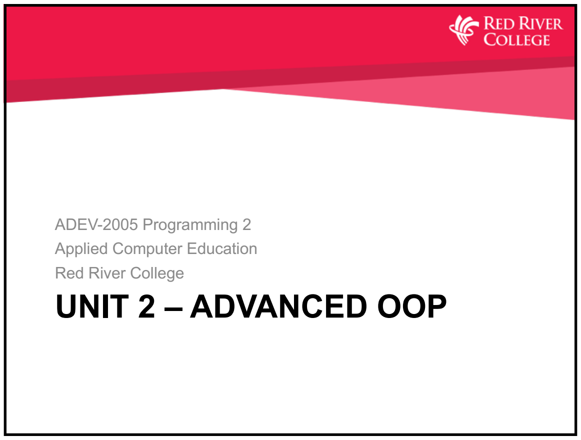
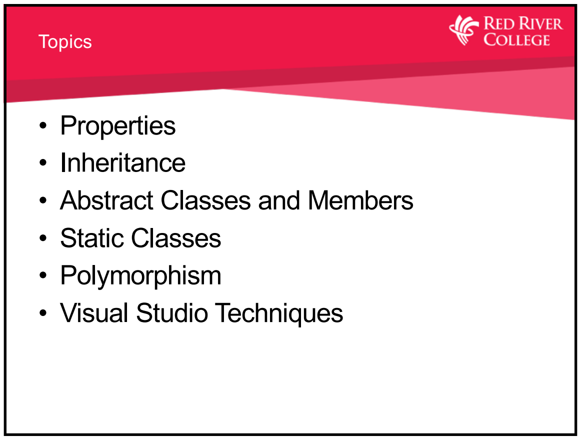
RED RIVER COLLEGE
OF APPLIED ARTS, SCIENCE AND TECHNOLOGY



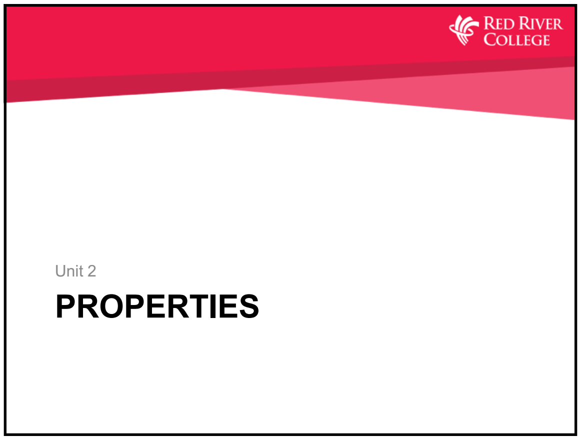
ADEV-2005 Programming 2
Applied Computer Education
Red River College

UNIT 2 – ADVANCED OOP



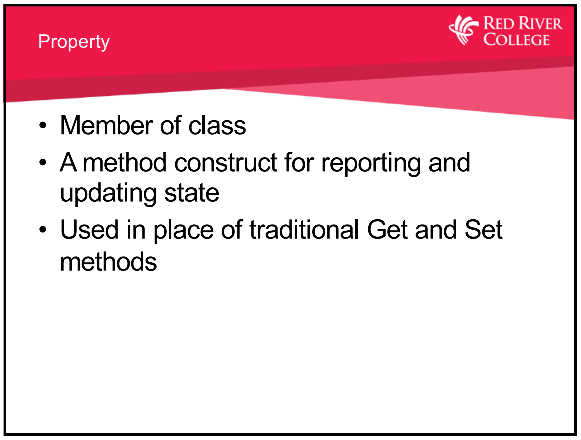
Topics

- Properties
- Inheritance
- Abstract Classes and Members
- Static Classes
- Polymorphism
- Visual Studio Techniques



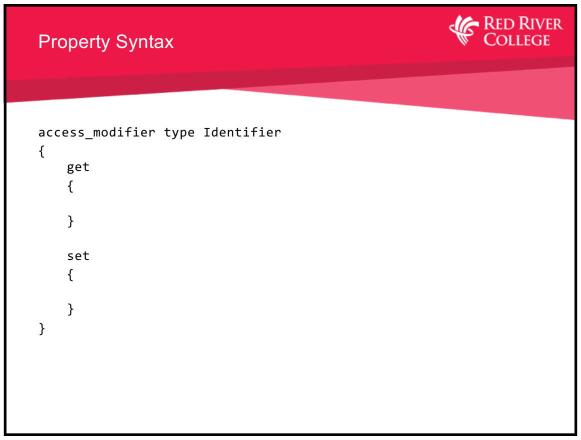
Unit 2

PROPERTIES



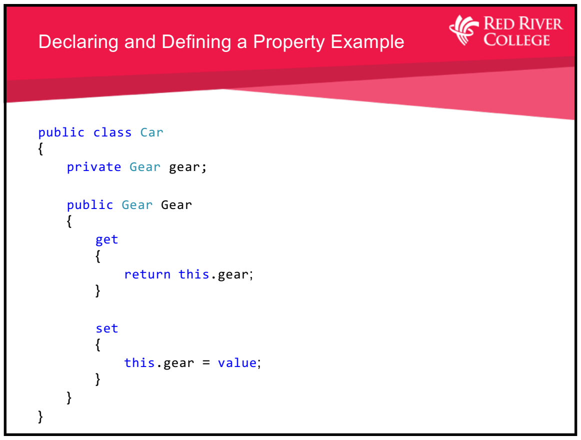
Property

- Member of class
- A method construct for reporting and updating state
- Used in place of traditional Get and Set methods



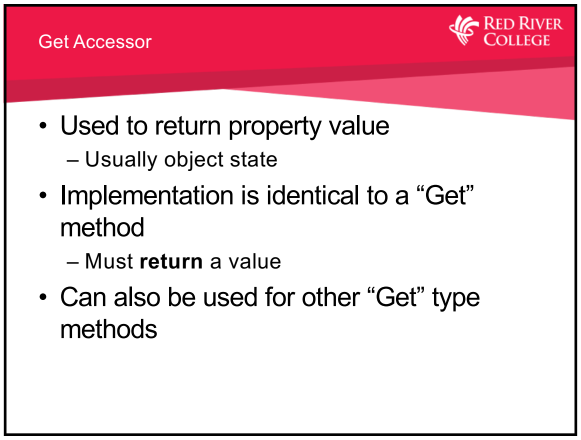
Property Syntax

```
access_modifier type Identifier
{
    get
    {

    }

    set
    {

    }
}
```



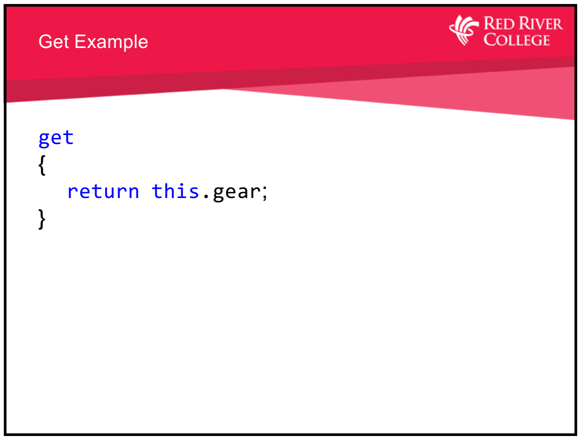
Declaring and Defining a Property Example

```
public class Car
{
    private Gear gear;

    public Gear Gear
    {
        get
        {
            return this.gear;
        }

        set
        {
            this.gear = value;
        }
    }
}
```



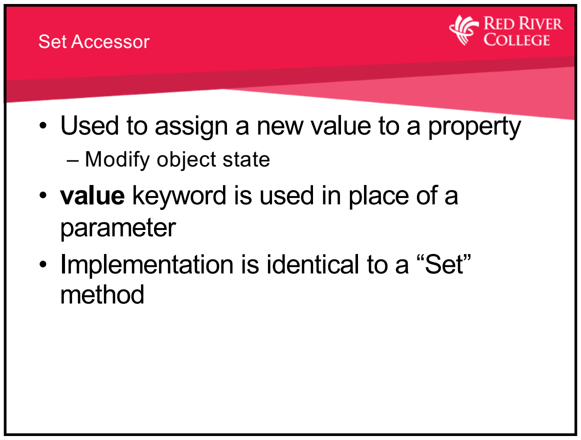
Get Accessor

- Used to return property value
  - Usually object state
- Implementation is identical to a “Get” method
  - Must **return** a value
- Can also be used for other “Get” type methods



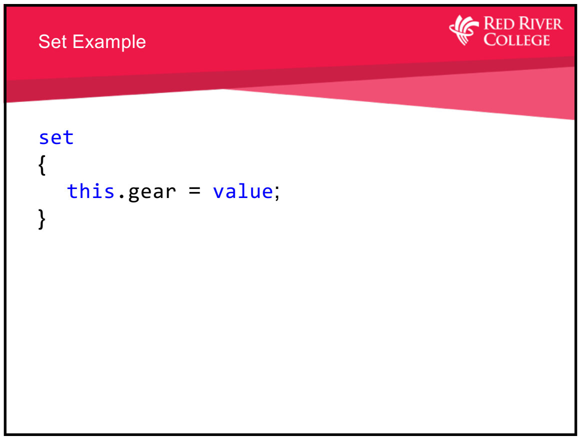
Get Example

```
get
{
    return this.gear;
}
```



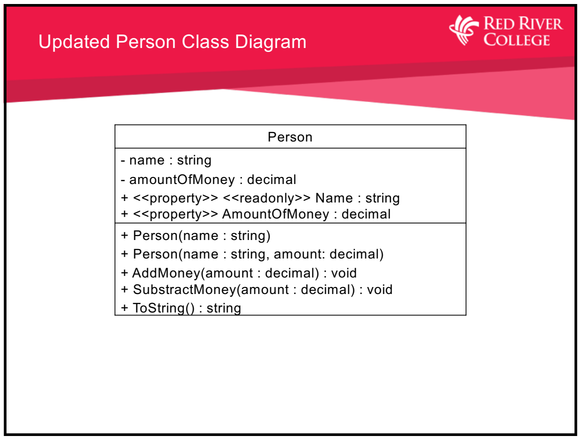
Set Accessor

- Used to assign a new value to a property
  - Modify object state
- **value** keyword is used in place of a parameter
- Implementation is identical to a “Set” method



Set Example

```
set
{
   this.gear = value;
}
```

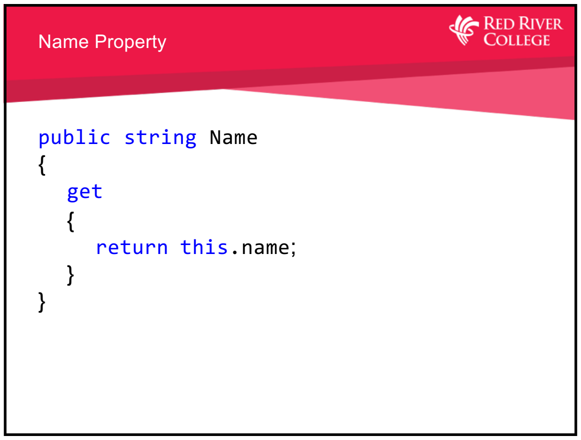


Updated Person Class Diagram

Person

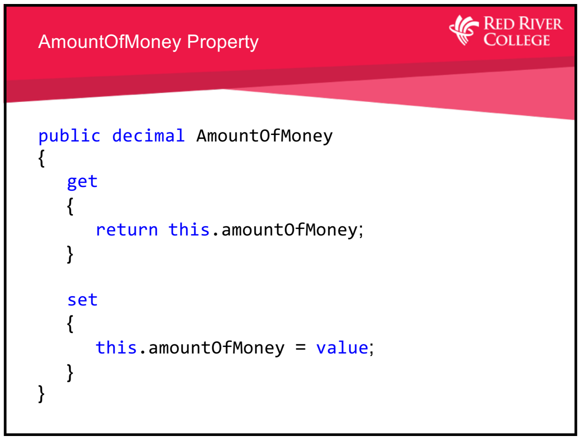
- name : string
- amountOfMoney : decimal
+ <<property>> <<readonly>> Name : string
+ <<property>> AmountOfMoney : decimal

+ Person(name : string)
+ Person(name : string, amount: decimal)
+ AddMoney(amount : decimal) : void
+ SubstractMoney(amount : decimal) : void
+ ToString() : string



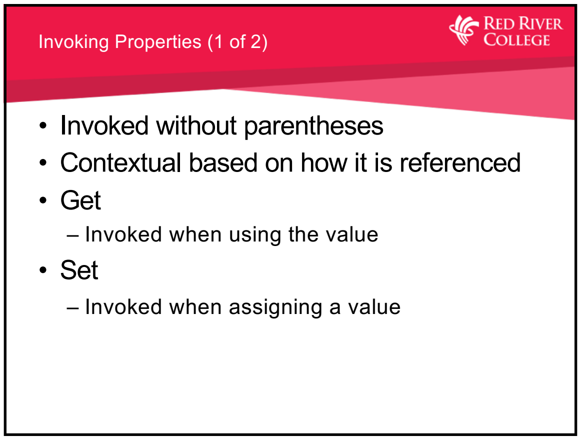
Name Property

```
public string Name
{
   get
   {
      return this.name;
   }
}
```



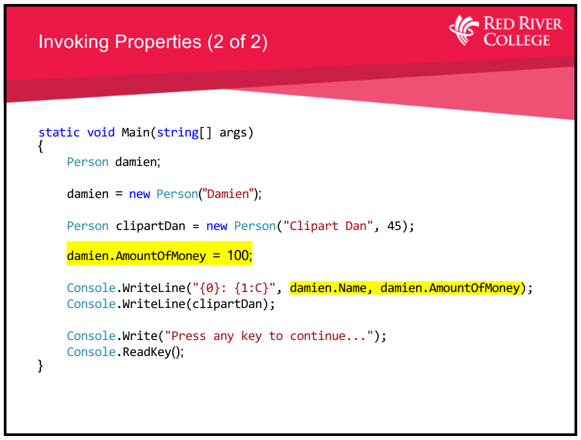
AmountOfMoney Property

```
public decimal AmountOfMoney
{
   get
   {
      return this.amountOfMoney;
   }

   set
   {
      this.amountOfMoney = value;
   }
}
```



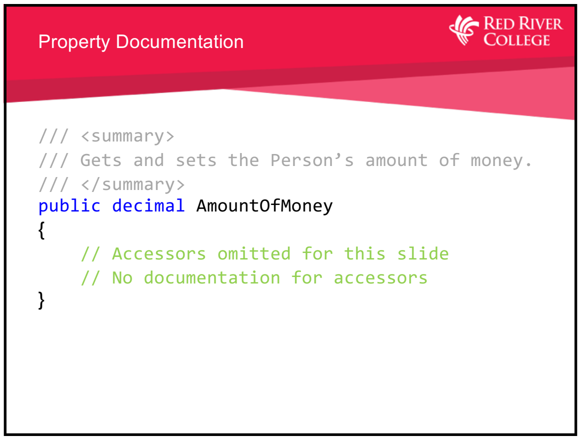
Invoking Properties (1 of 2)

- Invoked without parentheses
- Contextual based on how it is referenced
- Get
  - Invoked when using the value
- Set
  - Invoked when assigning a value



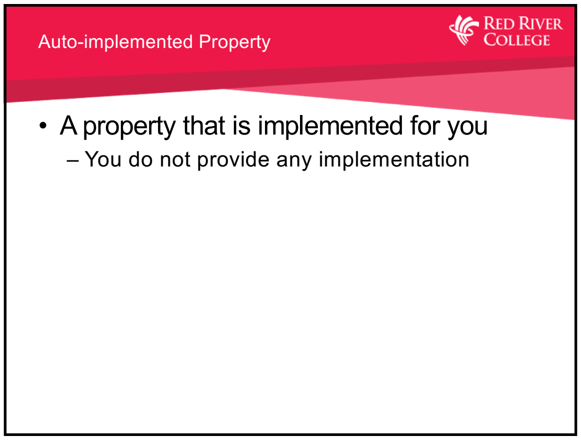
Invoking Properties (2 of 2)

```
static void Main(string[] args)
{
    Person damien;

    damien = new Person("Damien");

    Person clipartDan = new Person("Clipart Dan", 45);

    damien.AmountOfMoney = 100;

    Console.WriteLine("{0}: {1:C}", damien.Name, damien.AmountOfMoney);
    Console.WriteLine(clipartDan);

    Console.Write("Press any key to continue...");
    Console.ReadKey();
}
```



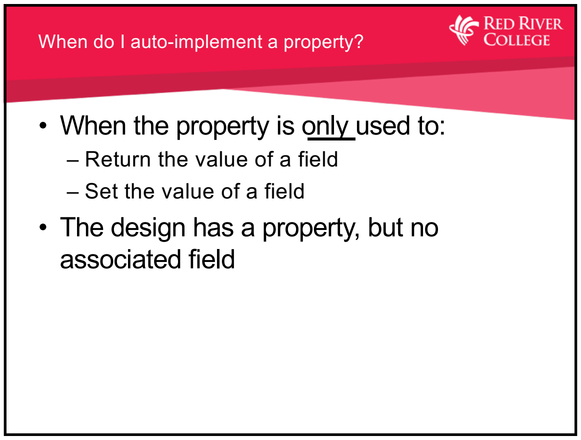
Property Documentation

```
/// <summary>
/// Gets and sets the Person’s amount of money.
/// </summary>
public decimal AmountOfMoney
{
    // Accessors omitted for this slide
    // No documentation for accessors
}
```



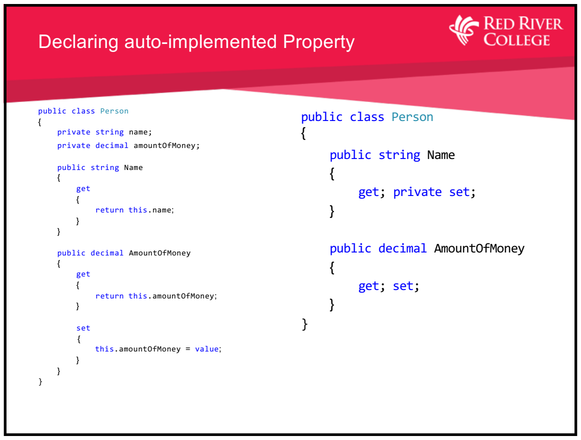
Auto-implemented Property

- A property that is implemented for you
  - You do not provide any implementation



When do I auto-implement a property?

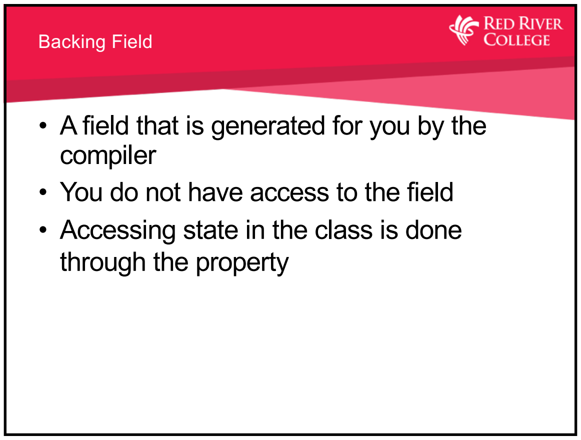
- When the property is only used to:
  - Return the value of a field
  - Set the value of a field
- The design has a property, but no associated field



Declaring auto-implemented Property

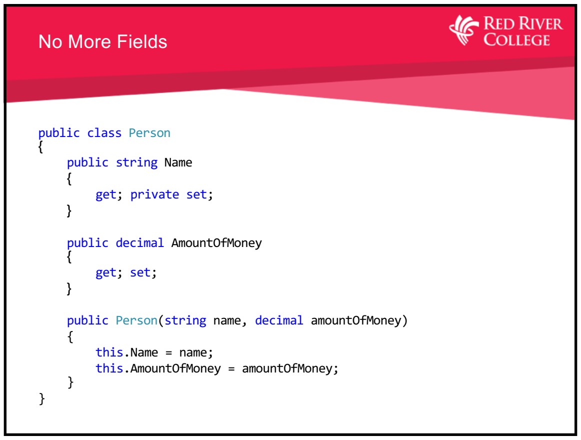
```
public class Person
{
    private string name;
    private decimal amountOfMoney;

    public string Name
    {
        get
        {
            return this.name;
        }
    }

    public decimal AmountOfMoney
    {
        get
        {
            return this.amountOfMoney;
        }

        set
        {
            this.amountOfMoney = value;
        }
    }
}
```

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

    public decimal AmountOfMoney
    {
        get; set;
    }
}
```



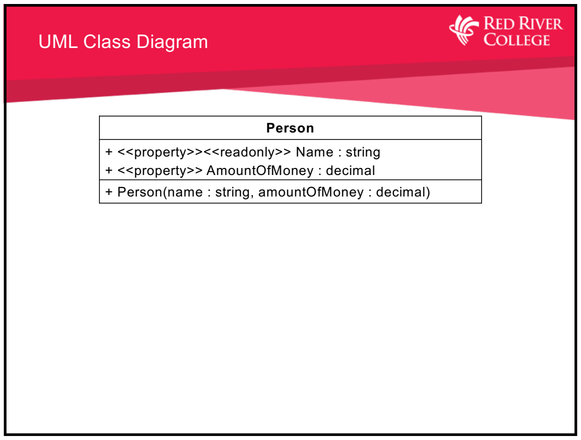
Backing Field

- A field that is generated for you by the compiler
- You do not have access to the field
- Accessing state in the class is done through the property



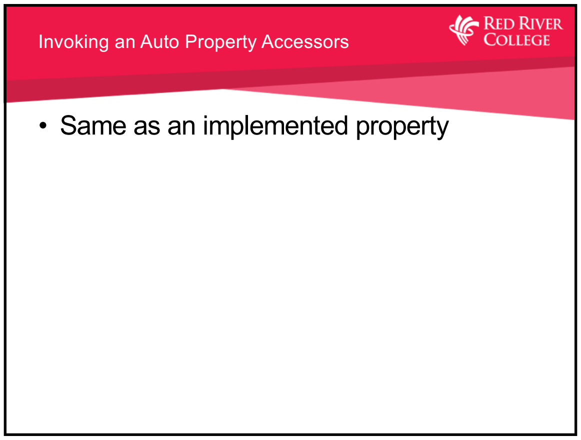
No More Fields

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

    public decimal AmountOfMoney
    {
        get; set;
    }

    public Person(string name, decimal amountOfMoney)
    {
        this.Name = name;
        this.AmountOfMoney = amountOfMoney;
    }
}
```



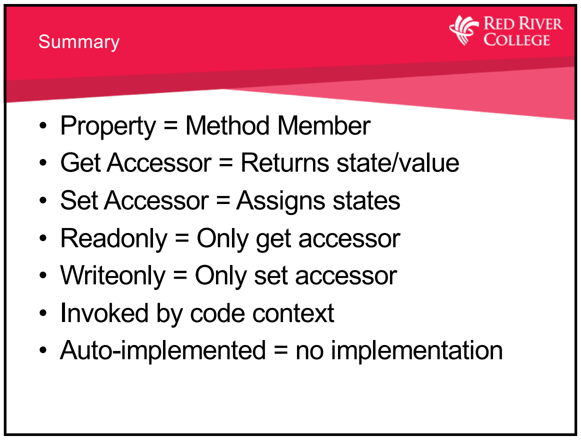
UML Class Diagram

Person
+ <<property>><<readonly>> Name : string + <<property>> AmountOfMoney : decimal
+ Person(name : string, amountOfMoney : decimal)



Invoking an Auto Property Accessors

- Same as an implemented property

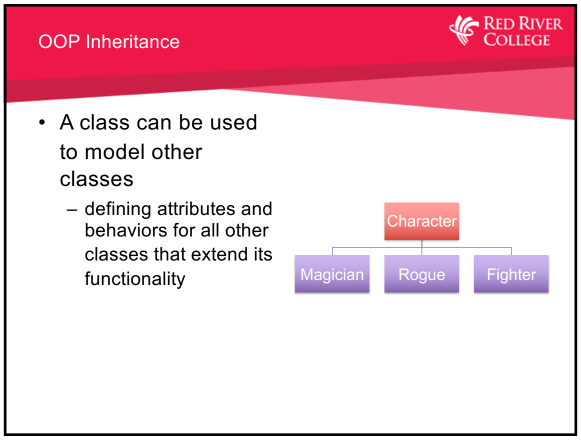


Summary

- Property = Method Member
- Get Accessor = Returns state/value
- Set Accessor = Assigns states
- Readonly = Only get accessor
- Writeonly = Only set accessor
- Invoked by code context
- Auto-implemented = no implementation

Unit 2

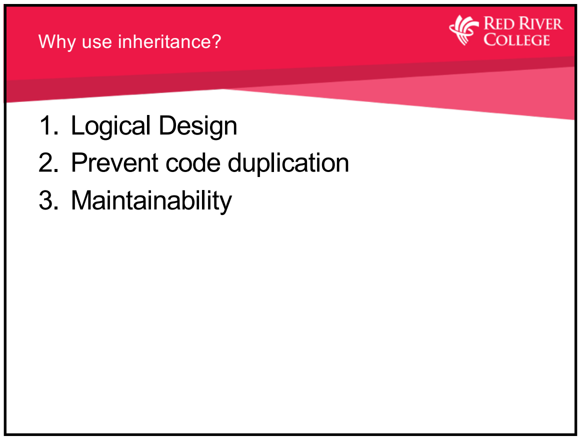
INHERITANCE



OOP Inheritance

- A class can be used to model other classes
  - defining attributes and behaviors for all other classes that extend its functionality

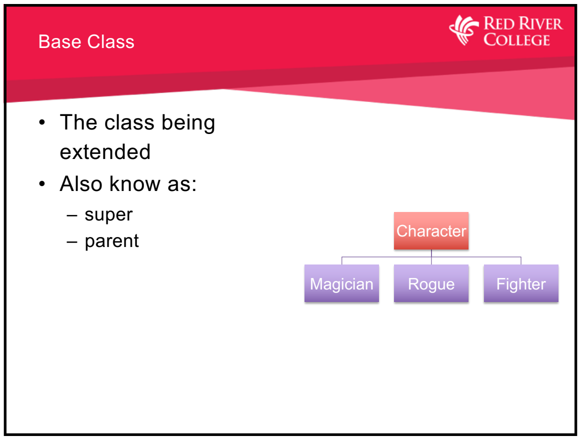
Character
Magician
Rogue
Fighter



RED RIVER COLLEGE

Why use inheritance?

1. Logical Design
2. Prevent code duplication
3. Maintainability



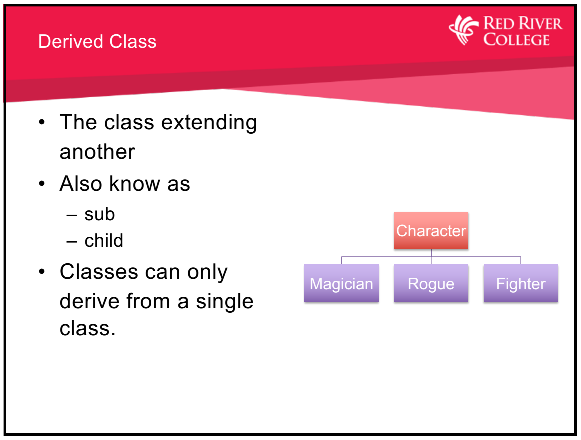
RED RIVER COLLEGE

Base Class

- The class being extended
- Also know as:
  - – super
  - – parent

Character

Magician Rogue Fighter



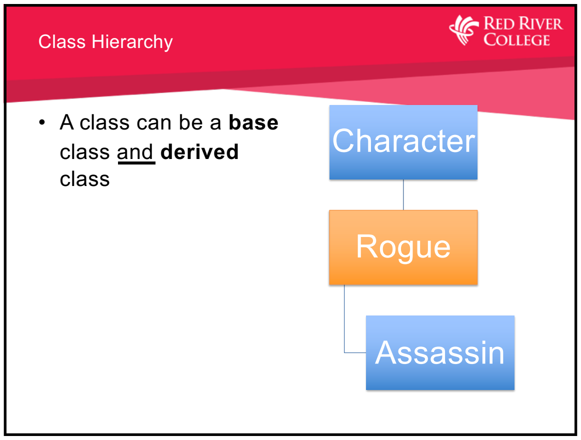
RED RIVER COLLEGE

Derived Class

- The class extending another
- Also know as
  - – sub
  - – child
- Classes can only derive from a single class.

Character

Magician Rogue Fighter



Class Hierarchy

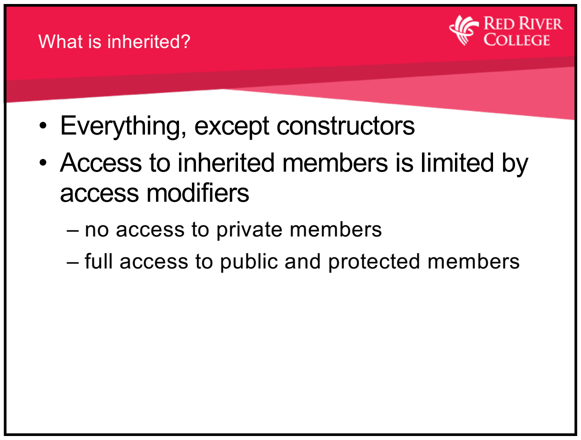
RED RIVER COLLEGE

- A class can be a **base** class <u>and</u> **derived** class

Character

Rogue

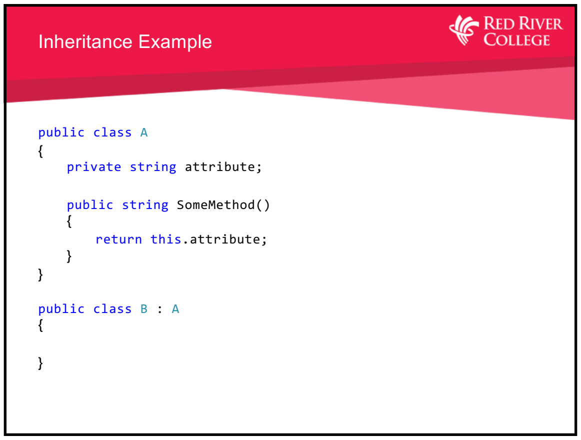
Assassin



What is inherited?

RED RIVER COLLEGE

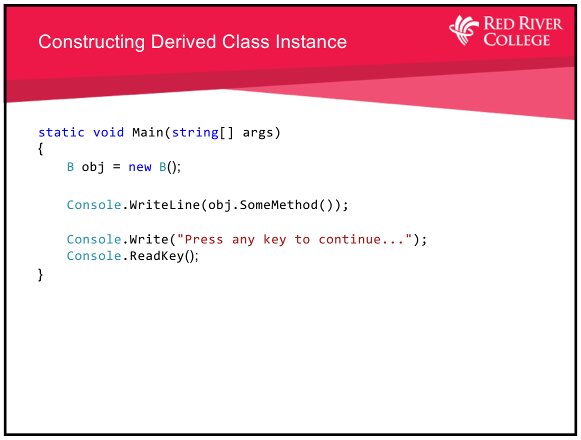
- Everything, except constructors
- Access to inherited members is limited by access modifiers
  - – no access to private members
  - – full access to public and protected members



Inheritance Example

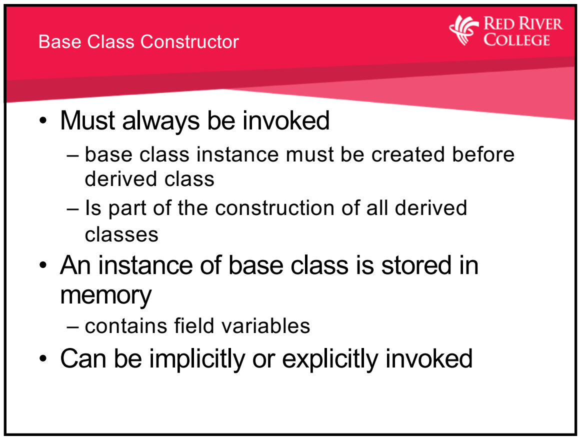
RED RIVER COLLEGE

```
public class A
{
    private string attribute;

    public string SomeMethod()
    {
        return this.attribute;
    }
}

public class B : A
{

}
```



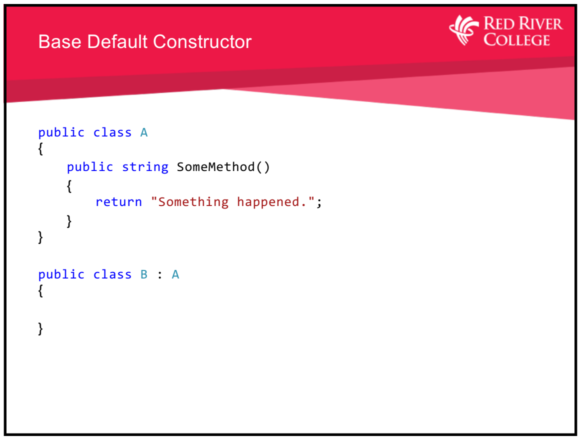
Constructing Derived Class Instance

```
static void Main(string[] args)
{
    B obj = new B();

    Console.WriteLine(obj.SomeMethod());

    Console.Write("Press any key to continue...");
    Console.ReadKey();
}
```



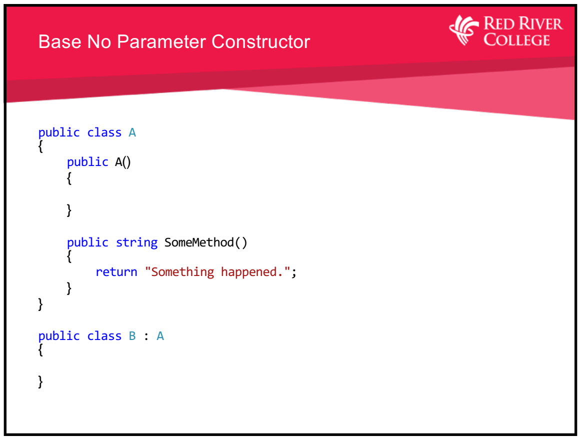
Base Class Constructor

- Must always be invoked
  - base class instance must be created before derived class
  - Is part of the construction of all derived classes
- An instance of base class is stored in memory
  - contains field variables
- Can be implicitly or explicitly invoked



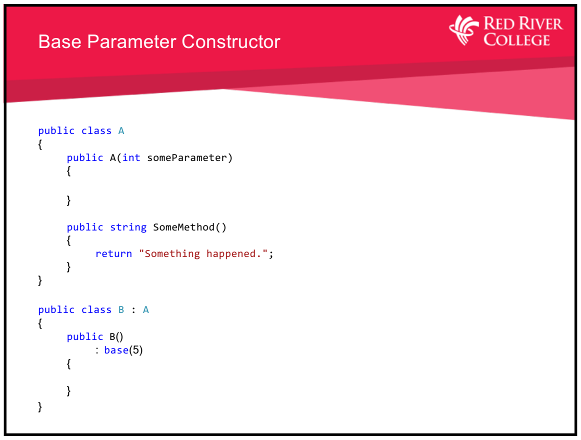
Base Default Constructor

```
public class A
{
    public string SomeMethod()
    {
        return "Something happened.";
    }
}

public class B : A
{

}
```



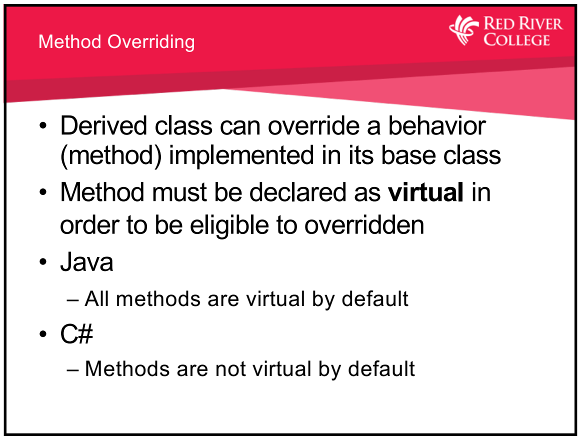
Base No Parameter Constructor

```
public class A
{
    public A()
    {

    }

    public string SomeMethod()
    {
        return "Something happened.";
    }
}

public class B : A
{

}
```



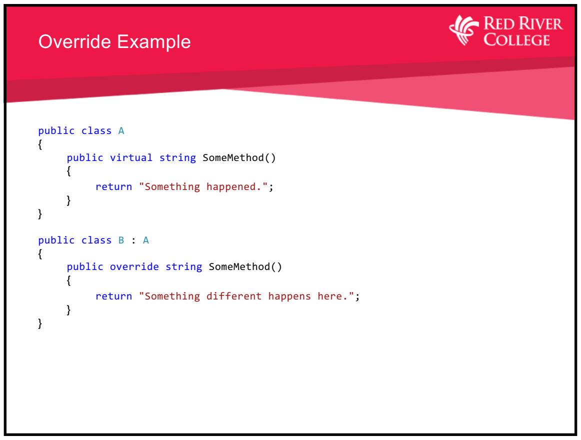
Base Parameter Constructor

```
public class A
{
    public A(int someParameter)
    {

    }

    public string SomeMethod()
    {
        return "Something happened.";
    }
}

public class B : A
{
    public B()
        : base(5)
    {

    }
}
```



Method Overriding

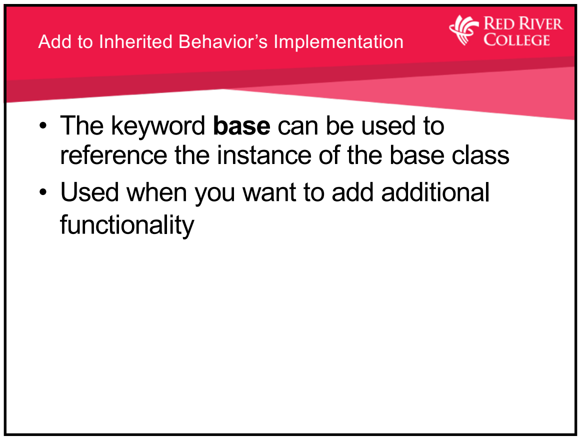
- Derived class can override a behavior (method) implemented in its base class
- Method must be declared as **virtual** in order to be eligible to overridden
- Java
  - All methods are virtual by default
- C#
  - Methods are not virtual by default



Override Example

RED RIVER COLLEGE

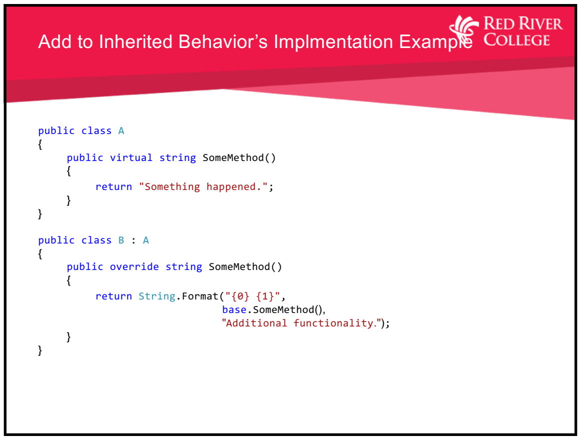
```
public class A
{
    public virtual string SomeMethod()
    {
        return "Something happened.";
    }
}

public class B : A
{
    public override string SomeMethod()
    {
        return "Something different happens here.";
    }
}
```



Add to Inherited Behavior’s Implementation

RED RIVER COLLEGE

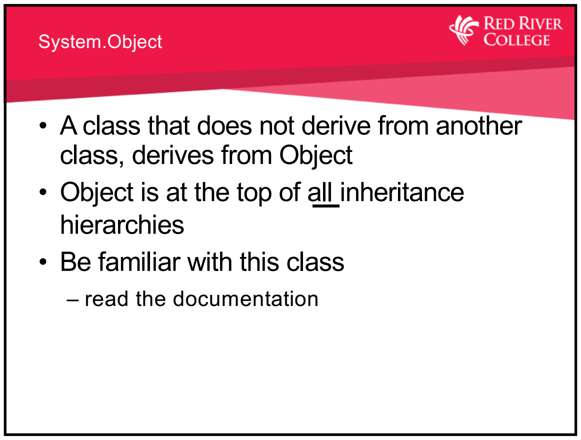
- The keyword **base** can be used to reference the instance of the base class
- Used when you want to add additional functionality



Add to Inherited Behavior’s Implmentation Example

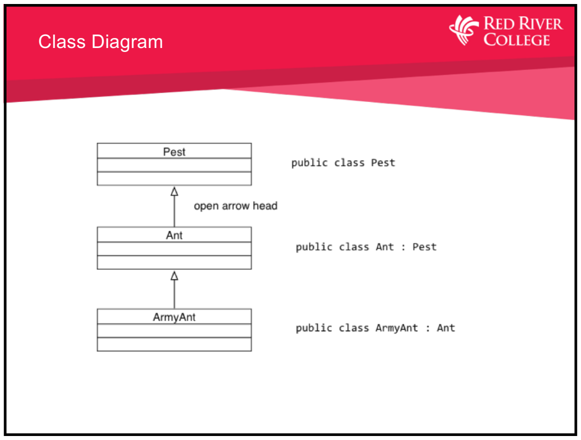
RED RIVER COLLEGE

```
public class A
{
    public virtual string SomeMethod()
    {
        return "Something happened.";
    }
}

public class B : A
{
    public override string SomeMethod()
    {
        return String.Format("{0} {1}",
                             base.SomeMethod(),
                             "Additional functionality.");
    }
}
```



System.Object

- A class that does not derive from another class, derives from Object
- Object is at the top of all inheritance hierarchies
- Be familiar with this class
  - read the documentation

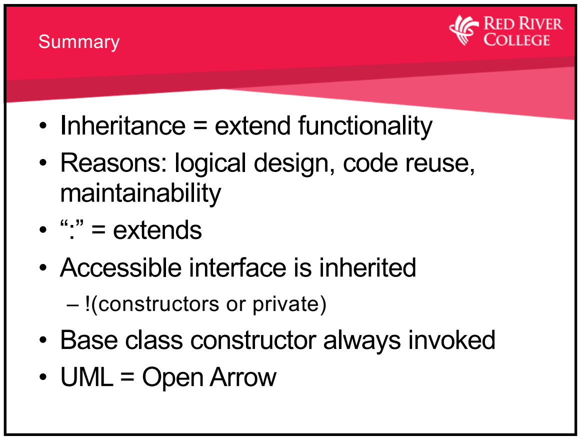


Class Diagram

```
public class Pest
```

```
public class Ant : Pest
```

```
public class ArmyAnt : Ant
```



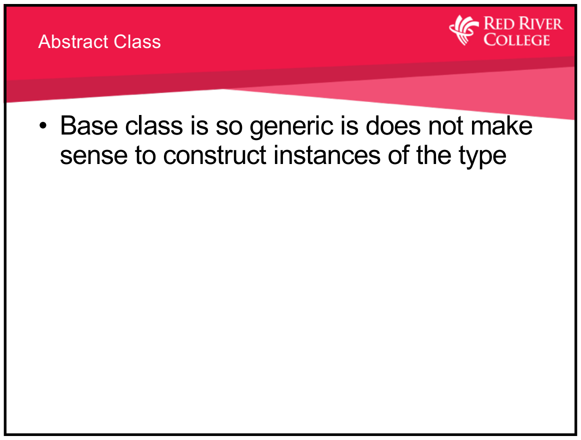
Summary

- Inheritance = extend functionality
- Reasons: logical design, code reuse, maintainability
- “:” = extends
- Accessible interface is inherited
  - !(constructors or private)
- Base class constructor always invoked
- UML = Open Arrow

RED RIVER COLLEGE

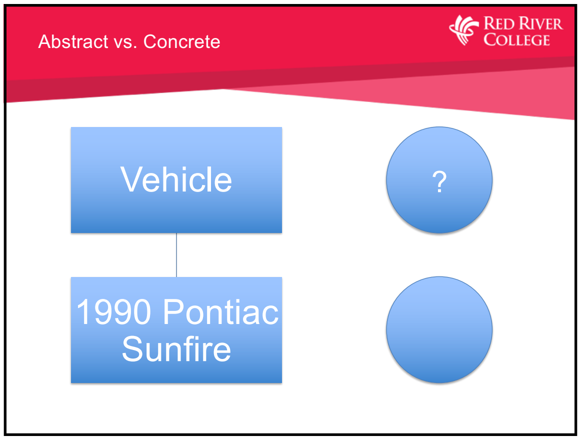
Unit 2

ABSTRACT CLASSES AND METHODS

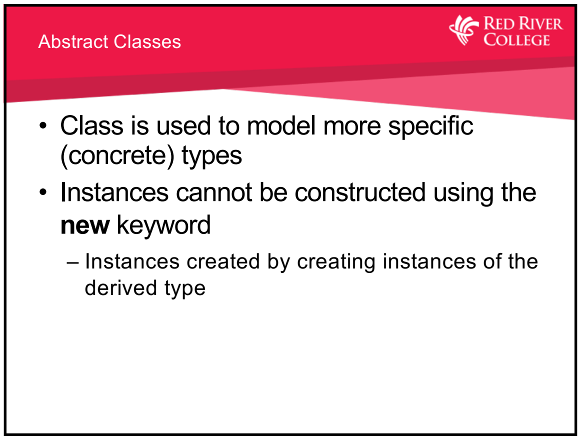

RED RIVER COLLEGE

Abstract Class

- Base class is so generic is does not make sense to construct instances of the type

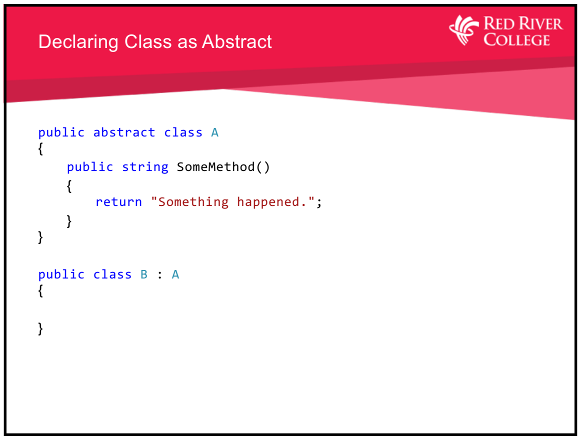

RED RIVER COLLEGE

Abstract vs. Concrete



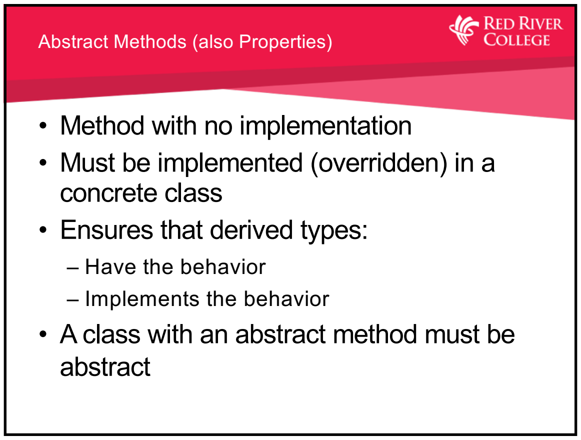
Abstract Classes

- Class is used to model more specific (concrete) types
- Instances cannot be constructed using the **new** keyword
  - Instances created by creating instances of the derived type



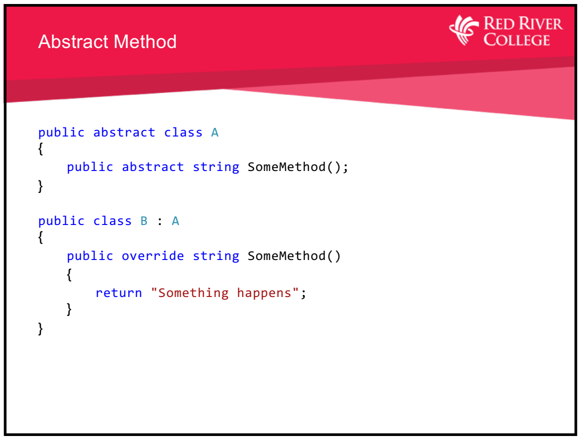
Declaring Class as Abstract

```
public abstract class A
{
    public string SomeMethod()
    {
        return "Something happened.";
    }
}

public class B : A
{

}
```



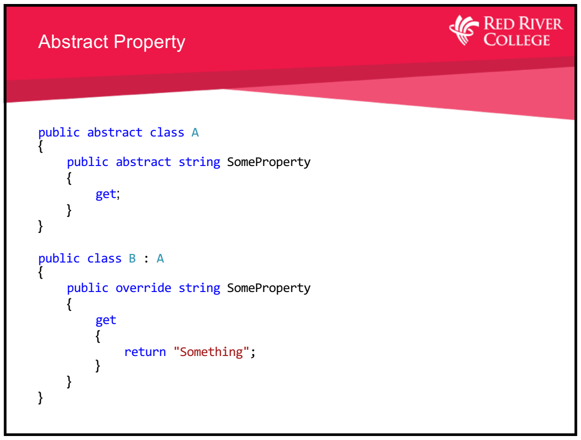
Abstract Methods (also Properties)

- Method with no implementation
- Must be implemented (overridden) in a concrete class
- Ensures that derived types:
  - Have the behavior
  - Implements the behavior
- A class with an abstract method must be abstract



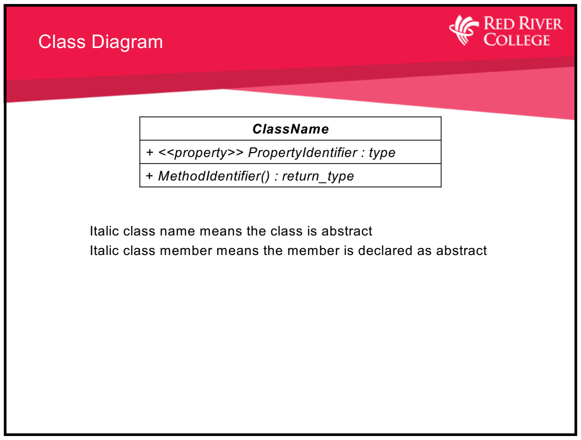
Abstract Method

```
public abstract class A
{
    public abstract string SomeMethod();
}

public class B : A
{
    public override string SomeMethod()
    {
        return "Something happens";
    }
}
```



Abstract Property

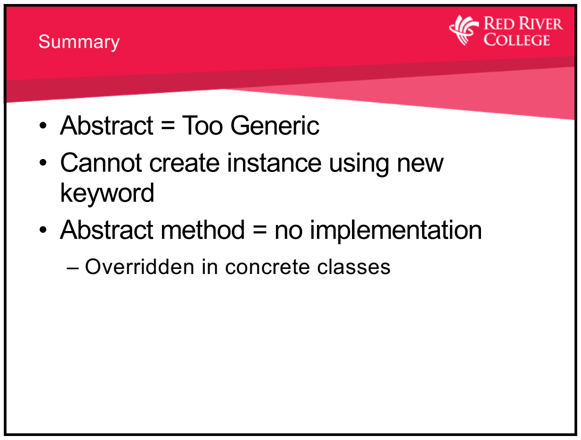
```
public abstract class A
{
    public abstract string SomeProperty
    {
        get;
    }
}

public class B : A
{
    public override string SomeProperty
    {
        get
        {
            return "Something";
        }
    }
}
```



Class Diagram

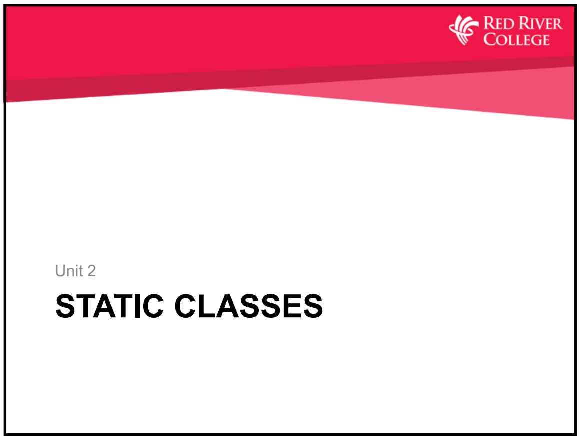
ClassName
+ *<<property>> PropertyIdentifier : type*
+ *MethodIdentifier() : return_type*

Italic class name means the class is abstract
Italic class member means the member is declared as abstract



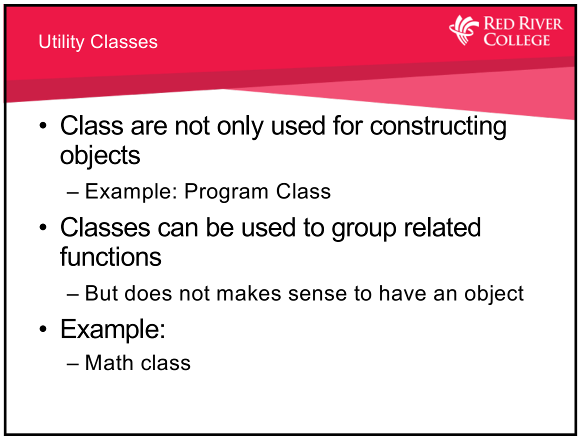
Summary

- Abstract = Too Generic
- Cannot create instance using new keyword
- Abstract method = no implementation
  - Overridden in concrete classes



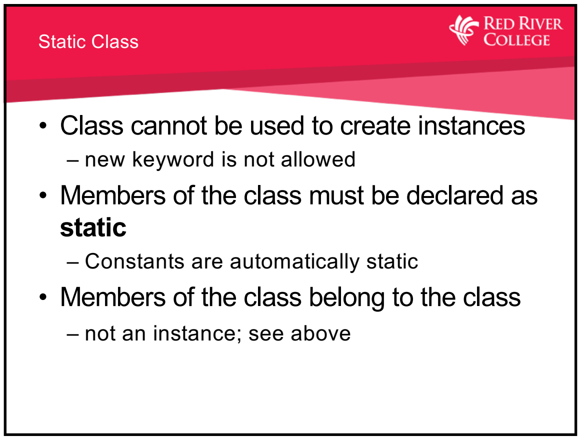
Unit 2

STATIC CLASSES



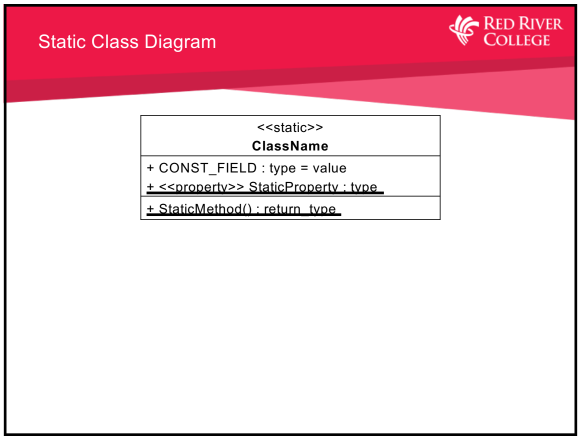
Utility Classes

- Class are not only used for constructing objects
  - Example: Program Class
- Classes can be used to group related functions
  - But does not makes sense to have an object
- Example:
  - Math class



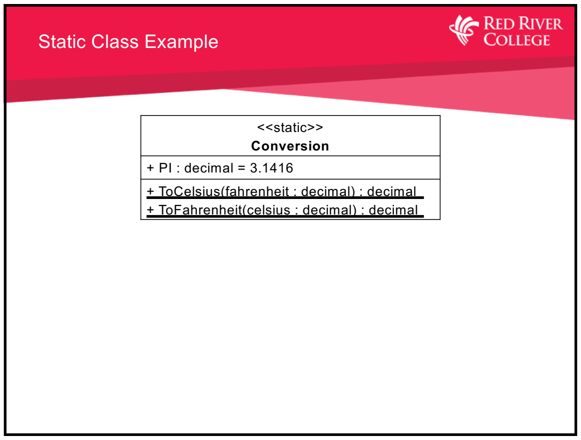
Static Class

- Class cannot be used to create instances
  - new keyword is not allowed
- Members of the class must be declared as **static**
  - Constants are automatically static
- Members of the class belong to the class
  - not an instance; see above



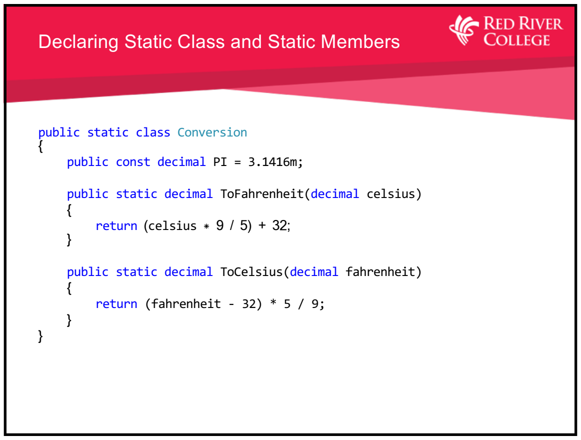
Static Class Diagram

```
<<static>>
ClassName
+ CONST_FIELD : type = value
+ <<property>> StaticProperty : type
+ StaticMethod() : return_type
```



Static Class Example

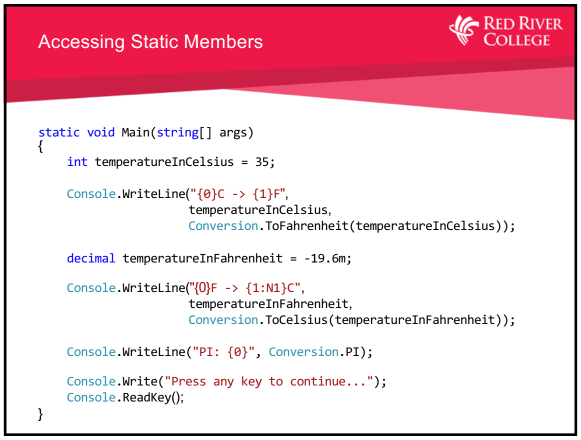
```
<<static>>
Conversion
+ PI : decimal = 3.1416
+ ToCelsius(fahrenheit : decimal) : decimal
+ ToFahrenheit(celsius : decimal) : decimal
```



Declaring Static Class and Static Members

RED RIVER COLLEGE

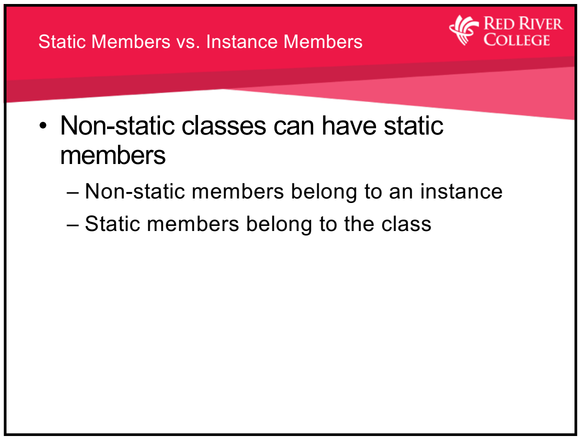
```
public static class Conversion
{
    public const decimal PI = 3.1416m;

    public static decimal ToFahrenheit(decimal celsius)
    {
        return (celsius * 9 / 5) + 32;
    }

    public static decimal ToCelsius(decimal fahrenheit)
    {
        return (fahrenheit - 32) * 5 / 9;
    }
}
```



Accessing Static Members

RED RIVER COLLEGE

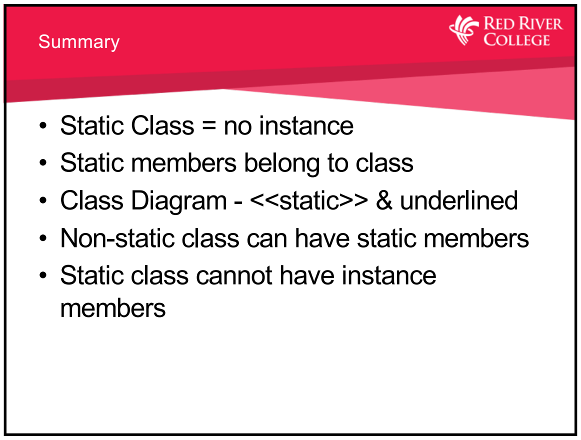
```
static void Main(string[] args)
{
    int temperatureInCelsius = 35;

    Console.WriteLine("{0}C -> {1}F",
                      temperatureInCelsius,
                      Conversion.ToFahrenheit(temperatureInCelsius));

    decimal temperatureInFahrenheit = -19.6m;

    Console.WriteLine("{0}F -> {1:N1}C",
                      temperatureInFahrenheit,
                      Conversion.ToCelsius(temperatureInFahrenheit));

    Console.WriteLine("PI: {0}", Conversion.PI);

    Console.Write("Press any key to continue...");
    Console.ReadKey();
}
```



Static Members vs. Instance Members

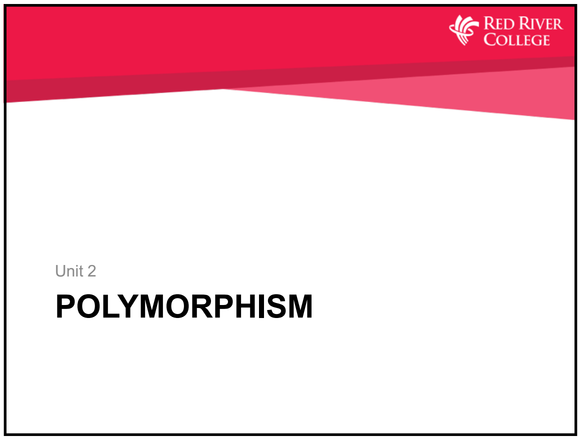
RED RIVER COLLEGE

- Non-static classes can have static members
  - Non-static members belong to an instance
  - Static members belong to the class



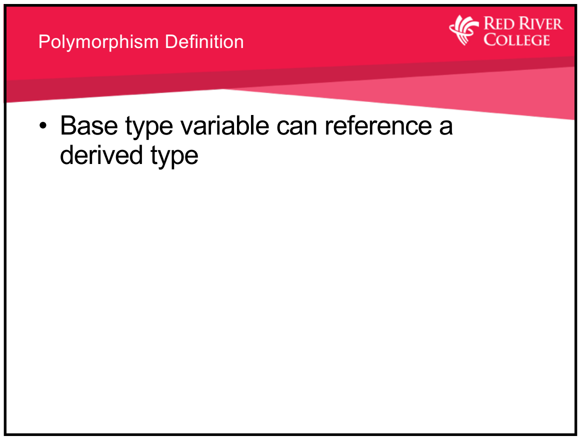
Summary

- Static Class = no instance
- Static members belong to class
- Class Diagram - <<static>> & underlined
- Non-static class can have static members
- Static class cannot have instance members



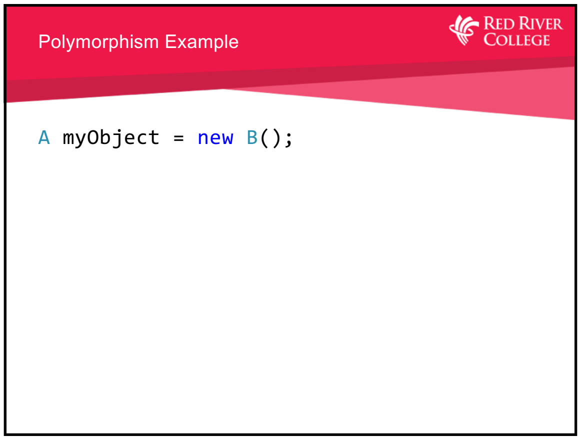
Unit 2

POLYMORPHISM



Polymorphism Definition

- Base type variable can reference a derived type

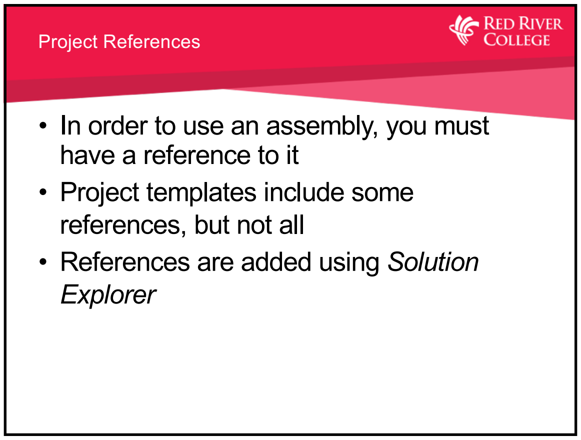


Polymorphism Example

```
A myObject = new B();
```

Unit 2

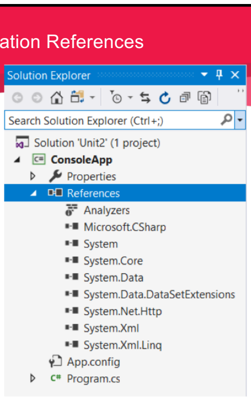
VISUAL STUDIO TECHNIQUES



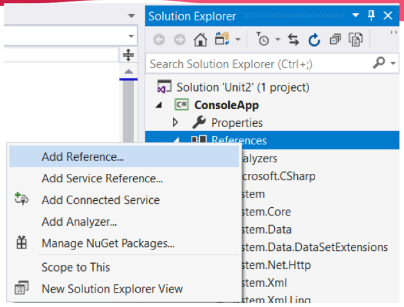
Project References

- In order to use an assembly, you must have a reference to it
- Project templates include some references, but not all
- References are added using *Solution Explorer*

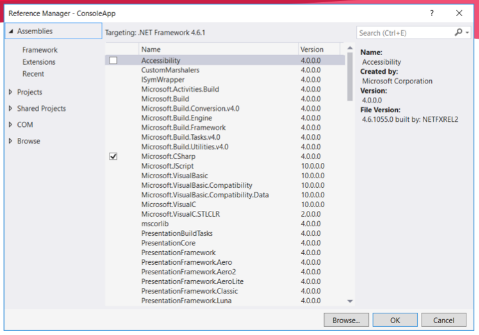
Console Application References



Adding a Reference (1 of 2)



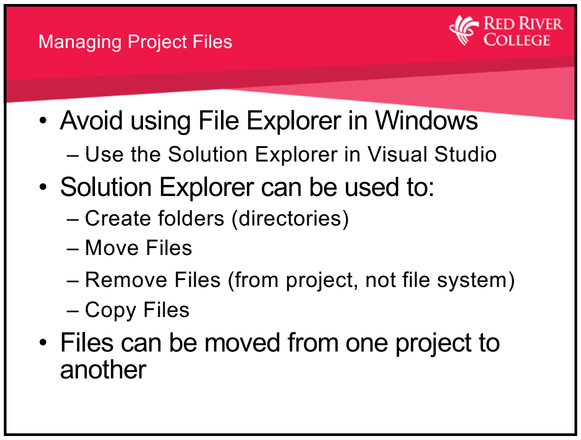
Adding a Reference (2 of 2)



Targeting: .NET Framework 4.6.1

	Name	Version
□	Accessibility	4.0.0.0
	CustomMarshalers	4.0.0.0
	ISymWrapper	4.0.0.0
	Microsoft.Activities.Build	4.0.0.0
	Microsoft.Build	4.0.0.0
	Microsoft.Build.Conversion.v4.0	4.0.0.0
	Microsoft.Build.Engine	4.0.0.0
	Microsoft.Build.Framework	4.0.0.0
	Microsoft.Build.Tasks.v4.0	4.0.0.0
	Microsoft.Build.Utilities.v4.0	4.0.0.0
☑	Microsoft.CSharp	4.0.0.0
	Microsoft.JScript	10.0.0.0
	Microsoft.VisualBasic	10.0.0.0
	Microsoft.VisualBasic.Compatibility	10.0.0.0
	Microsoft.VisualBasic.Compatibility.Data	10.0.0.0
	Microsoft.VisualC	10.0.0.0
	Microsoft.VisualC.STLCLR	2.0.0.0
	mscorlib	4.0.0.0
	PresentationBuildTasks	4.0.0.0
	PresentationCore	4.0.0.0
	PresentationFramework	4.0.0.0
	PresentationFramework.Aero	4.0.0.0
	PresentationFramework.Aero2	4.0.0.0
	PresentationFramework.AeroLite	4.0.0.0
	PresentationFramework.Classic	4.0.0.0
	PresentationFramework.Luna	4.0.0.0

Name:
Accessibility
Created by:
Microsoft Corporation
Version:
4.0.0.0
File Version:
4.6.1055.0 built by: NETFXREL2



Managing Project Files

- Avoid using File Explorer in Windows
  - Use the Solution Explorer in Visual Studio
- Solution Explorer can be used to:
  - Create folders (directories)
  - Move Files
  - Remove Files (from project, not file system)
  - Copy Files
- Files can be moved from one project to another