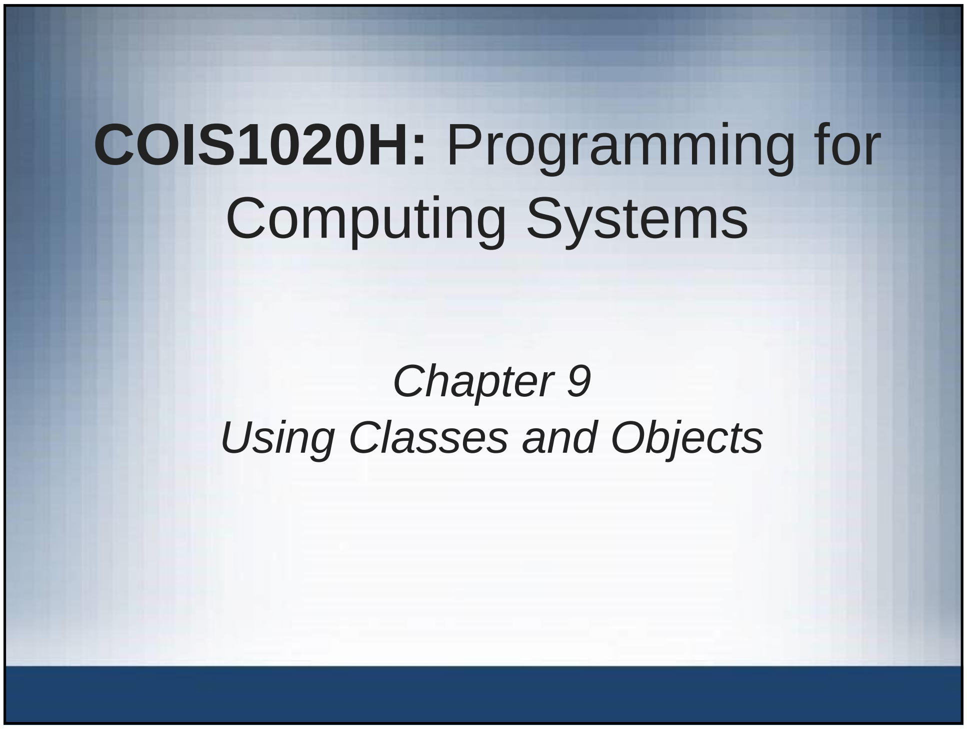
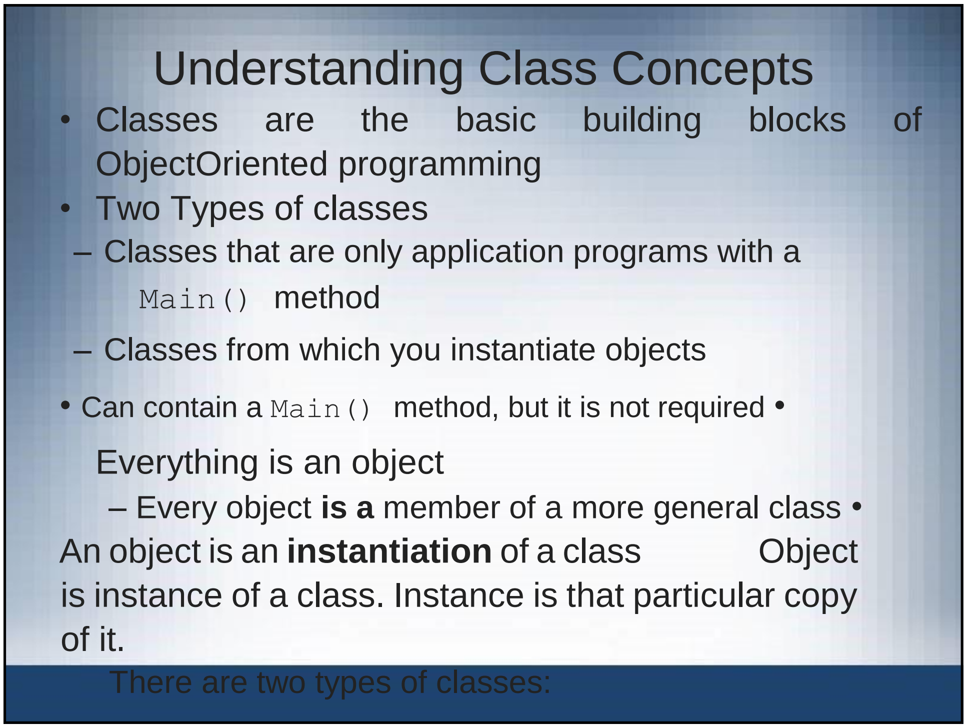


COIS1020H: Programming for Computing Systems

Chapter 9
Using Classes and Objects



Understanding Class Concepts

- Classes are the basic building blocks of ObjectOriented programming
- Two Types of classes
  – Classes that are only application programs with a `Main()` method
  – Classes from which you instantiate objects

• Can contain a `Main()` method, but it is not required •

Everything is an object

– Every object **is a** member of a more general class •

An object is an **instantiation** of a class Object is instance of a class. Instance is that particular copy of it.

There are two types of classes:

1)Static: we cannot create objects from these classes eg: Array, Console (Console.Write, Console.WriteLine..), Math

2) Dynamic: we can create objects from these classes

Understanding Class Concepts (cont'd.)

- **Instance variables (**also called **fields)**
  - Data components of a class
- **State**
  - Set of contents of an object’s instance variables
- **Instance methods**
  - Methods associated with objects
  - Every instance of the class has the same methods
- **Class client** or **class user**
  - Program or class that instantiates objects of another prewritten class (such as **Console**)

Creating a Class from Which Objects Can Be Instantiated

- **Class header** or **class definition** parts
  - An optional access modifier
  - The keyword `class`
  - Any legal identifier for the name of your class
- **Class access modifiers**

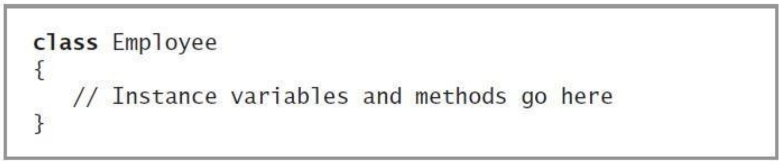
-public: anybody within the entire project can use it.(array, math, console)

-protected

-internal: somewhere in between public and private

-private

Creating a Class from Which Objects Can Be Instantiated (cont'd.)



```
class Employee
{
   // Instance variables and methods go here
}
```

Figure 9-1 Employee class shell

```
public class Employee
{
      // Instance variables and methods go here
}
```

- Better to use a modifier as default is `internal`

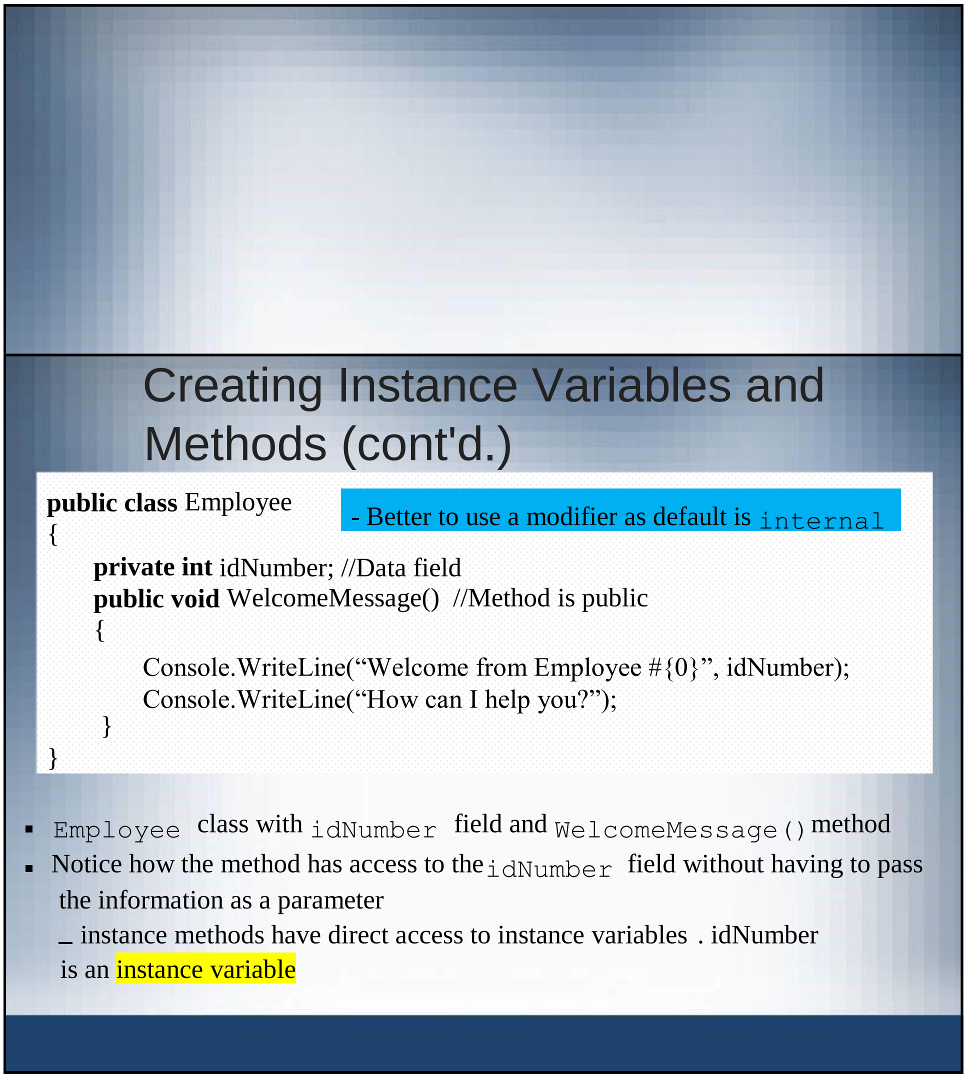
Classes are just definitions until objects are created from them.
Every program has to have a class that contains Main Method. We can have more user defined classes

Creating Instance Variables and Methods

- When creating a class, define both its fields and its methods
- Field access modifiers
  - new, **public**, protected, internal, **private**, static, readonly, and volatile
- Most class fields are nonstatic and private

Creating Instance Variables and Methods (cont'd.)

- Using private fields within classes is an example of **information hiding**
- Most class methods are public
  - – Provides the highest level of security. When a class is private, it protects our data as a private field object can be accessed by providing a message required to access the private field object.
- private data / public method arrangement
  - – Allows you to control outside access to your data
- Like using a gas gauge to "see" the level of the gas
  - – The private data fields are manipulated by welldefined (and programmer-defined) interfaces provided by the public methods



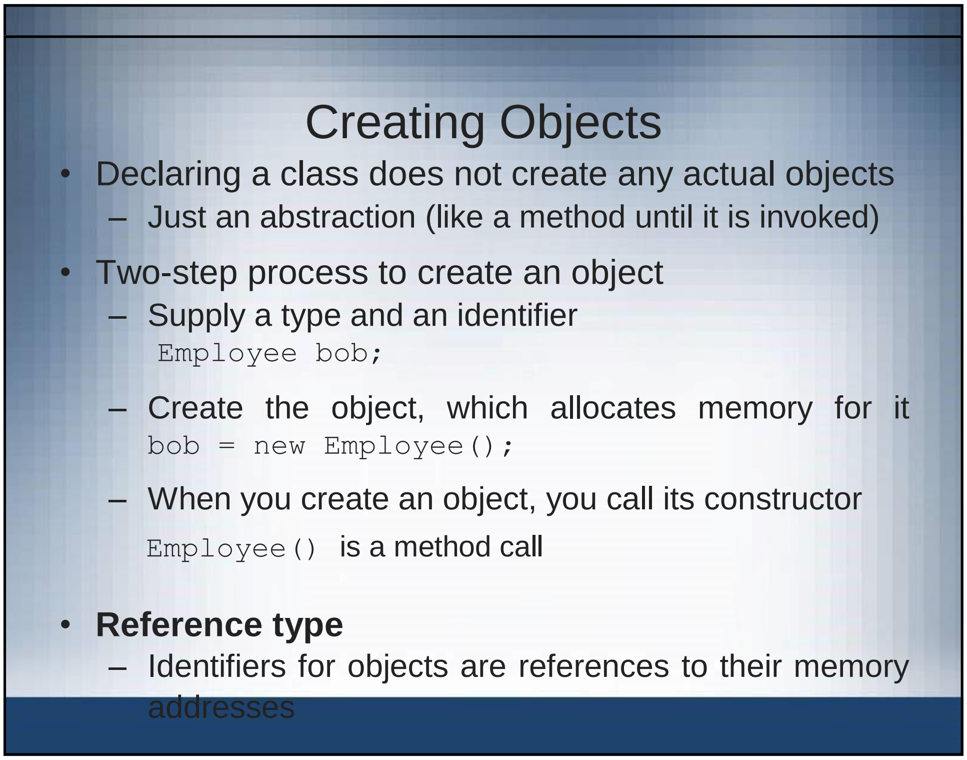
Creating Instance Variables and Methods (cont'd.)

- Better to use a modifier as default is `internal`

```
public class Employee
{
      private int idNumber; //Data field
      public void WelcomeMessage()  //Method is public
      {
            Console.WriteLine("Welcome from Employee #{0}", idNumber);
            Console.WriteLine("How can I help you?");
       }
}
```

- `Employee` class with `idNumber` field and `WelcomeMessage()` method
- Notice how the method has access to the `idNumber` field without having to pass the information as a parameter
  - instance methods have direct access to instance variables . idNumber is an instance variable
- Also notice that **static** is NOT part of the method header – instance method

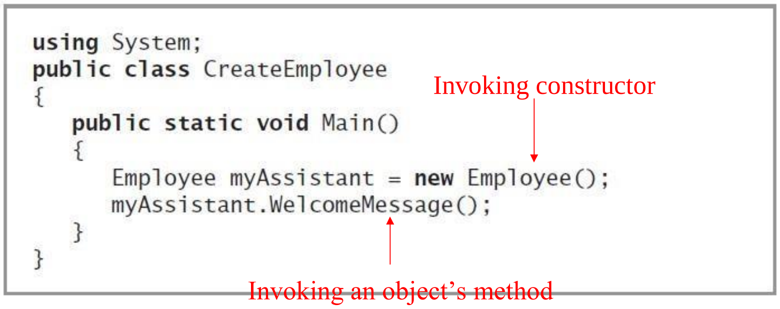
This class is just a definition as we are never using It because there Is no main method(like making a method but never calling it.



Creating Objects

- Declaring a class does not create any actual objects
  - Just an abstraction (like a method until it is invoked)
- Two-step process to create an object
  - Supply a type and an identifier
    `Employee bob;`
  - Create the object, which allocates memory for it
    `bob = new Employee();`
  - When you create an object, you call its constructor
    `Employee()` is a method call
- **Reference type**
  - Identifiers for objects are references to their memory addresses

Object.Method();

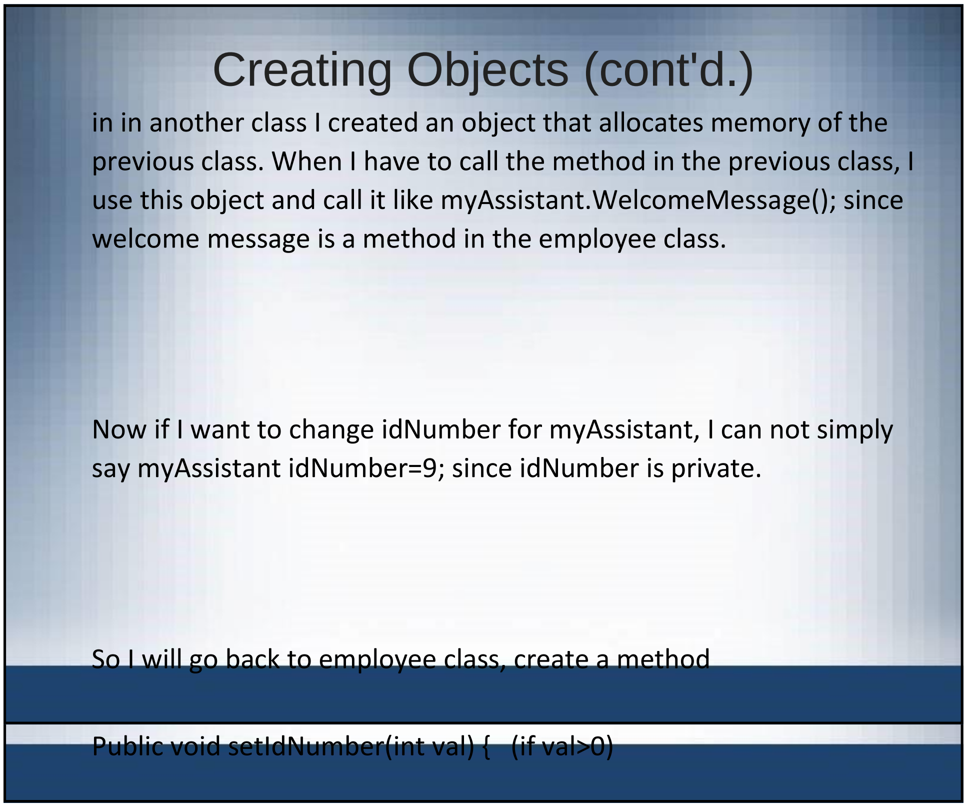


```
using System;
public class CreateEmployee
{
   public static void Main()
   {
      Employee myAssistant = new Employee();
      myAssistant.WelcomeMessage();
   }
}
```

Invoking constructor

Invoking an object's method

Figure 9-4 The `CreateEmployee` program



Creating Objects (cont'd.)

in in another class I created an object that allocates memory of the previous class. When I have to call the method in the previous class, I use this object and call it like myAssistant.WelcomeMessage(); since welcome message is a method in the employee class.

Now if I want to change idNumber for myAssistant, I can not simply say myAssistant idNumber=9; since idNumber is private.

So I will go back to employee class, create a method

Public void setIdNumber(int val) {   (if val>0)

idNumber =val;   else   idNumber=0; }

in Employee class and call it as myAssistant.setIdNumber(9); Now the idNumber will be changed for myAssistant.

We didn’t use static as we are trying to access it through an object from a non static field.

Also, we are able to use idNumber in all the methods without any need to pass it because it was defined in the beginning of the class and is applicable for the entire class.

Passing Objects to Methods

- You can pass objects to methods
  - Just as you can simple data types
  - Any object passed is always passed as reference.

aWorker is the argument which should match the formal parameter Employee in the DisplayMessage()

```
using  System;
publicclass  CreateEmployee
{                         publicstaticvoid Main()
   {
      Employee aWorker = new Employee();
      bWorker = new Employee();
      DisplayMessage("First", aWorker) ; //””for string
      DisplayMessage("Second", bWorker);
   }
   public static void  DisplayMessage(string mess, Employee emp)
   {
      Console.WriteLine("\n{0} employee's message", mess);
      emp.WelcomeMessage();
   }
}
```

```
First employee's message
Welcome from Employee #0

Second employee's message
me from Employee #0 Employee
u?
```

So if we change emp.setidNumber(100) in DisplayMessage then the idNumber will be changed for the worker everywhere

Creating Properties

- **Property**
  - A member of a class that provides access to a field of a class (very helpful for `private` fields)
  - They behave like a variable. There are no (); like while calling a Method. In method we can put a code to validate, however nothing such can be done in for a variable. Method gives ability to add code to protect on from what's going on.
  - Eg:- Array.Length, length Is a property.
  - Defines how fields will be set and retrieved
- Properties have **accessors**

–`set` **accessors** for setting an object's fields

- setters (also called mutators) they change the value

–`get` **accessors** for retrieving the stored values

- getters (or just accessors) they return the value

- **Read-only property**

– Has only a `get` accessor

--Notice we use a capital letter for property (IdNumber) and idNumber is from smaller case letter which is an instance variable.

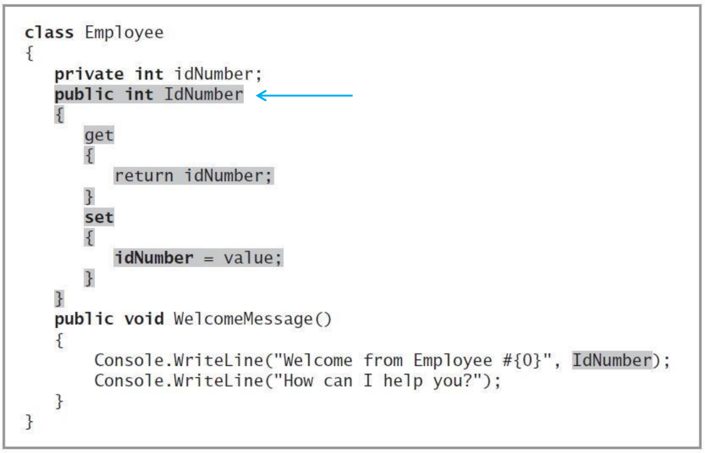
--Use capital letter for class and property.

12

Creating Properties (cont'd.)

public

Notice nomenclature



```
class Employee
{
    private int idNumber;
    public int IdNumber
    {
        get
        {
            return idNumber;
        }
        set
        {
            idNumber = value;
        }
    }
    public void WelcomeMessage()
    {
        Console.WriteLine("Welcome from Employee #{0}", IdNumber);
        Console.WriteLine("How can I help you?");
    }
}
```

Figure 9-8 Employee class with defined property

hand side of the assignment statement.

--

is a property and NOT a method.

ue will take whateve

od BUT there is no ()

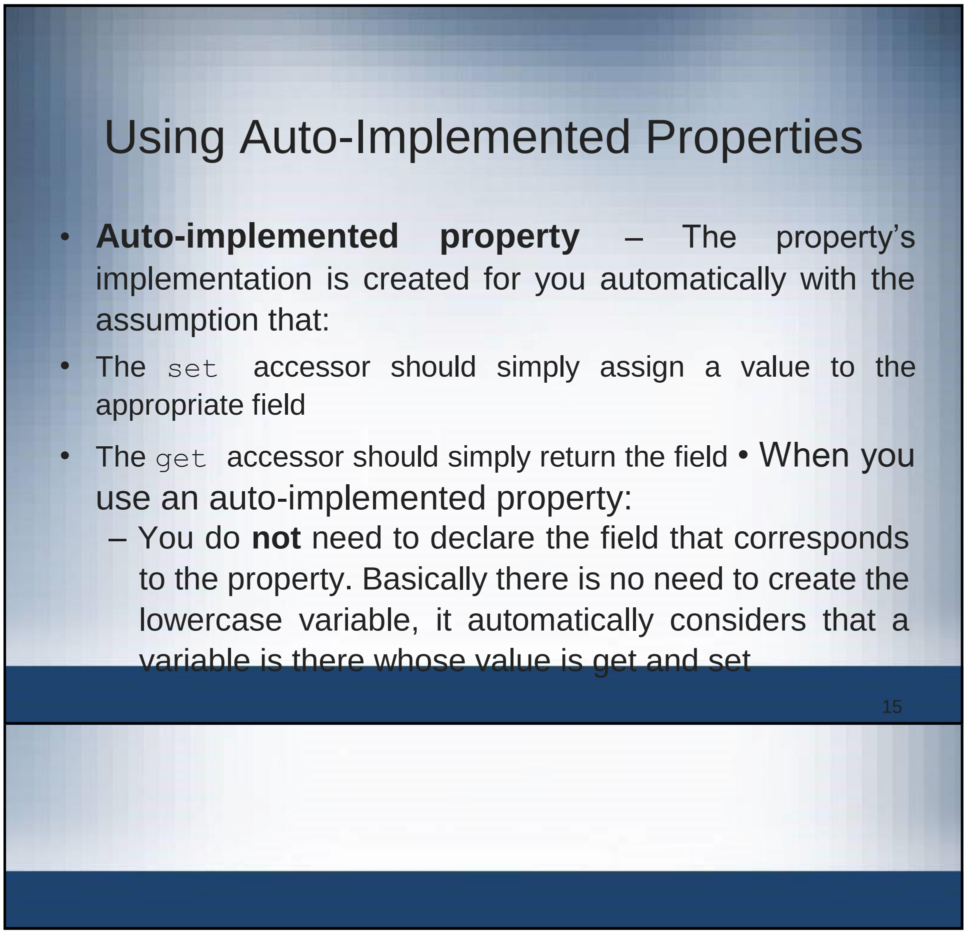
it

Creating Properties (cont'd.)

```
using  System;
publicclass  UseEmployeeProperties
{                                    publicstaticvoid Main()
   {
      Employee myEmployee = new Employee();
      myEmployee.IdNumber = 9;
      Console.WriteLine("ID number is {0}", myEmployee.IdNumber);
      myEmployee.WelcomeMessage();
   }
}
```

```
ID number is 9
How can I help you?
```

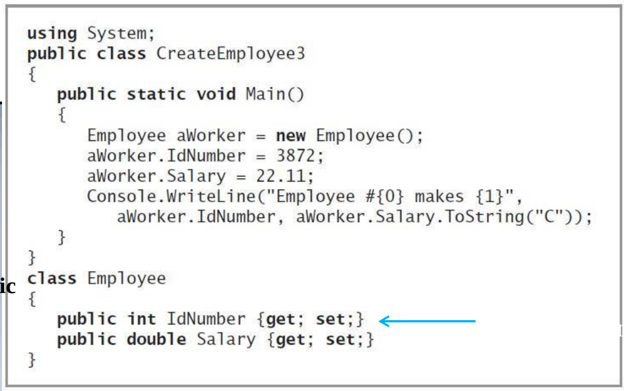
- Notice how the property is used by an object
  myEmployee.idNumber = 9; // this would result in an error
- **Implicit parameter:** one that is undeclared and that gets its value automatically
  –`value` becomes 9 in this case i.e. 9 is the implicit parameter



Using Auto-Implemented Properties

- **Auto-implemented property** – The property's implementation is created for you automatically with the assumption that:
- The `set` accessor should simply assign a value to the appropriate field
- The `get` accessor should simply return the field • When you use an auto-implemented property:
  - – You do **not** need to declare the field that corresponds to the property. Basically there is no need to create the lowercase variable, it automatically considers that a variable is there whose value is get and set

Using Auto-Implemented Properties (cont'd.)



```
using System;
public class CreateEmployee3
{
   public static void Main()
   {
      Employee aWorker = new Employee();
      aWorker.IdNumber = 3872;
      aWorker.Salary = 22.11;
      Console.WriteLine("Employee #{0} makes {1}",
         aWorker.IdNumber, aWorker.Salary.ToString("C"));
   }
}
class Employee
{
   public int IdNumber {get; set;}
   public double Salary {get; set;}
}
```

public

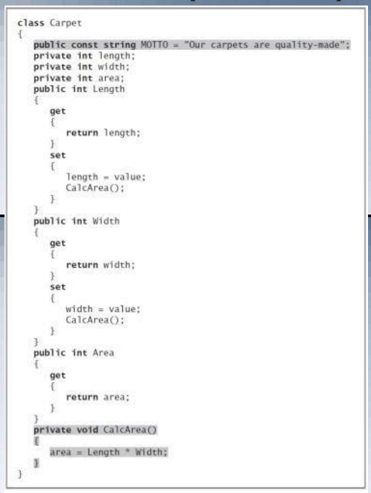
ivate fields

Figure 9-11 An Employee class with no declared fields and auto-implemented properties, and a program that uses them

More About public and private Access Modifiers

- Occasionally you need to create public fields or private methods
  - You can create a public data field when you want all objects of a class to be able to access it
- A named constant within a class is always static without having to declare it so
  - Belongs to the entire class, not to any particular instance

More About `public` and `private` Access Modifiers (cont'd.)



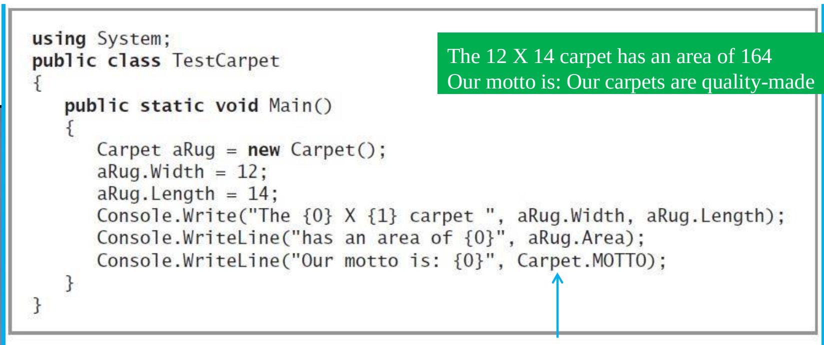
```
class Carpet
{
   public const string MOTTO = "Our carpets are quality-made";
   private int length;
   private int width;
   private int area;
   public int Length
   {
      get
      {
         return length;
      }
      set
      {
         length = value;
         CalcArea();
      }
   }
   public int Width
   {
      get
      {
         return width;
      }
      set
      {
         width = value;
         CalcArea();
      }
   }
   public int Area
   {
      get
      {
         return area;
      }
   }
   private void CalcArea()
   {
      area = Length * Width;
   }
}
```

public Figure 9-13 The Carpet class

All the parameters are in the same class and so can see each other, that means, they have the access to each other easily. These variables are called GLOBAL VARIABLE.

Something static is not shared in a dynamic class. Static variable doesn't have a copy. There is only one of it. There can be 100 objects and all can see it.

More About `public` and `private` Access Modifiers (cont'd.)



```
using System;
public class TestCarpet
{
   public static void Main()
   {
      Carpet aRug = new Carpet();
      aRug.Width = 12;
      aRug.Length = 14;
      Console.Write("The {0} X {1} carpet ", aRug.Width, aRug.Length);
      Console.WriteLine("has an area of {0}", aRug.Area);
      Console.WriteLine("Our motto is: {0}", Carpet.MOTTO);
   }
}
```

```
The 12 X 14 carpet has an area of 164
Our motto is: Our carpets are quality-made
```

Figure 9-14 The TestCarpet class

Notice how the constant MOTTO is accessed

Understanding the `this` Reference

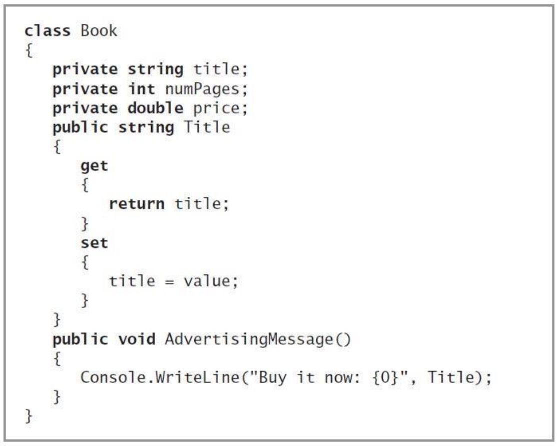
- You might eventually create thousands of objects from a class
  - Each object does not need to store its own copy of each property and method
- **`this` reference**
  - Implicitly passed reference
- When you call a method, you automatically pass the `this` reference to the method
  - Tells the method which instance of the class to use

Understanding the `this` Reference

Understanding the `this` Reference

(cont'd.)

public

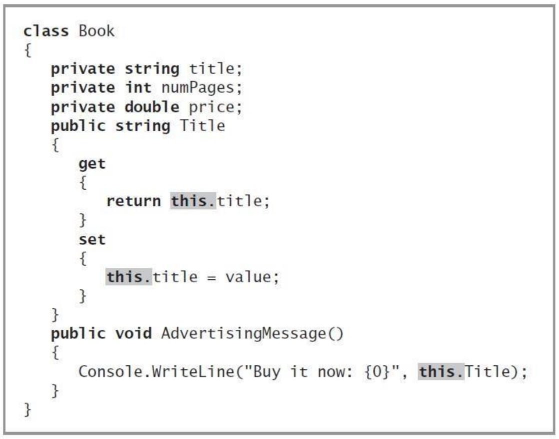


```
class Book
{
   private string title;
   private int numPages;
   private double price;
   public string Title
   {
      get
      {
         return title;
      }
      set
      {
         title = value;
      }
   }
   public void AdvertisingMessage()
   {
      Console.WriteLine("Buy it now: {0}", Title);
   }
}
```

Figure 9-16 Partially developed Book class

Without `this` reference

21



```
class Book
{
   private string title;
   private int numPages;
   private double price;
   public string Title
   {
      get
      {
         return this.title;
      }
      set
      {
         this.title = value;
      }
   }
   public void AdvertisingMessage()
   {
      Console.WriteLine("Buy it now: {0}", this.Title);
   }
}
```

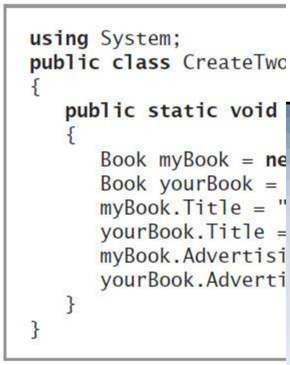
Figure 9-17 Book class with methods explicitly using this references

(cont'd.)

public

With this reference

Understanding the this Reference (cont'd.)



```
using System;
public class CreateTwc
{
   public static void
   {
      Book myBook = ne
      Book yourBook =
      myBook.Title = "
      yourBook.Title =
      myBook.Advertisi
      yourBook.Adverti
   }
}
```

Figure 9-18 Program that de

Buy it now: Silas Marner

Buy it now: The Time Traveler's Wife

23

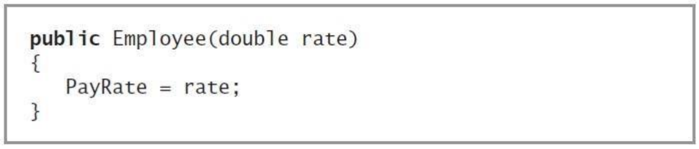
Understanding Constructors

- **Constructor**
  - Method that instantiates an object
    - **Default constructor**
      - Automatically supplied constructor without parameters
      - The <u>only </u>time that C# provides a default constructor is when there are no programmer-defined constructors
    - **Default value of the object**
      - The value of an object initialized with a default constructor

- Numeric fields are set to 0
- Character fields are set to ‘\0’
- Boolean fields are set to false
- References (strings and objects) are set to `null`

Passing Parameters to Constructors

- You can create a constructor that receives arguments



```
public Employee(double rate)
{
   PayRate = rate;
}
```

Figure 9-22 Employee constructor with parameter

- Using the constructor

```
Employee partTimeWorker = new Employee(12.50);
```

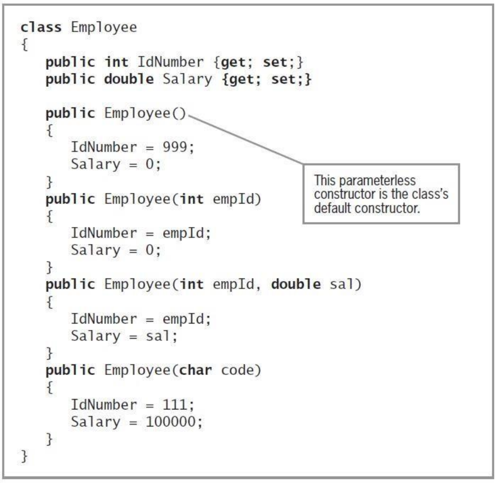
Overloading Constructors

- C# automatically provides a default constructor
  – Until you provide your own constructor

- Constructors can be overloaded
  – You can write as many constructors as you want

- As long as their argument lists do not cause ambiguity• Chooses constructor based on the signature

Overloading Constructors (cont'd.)

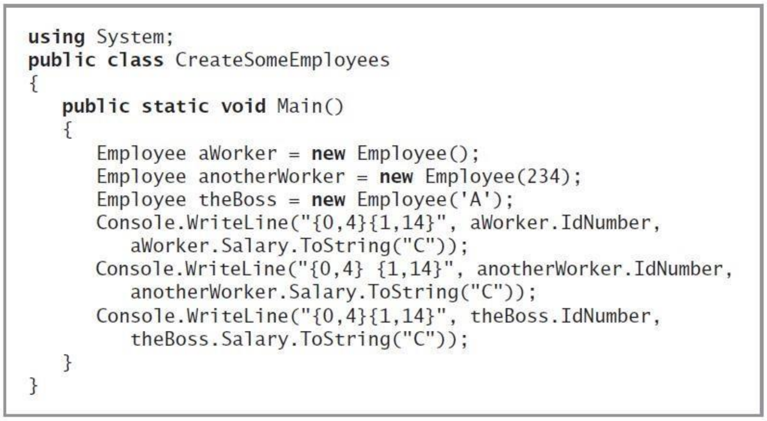


```
class Employee
{
   public int IdNumber {get; set;}
   public double Salary {get; set;}

   public Employee()
   {
      IdNumber = 999;
      Salary = 0;
   }
   public Employee(int empId)
   {
      IdNumber = empId;
      Salary = 0;
   }
   public Employee(int empId, double sal)
   {
      IdNumber = empId;
      Salary = sal;
   }
   public Employee(char code)
   {
      IdNumber = 111;
      Salary = 100000;
   }
}
```

This parameterless constructor is the class's default constructor.

public Figure 9-23 Employee class with four constructors

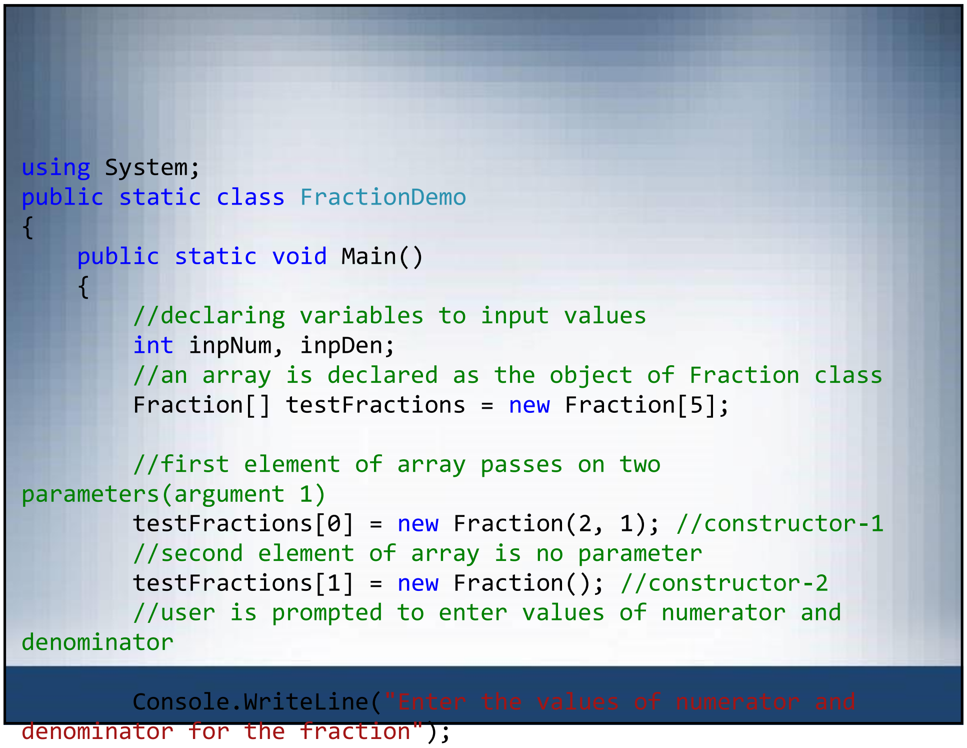
Overloading Constructors (cont'd.)



```
using System;
public class CreateSomeEmployees
{
   public static void Main()
   {
      Employee aWorker = new Employee();
      Employee anotherWorker = new Employee(234);
      Employee theBoss = new Employee('A');
      Console.WriteLine("{0,4}{1,14}", aWorker.IdNumber,
         aWorker.Salary.ToString("C"));
      Console.WriteLine("{0,4} {1,14}", anotherWorker.IdNumber,
         anotherWorker.Salary.ToString("C"));
      Console.WriteLine("{0,4}{1,14}", theBoss.IdNumber,
         theBoss.Salary.ToString("C"));
   }
}
```

Figure 9-24 CreateSomeEmployees program

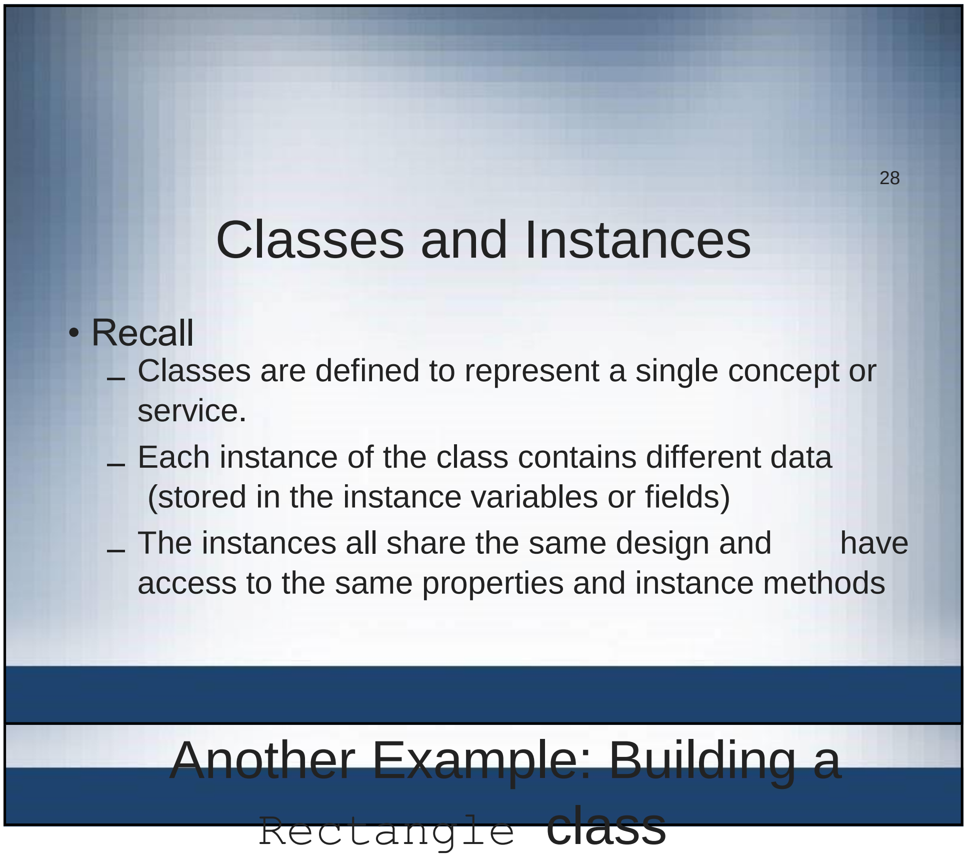
```
999                                                        $0.00
234                                                        $0.00
                                             111     $100,000.00
```



```
using System;
public static class FractionDemo
{
    public static void Main()
    {
        //declaring variables to input values
        int inpNum, inpDen;
        //an array is declared as the object of Fraction class
        Fraction[] testFractions = new Fraction[5];

        //first element of array passes on two 
parameters(argument 1)
        testFractions[0] = new Fraction(2, 1); //constructor-1
        //second element of array is no parameter
        testFractions[1] = new Fraction(); //constructor-2
        //user is prompted to enter values of numerator and 
denominator

        Console.WriteLine("Enter the values of numerator and 
denominator for the fraction");
        //do-while loop to validate value of numerator is 
positive
        do
        {
            Console.WriteLine("numerator : ");
            //value entered is stored in int variable NUMERATOR
            inpNum = Convert.ToInt32(Console.ReadLine());
            testFractions[1].Numerator = inpNum;//the public 
property Numerator of Fraction class is given the value entered 
by the user
        } while (inpNum < 0); //value shouldn't be negative

        //do-while loop to validate the value of denominator is 
positive and not equal to zero
        do
        {
            Console.WriteLine("denominator : ");
            //value is stored in int variable denominator
            inpDen = Convert.ToInt32(Console.ReadLine());
            testFractions[1].Denominator = inpDen; //public 
property Denominator of Fraction class is given the value 
entered
        } while (inpDen <= 0); //value shouldn't be negative

        //printing out the Fractions
        Console.WriteLine("The two Fractions are:");
        Console.WriteLine("Fraction 1 : {0} ", 
testFractions[0]);
```

```
        Console.WriteLine("Fraction 2 : {0} ",
testFractions[1]);

        //Overload addition operator
        testFractions[2] = testFractions[0] + testFractions[1];
        //Overload multiplication operator
        testFractions[3] = testFractions[0] * testFractions[1];

        //printing out addition and multiplication of the two
fractions.
        Console.WriteLine("Addition of the two fractions : {0}
", testFractions[2]);
        Console.WriteLine("Multiplication of the two fractions :
{0} ", testFractions[3]);

        //<= and >= overload operators and printing the result
        if (testFractions[0] <= testFractions[1])
            Console.WriteLine("{0} is less than or equal to
{1}",testFractions[0],testFractions[1]);
        else
            Console.WriteLine("{0} is less than or equal to
{1}", testFractions[1], testFractions[0]);

        if (testFractions[0] >= testFractions[1])
            Console.WriteLine("{0} is greater than or equal to
{1}", testFractions[0], testFractions[1]);
        else
            Console.WriteLine("{0} is greater than or equal to
{1}", testFractions[1], testFractions[0]);

        Console.ReadLine();
    }
}
```

```
using System;
public class Fraction //Fraction class
{
    //private fields in Fraction class
    private int numerator;
    private int denominator;

    //no argument constructor, constructor-1
    public Fraction()
    {
        numerator = 0;
        denominator = 1;
    }

    //constructor-2 taking in two arguments
    public Fraction(int num, int den)
    {
        Reduce(ref num, ref den); //calling Reduce Method and 
passing values by reference
        numerator = num;
        denominator = den;
    }

    public int Numerator //property Numerator
    {
        //setting the value of the numerator entered by the user
        set
        {
            //validating, the value entered by user must be 
positive
            if (value >= 0)
                numerator = value;
            else
                numerator = 1;

        }

    }

    public int Denominator //property denominator
    {
        //setting the value of the denominator entered by the 
user
        set
        {
            //validating, the value entered by user must be 
positive
            if (value > 0)
                denominator = value;
            else
                denominator = 1;
            Reduce(ref numerator, ref denominator); //calling 
Reduce() Method once both Numerator and Denominator values have 
been entered by the user
        }

    }

    //Method : Reduce Method takes two arguments by reference
    //returns : void
    //num, den : int reference variables storing the values of 
numerator and denominator
    private void Reduce(ref int num, ref int den)
    {
        //finding the greatest common divisor if numerator is 
larger
        if (num > den)
        {
            int k = num;
            int m = den;

            while(m>0)
            {
                int remainder = k % m;
                k = m;
                m = remainder;
            }

            num = num / k;
            den = den / k;

        }
        else
        //finding the greatest common divisor if denominator is 
larger
        {
            int k = den;
            int m = num;

            while (m > 0)
            {
                int remainder = k % m;
                k = m;
                m = remainder;
            }

            num = num / k;
            den = den / k;
        }


    }

    // ToString() Method to print out the fractions
    public override string ToString()
    {
        return numerator + "/" + denominator;
    }

    //Overloading Multiplication Operator
    public static Fraction operator *(Fraction f1, Fraction f2)
    {
        //takes the two Fraction objects and used to find value 
after their multiplication
        int num;
        int den;

        num = f1.numerator * f2.numerator;
        den = f1.denominator * f2.denominator;

        //a new object made and given values after 
multiplication
        Fraction f3 = new Fraction(num, den);
        return f3; //value returned

    }

    //Overloading Addition Operator
    public static Fraction operator +(Fraction f1, Fraction f2)
    {
        //takes the two Fraction objects and used to find value
after their addition and the reutning the value
        return new Fraction(f1.numerator * f2.denominator +
f2.numerator * f1.denominator, f1.denominator * f2.denominator);
    }

    //Overloading less than or equal to Operator using boolean
    public static bool operator <=(Fraction f1, Fraction f2)
    {
        //finding double values of fractions for easy
comparision
        double value1 = f1.numerator / (double)f1.denominator;
        double value2 = f2.numerator / (double)f2.denominator;


        if (value1 <= value2)
        {

            return true;
        }

        else
            return false;
    }

    //Overloading greater than or equal to Operator using
boolean
    public static bool operator >=(Fraction f1, Fraction f2)
    {
        //finding double values of fractions for easy
comparision
        double value1 = f1.numerator / (double)f1.denominator;
        double value2 = f2.numerator / (double)f2.denominator;

        if (value1 >= value2)
            return true;
        else
            return false;

    }


}
```



Classes and Instances

- Recall
  - Classes are defined to represent a single concept or service.
  - Each instance of the class contains different data (stored in the instance variables or fields)
  - The instances all share the same design and have access to the same properties and instance methods

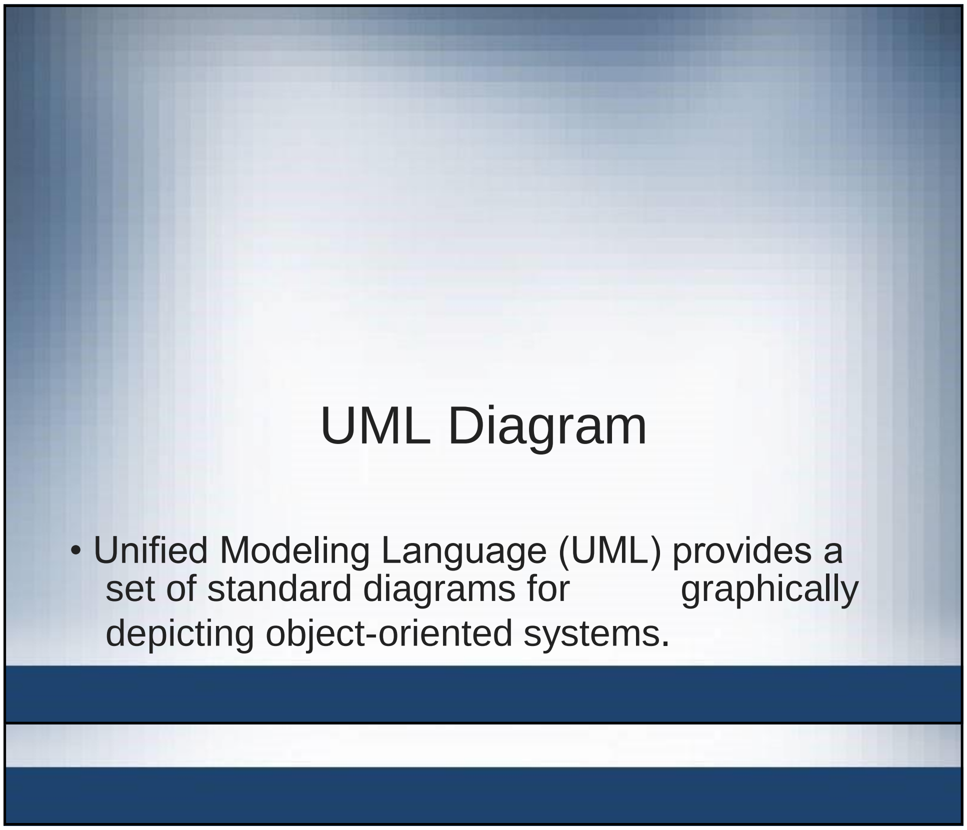
Another Example: Building a `Rectangle` class

- A `Rectangle` object will have the following fields:
  - `length` - holds the rectangle's length.
  - `width` - holds the rectangle's width.

Building a `Rectangle` class

- The `Rectangle` class will also have the following methods (no Properties) yet:

–`SetLength` - sets a value in an object's length field. –
`SetWidth` - sets a value in an object's width field. –
`GetLength` - returns the value in an object's length field.
–`GetWidth` - returns the value in an object's width field.
–`GetArea` - returns the area of the rectangle, which is the result of the object's length multiplied by its width.



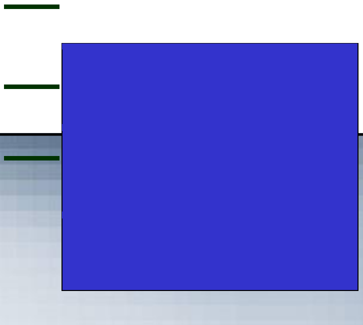
UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.

Class name goes here

Fields are listed here

Methods are listed here



UML Diagram for `Rectangle` class

Rectangle

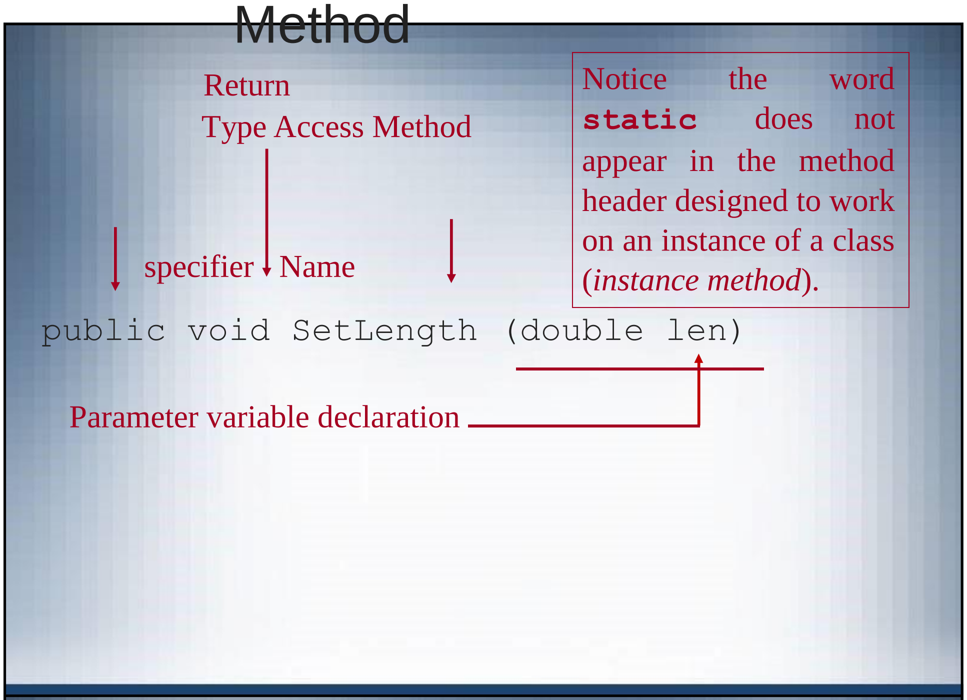
- length
- width

+ SetLength()
+ SetWidth()
+ GetLength()
+ GetWidth()
+ GetArea()

Writing the Code for the Class Fields

```
public class Rectangle
{ private double length;
     private double width;
}
```

Header for the `SetLength` Method



Return

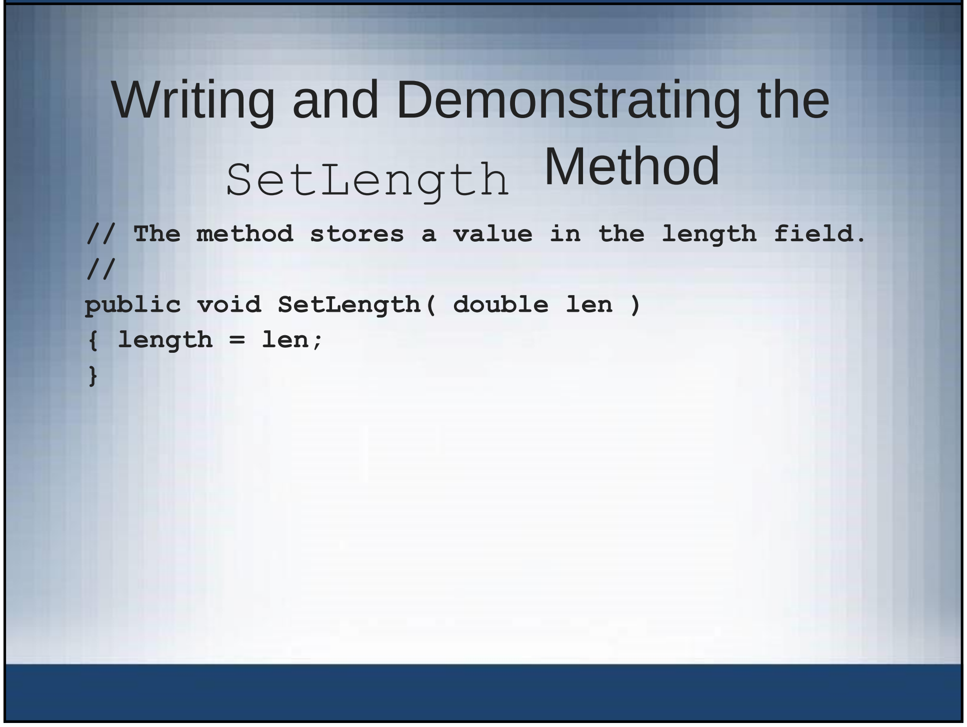
Type Access Method

specifier Name

```
public void SetLength (double len)
```

Parameter variable declaration

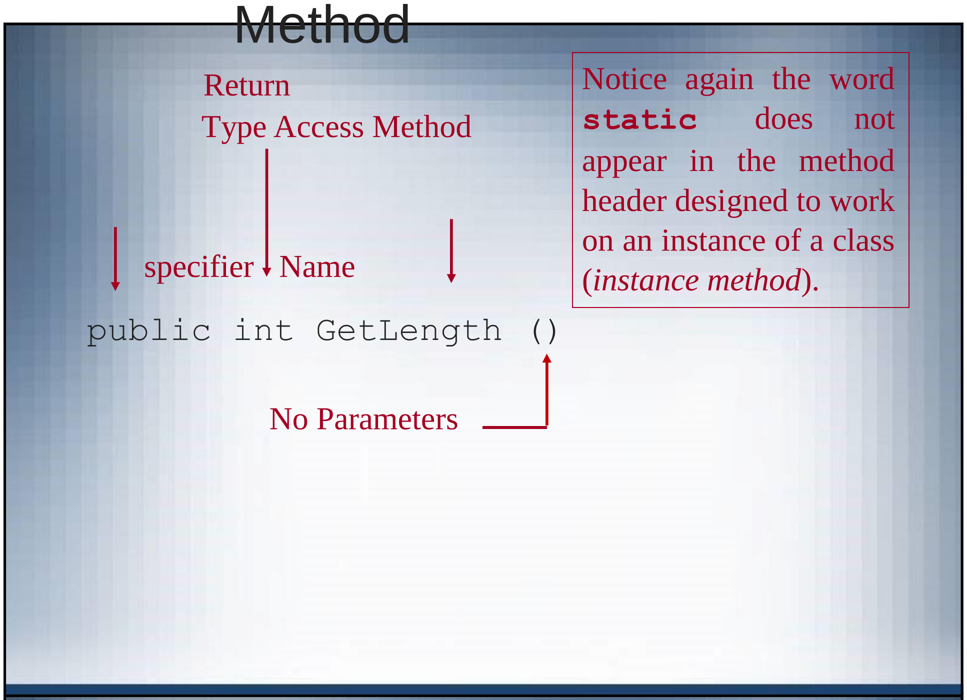
Notice the word **`static`** does not appear in the method header designed to work on an instance of a class (*instance method*).



Writing and Demonstrating the `SetLength` Method

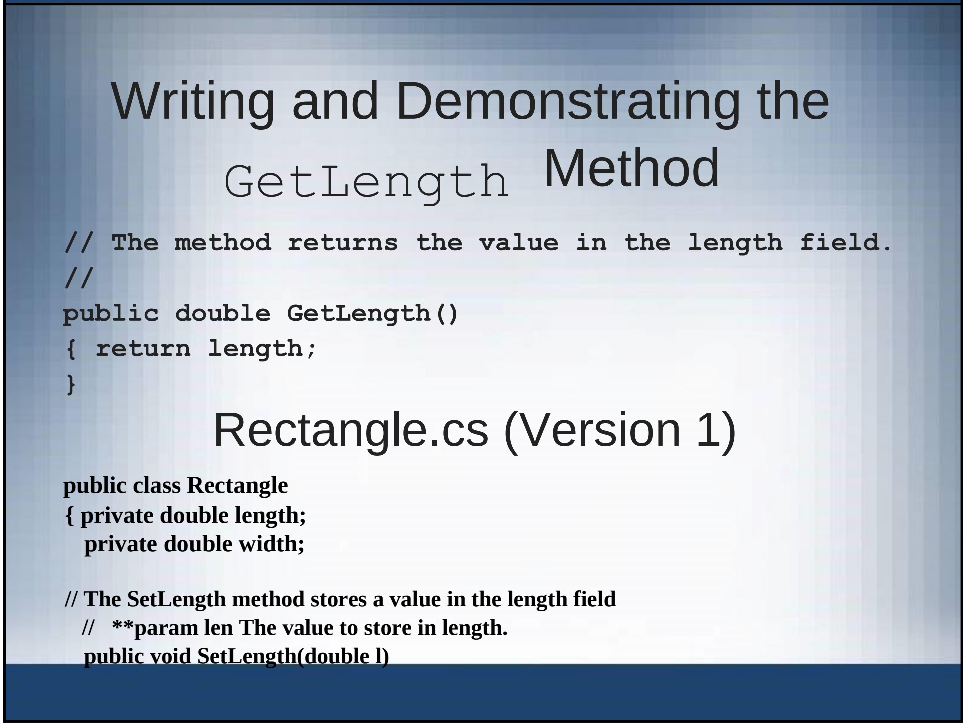
```
// The method stores a value in the length field.
//
public void SetLength( double len )
{ length = len;
}
```

Header for the GetLength Method



```
public int GetLength ()
```

Notice again the word **static** does not appear in the method header designed to work on an instance of a class (*instance method*).



Writing and Demonstrating the GetLength Method

```
// The method returns the value in the length field.
//
public double GetLength()
{ return length;
}
```

Rectangle.cs (Version 1)

```
public class Rectangle
{ private double length;
   private double width;

// The SetLength method stores a value in the length field
   //   **param len The value to store in length.
   public void SetLength(double l)
   { length = l;
   }

   // The SetWidth method stores a value in the width field
   //   **param w The value to store in width.
   public void SetWidth(double w)
   { width = w;
```

```
      }
```

Rectangle.cs (cont'd)

```
// The GetLength method returns a Rectangle object's length
  //   *return The value in the length field.
  public double GetLength()
  { return length;
  }

// The GetWidth method returns a Rectangle object's width //
**return The value in the width field.
  public double GetWidth()
  { return width;
  }

  // The GetArea method returns a Rectangle object's area.
  //   **return The product of length times width.
  public double GetArea()
  { return length * width;
  }
}
```

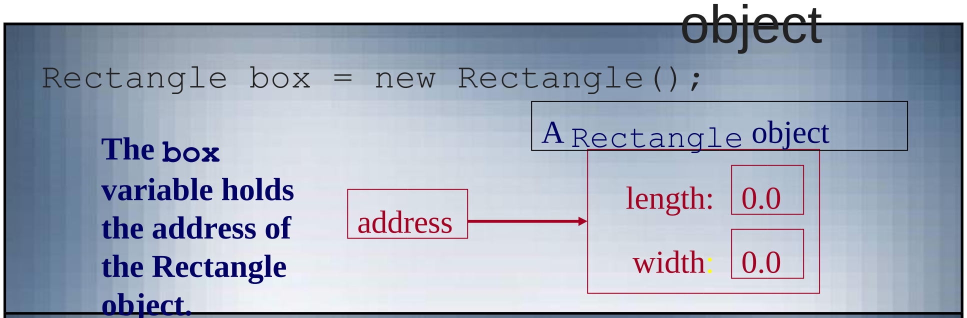
RectangleDemo.cs

```
    // This program demonstrates the Rectangle class'
    // SetLength, SetWidth, GgetLength, GetWidth, and getArea methods.

    using System; public class RectangleDemo
    { public static void Main()
       {
         Rectangle box = new Rectangle();

         box.SetLength(10.0);
         box.SetWidth(20.0);

         Console.WriteLine("The box's length is {0}", box.GetLength());
         Console.WriteLine("The box's width is {0}", box.GetWidth());
         Console.WriteLine("The box's area is {0}", box.GetArea());

         Console.ReadLine();
       }
    }
```

Output:

```
The box's length is 10
The box's width is 20
The box's area is 200
```

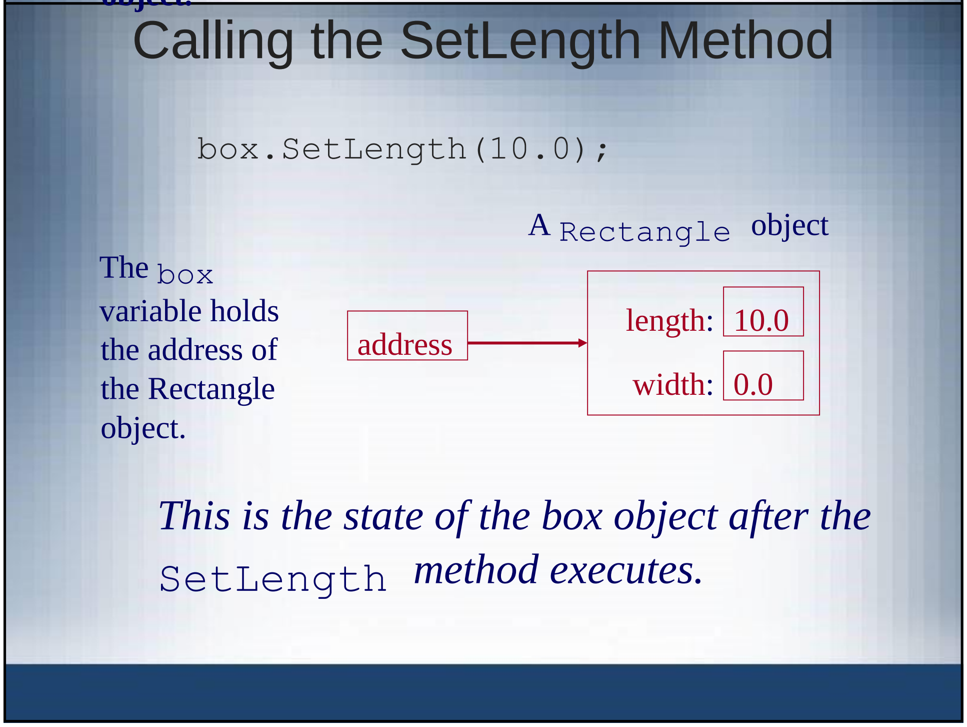
Create a `Rectangle` class object



`Rectangle box = new Rectangle();`

The `box` variable holds the address of the Rectangle object.

Calling the SetLength Method



`box.SetLength(10.0);`

The `box` variable holds the address of the Rectangle object.

This is the state of the box object after the `SetLength` *method executes.*

Accessor and Mutator Methods

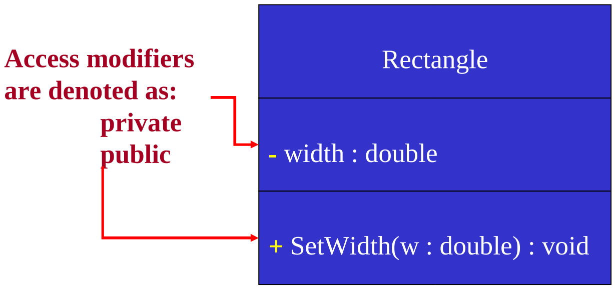
- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called *accessors*.
  - Each field that the programmer wishes to be viewed by other classes needs an accessor.
- The methods that modify the data of fields are called *mutators*.
  - Each field that the programmer wishes to be modified by other classes needs a mutator.

Accessors and Mutators

- For the rectangle example, the accessors and mutators are:
  - SetLength : Sets the value of the length field. public void SetLength(double len) …
  - SetWidth : Sets the value of the width field. public void SetWidth(double w) …
  - GetLength : Returns the value of the length field. public double GetLength() …
  - GetWidth : Returns the value of the width field. public double GetWidth() …
- Other names for these methods are *getters* and *setters*.

UML Data Type and Parameter Notation

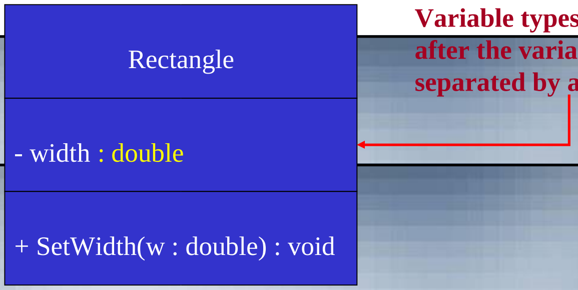
- UML diagrams are language independent.
- UML diagrams use an independent notation to show return types, access modifiers, etc.


Access modifiers are denoted as:
private
public
Rectangle
- width : double
+ SetWidth(w : double) : void

-
+

UML Data Type and

Parameter Notation



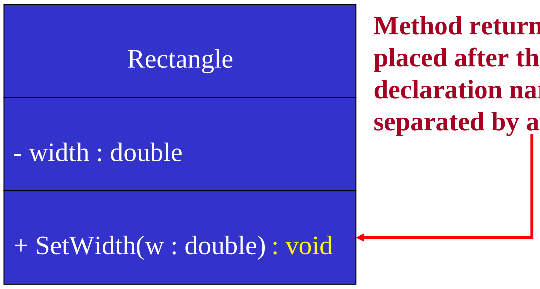
Rectangle

- width : double

+ SetWidth(w : double) : void

Variable types are placed after the variable name, separated by a colon.

UML Data Type and Parameter Notation



Rectangle

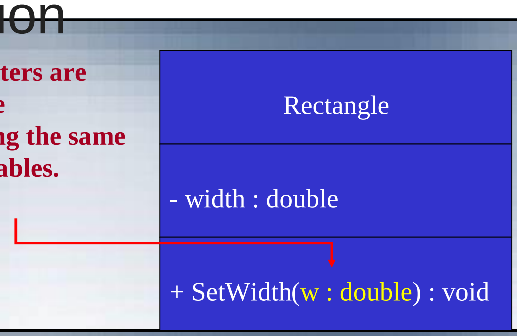
- width : double

+ SetWidth(w : double) : void

Method return types are placed after the method declaration name, separated by a colon.

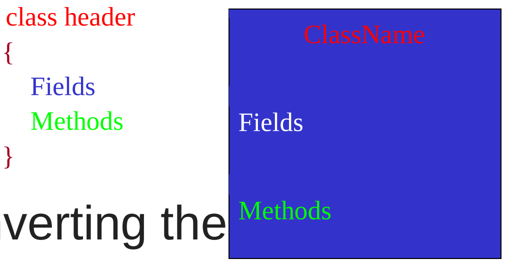
UML Data Type and Parameter Notation

Method parameters are shown inside the parentheses using the same notation as variables.



Converting the UML Diagram to Code

- Putting all of this information together, a C# class file can be built easily using the UML diagram.
- The UML diagram parts match the C# class file structure.

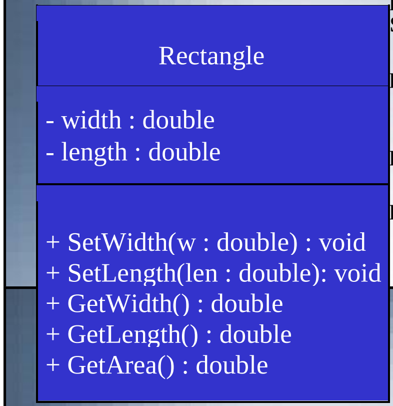


Converting the UML

Diagram to Code

The structure of the class can be public class Rectangle compiled and
tested without having { private double width; bodies for the methods.
Just be sure to private double length;
put in dummy return values for methods
that have a return type other than void.

public void SetWidth(double w) { } public void
SetLength(double len) { }



Rectangle

- width : double
- length : double

+ SetWidth(w : double) : void
+ SetLength(len : double): void
+ GetWidth() : double
+ GetLength() : double
+ GetArea() : double

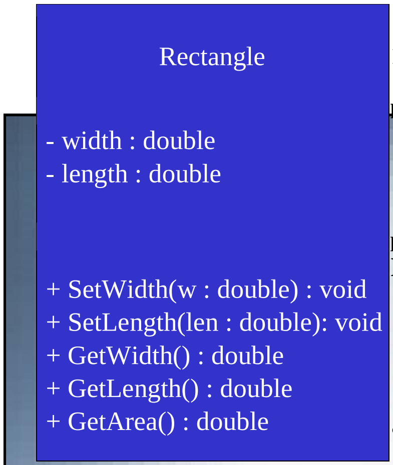
public double GetWidth() { return 0.0;
}

public double GetLength() { return 0.0;
}
public double GetArea() { return 0.0;
}
}

Converting the UML Diagram to Code

public class Rectangle
Once the class structure has been { tested, the method bodies can
be private double width; written and tested. private double length;

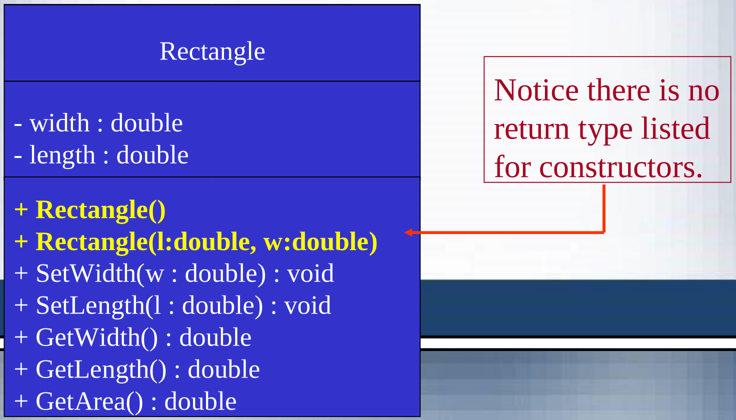
public void SetWidth(double w) { width = w;



```
}
public void SetLength(double len) { length = len;
}
public double GetWidth() { return width;
  } public double
  GetLength() { return
        length;
}
public double GetArea() { return length * width;
} }
```

Constructors in UML

• In UML, the constructors are defined as follows:



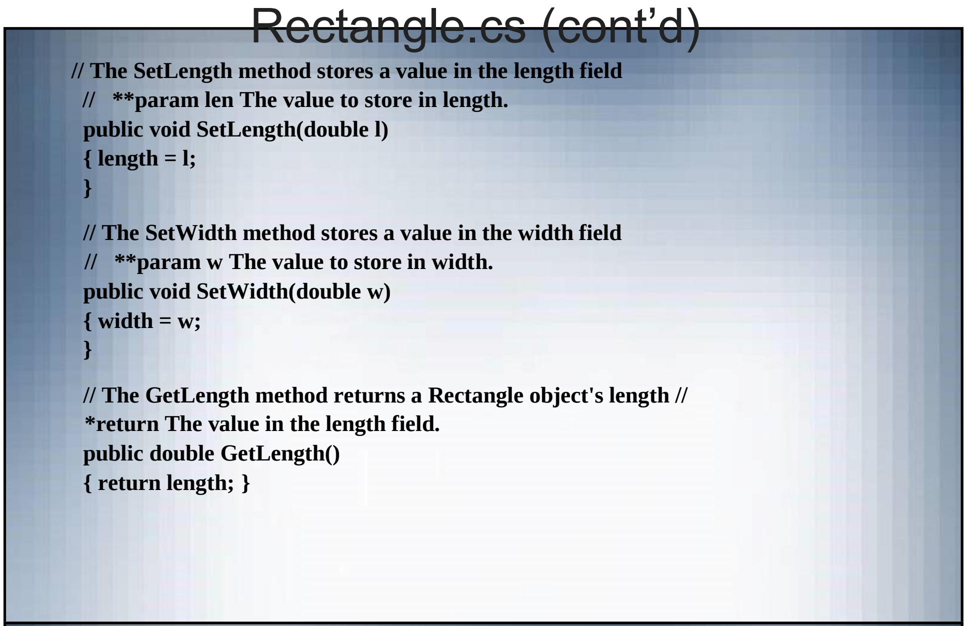
Notice there is no return type listed for constructors.

Rectangle.cs (Version 2)

```
public class Rectangle
{ private double length;
  private double width;

  // Parameterless constructor
  public Rectangle()
  { length = 0;
     width = 0;
  }
```

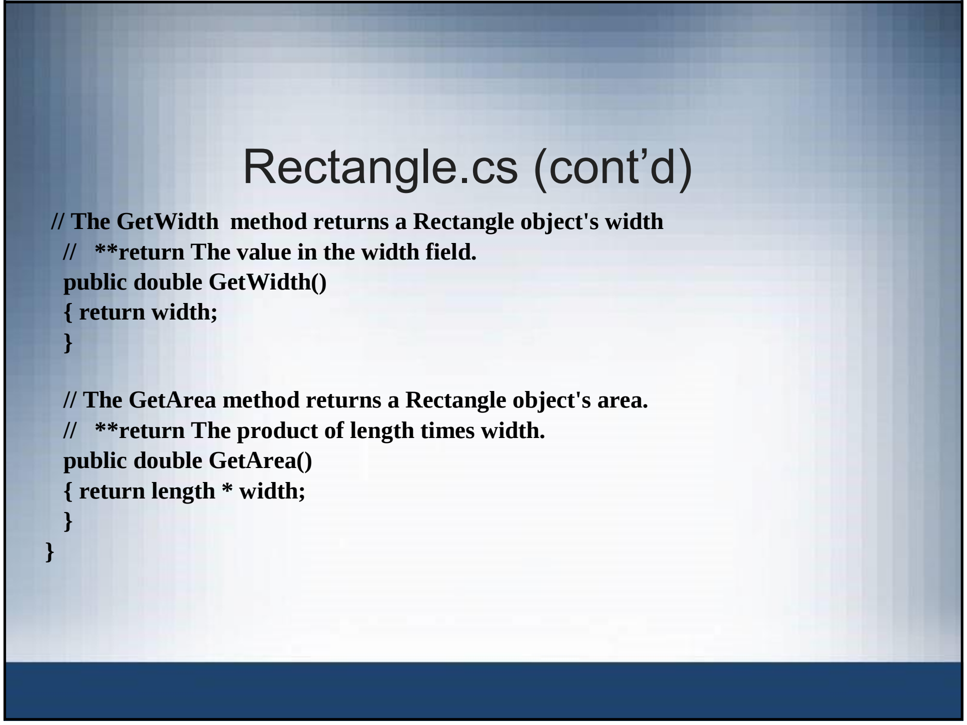
```
  // Parameter Constructor //   ** l
  The length of the rectangle. //   ** w
  The width of the rectangle. public
  Rectangle(double l, double w)
```

```
{ length = l; width
= w; }
```



Rectangle.cs (cont'd)

```
// The SetLength method stores a value in the length field
  //   **param len The value to store in length.
  public void SetLength(double l)
  { length = l;
  }

  // The SetWidth method stores a value in the width field
  //   **param w The value to store in width.
  public void SetWidth(double w)
  { width = w;
  }

  // The GetLength method returns a Rectangle object's length //
  *return The value in the length field.
  public double GetLength()
  { return length; }
```



Rectangle.cs (cont'd)

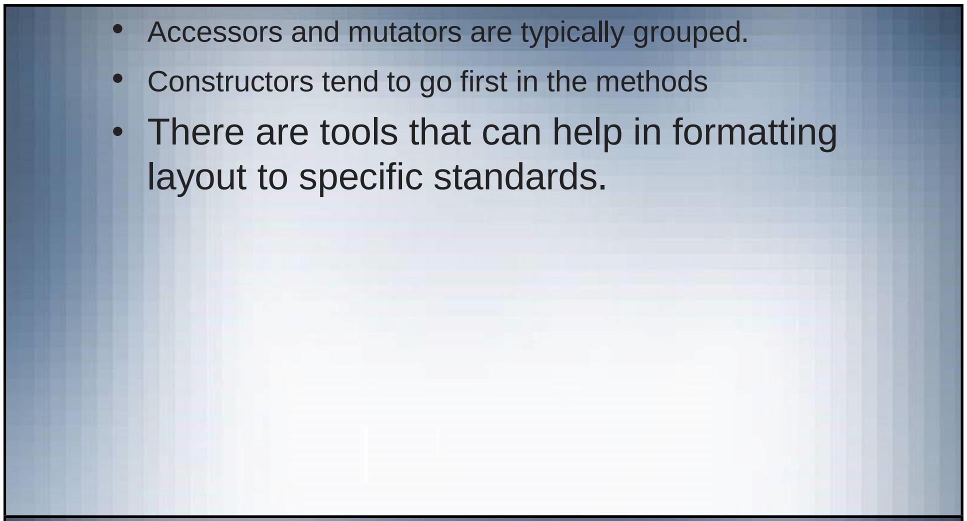
```
 // The GetWidth  method returns a Rectangle object's width
   //   **return The value in the width field.
   public double GetWidth()
   { return width;
   }

   // The GetArea method returns a Rectangle object's area.
   //   **return The product of length times width.
   public double GetArea()
   { return length * width;
   }
}
```

Class Layout Conventions

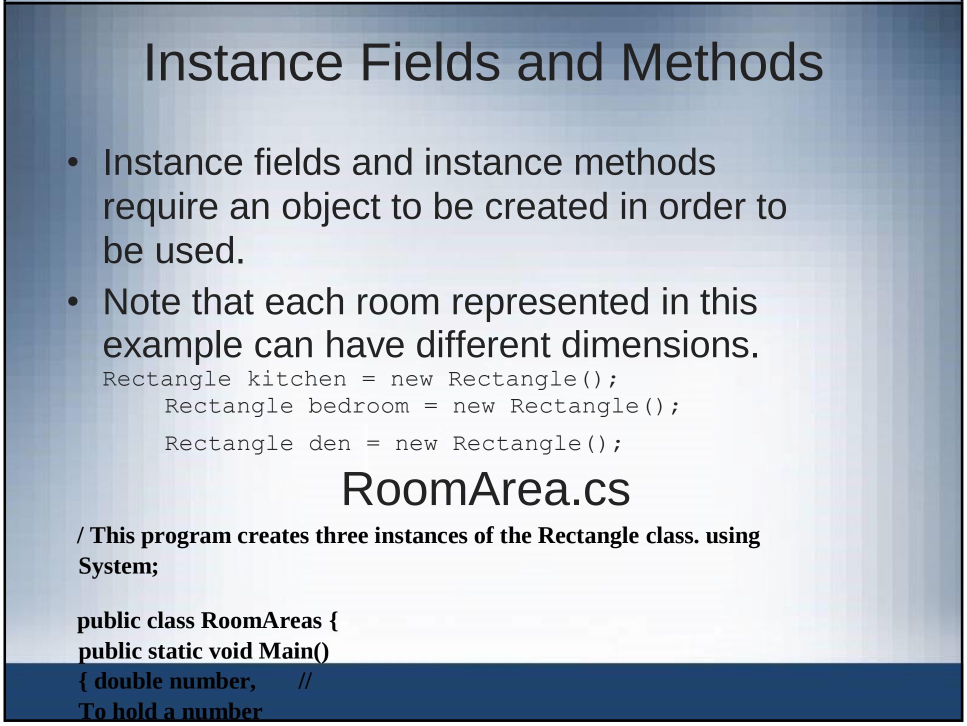
- The layout of a source code file can vary by employer or instructor.
- Typically the layout is as follows:
  - Fields are listed first.

- Methods are listed second.

- The main method is sometimes first, sometimes last.



- Accessors and mutators are typically grouped.
- Constructors tend to go first in the methods
- There are tools that can help in formatting layout to specific standards.



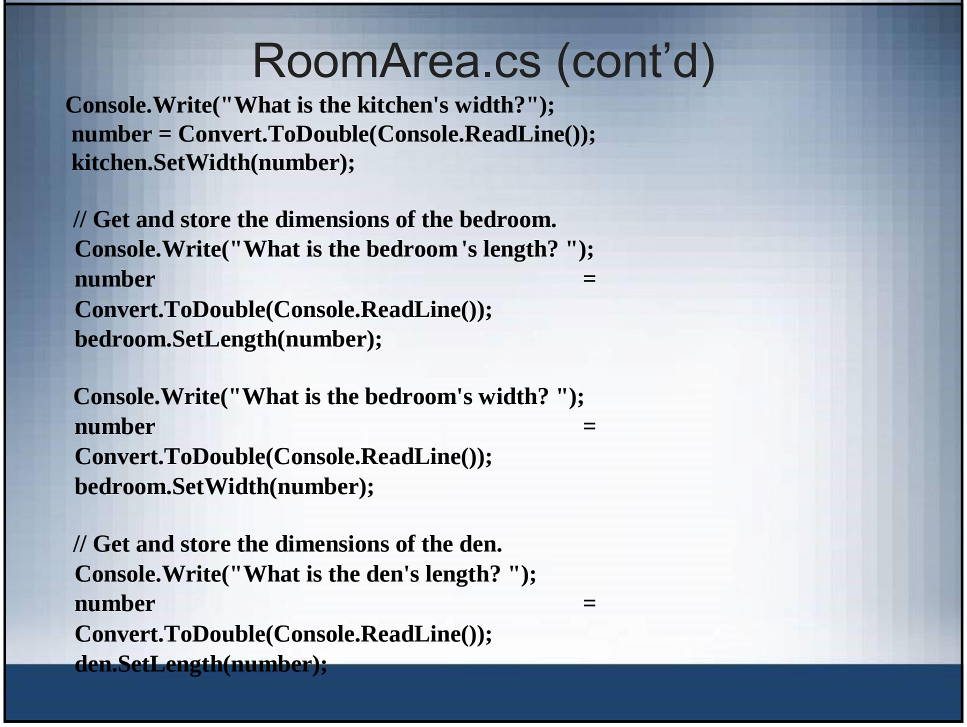
Instance Fields and Methods

- Instance fields and instance methods require an object to be created in order to be used.
- Note that each room represented in this example can have different dimensions.

```
Rectangle kitchen = new Rectangle();
    Rectangle bedroom = new Rectangle();
    Rectangle den = new Rectangle();
```

RoomArea.cs

```
/ This program creates three instances of the Rectangle class. using
System;

public class RoomAreas {
public static void Main()
{ double number,      //
To hold a number
totalArea;   // The total
area

  // Create three Rectangle objects.
  Rectangle kitchen = new Rectangle();
  Rectangle   bedroom   =   new   Rectangle();
  Rectangle den = new Rectangle();

  // Get and store the dimensions of the kitchen.
  Console.Write("What is the kitchen's length?"); number
  =
  Convert.ToDouble(Console.ReadLine());
  kitchen.SetLength(number);
```



RoomArea.cs (cont’d)

```
Console.Write("What is the kitchen's width?");
 number = Convert.ToDouble(Console.ReadLine());
 kitchen.SetWidth(number);

 // Get and store the dimensions of the bedroom.
 Console.Write("What is the bedroom's length? ");
 number                                          =
 Convert.ToDouble(Console.ReadLine());
 bedroom.SetLength(number);

 Console.Write("What is the bedroom's width? ");
 number                                          =
 Convert.ToDouble(Console.ReadLine());
 bedroom.SetWidth(number);

 // Get and store the dimensions of the den.
 Console.Write("What is the den's length? ");
 number                                          =
 Convert.ToDouble(Console.ReadLine());
 den.SetLength(number);
```

RoomArea.cs (cont’d)

```
Console.Write("What is the den's width?"); number
=      Convert.ToDouble(Console.ReadLine());
den.SetWidth(number);

// Calculate the total area of the rooms.
totalArea = kitchen.GetArea() + bedroom.GetArea() + den.GetArea();

// Display the total area of the rooms.
Console.WriteLine("\nThe total area of the rooms is {0}", totalArea);
```

```
        Console.ReadLine();
    }
}
```

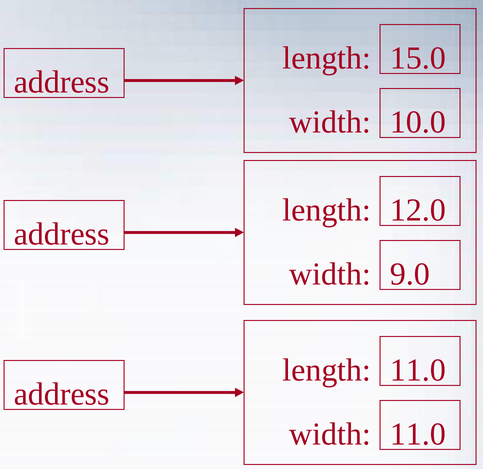
Output:

```
What is the kitchen's length?15
What is the kitchen's width?10
What is the bedroom's length? 12
What is the bedroom's width? 9
What is the den's length? 11
What is the den's width?11

The total area of the rooms is 379
```

States of Three Different Rectangle Objects

The `kitchen` variable holds the address of a Rectangle Object.

The `bedroom` variable holds the address of a Rectangle Object.



address

length: 15.0

width: 10.0

address

length: 12.0

width: 9.0

address

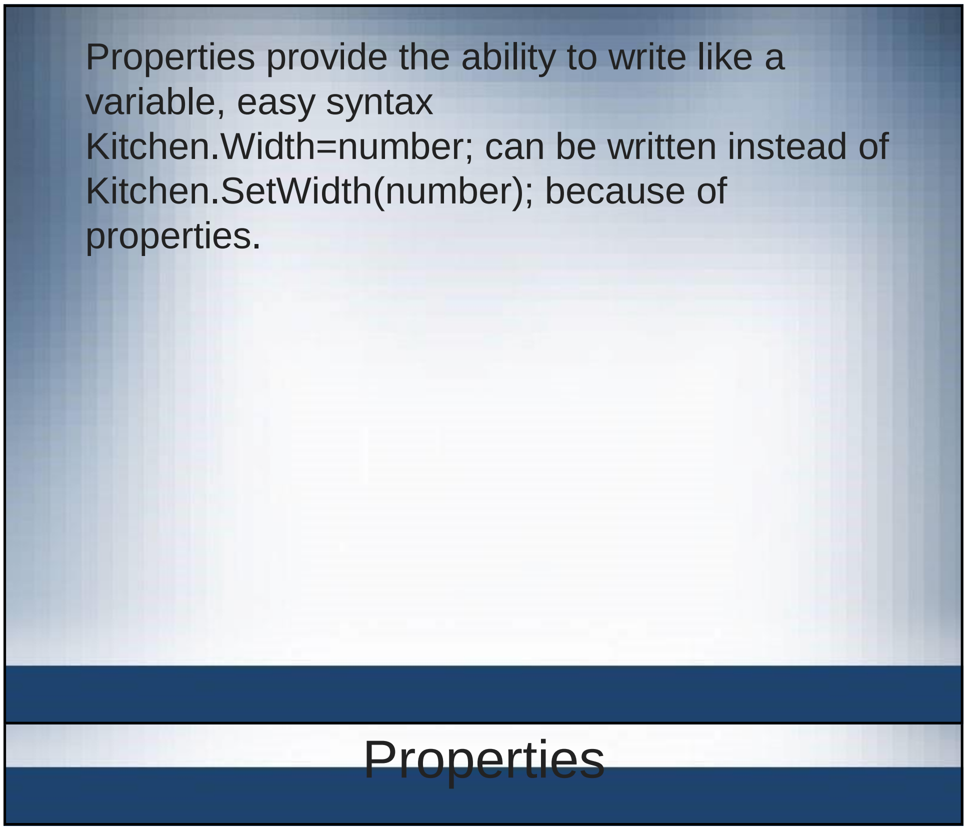
length: 11.0

width: 11.0

The `den` variable holds the address of a Rectangle Object.

Properties

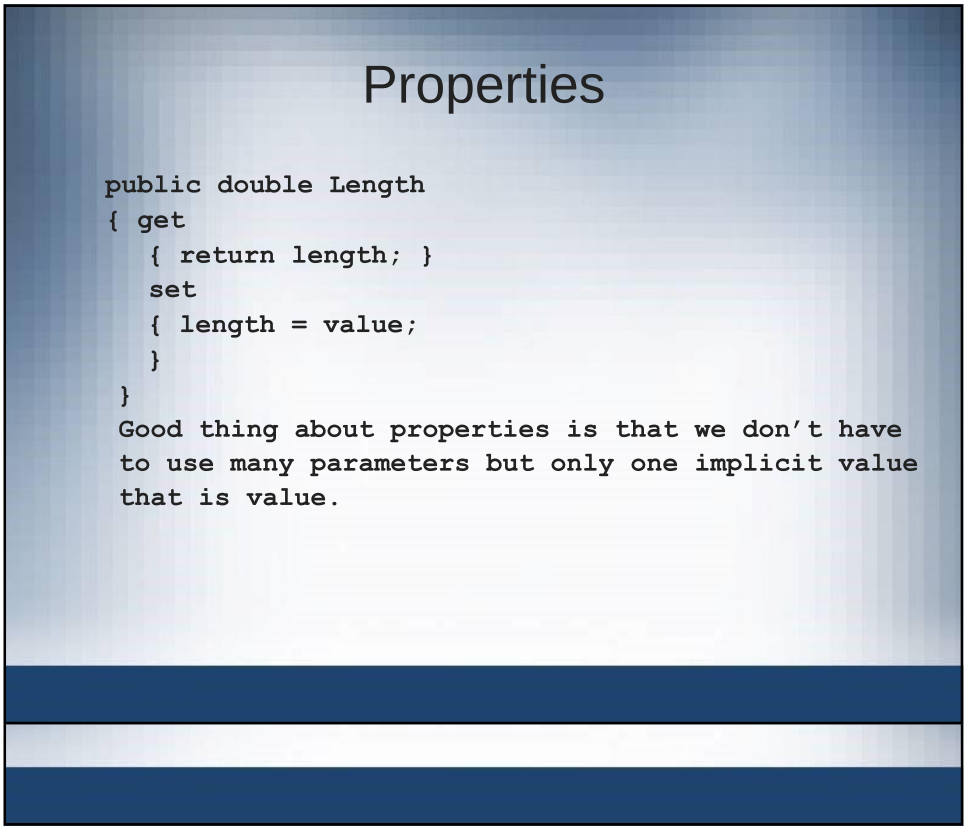
- C# has an alternative to using accessor and mutator methods, called *properties.*
- Properties allow us to modify and access private data members like accessor and mutator methods without using public methods.



Properties provide the ability to write like a variable, easy syntax Kitchen.Width=number; can be written instead of Kitchen.SetWidth(number); because of properties.

Properties

- We use the key words `get`, `set` and `value` when defining properties.
- The `get` and `set` blocks replace the function of the accessor and mutator methods.
- The `get` block uses a variable named `value` that returns the actual value of the instance field. (get is used when we want a readonly)
- The `set` block uses the `value` to set the value of the instance field.



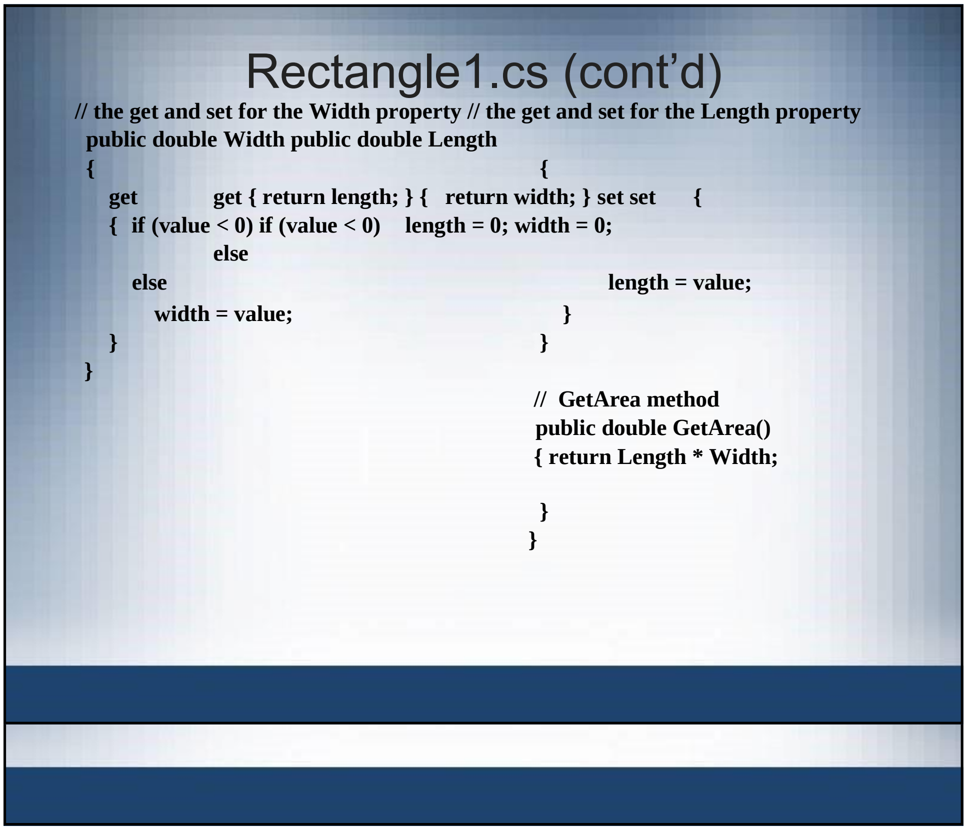
Properties

```
public double Length
{ get
   { return length; }
   set
   { length = value;
   }
 }
```

Good thing about properties is that we don’t have to use many parameters but only one implicit value that is value.

Rectangle.cs (Version 3)

```
public class Rectangle
{ //two data fields private
   double length;  private
   double width;

// No-Arg constructor public
   Rectangle()
  { length = 0; width
     = 0;
  }
  // Parameter Constructor public Rectangle(double
  l, double w)
  { length = l; width
  = w; }
```



Rectangle1.cs (cont’d)

```
// the get and set for the Width property // the get and set for the Length property
  public double Width public double Length
  {                                                   {
     get            get { return length; } {   return width; } set set      {
     {  if (value < 0) if (value < 0)    length = 0; width = 0;
                    else
        else                                                  length = value;
           width = value;                                 }
     }                                                  }
  }
                                                       //  GetArea method
                                                       public double GetArea()
                                                       { return Length * Width;

                                                        }
                                                      }
```

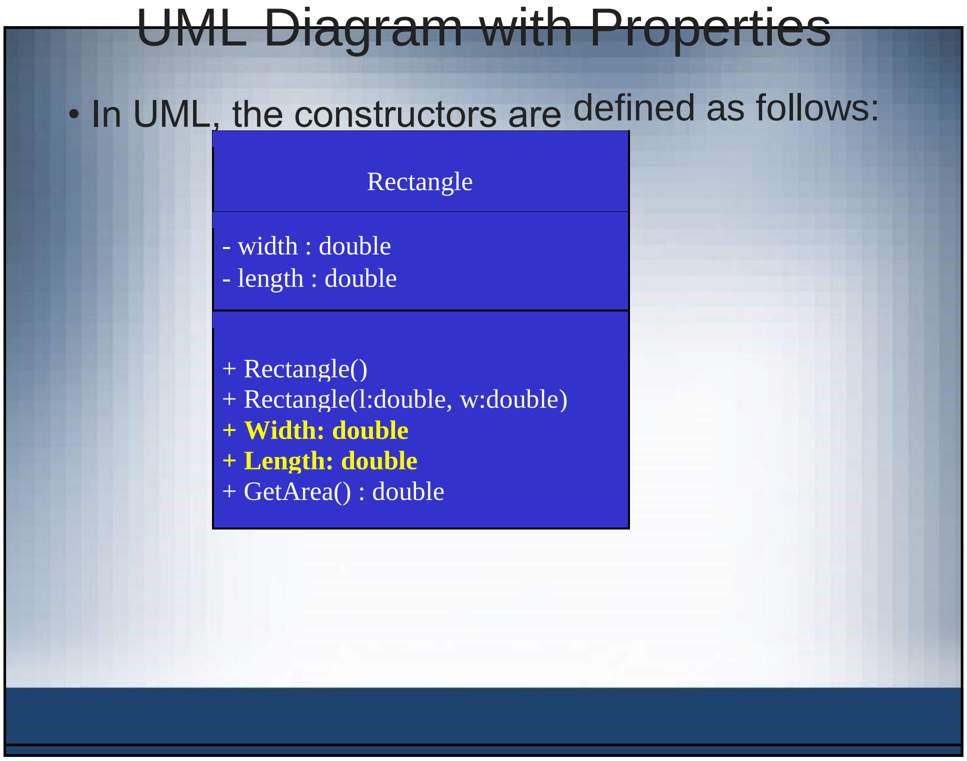
PropertyTest.cs

```
// This program demonstrates Rectangle1.cs.
// This tests C# properties. using
System;

public class PropertyDemo
{ public static void Main()
  {
    Rectangle box = new Rectangle(); box.Length
    = 5.0; box.Width = 15.0;

    Console.WriteLine("The box's length is {0}", box.Length);
    Console.WriteLine("The box's width is {0}", box.Width);
    Console.WriteLine("The box's area is {0}", box.GetArea());

    Console.ReadLine();
```

Output:

```
The box's length is 5
The box's width is 15
The box's area is 75
```

```
   }
}
```



UML Diagram with Properties

- In UML, the constructors are defined as follows:

Rectangle

- width : double
- length : double

+ Rectangle()
+ Rectangle(l:double, w:double)
+ Width: double
+ Length: double
+ GetArea() : double

Overloading Operators

- Overload operators
  – Enable you to use arithmetic symbols with your own objects

- Overloadable unary operators:

  `+ - ! ~ ++ -- true false`

- Overloadable binary operators:

  `+ - * / % & | ^ == != > < >= <=`

- Cannot overload the following operators:

  `= && || ?? ?: checked unchecked new typeof as is`

- Cannot overload an operator for a built-in data type

– Cannot change the meaning of `+` for `ints`

Overloading Operators (cont'd.)

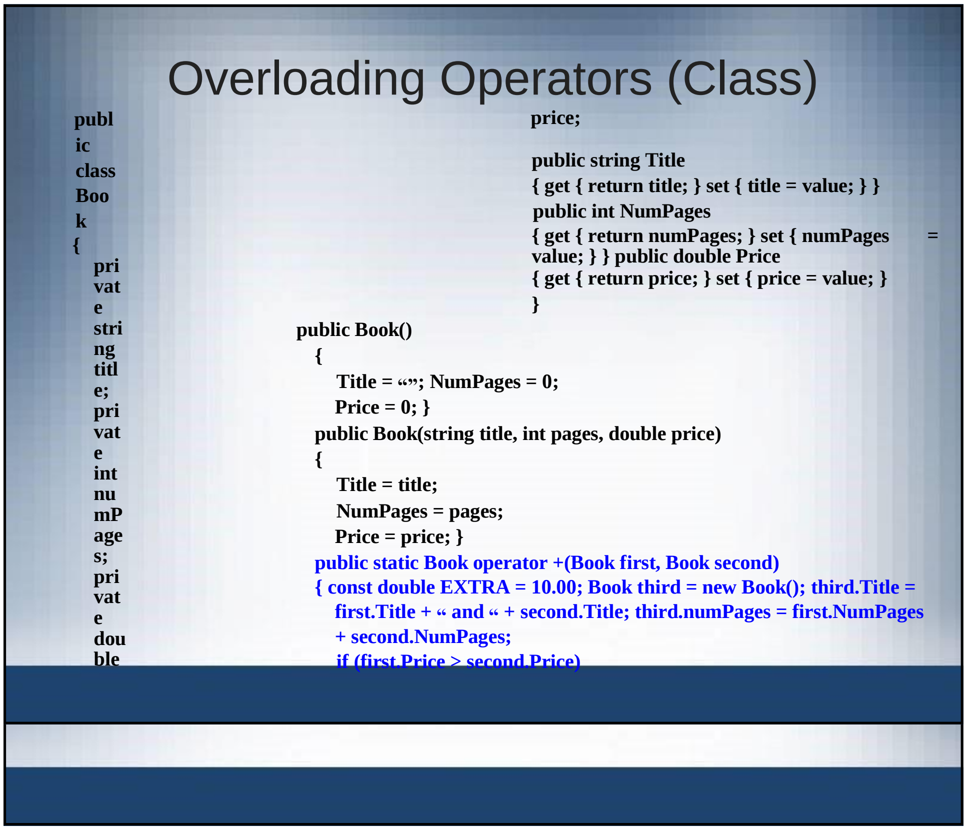
- When a binary operator is overloaded and has a corresponding assignment operator:
  - It is also overloaded

- Some operators must be overloaded in pairs:`==` with `!=`, and `<` with `>`

- Syntax to overload unary operators:*type* `operator` *overloadable-operator* (*type identifier*)

- Syntax to overload binary operators:

  type `operator` *overloadable-operator* (*type identifier, type operand*)



Overloading Operators (Class)

```
public class Book
{
    private string title; private int numPages; private double
    price;

    public string Title
    { get { return title; } set { title = value; } }
    public int NumPages
    { get { return numPages; } set { numPages = value; } } public double Price
    { get { return price; } set { price = value; } }
    }
public Book()
   {
      Title = “”; NumPages = 0;
      Price = 0; }
   public Book(string title, int pages, double price)
   {
      Title = title;
      NumPages = pages;
      Price = price; }
   public static Book operator +(Book first, Book second)
   { const double EXTRA = 10.00; Book third = new Book(); third.Title =
      first.Title + “ and “ + second.Title; third.numPages = first.NumPages
      + second.NumPages;
      if (first.Price > second.Price)
```

third.Price = first.Price + EXTRA; } else } third.Price = second.Price + EXTRA; return third;

72 Overloading Operators (Driver)

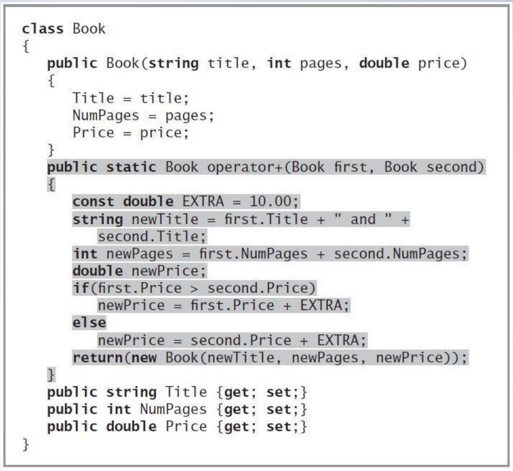
```
using System; public
class AddBooks
{ public static void Main()
   {
      Book book1 = new Book(“Visual C#”, 840, 75.00);
      Book book2 = new Book(“Moby Dick”, 250, 16.00);
      Book book3; book3 = book1 + book2;
      Console.WriteLine(“The new book is \”{0}\””, book3.Title);
      Console.WriteLine(“It has {0} pages and costs {1:C}”, book3.NumPages, book3.Price);
      Console.ReadLine();
   }
}
```

Output:

```
The new book is “Visual C# and Moby Dick”
It has 1090 pages and costs $85.00
```

73

Overloading Operators (Textbook version)

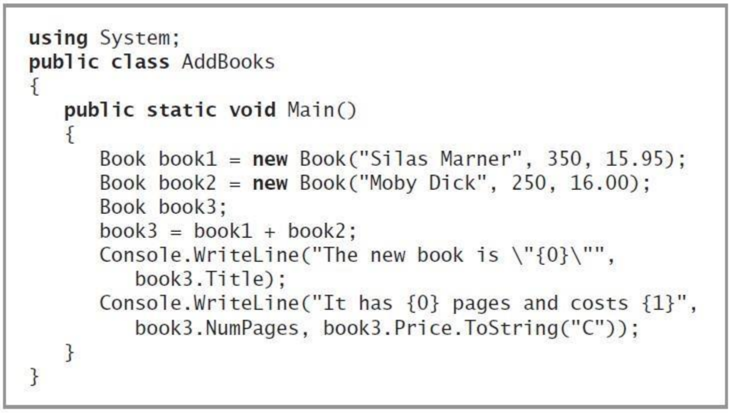


```
class Book
{
   public Book(string title, int pages, double price)
   {
      Title = title;
      NumPages = pages;
      Price = price;
   }
   public static Book operator+(Book first, Book second)
   {
      const double EXTRA = 10.00;
      string newTitle = first.Title + " and " +
         second.Title;
      int newPages = first.NumPages + second.NumPages;
      double newPrice;
      if(first.Price > second.Price)
         newPrice = first.Price + EXTRA;
      else
         newPrice = second.Price + EXTRA;
      return(new Book(newTitle, newPages, newPrice));
   }
   public string Title {get; set;}
   public int NumPages {get; set;}
   public double Price {get; set;}
}
```

Figure 9-32 Book class with overloaded + operator

74

Overloading Operators (Textbook version)



```
using System;
public class AddBooks
{
   public static void Main()
   {
      Book book1 = new Book("Silas Marner", 350, 15.95);
      Book book2 = new Book("Moby Dick", 250, 16.00);
      Book book3;
      book3 = book1 + book2;
      Console.WriteLine("The new book is \"{0}\"",
         book3.Title);
      Console.WriteLine("It has {0} pages and costs {1}",
         book3.NumPages, book3.Price.ToString("C"));
   }
}
```

Figure 9-33 The AddBooks program

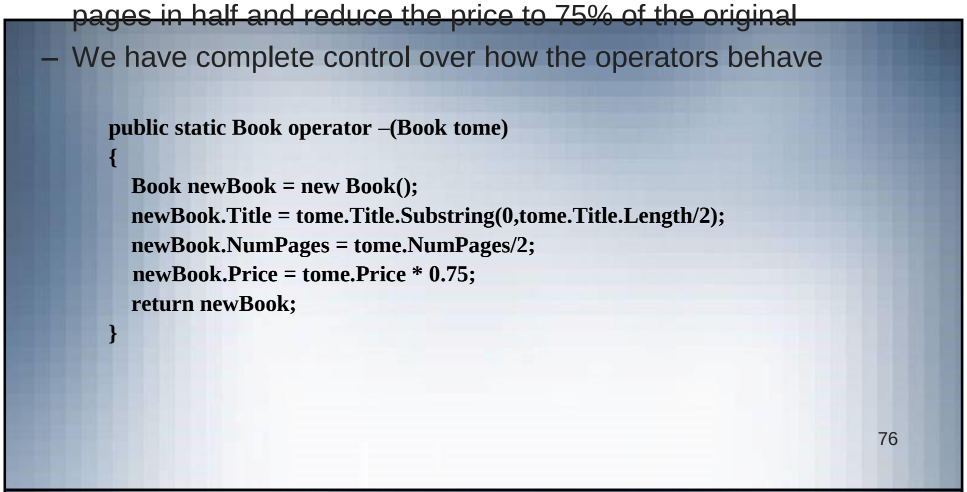
```
The new book is Silas Marner and Moby Dick It has 600 pages and costs $26.00
```

75

Overloading Operators (more)

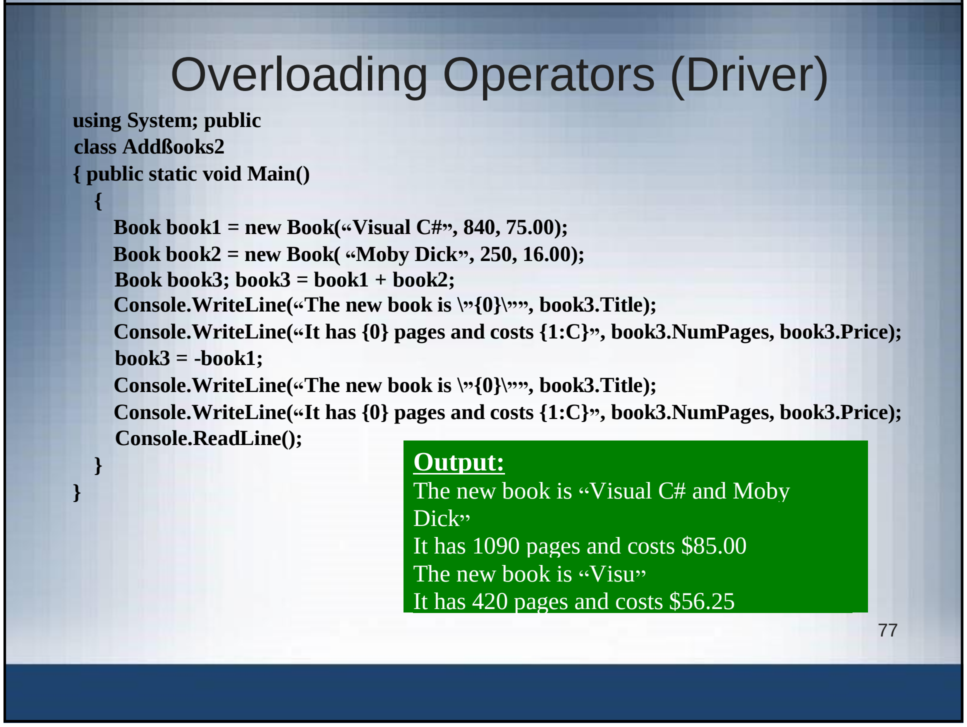
- Let’s add a unary operator (-) to the Book class

– Assume we define this operator to cut the title and number of pages in half and reduce the price to 75% of the original



– We have complete control over how the operators behave

```
public static Book operator –(Book tome)
{
   Book newBook = new Book();
   newBook.Title = tome.Title.Substring(0,tome.Title.Length/2);
   newBook.NumPages = tome.NumPages/2;
   newBook.Price = tome.Price * 0.75;
   return newBook;
}
```



Overloading Operators (Driver)

```
using System; public
class Addßooks2
{ public static void Main()
  {
     Book book1 = new Book(“Visual C#”, 840, 75.00);
     Book book2 = new Book( “Moby Dick”, 250, 16.00);
     Book book3; book3 = book1 + book2;
     Console.WriteLine(“The new book is \”{0}\””, book3.Title);
     Console.WriteLine(“It has {0} pages and costs {1:C}”, book3.NumPages, book3.Price);
     book3 = -book1;
     Console.WriteLine(“The new book is \”{0}\””, book3.Title);
     Console.WriteLine(“It has {0} pages and costs {1:C}”, book3.NumPages, book3.Price);
     Console.ReadLine();
  }
}
```

Output:

```
The new book is “Visual C# and Moby
Dick”
It has 1090 pages and costs $85.00
The new book is “Visu”
It has 420 pages and costs $56.25
```

Overloading Operators (one more)

- Add relational operators (>= and <=) to the Book class

– They have to be added in pairs and unlike the first two which create a new Book, these return true or false

– Notice that we can define the operators anyway we want

```
public static bool operator <=(Book b1, Book b2)
{ if ((b1.NumPages / b1.Price) <= (b2.NumPages / b2.Price)) return
      true;
   else
      return false;
} public static bool operator >=(Book b1, Book
b2)
```

```
{ if ((b1.Title.Length > b2.Title.Length) return
      true;
   else
      return false;
}
```

Overloading Operators (Driver)

```
using System; public
class Addßooks3
{ public static void Main()
   {
      Book book1 = new Book(«Visual C#: For Fun and Profit», 840, 75.00);
      Book book2 = new Book(«Moby Dick», 250, 16.00); if (book1 >=book2)
         Console.WriteLine(«Purchase Book 1»);
      else
         Console.WriteLine(«Purchase Book 2»);
      if (book1 <= book2)
         Console.WriteLine(«Purchase Book 1»);
      else
         Console.WriteLine(«Purchase Book 2»);
      Console.ReadLine();
   }

}
```

Output:

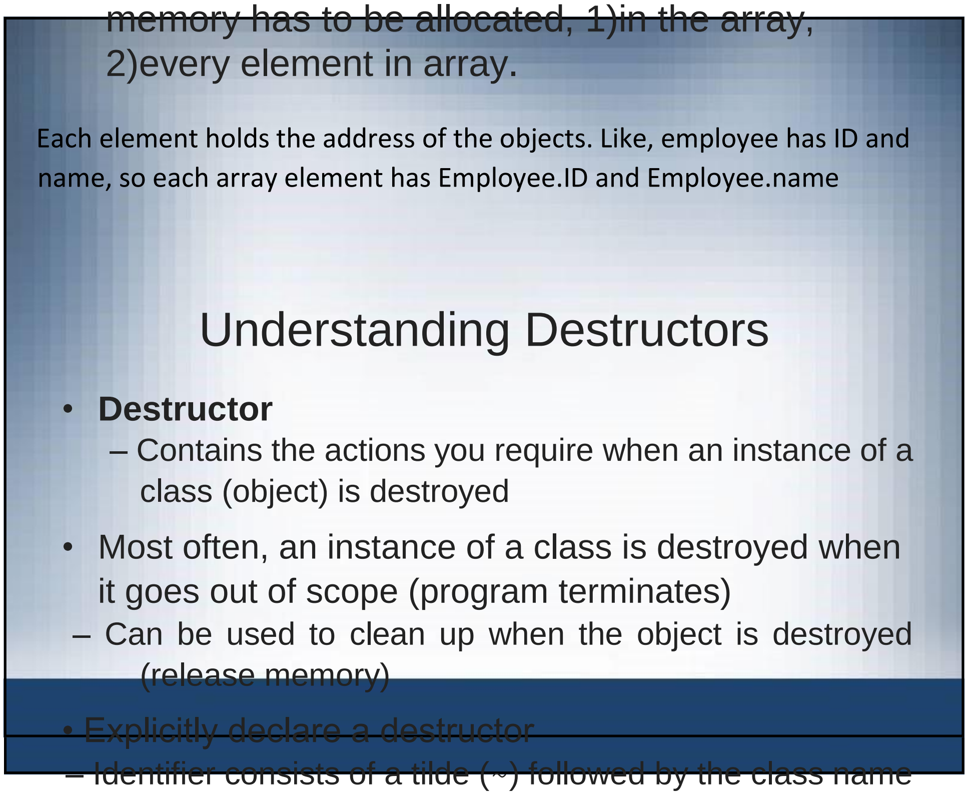
```
Purchase Book1
Purchase Book 1
```

Declaring an Array of Objects

- You can declare arrays that hold elements of any type
  - Including objects
- Example

```
Employee[] empArray = new Employee[7];

for(int x = 0; x < empArray.Length; ++x)
   empArray[x] = new Employee();
```

when declaring array of objects, two places memory has to be allocated, 1)in the array, 2)every element in array.



Each element holds the address of the objects. Like, employee has ID and name, so each array element has Employee.ID and Employee.name

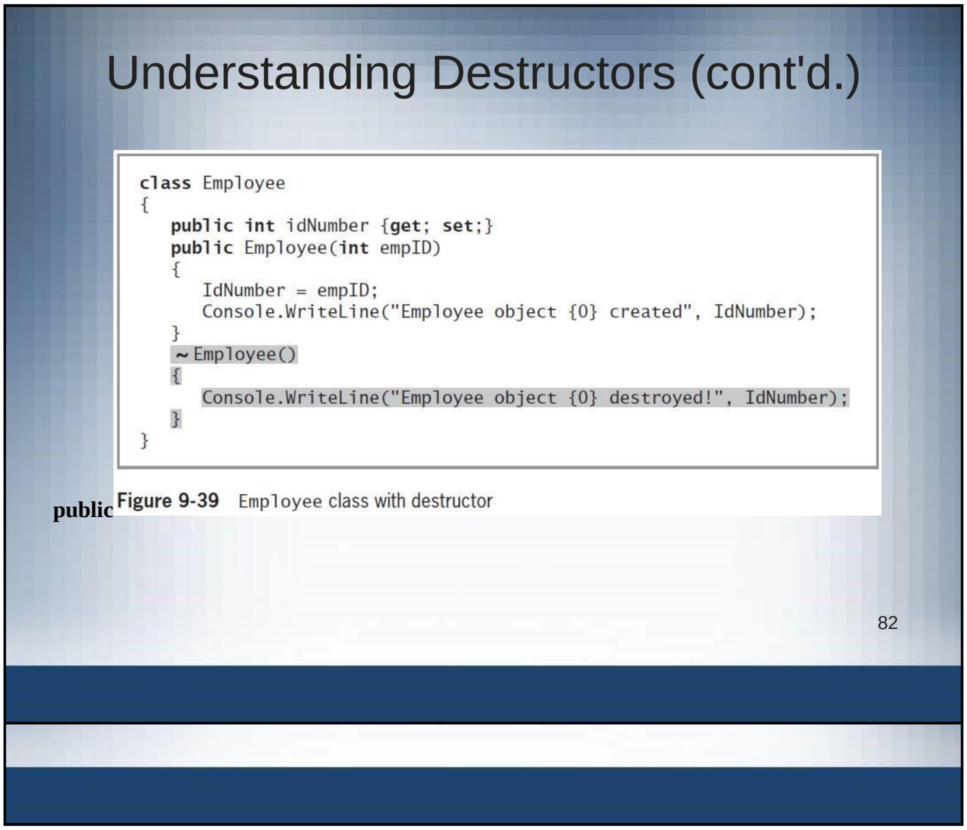
Understanding Destructors

- **Destructor**
  - Contains the actions you require when an instance of a class (object) is destroyed
- Most often, an instance of a class is destroyed when it goes out of scope (program terminates)
  - Can be used to clean up when the object is destroyed (release memory)
- Explicitly declare a destructor
  - Identifier consists of a tilde (~) followed by the class name

(1) A constructor takes the name of a class. Similarly, destructor takes name of the class with a tilde in front of it.

(2) Destructor doesn't take parameters

(3) Now if we constructed object 101 first then 202, so when a destructor is called, the 202 object gets destroyed first then 101, because like in methods, the last method called is on first



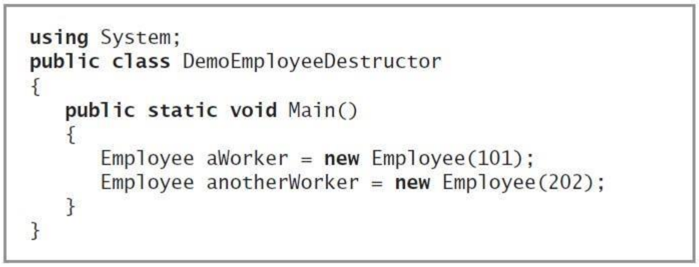
Understanding Destructors (cont'd.)

```
class Employee
{
   public int idNumber {get; set;}
   public Employee(int empID)
   {
      IdNumber = empID;
      Console.WriteLine("Employee object {0} created", IdNumber);
   }
   ~Employee()
   {
      Console.WriteLine("Employee object {0} destroyed!", IdNumber);
   }
}
```

public **Figure 9-39** Employee class with destructor

82

Understanding Destructors (cont'd.)



```
using System;
public class DemoEmployeeDestructor
{
   public static void Main()
   {
      Employee aWorker = new Employee(101);
      Employee anotherWorker = new Employee(202);
   }
}
```

Figure 9-40 DemoEmployeeDestructor program

```
Employee object 101 created
Employee object 202 created
Employee object 202 destroyed
Employee object 101 destroyed
```

83

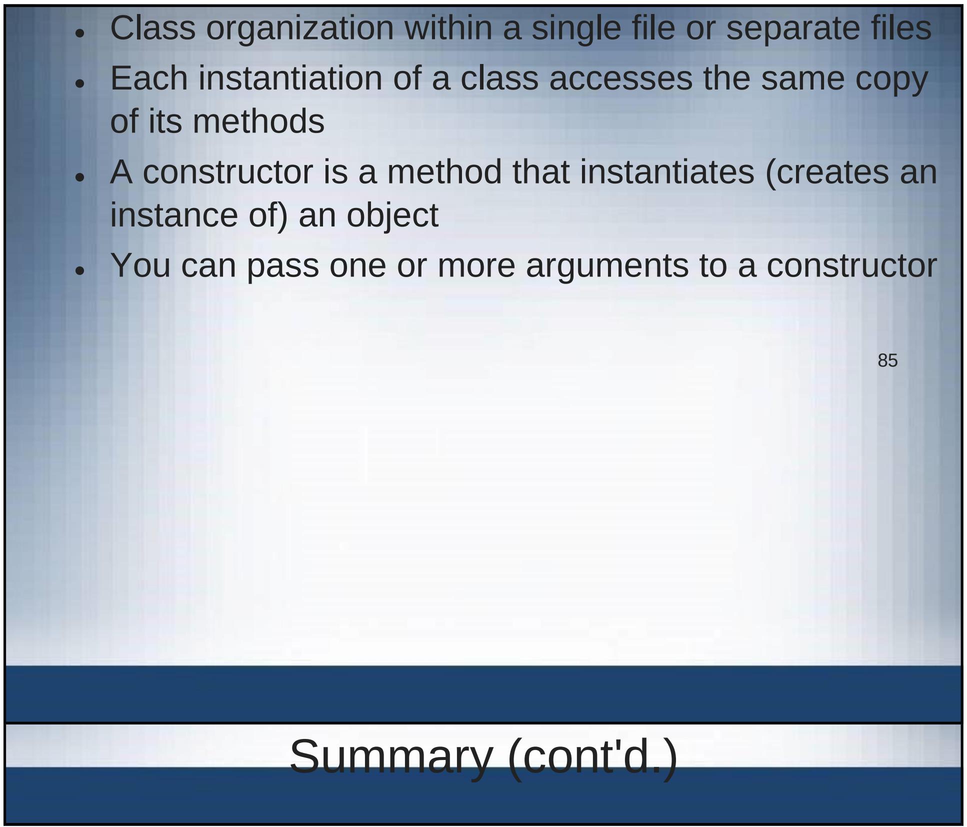
Summary

- You can create classes that are only programs with a `Main()` method and classes from which you instantiate objects
- When creating a class:
  - Must assign a name to it and determine what data and methods will be part of the class – Usually declare instance variables to be `private` and instance methods to be `public`
- When creating an object:
  - Supply a type and an identifier, and you allocate computer memory for that object

84

Summary (cont'd.)

- A property is a member of a class that provides access to a field of a class



- Class organization within a single file or separate files
- Each instantiation of a class accesses the same copy of its methods
- A constructor is a method that instantiates (creates an instance of) an object
- You can pass one or more arguments to a constructor

85

Summary (cont'd.)

- Constructors can be overloaded
- You can pass objects to methods just as you can simple data types
- You can overload operators to use with objects
- You can declare arrays that hold elements of any type, including objects
- A destructor contains the actions you require when an instance of a class is destroyed

86

```
using System;
public static class BankAccountDemo
{
    public static void Main()
    {
        int acctNumber;
        double amount;
        BankAccount savings = new BankAccount();
        BankAccount chequing = new BankAccount(12345, 350.45);
        BankAccount newAcct;

        // input a 5 digit account number and balance for savings
        do
        {
            Console.Write("Enter a 5-digit account number => ");
            acctNumber = Convert.ToInt32(Console.ReadLine());
        } while ((acctNumber < 10000) || (acctNumber > 99999));
        savings.AcctNum = acctNumber;

        // print out the account information
        Console.WriteLine("Account {0} contains {1:C2}", savings.AcctNum,
savings.Balance);
        Console.WriteLine("Account {0} contains {1:C2}", chequing.AcctNum,
chequing.Balance);

        // prompt the user to enter an amount to deposit to savings
        // *** Insert code
        double deposit;
        do
        {
            Console.WriteLine("Enter the amount you want to deposit : ");
            deposit = Convert.ToDouble(Console.ReadLine());

        } while (deposit < 0);

        // perform the deposit to savings
        // *** Insert code
        savings.Deposit(deposit);

        // print out the savings account information
        // *** Insert code
        Console.WriteLine("The amount {0:C} is depositted in account number
{1} and the new balance is {2:C} ", deposit, savings.AcctNum,
savings.Balance);

        // prompt the user to enter an amount to withdraw from chequing
        // *** Insert code
        Console.WriteLine("Enter the amount you want to withdraw : ");
        amount = Convert.ToDouble(Console.ReadLine());


        // perform the withdrawal from chequing
        // *** Insert code
        chequing.Withdrawal(amount);

        // print out the chequing account information
        // *** Insert code
        Console.WriteLine("The balance of chequing account is
{0:C}",chequing.Balance);


        // apply the interest to savings
        // *** Insert code
        savings.Interest();
        // print out the savings account information
        // *** Insert code
        Console.WriteLine("The savings account has {0:C}",savings.Balance);

        // combine chequing and savings into newAcct using overloaded
operator
        // *** Insert code
        newAcct = chequing + savings;

        // print out the newAcct account information
        Console.WriteLine("Account {0} contains {1:C2}", newAcct.AcctNum,
newAcct.Balance);
        Console.ReadLine();

        Console.WriteLine("Enter the amount to be e-transferred : ");
        amount = Convert.ToDouble(Console.ReadLine());

        //e-transfer
        chequing.etransfer(amount,savings);


        Console.WriteLine("The amount {0:C} was transferred to {1}", amount,
savings.AcctNum);

        Console.WriteLine("The chequing account has
{0:C}",chequing.Balance);

    }
}

using System;
public class BankAccount
{
    private int acctNum;
    private double balance;

    public const double SERVICE_CHARGE = 1.00;  // for Withdrawals only
    public const double INTEREST_RATE = 0.015;  // fixed interest rate

    // no arg constructor
    public BankAccount()
    {
        acctNum = 0;
        balance = 0;
    }

    // two arg constructor
    public BankAccount(int aNumber, double bal)
    {
        acctNum = aNumber;
        // ensuring the balance is not negative
        if (bal < 0)
            balance = 0;
        else
            balance = bal;
    }

    // AcctNum Property
```

```
public int AcctNum
{
    set
    { acctNum = value; }
    get
    { return acctNum; }
}

// Balance Property (read-only)
public double Balance
{
    get
    { return balance; }
}

// Deposit Method
public void Deposit(double amt)
{
    // check to see that the deposit amount is positive
    if (amt > 0)
        balance += amt;
}

// Withdrawal Method (a Service Charge)
public void Withdrawal(double amt)
{
    // *** Insert code
    if((amt+SERVICE_CHARGE)<balance)
    {
        balance -= (amt+SERVICE_CHARGE);
    }

}

// instance method to add interest onto the balance
public void Interest()
{
    // *** Insert code
    balance = balance * (1 + INTEREST_RATE);
}

// overloaded operator + to combine the contents of two accounts
//        Assume new account number will be the average of the
//        two account numbers and the balance will be the sum
// Parameters: the two accounts to be combined
// Returns: the new account with the
public static BankAccount operator +(BankAccount acc1, BankAccount acc2)
{
    // *** Insert code
    double x;
    int newAcctNum;
    x = acc1.balance + acc2.balance;
    newAcctNum = (acc1.AcctNum + acc2.AcctNum) / 2;

    BankAccount NewAcct = new BankAccount(newAcctNum, x);
    return NewAcct;
}
```

```
        //e-transfer method
        public void etransfer(double amt, BankAccount acct)
        {
            if (amt<balance)
            {
                Withdrawal(amt);

                acct.Deposit(amt);
            }
        }


    }
```