Age { get; set; } = 21;`
- Can be readonly (note lack of set!) so helps with encapsulation.
  - `public int Age { get; } = 21;`
- AutoProps code snippet: type **prop** followed by two tabs in VS!!
- Can be used with static
- Example: StaticDataAndMembers

```
// A static point of data.  
private static double currInterestRate = 0.04;  
// A static property.  
public static double InterestRate  
{  
    get { return currInterestRate; }  
    set { currInterestRate = value; }  
}
```

# Object Initialization

```
Console.WriteLine("***** Fun with Object Init Syntax *****\n");
```

- Useful when a number of properties need to get set at once.
- Author may have a lot of properties in a class, and does not need or want all possible permutations of constructors.
- Use
  - `new MyObject() { Prop1 = x, Prop2 = y }`
- Example
  - `ObjectInitializers`

```
// Make a Point by setting each property manually.  
Point firstPoint = new Point();  
firstPoint.X = 10;  
firstPoint.Y = 10;  
firstPoint.DisplayStats();  
Console.WriteLine();
```

```
// Or make a Point via a custom constructor.  
Point anotherPoint = new Point(20, 20);  
anotherPoint.DisplayStats();  
Console.WriteLine();
```

```
// Or make a Point using object init syntax.  
Point finalPoint = new Point { X = 30, Y = 30 };  
finalPoint.DisplayStats();  
Console.WriteLine();
```

```
// Calling a more interesting custom constructor with init  
syntax.  
Point goldPoint = new Point(PointColor.Gold) { X = 90, Y = 20 };  
goldPoint.DisplayStats();  
Console.WriteLine();
```

```
// Create and initialize a Rectangle.  
Rectangle myRect = new Rectangle  
{  
    TopLeft = new Point { X = 10, Y = 10 },  
    BottomRight = new Point { X = 200, Y = 200 }  
};
```

# Const and readonly

- const keyword
  - Variable cannot be modified and must be initialized at compile time
- readonly keyword
  - Variable cannot be modified
  - Can be initialized at run time ONLY in the constructor
- Both can be made static as part of a class
  - Allows groups of constants to be defined in a particular class – eg, Color
- Example
  - ConstData



# Partial Classes

- Allows for splitting of code across files
- Windows Forms and VS produces partial class files, with some fields, properties, methods in one file, and with a 2<sup>nd</sup> file where you fill in the various methods to complete the class.
- See
  - Employees