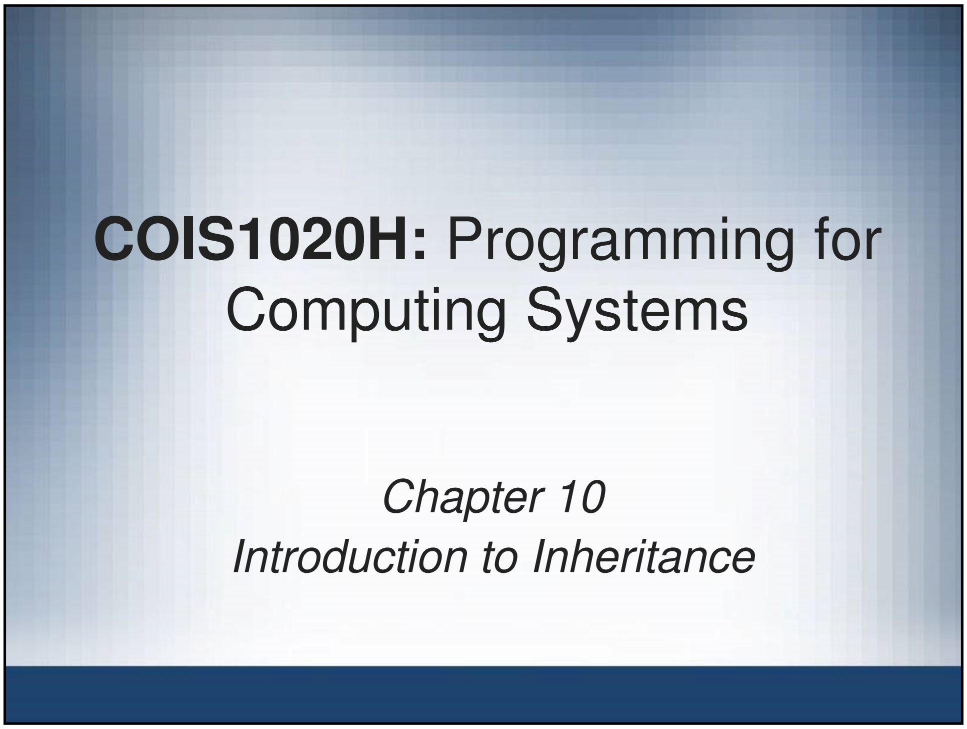
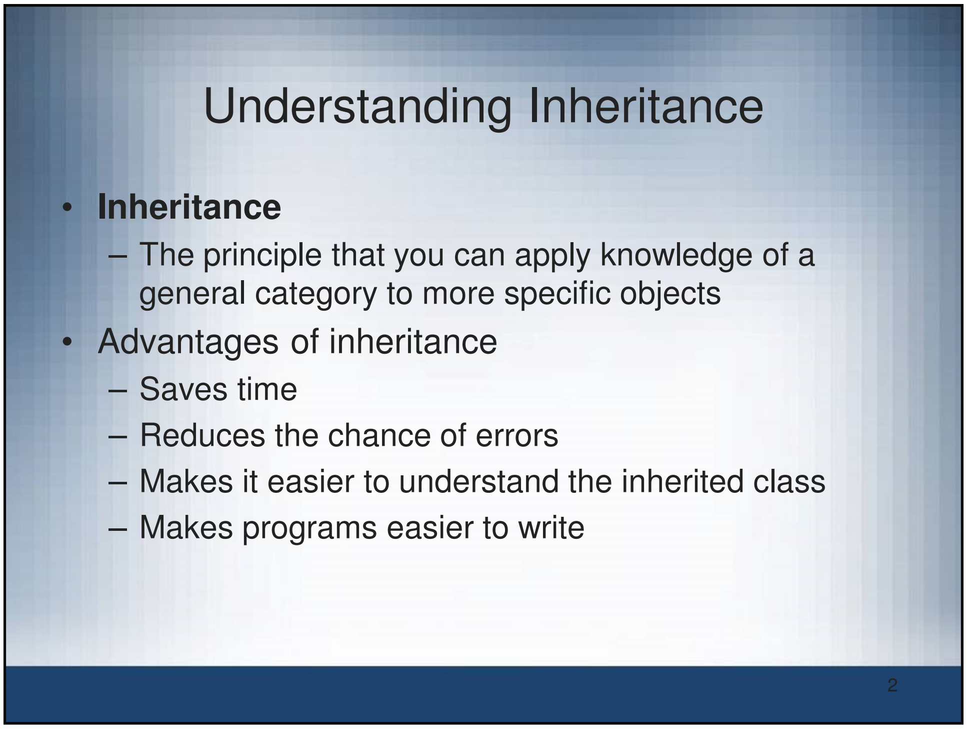


COIS1020H: Programming for Computing Systems

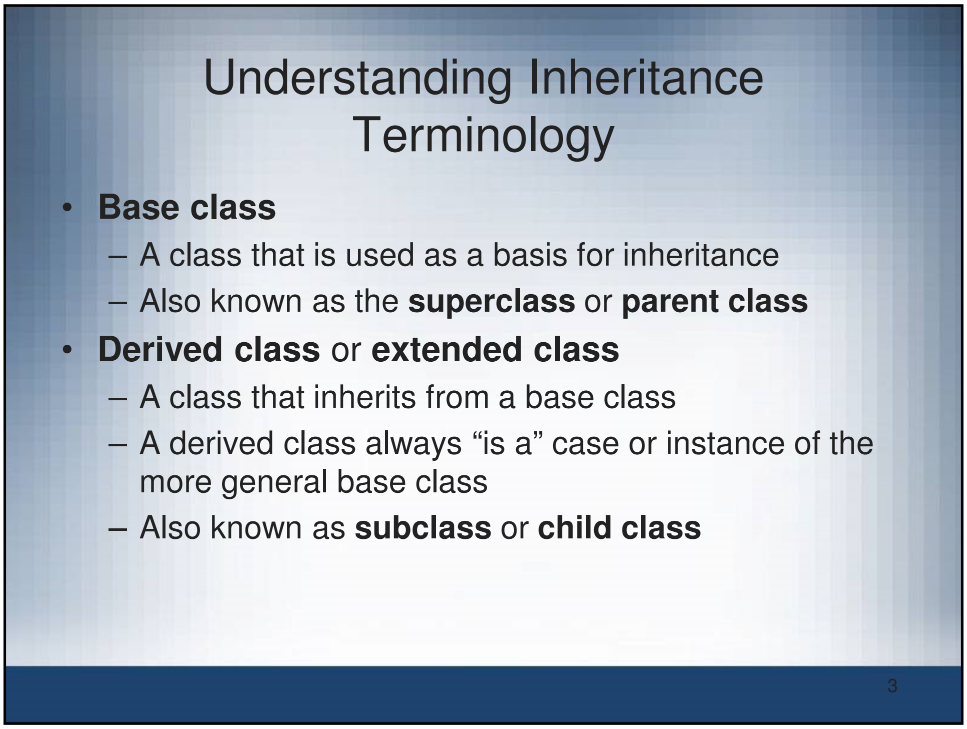
Chapter 10

Introduction to Inheritance



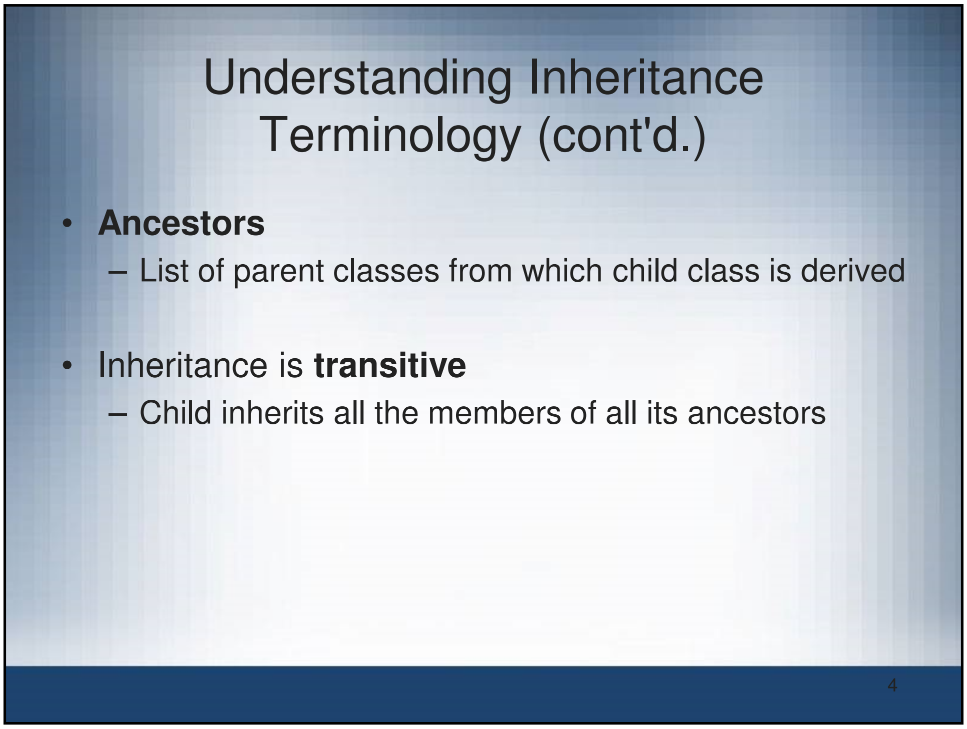
Understanding Inheritance

- **Inheritance**
  - The principle that you can apply knowledge of a general category to more specific objects
- Advantages of inheritance
  - Saves time
  - Reduces the chance of errors
  - Makes it easier to understand the inherited class
  - Makes programs easier to write



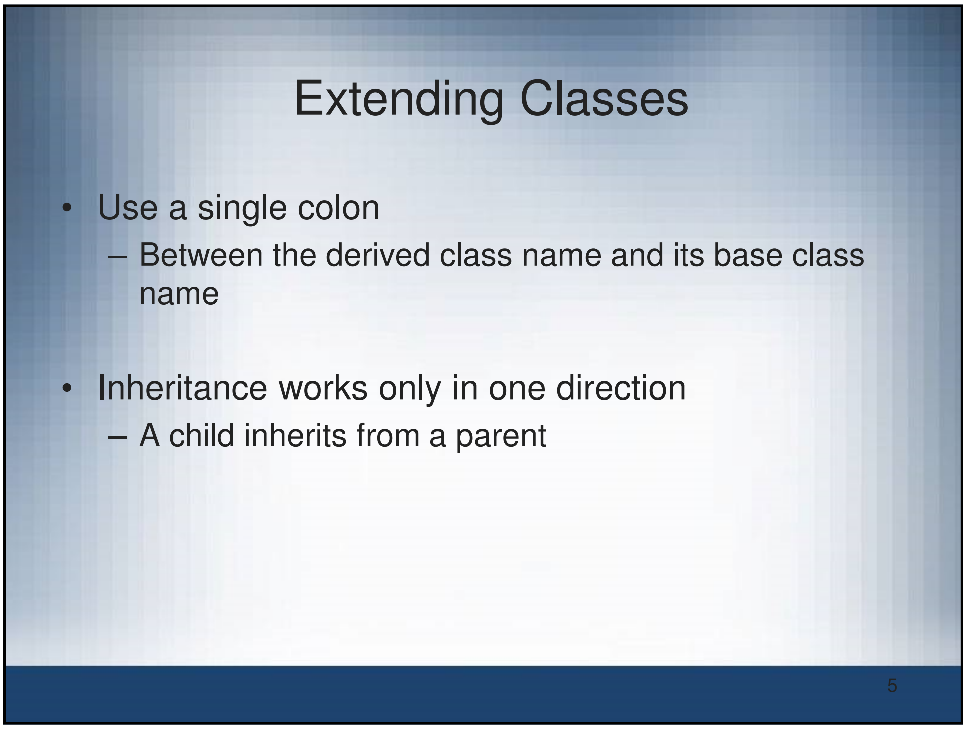
Understanding Inheritance Terminology

- **Base class**
  - A class that is used as a basis for inheritance
  - Also known as the **superclass** or **parent class**
- **Derived class** or **extended class**
  - A class that inherits from a base class
  - A derived class always “is a” case or instance of the more general base class
  - Also known as **subclass** or **child class**



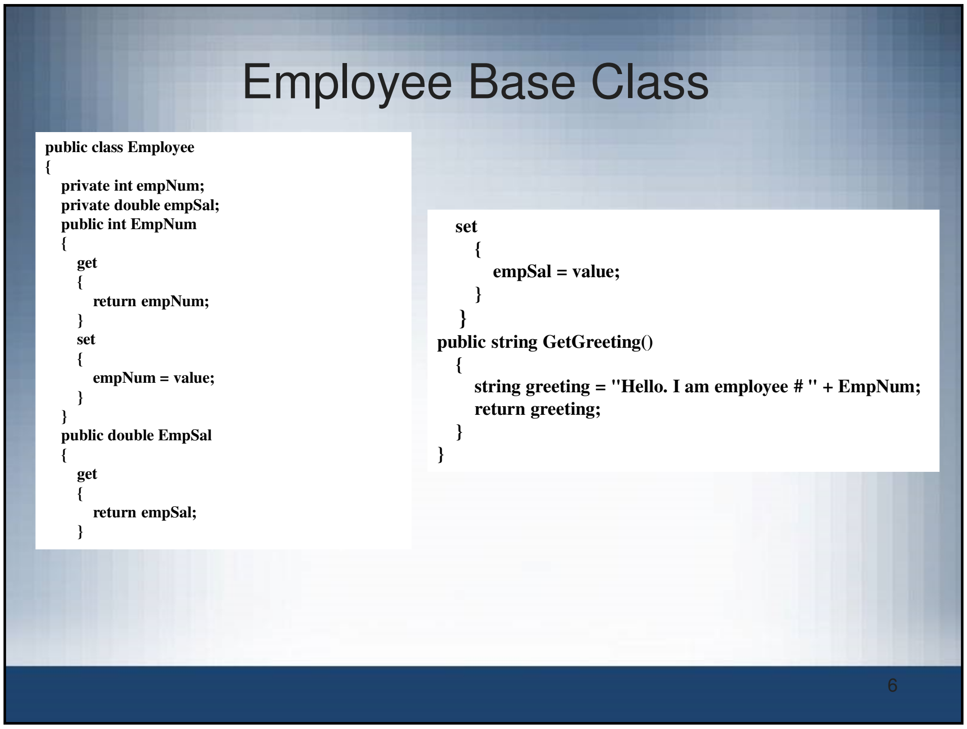
Understanding Inheritance Terminology (cont'd.)

- **Ancestors**
  - List of parent classes from which child class is derived
- Inheritance is **transitive**
  - Child inherits all the members of all its ancestors



Extending Classes

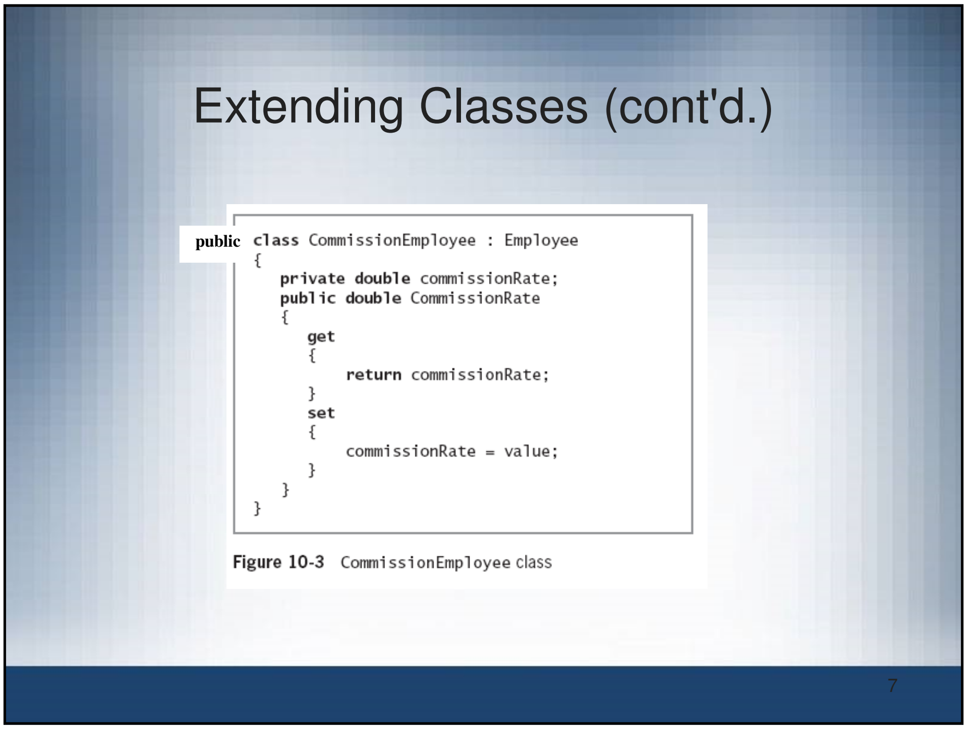
- Use a single colon
  - Between the derived class name and its base class name
- Inheritance works only in one direction
  - A child inherits from a parent



Employee Base Class

```
public class Employee
{
   private int empNum;
   private double empSal;
   public int EmpNum
   {
      get
      {
         return empNum;
      }
      set
      {
         empNum = value;
      }
   }
   public double EmpSal
   {
      get
      {
         return empSal;
      }
```

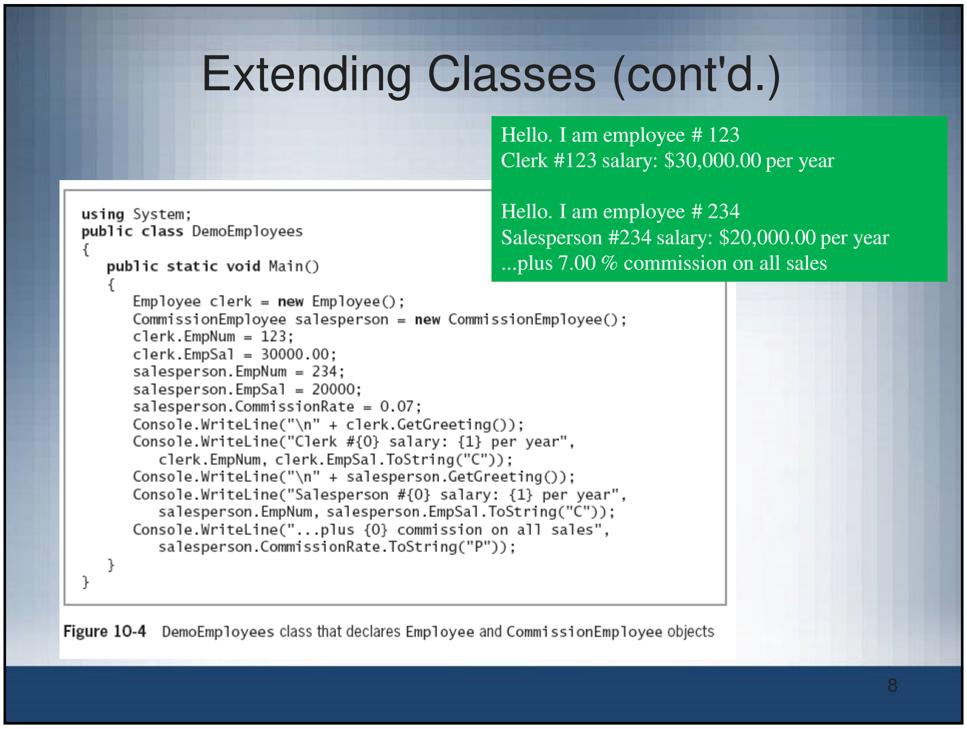
```
      set
      {
         empSal = value;
      }
   }
public string GetGreeting()
   {
      string greeting = "Hello. I am employee # " + EmpNum;
      return greeting;
   }
}
```



Extending Classes (cont'd.)

```
public class CommissionEmployee : Employee
{
   private double commissionRate;
   public double CommissionRate
   {
      get
      {
         return commissionRate;
      }
      set
      {
         commissionRate = value;
      }
   }
}
```

Figure 10-3 CommissionEmployee class



Extending Classes (cont'd.)

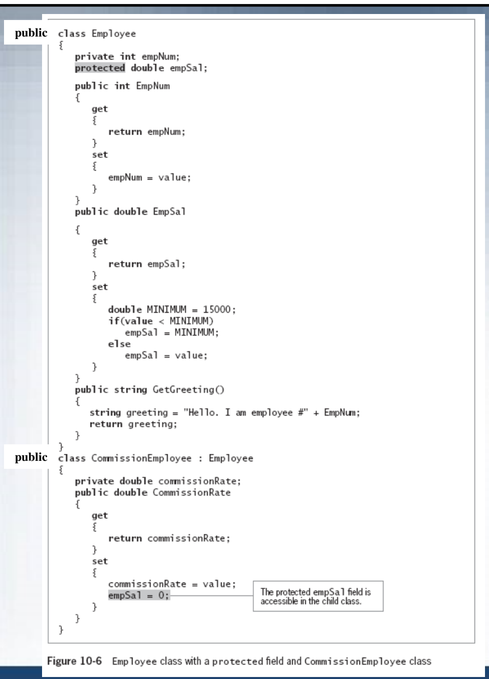
```
using System;
public class DemoEmployees
{
   public static void Main()
   {
      Employee clerk = new Employee();
      CommissionEmployee salesperson = new CommissionEmployee();
      clerk.EmpNum = 123;
      clerk.EmpSal = 30000.00;
      salesperson.EmpNum = 234;
      salesperson.EmpSal = 20000;
      salesperson.CommissionRate = 0.07;
      Console.WriteLine("\n" + clerk.GetGreeting());
      Console.WriteLine("Clerk #{0} salary: {1} per year",
         clerk.EmpNum, clerk.EmpSal.ToString("C"));
      Console.WriteLine("\n" + salesperson.GetGreeting());
      Console.WriteLine("Salesperson #{0} salary: {1} per year",
         salesperson.EmpNum, salesperson.EmpSal.ToString("C"));
      Console.WriteLine("...plus {0} commission on all sales",
         salesperson.CommissionRate.ToString("P"));
   }
}
```

```
Hello. I am employee # 123
Clerk #123 salary: $30,000.00 per year

Hello. I am employee # 234
Salesperson #234 salary: $20,000.00 per year
...plus 7.00 % commission on all sales
```

Figure 10-4 DemoEmployees class that declares Employee and CommissionEmployee objects

Using the `protected` Access Specifier

- Any derived class inherits all the data and methods of its base class
  - Including `private` data and methods
  - Cannot use or modify `private` data and methods directly
- A **`protected`** data field or method
  - Can be used within its own class or in any classes extended from that class
  - Cannot be used by “outside” classes
    - Only used within the “family”
- `protected` methods should be used sparingly

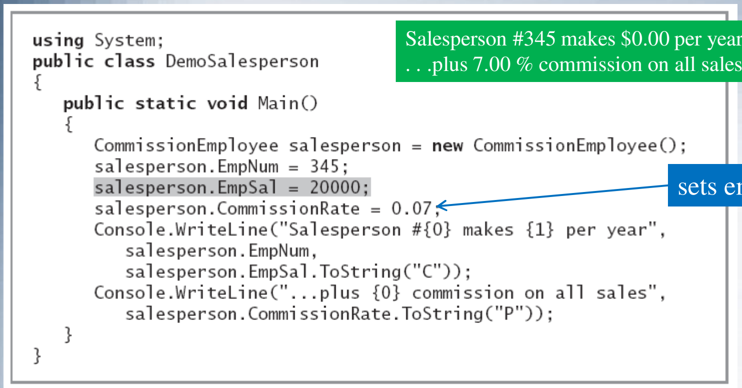


```
public class Employee
{
   private int empNum;
   protected double empSal;

   public int EmpNum
   {
      get
      {
         return empNum;
      }
      set
      {
         empNum = value;
      }
   }
   public double EmpSal

   {
      get
      {
         return empSal;
      }
      set
      {
         double MINIMUM = 15000;
         if(value < MINIMUM)
            empSal = MINIMUM;
         else
            empSal = value;
      }
   }
   public string GetGreeting()
   {
     string greeting = "Hello. I am employee #" + EmpNum;
     return greeting;
   }
}
public class CommissionEmployee : Employee
{
   private double commissionRate;
   public double CommissionRate
   {
      get
      {
         return commissionRate;
      }
      set
      {
         commissionRate = value;
         empSal = 0;
      }
   }
}
```

The protected empSal field is accessible in the child class.

Figure 10-6 Employee class with a protected field and CommissionEmployee class

Using the `protected` Access Specifier (cont'd.)



```
using System;
public class DemoSalesperson
{
   public static void Main()
   {
      CommissionEmployee salesperson = new CommissionEmployee();
      salesperson.EmpNum = 345;
      salesperson.EmpSal = 20000;
      salesperson.CommissionRate = 0.07;
      Console.WriteLine("Salesperson #{0} makes {1} per year",
         salesperson.EmpNum,
         salesperson.EmpSal.ToString("C"));
      Console.WriteLine("...plus {0} commission on all sales",
         salesperson.CommissionRate.ToString("P"));
   }
}
```

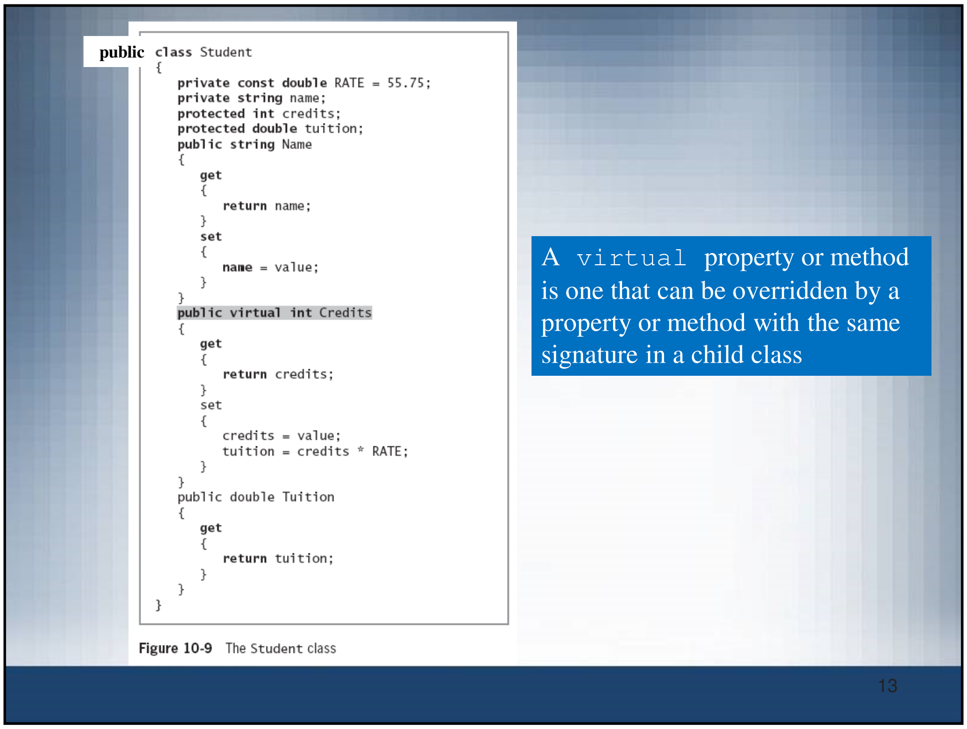
Figure 10-7 The DemoSalesperson program

```
Salesperson #345 makes $0.00 per year
. . .plus 7.00 % commission on all sales
```

sets empSal to 0

Overriding Base Class Methods

- Derived class contains data and methods defined in the original class
- **Polymorphism**
  - Using the same method or property name to indicate different implementations
  - Eg. Although both are vehicles and have an `Operate()` method, a bicycle is operated differently than a truck.
- Derived class can override and “hide” methods and data from the base class

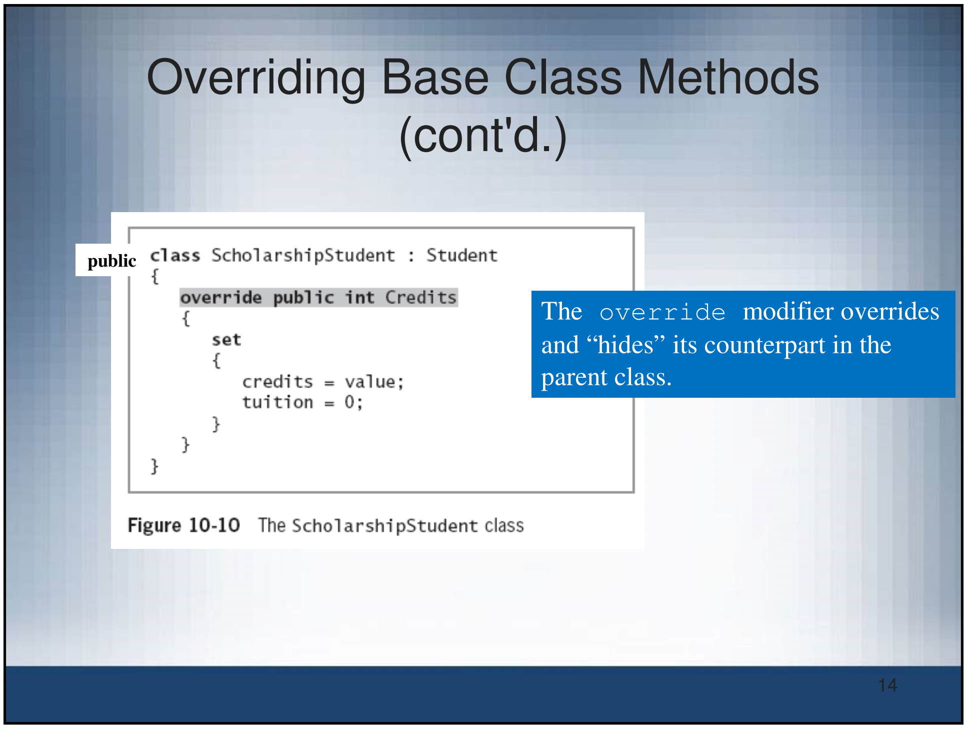


```
public class Student
{
    private const double RATE = 55.75;
    private string name;
    protected int credits;
    protected double tuition;
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    public virtual int Credits
    {
        get
        {
            return credits;
        }
        set
        {
            credits = value;
            tuition = credits * RATE;
        }
    }
    public double Tuition
    {
        get
        {
            return tuition;
        }
    }
}
```

Figure 10-9 The Student class

A `virtual` property or method is one that can be overridden by a property or method with the same signature in a child class

13



Overriding Base Class Methods (cont'd.)

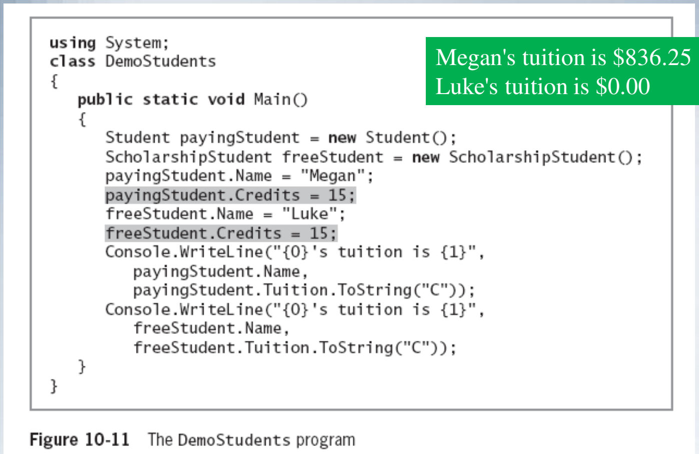
```
public class ScholarshipStudent : Student
{
    override public int Credits
    {
        set
        {
            credits = value;
            tuition = 0;
        }
    }
}
```

Figure 10-10 The ScholarshipStudent class

The `override` modifier overrides and “hides” its counterpart in the parent class.

14

Overriding Base Class Methods (cont'd.)



```
using System;
class DemoStudents
{
   public static void Main()
   {
      Student payingStudent = new Student();
      ScholarshipStudent freeStudent = new ScholarshipStudent();
      payingStudent.Name = "Megan";
      payingStudent.Credits = 15;
      freeStudent.Name = "Luke";
      freeStudent.Credits = 15;
      Console.WriteLine("{0}'s tuition is {1}",
         payingStudent.Name,
         payingStudent.Tuition.ToString("C"));
      Console.WriteLine("{0}'s tuition is {1}",
         freeStudent.Name,
         freeStudent.Tuition.ToString("C"));
   }
}
```

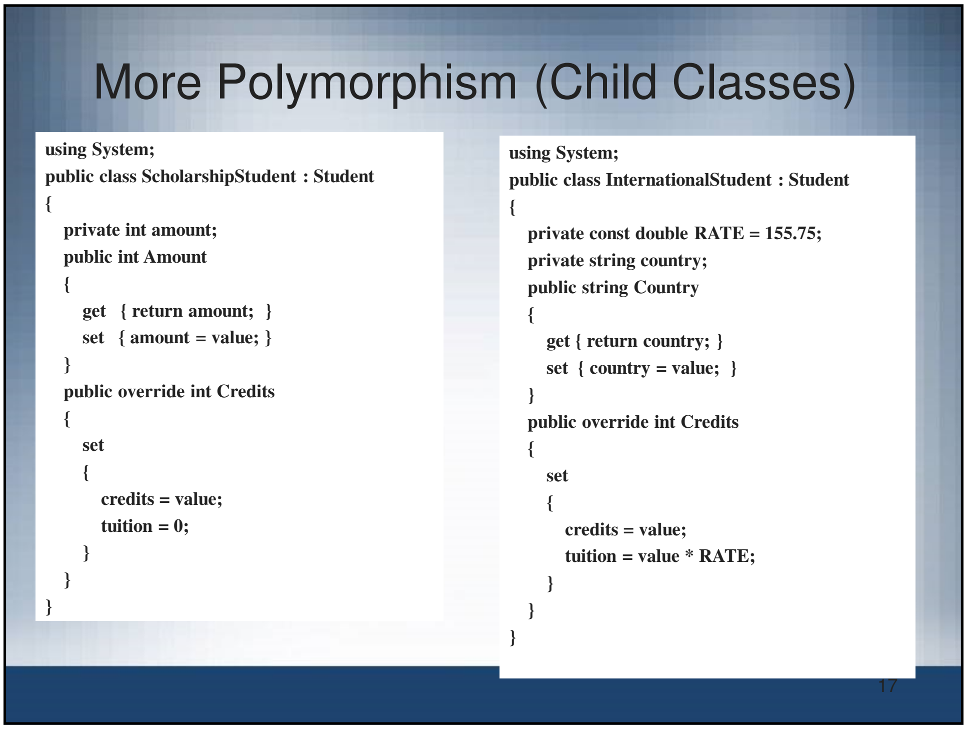
Figure 10-11 The DemoStudents program

```
Megan's tuition is $836.25
Luke's tuition is $0.00
```

More Polymorphism (Parent Class)

```
public class Student
{
   private const double RATE = 55.75;
   private string name;
   protected int credits;
   protected double tuition;
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }
```

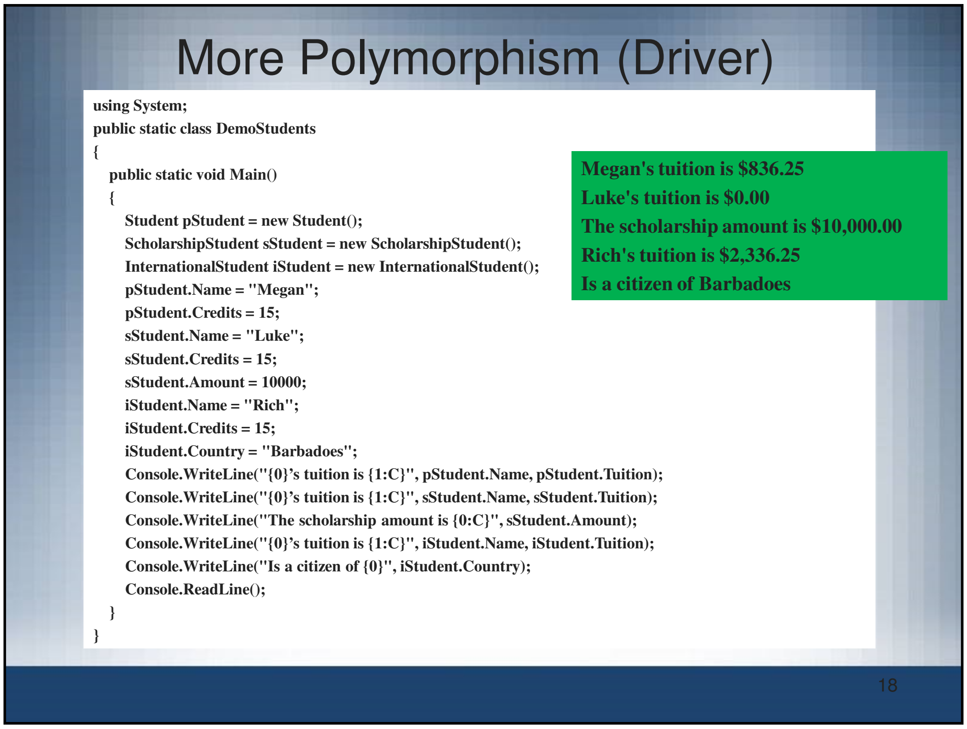
```
public virtual int Credits
  {
     get
     {
        return credits;
     }

     set
     {
        credits = value;
        tuition = credits * RATE;
     }
  }
  public double Tuition
  {
     get
     {
        return tuition;
     }
  }
}
```



More Polymorphism (Child Classes)

```
using System;
public class ScholarshipStudent : Student
{
   private int amount;
   public int Amount
   {
      get   { return amount;  }
      set   { amount = value; }
   }
   public override int Credits
   {
      set
      {
         credits = value;
         tuition = 0;
      }
   }
}
```

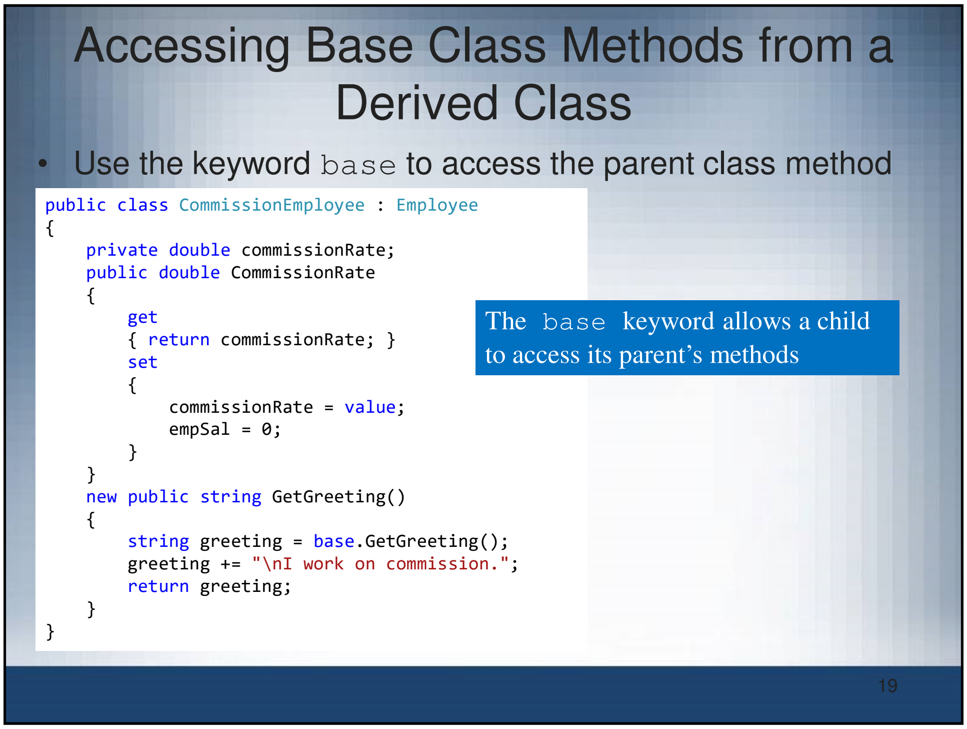
```
using System;
public class InternationalStudent : Student
{
   private const double RATE = 155.75;
   private string country;
   public string Country
   {
      get { return country; }
      set  { country = value;  }
   }
   public override int Credits
   {
      set
      {
         credits = value;
         tuition = value * RATE;
      }
   }
}
```



More Polymorphism (Driver)

```
using System;
public static class DemoStudents
{
   public static void Main()
   {
      Student pStudent = new Student();
      ScholarshipStudent sStudent = new ScholarshipStudent();
      InternationalStudent iStudent = new InternationalStudent();
      pStudent.Name = "Megan";
      pStudent.Credits = 15;
      sStudent.Name = "Luke";
      sStudent.Credits = 15;
      sStudent.Amount = 10000;
      iStudent.Name = "Rich";
      iStudent.Credits = 15;
      iStudent.Country = "Barbadoes";
      Console.WriteLine("{0}'s tuition is {1:C}", pStudent.Name, pStudent.Tuition);
      Console.WriteLine("{0}'s tuition is {1:C}", sStudent.Name, sStudent.Tuition);
      Console.WriteLine("The scholarship amount is {0:C}", sStudent.Amount);
      Console.WriteLine("{0}'s tuition is {1:C}", iStudent.Name, iStudent.Tuition);
      Console.WriteLine("Is a citizen of {0}", iStudent.Country);
      Console.ReadLine();
   }
}
```

```
Megan's tuition is $836.25
Luke's tuition is $0.00
The scholarship amount is $10,000.00
Rich's tuition is $2,336.25
Is a citizen of Barbadoes
```

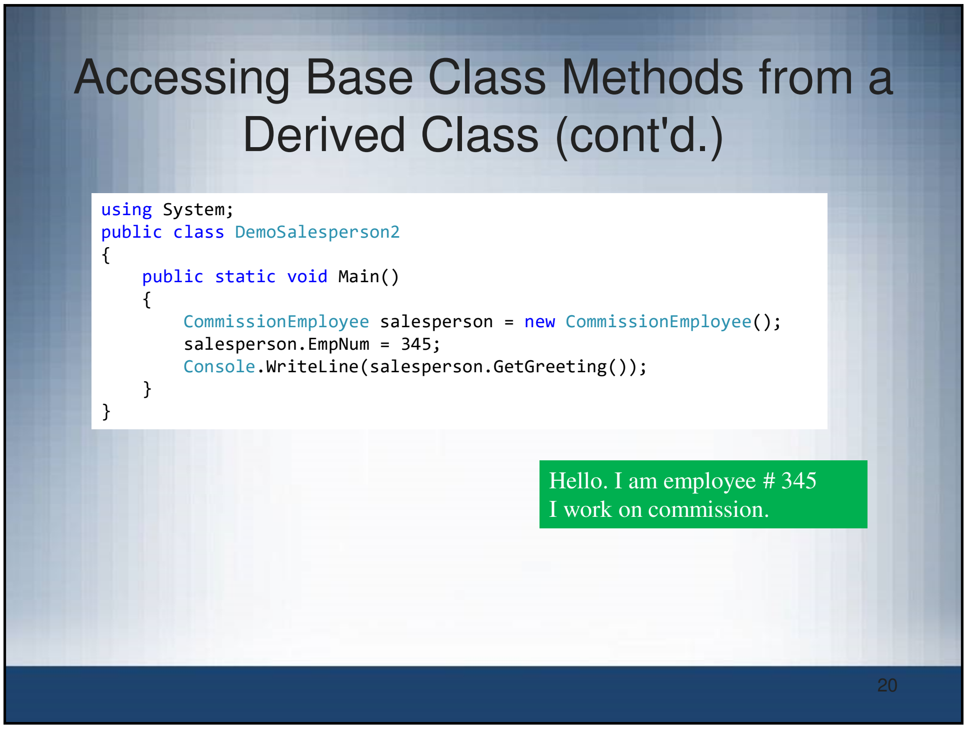


Accessing Base Class Methods from a Derived Class

- Use the keyword `base` to access the parent class method

```
public class CommissionEmployee : Employee
{
    private double commissionRate;
    public double CommissionRate
    {
        get
        { return commissionRate; }
        set
        {
            commissionRate = value;
            empSal = 0;
        }
    }
    new public string GetGreeting()
    {
        string greeting = base.GetGreeting();
        greeting += "\nI work on commission.";
        return greeting;
    }
}
```

The `base` keyword allows a child to access its parent's methods



Accessing Base Class Methods from a Derived Class (cont'd.)

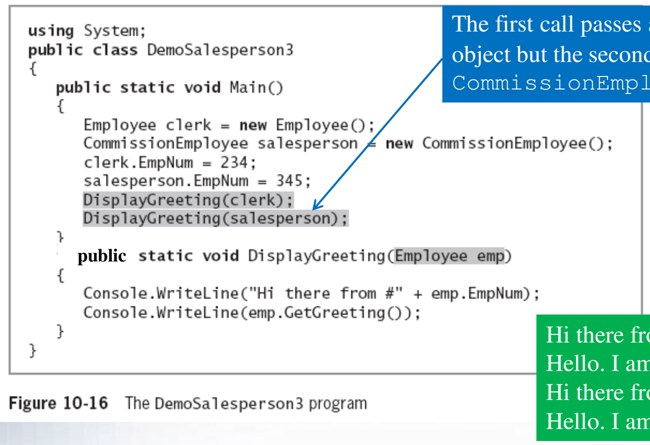
```
using System;
public class DemoSalesperson2
{
    public static void Main()
    {
        CommissionEmployee salesperson = new CommissionEmployee();
        salesperson.EmpNum = 345;
        Console.WriteLine(salesperson.GetGreeting());
    }
}
```

```
Hello. I am employee # 345
I work on commission.
```

Understanding How a Derived Class Object "is an" Instance of the Base Class

- Every derived class object "is a" specific instance of both the derived class and the base class
- You can assign a derived class object to an object of any of its superclass types
  - C# makes an **implicit conversion** from derived class to base class

Understanding How a Derived Class Object "is an" Instance of the Base Class (cont'd.)

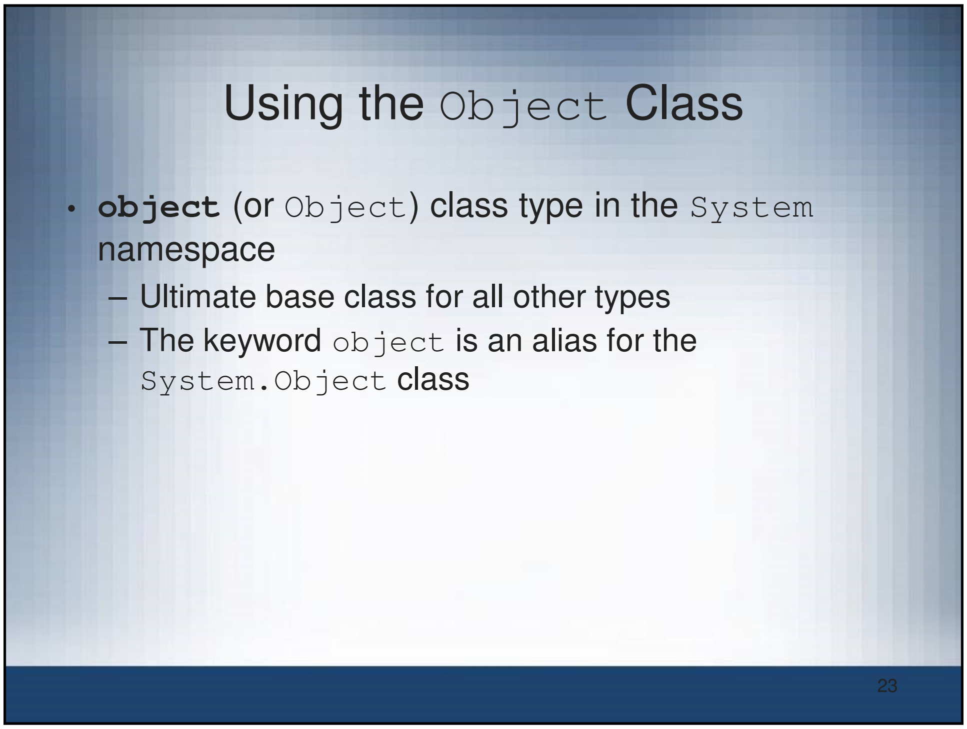


```
using System;
public class DemoSalesperson3
{
   public static void Main()
   {
      Employee clerk = new Employee();
      CommissionEmployee salesperson = new CommissionEmployee();
      clerk.EmpNum = 234;
      salesperson.EmpNum = 345;
      DisplayGreeting(clerk);
      DisplayGreeting(salesperson);
   }
   public static void DisplayGreeting(Employee emp)
   {
      Console.WriteLine("Hi there from #" + emp.EmpNum);
      Console.WriteLine(emp.GetGreeting());
   }
}
```

Figure 10-16 The DemoSalesperson3 program

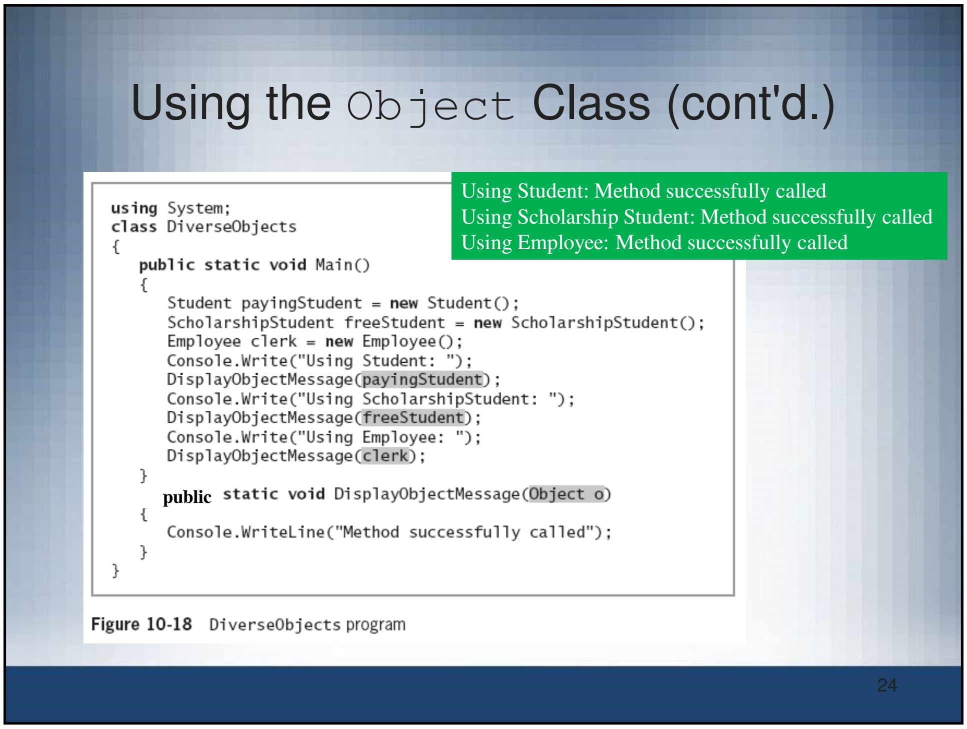
The first call passes an `Employee` object but the second call passes a `CommissionEmployee` object

```
Hi there from #234
Hello. I am employee # 234
Hi there from #345
Hello. I am employee # 345
```



Using the `Object` Class

- **`object`** (or `Object`) class type in the `System` namespace
  - Ultimate base class for all other types
  - The keyword `object` is an alias for the `System.Object` class



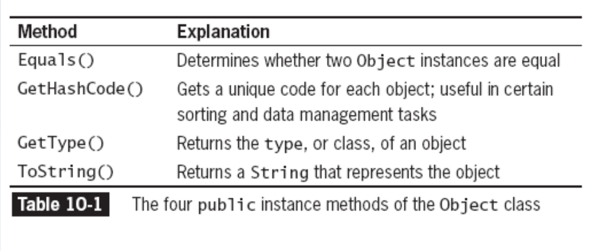
Using the `Object` Class (cont'd.)

```
using System;
class DiverseObjects
{
   public static void Main()
   {
      Student payingStudent = new Student();
      ScholarshipStudent freeStudent = new ScholarshipStudent();
      Employee clerk = new Employee();
      Console.Write("Using Student: ");
      DisplayObjectMessage(payingStudent);
      Console.Write("Using ScholarshipStudent: ");
      DisplayObjectMessage(freeStudent);
      Console.Write("Using Employee: ");
      DisplayObjectMessage(clerk);
   }
   public static void DisplayObjectMessage(Object o)
   {
      Console.WriteLine("Method successfully called");
   }
}
```

```
Using Student: Method successfully called
Using Scholarship Student: Method successfully called
Using Employee: Method successfully called
```

Figure 10-18 DiverseObjects program

Using the Object Class (cont'd.)



Method	Explanation
Equals()	Determines whether two Object instances are equal
GetHashCode()	Gets a unique code for each object; useful in certain sorting and data management tasks
GetType()	Returns the type, or class, of an object
ToString()	Returns a String that represents the object

Table 10-1 The four public instance methods of the Object class

Using the Object Class's GetType() Method

- GetType() method
  - Returns an object's type, or class
- Example

```
Employee someWorker = new Employee();
Console.WriteLine(someWorker.GetType());
```

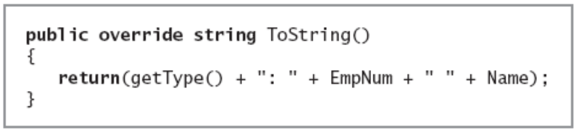
Would output Employee

```
Student csStudent = new Student();
Console.WriteLine(csStudent.GetType());
```

Would output Student

Using the `Object` Class's `ToString()` Method

- `ToString()` method
  - Returns a string that holds the class name
    - Just as `GetType()` does
- You should override this method in your own class to make it more meaningful (Employee: 1234 Hurley)

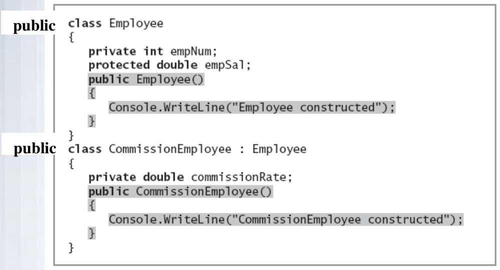


```
public override string ToString()
{
   return(getType() + ": " + EmpNum + " " + Name);
}
```

Figure 10-20 An Employee class ToString() method

Working with Base Class Constructors

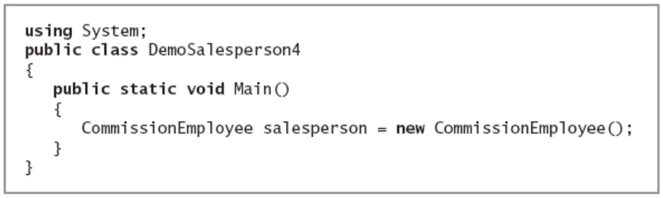
- Instantiating an object of a derived class
  - Calls the constructor for both the base class and the derived class
    - The base class constructor must execute first



```
public class Employee
{
   private int empNum;
   protected double empSal;
   public Employee()
   {
      Console.WriteLine("Employee constructed");
   }
}
public class CommissionEmployee : Employee
{
   private double commissionRate;
   public CommissionEmployee()
   {
      Console.WriteLine("CommissionEmployee constructed");
   }
}
```

Figure 10-24 Employee and CommissionEmployee classes with parameterless constructors

Working with Base Class Constructors (cont'd.)



```
using System;
public class DemoSalesperson4
{
   public static void Main()
   {
      CommissionEmployee salesperson = new CommissionEmployee();
   }
}
```

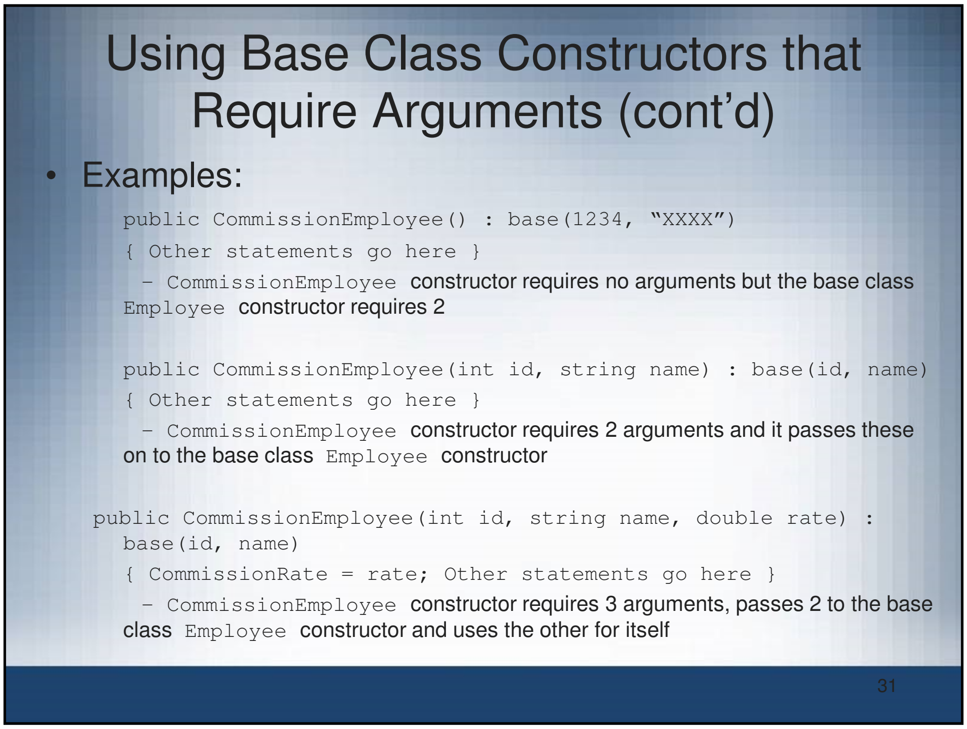
Figure 10-25 The DemoSalesperson4 program

Employee constructed
ComissionEmployee constructed

Using Base Class Constructors that Require Arguments

- When a base class constructor requires arguments:
  - Include a constructor for each derived class you create
- The derived class constructor can contain any number of statements
  - Within the header, provide values for any arguments required by the base class constructor
    - Using the keyword **`base`**
- Example
  - Assume you have a base class Employee constructor that takes two parameters (int, string)

```
public Employee (int eNum, string eName) { code }
```



Using Base Class Constructors that Require Arguments (cont'd)

- Examples:

```
public CommissionEmployee() : base(1234, "XXXX")
{ Other statements go here }
```

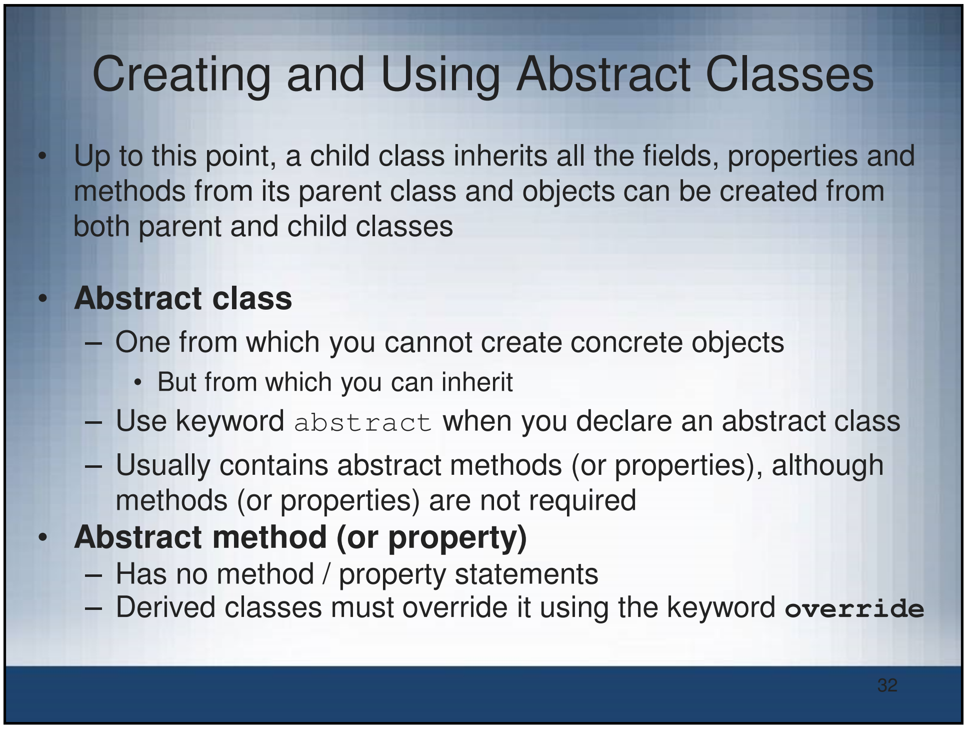
  – `CommissionEmployee` constructor requires no arguments but the base class `Employee` constructor requires 2

```
public CommissionEmployee(int id, string name) : base(id, name)
{ Other statements go here }
```

  – `CommissionEmployee` constructor requires 2 arguments and it passes these on to the base class `Employee` constructor

```
public CommissionEmployee(int id, string name, double rate) :
   base(id, name)
   { CommissionRate = rate; Other statements go here }
```

  – `CommissionEmployee` constructor requires 3 arguments, passes 2 to the base class `Employee` constructor and uses the other for itself



Creating and Using Abstract Classes

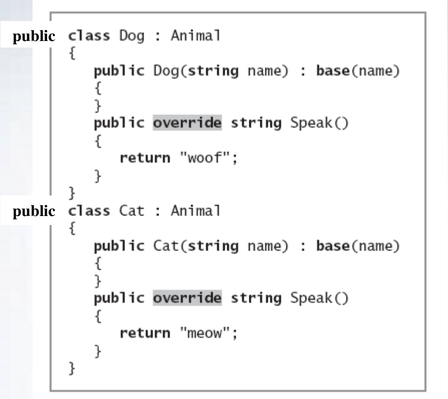
- Up to this point, a child class inherits all the fields, properties and methods from its parent class and objects can be created from both parent and child classes

- **Abstract class**
  - One from which you cannot create concrete objects
    - But from which you can inherit
  - Use keyword `abstract` when you declare an abstract class
  - Usually contains abstract methods (or properties), although methods (or properties) are not required
- **Abstract method (or property)**
  - Has no method / property statements
  - Derived classes must override it using the keyword **`override`**

Creating and Using Abstract Classes (cont'd.)

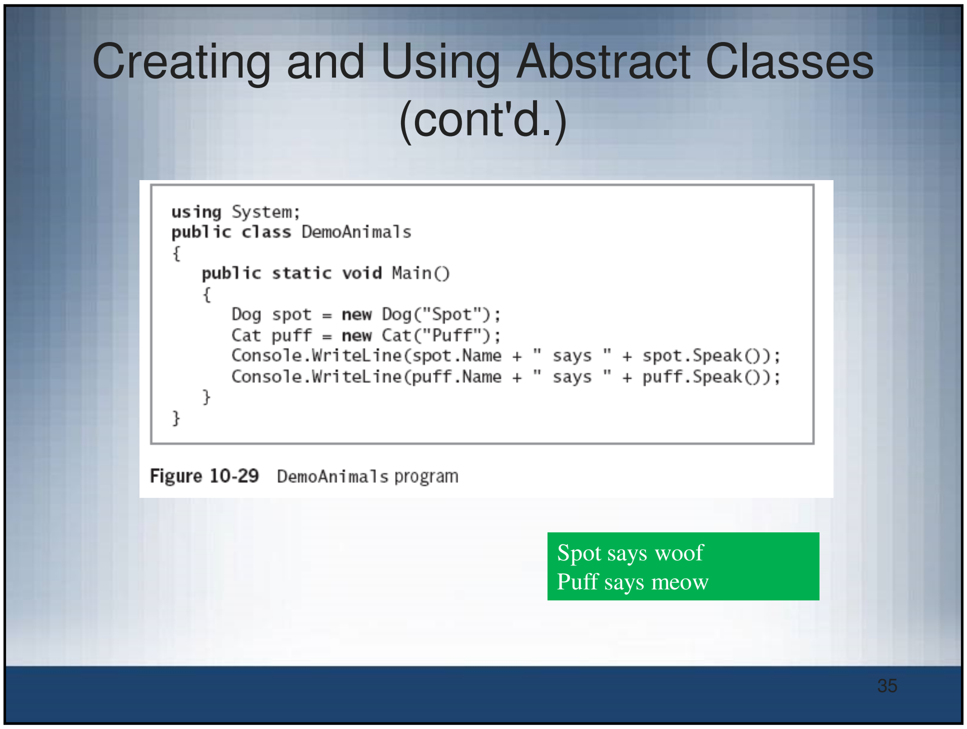
```
public abstract class Animal
{
    private string name;
    public Animal (string valName)
    {
       name = valName;
     }
    public string Name
    {
       get
       {
          return name;
       }
    }
    public abstract string Speak();
}
```

Creating and Using Abstract Classes (cont'd.)



```
public class Dog : Animal
{
   public Dog(string name) : base(name)
   {
   }
   public override string Speak()
   {
      return "woof";
   }
}
public class Cat : Animal
{
   public Cat(string name) : base(name)
   {
   }
   public override string Speak()
   {
      return "meow";
   }
}
```

Figure 10-28 Dog and Cat classes

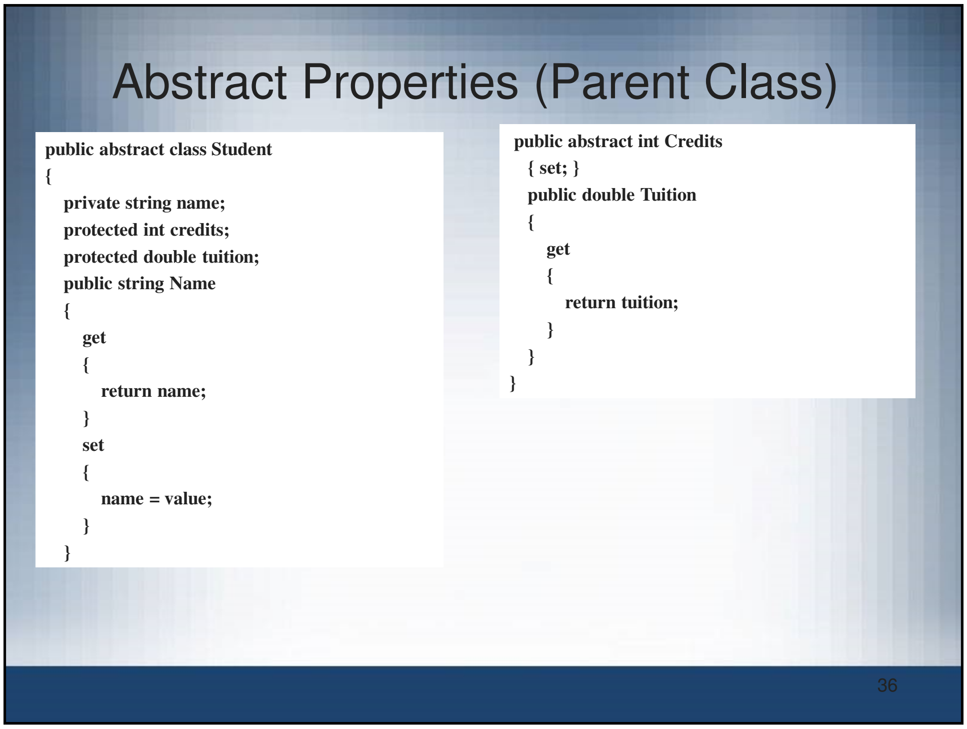


Creating and Using Abstract Classes (cont'd.)

```
using System;
public class DemoAnimals
{
   public static void Main()
   {
      Dog spot = new Dog("Spot");
      Cat puff = new Cat("Puff");
      Console.WriteLine(spot.Name + " says " + spot.Speak());
      Console.WriteLine(puff.Name + " says " + puff.Speak());
   }
}
```

Figure 10-29 DemoAnimals program

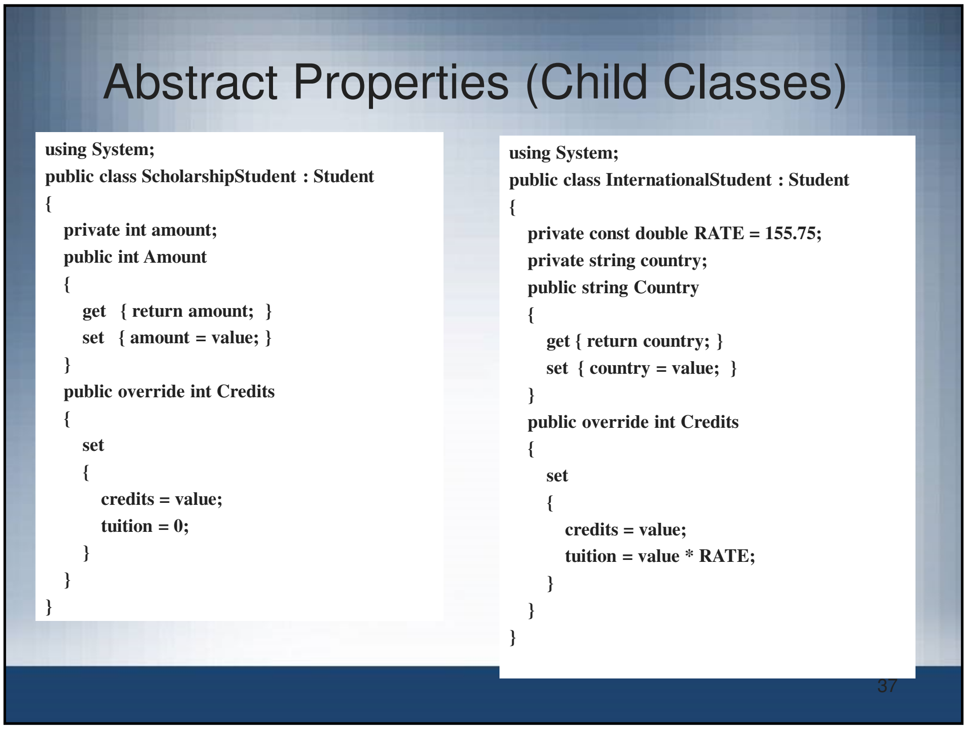
```
Spot says woof
Puff says meow
```



Abstract Properties (Parent Class)

```
public abstract class Student
{
   private string name;
   protected int credits;
   protected double tuition;
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }
```

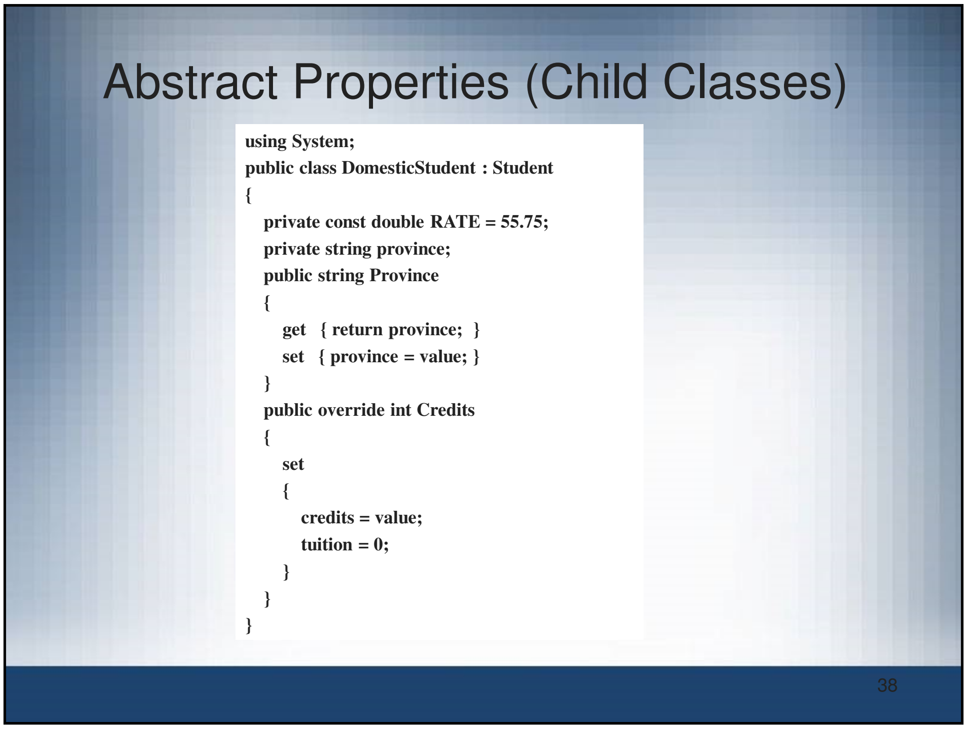
```
public abstract int Credits
   { set; }
   public double Tuition
   {
      get
      {
         return tuition;
      }
   }
}
```



Abstract Properties (Child Classes)

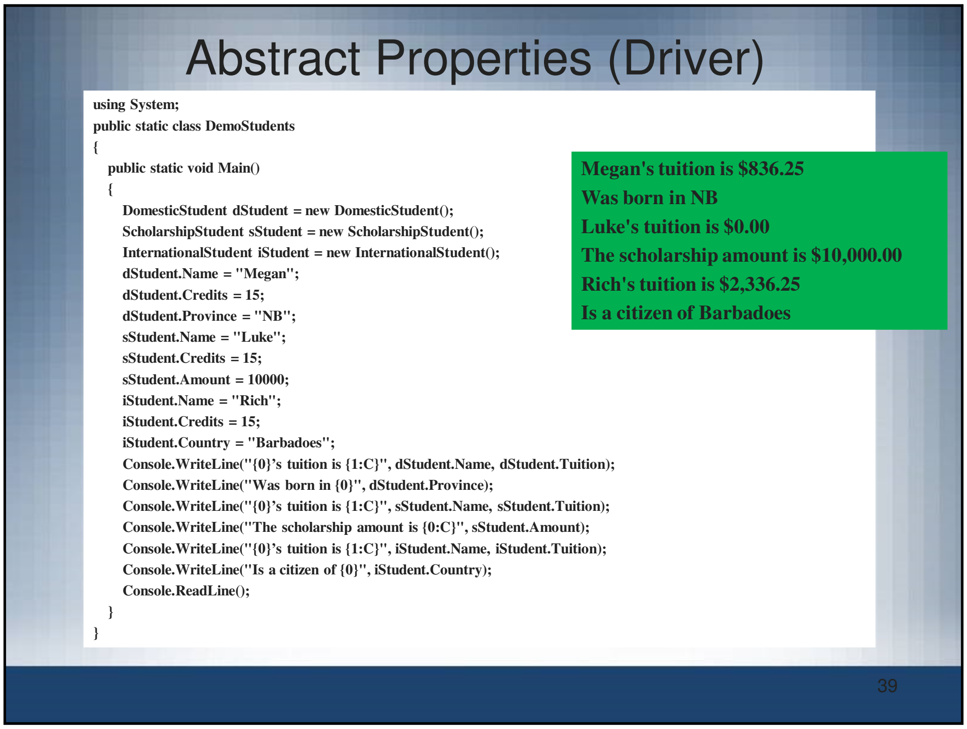
```
using System;
public class ScholarshipStudent : Student
{
   private int amount;
   public int Amount
   {
      get   { return amount;  }
      set   { amount = value; }
   }
   public override int Credits
   {
      set
      {
         credits = value;
         tuition = 0;
      }
   }
}
```

```
using System;
public class InternationalStudent : Student
{
   private const double RATE = 155.75;
   private string country;
   public string Country
   {
      get { return country; }
      set  { country = value;  }
   }
   public override int Credits
   {
      set
      {
         credits = value;
         tuition = value * RATE;
      }
   }
}
```



Abstract Properties (Child Classes)

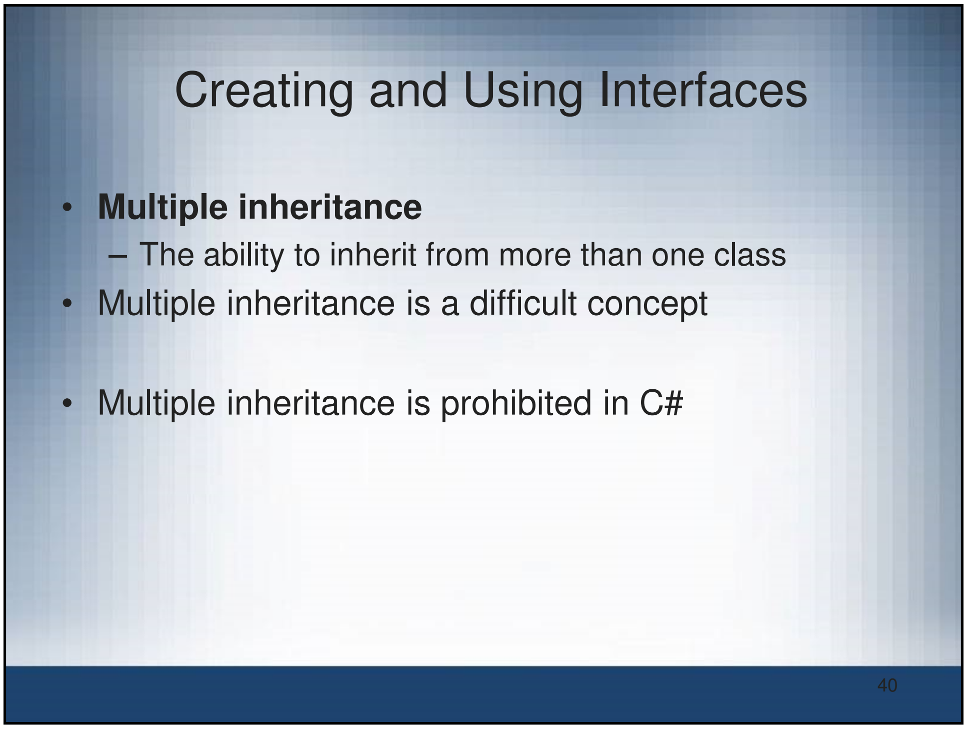
```
using System;
public class DomesticStudent : Student
{
   private const double RATE = 55.75;
   private string province;
   public string Province
   {
      get   { return province;  }
      set   { province = value; }
   }
   public override int Credits
   {
      set
      {
         credits = value;
         tuition = 0;
      }
   }
}
```



Abstract Properties (Driver)

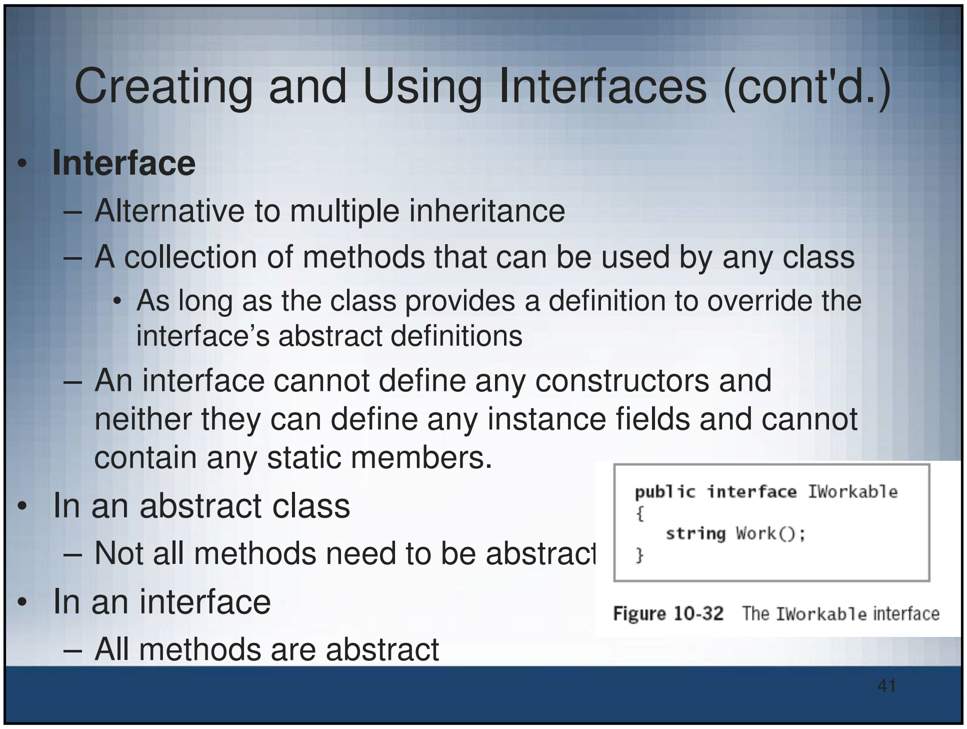
```
using System;
public static class DemoStudents
{
   public static void Main()
   {
      DomesticStudent dStudent = new DomesticStudent();
      ScholarshipStudent sStudent = new ScholarshipStudent();
      InternationalStudent iStudent = new InternationalStudent();
      dStudent.Name = "Megan";
      dStudent.Credits = 15;
      dStudent.Province = "NB";
      sStudent.Name = "Luke";
      sStudent.Credits = 15;
      sStudent.Amount = 10000;
      iStudent.Name = "Rich";
      iStudent.Credits = 15;
      iStudent.Country = "Barbadoes";
      Console.WriteLine("{0}'s tuition is {1:C}", dStudent.Name, dStudent.Tuition);
      Console.WriteLine("Was born in {0}", dStudent.Province);
      Console.WriteLine("{0}'s tuition is {1:C}", sStudent.Name, sStudent.Tuition);
      Console.WriteLine("The scholarship amount is {0:C}", sStudent.Amount);
      Console.WriteLine("{0}'s tuition is {1:C}", iStudent.Name, iStudent.Tuition);
      Console.WriteLine("Is a citizen of {0}", iStudent.Country);
      Console.ReadLine();
   }
}
```

```
Megan's tuition is $836.25
Was born in NB
Luke's tuition is $0.00
The scholarship amount is $10,000.00
Rich's tuition is $2,336.25
Is a citizen of Barbadoes
```



Creating and Using Interfaces

- **Multiple inheritance**
  - The ability to inherit from more than one class
- Multiple inheritance is a difficult concept
- Multiple inheritance is prohibited in C#

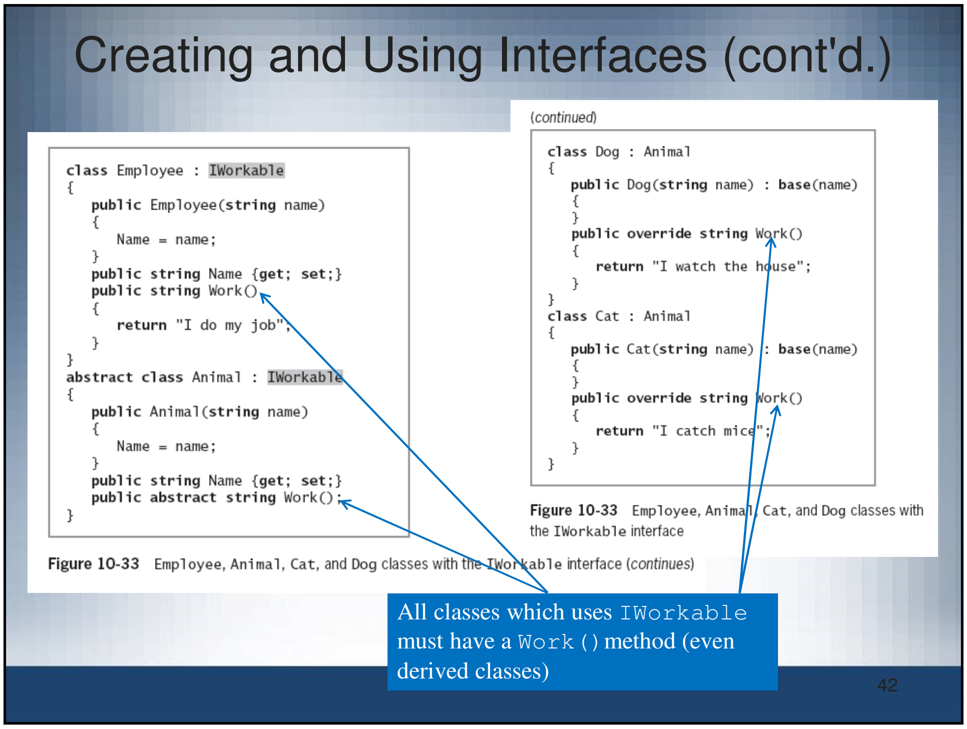


Creating and Using Interfaces (cont'd.)

- **Interface**
  - Alternative to multiple inheritance
  - A collection of methods that can be used by any class
    - As long as the class provides a definition to override the interface's abstract definitions
  - An interface cannot define any constructors and neither they can define any instance fields and cannot contain any static members.
- In an abstract class
  - Not all methods need to be abstract
- In an interface
  - All methods are abstract

```
public interface IWorkable
{
    string Work();
}
```

Figure 10-32 The IWorkable interface



Creating and Using Interfaces (cont'd.)

```
class Employee : IWorkable
{
   public Employee(string name)
   {
      Name = name;
   }
   public string Name {get; set;}
   public string Work()
   {
      return "I do my job";
   }
}
abstract class Animal : IWorkable
{
   public Animal(string name)
   {
      Name = name;
   }
   public string Name {get; set;}
   public abstract string Work();
}
```

Figure 10-33 Employee, Animal, Cat, and Dog classes with the IWorkable interface (*continues*)

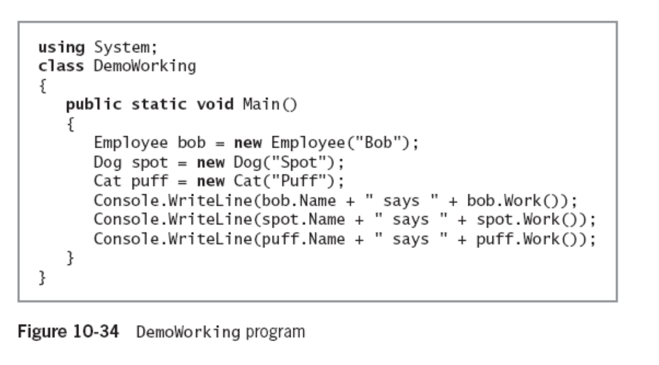
(*continued*)

```
class Dog : Animal
{
   public Dog(string name) : base(name)
   {
   }
   public override string Work()
   {
      return "I watch the house";
   }
}
class Cat : Animal
{
   public Cat(string name) : base(name)
   {
   }
   public override string Work()
   {
      return "I catch mice";
   }
}
```

Figure 10-33 Employee, Animal, Cat, and Dog classes with the IWorkable interface

All classes which uses IWorkable must have a Work() method (even derived classes)

Creating and Using Interfaces (cont'd.)



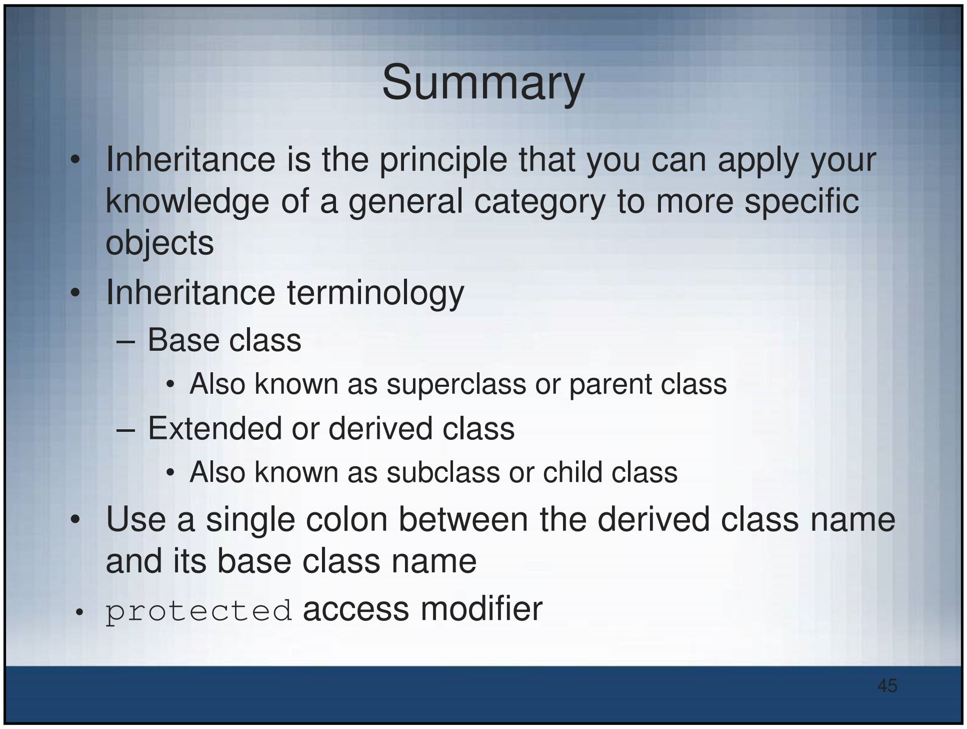
```
using System;
class DemoWorking
{
   public static void Main()
   {
      Employee bob = new Employee("Bob");
      Dog spot = new Dog("Spot");
      Cat puff = new Cat("Puff");
      Console.WriteLine(bob.Name + " says " + bob.Work());
      Console.WriteLine(spot.Name + " says " + spot.Work());
      Console.WriteLine(puff.Name + " says " + puff.Work());
   }
}
```

Figure 10-34 DemoWorking program

```
Bob says I do my job
Spot says I watch the house
Puff says I catch mice
```

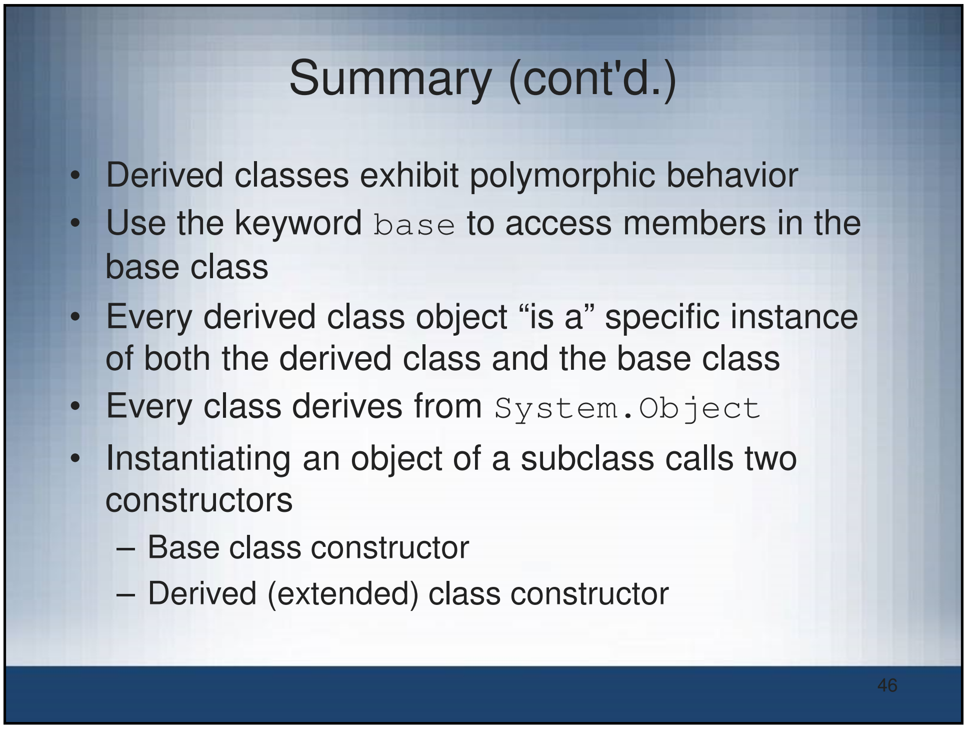
Creating and Using Interfaces (cont'd.)

- You cannot instantiate concrete objects from either abstract classes or interfaces
- A class can inherit from only one base class
  - However, it can implement any number of interfaces
- You create an interface when you want derived classes to override every method
- Interfaces provide you with another way to exhibit polymorphic behavior



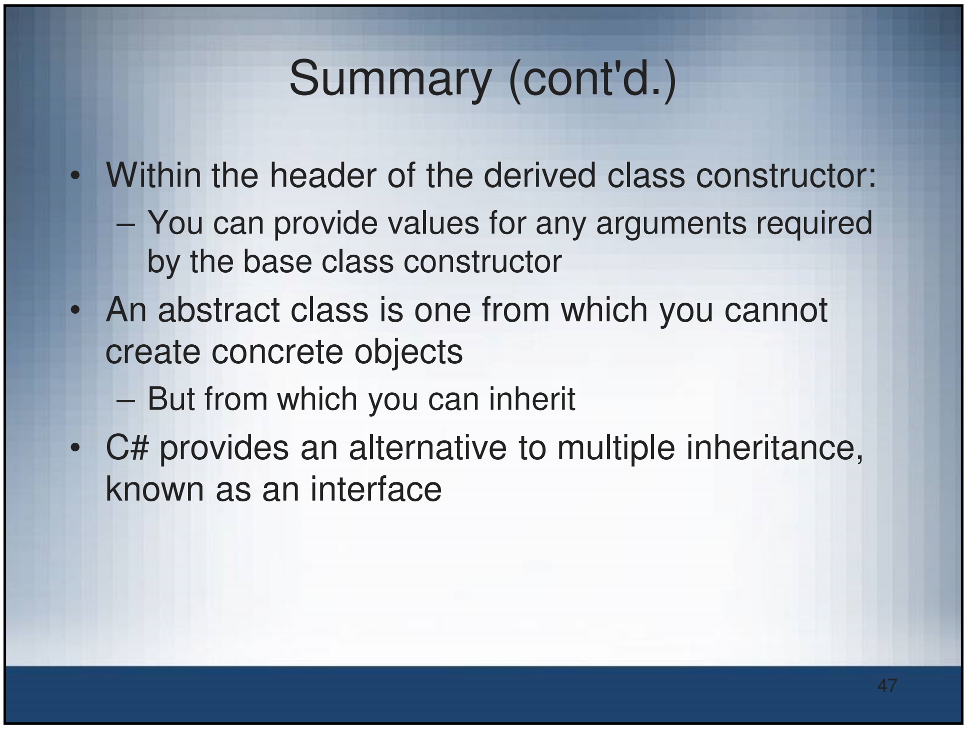
Summary

- Inheritance is the principle that you can apply your knowledge of a general category to more specific objects
- Inheritance terminology
  - Base class
    - Also known as superclass or parent class
  - Extended or derived class
    - Also known as subclass or child class
- Use a single colon between the derived class name and its base class name
- `protected` access modifier



Summary (cont'd.)

- Derived classes exhibit polymorphic behavior
- Use the keyword `base` to access members in the base class
- Every derived class object “is a” specific instance of both the derived class and the base class
- Every class derives from `System.Object`
- Instantiating an object of a subclass calls two constructors
  - Base class constructor
  - Derived (extended) class constructor



Summary (cont'd.)

- Within the header of the derived class constructor:
  - You can provide values for any arguments required by the base class constructor
- An abstract class is one from which you cannot create concrete objects
  - But from which you can inherit
- C# provides an alternative to multiple inheritance, known as an interface

47