Chapter 9:

Using Classes and Objects

Objectives

- Learn about class concepts
- Create classes from which objects can be instantiated
- Create objects
- Create properties, including auto-implemented properties
- Learn more about using `public` and `private` access modifiers
- Learn about the `this` reference

Objectives (cont'd.)

- Write and use constructors
- Use object initializers
- Overload operators
- Declare an array of objects and use the `Sort()` and `BinarySearch()` methods with them
- Write destructors

Understanding Class Concepts

- 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
- **Instance variables** (also called fields)
  - Object attributes
  - Data components of a class

Understanding Class Concepts (cont'd.)

- **State**
  - A 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**
  - A program or class that instantiates objects of another prewritten class

Creating a Class from Which Objects Can Be Instantiated

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

Creating Instance Variables and Methods

- When creating a class, define both its attributes and its methods
- Field access modifiers
  - `new`, `public`, `protected`, `internal`, `private`, `static`, `readonly`, and `volatile`
- Most class fields are nonstatic and `private`
  - Provides the highest level of security

```
class Employee
{
   private int idNumber;
}
```

Figure 9-2 `Employee` class containing `idNumber` field

Creating Instance Variables and Methods (cont'd.)

- Using `private` fields within classes is an example of **information hiding**
- Most class methods are `public`
- `private` data/`public` method arrangement
  - Allows you to control outside access to your data
- **Composition**
  - Using an object within another object
  - Defines a **has-a relationship**

Creating Instance Variables and Methods (cont'd.)

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

Figure 9-3 Employee class with idNumber field and WelcomeMessage() method

Creating Objects

- Declaring a class does not create any actual objects
- The two-step process to create an object:
  - Supply a type and an identifier
  - Create the object, which allocates memory for it
- **Reference type**
  - Identifiers for objects are references to their memory addresses
- When you create an object, you call its constructor

Creating Objects (cont'd.)

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

Figure 9-4 The CreateEmployee program

Creating Objects (cont'd.)

Figure 9-5 Output of the `CreateEmployee` program

Passing Objects to Methods

- You can pass objects to methods just as you can simple data types

```
using System;
class CreateTwoEmployees
{
   static void Main()
   {
      Employee aWorker = new Employee();
      Employee anotherWorker = new Employee();
      DisplayEmployeeData("First", aWorker);
      DisplayEmployeeData("Second", anotherWorker);
   }
   static void DisplayEmployeeData(string order, Employee emp)
   {
      Console.WriteLine("\n{0} employee's message:", order);
      emp.WelcomeMessage();
   }
}
```

Figure 9-6 The CreateTwoEmployees program

Passing Objects to Methods (cont'd.)

Figure 9-7 Output of the `CreateTwoEmployees` program

Creating Properties

- **Property**
  - A member of a class that provides access to a field of a class
  - Defines how fields will be set and retrieved
- Properties have **accessors**
  - **`set` accessors** for setting an object's fields
  - **`get` accessors** for retrieving the stored values
- **Read-only property**
  - Has only a `get` accessor

Creating Properties (cont'd.)

```
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

Creating Properties (cont'd.)

- **Implicit parameter**
  - One that is undeclared and that gets its value automatically
- **Contextual keywords**
  - Identifiers that act like keywords in specific circumstances
  - `get`, `set`, `value`, `partial`, `where`, and `yield`

Creating Properties (cont'd.)

```
using System;
class CreateEmployee2
{
   static void Main()
   {
        Employee myChef = new Employee();
        myChef.IdNumber = 2345;
        Console.WriteLine("ID number is {0}",
           myChef.IdNumber);
        myChef.WelcomeMessage();
   }
}
```

Figure 9-9 The `CreateEmployee2` application that uses the `Employee` class containing a property

Creating Properties (cont'd.)

Figure 9-10 Output of the `CreateEmployee2` application

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

Using Auto-Implemented Properties (cont'd.)

```
using System;
class CreateEmployee3
{
   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;}
}
```

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

Using Auto-Implemented Properties (cont'd.)

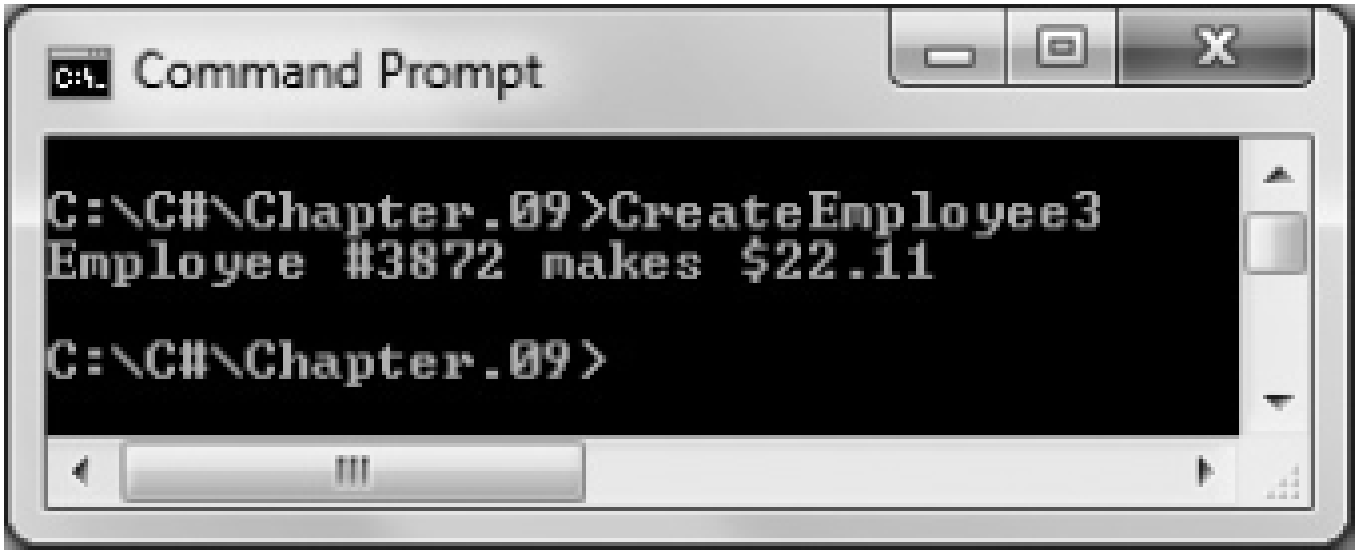

Figure 9-12 Output of the `CreateEmployee3` application

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 contain the same value
- A named constant within a class is always `static`
  - Belongs to the entire class, not to any particular instance

```
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;
   }
}
```

Figure 9-14 The Carpet class

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

```
using System;
class TestCarpet
{
   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);
   }
}
```

Figure 9-15 The `TestCarpet` class

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

Figure 9-16 Output of the TestCarpet program

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**
  - An implicitly passed reference
- When you call a method, you automatically pass the `this` reference to the method
  - It tells the method which instance of the class to use

Understanding the `this` Reference (cont'd.)

```
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 9-17 Partially developed Book class

Understanding the `this` Reference (cont'd.)

```
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 9-18 Book class with methods explicitly using `this` references

Understanding the `this` Reference (cont'd.)

```
using System;
class CreateTwoBooks
{
    static void Main()
    {
        Book myBook = new Book();
        Book yourBook = new Book();
        myBook.Title = "Silas Marner";
        yourBook.Title = "The Time Traveler's Wife";
        myBook.AdvertisingMessage();
        yourBook.AdvertisingMessage();
    }
}
```

Figure 9-19 Program that declares two Book objects

Understanding the `this` Reference (cont'd.)

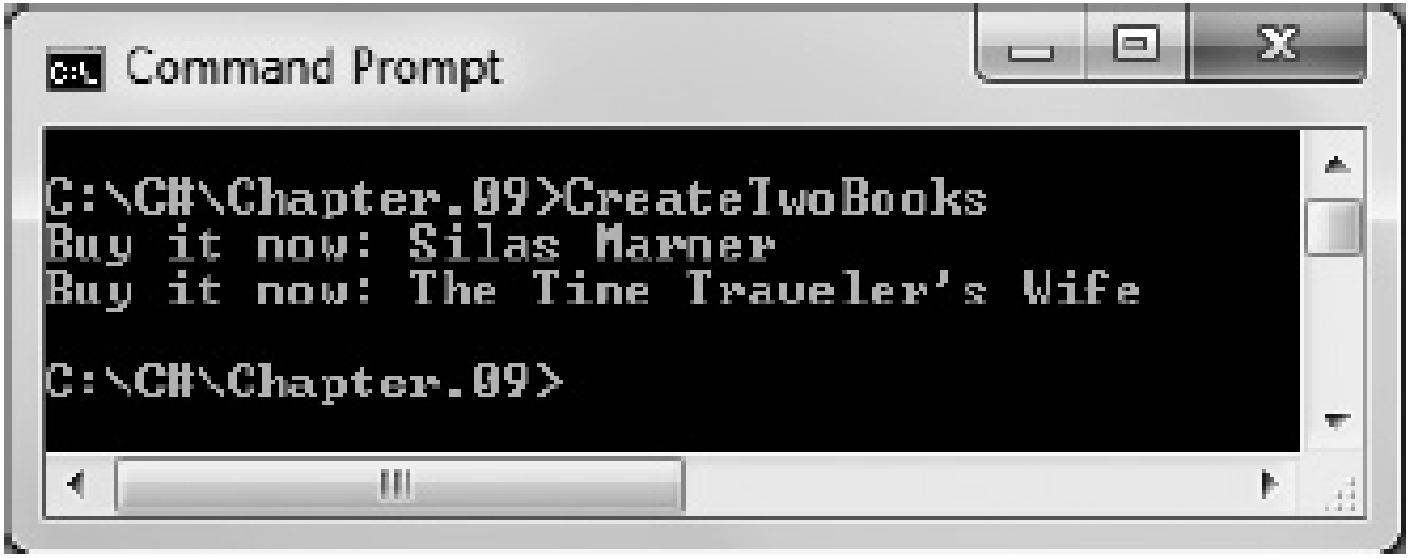

Figure 9-20 Output of the CreateTwoBooks program

Understanding the `this` Reference (cont'd.)

- Sometimes you must explicitly code the `this` reference

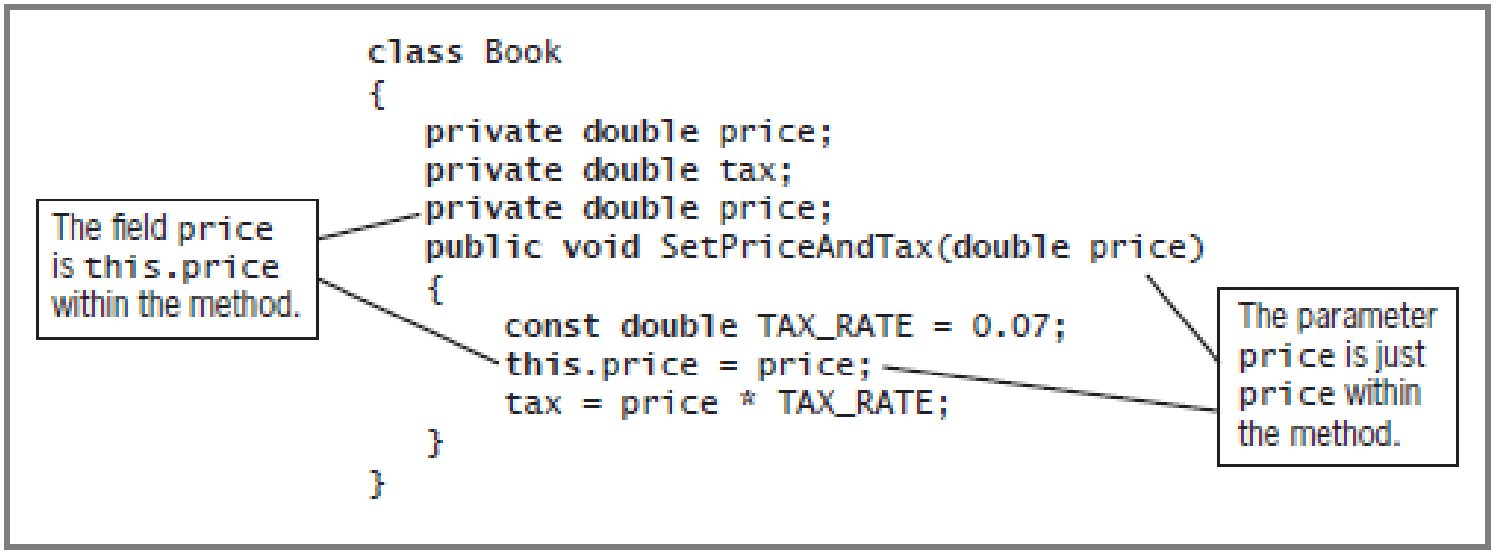

Figure 9-21 Book class that must explicitly use the `this` reference

Understanding Constructors

- **Constructor**
  - A method that instantiates an object
- **Default constructor**
  - An automatically supplied constructor without parameters
- **Default value of the object**
  - The value of an object initialized with a default constructor

Passing Parameters to Constructors

- **Parameterless constructor**
  - A constructor that takes no arguments

```
class Employee
{
    private int idNumber;
    private string name;
    public Employee()
    {
        PayRate = 9.99;
    }
    public double PayRate {get; set;}
    //  Other class members can go here
}
```

Figure 9-22 Employee class with a parameterless constructor

Passing Parameters to Constructors (cont'd.)

- You can create a constructor that receives argument(s)

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

Figure 9-23 Employee constructor with parameter

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

```
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.

Figure 9-24 Employee class with four constructors

Overloading Constructors (cont'd.)

```
using System;
class CreateSomeEmployees
{
   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 9-25 The CreateSomeEmployees program

Overloading Constructors (cont'd.)

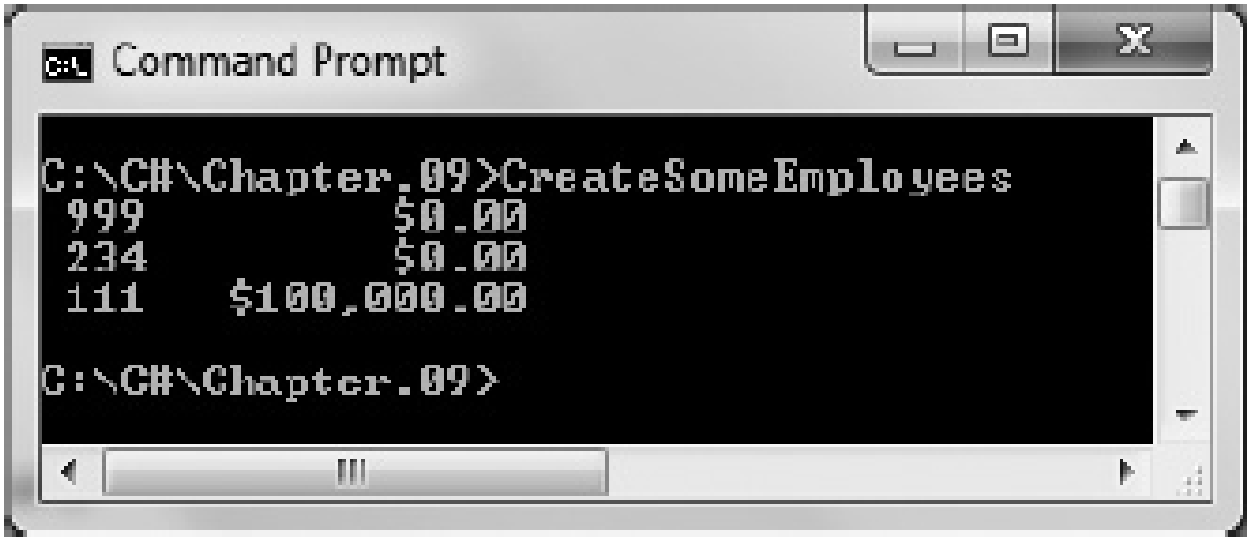

Figure 9-26 Output of the `CreateSomeEmployees` program

Using Constructor Initializers

- **Constructor initializer**
  - A clause that indicates another instance of a class constructor should be executed before any statements in the current constructor body

Using Constructor Initializers (cont'd.)

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

Figure 9-27 Employee class with constructor initializers

Using the `readonly` Modifier in a Constructor

- `readonly` modifiers are like named constants
- They are assigned a value that cannot be changed
  - Their value can be assigned at run time rather than at compile time
  - They can get their value from user input or the operating system

Using Object Initializers

- **Object initializer**
  - Allows you to assign values to any accessible members or properties of a class at the time of instantiation without calling a constructor with parameters
- For you to use object initializers, a class must have a default constructor

Using Object Initializers (cont'd.)

```
using System;
class DemoObjectInitializer
{
   static void Main()
   {
      Employee aWorker = new Employee {IdNumber = 101};
      Console.WriteLine("Employee #{0} exists. Salary is {1}.",
         aWorker.IdNumber, aWorker.Salary);
   }
}
class Employee
{
   public int IdNumber {get; set;}
   public double Salary {get; set;}
   public Employee()
   {
      Salary = 99.99;
      Console.WriteLine("Employee #{0} created. Salary is {1}.",
         IdNumber, Salary);
   }
}
```

Figure 9-29 The DemoObjectInitializer program

Using Object Initializers (cont'd.)

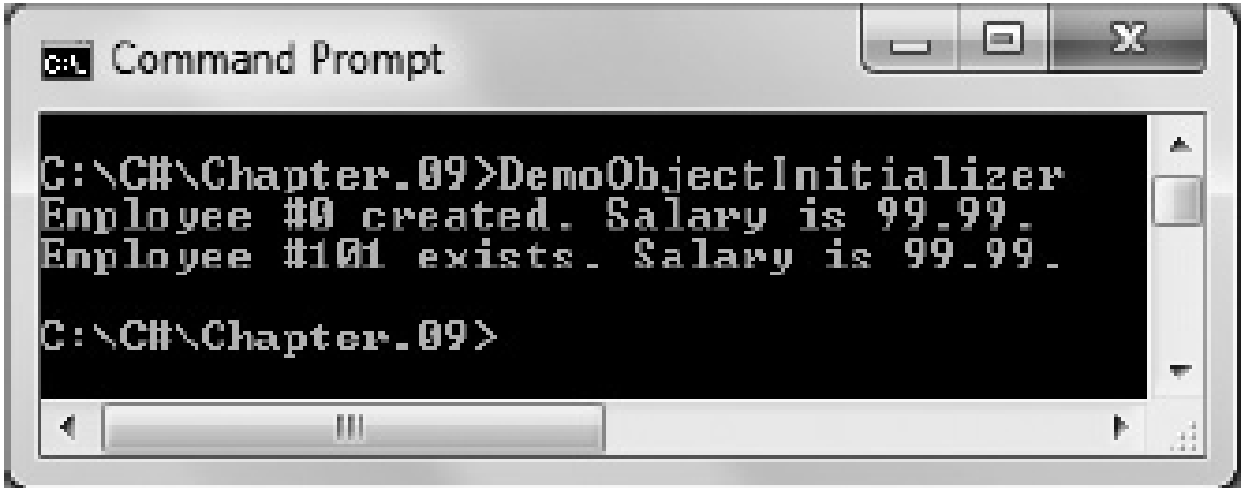

Figure 9-30 Output of the `DemoObjectInitializer` program

Using Object Initializers (cont'd.)

- Using object initializers allows you to:
  - Create multiple objects with different initial assignments without having to provide multiple constructors to cover every possible situation
  - Create objects with different starting values for different properties of the same data type

Using Object Initializers (cont'd.)

```
class Box
{
    public int Height {get; set;}
    public int Width {get; set;}
    public int Depth {get; set;}
    public Box()
    {
        Height = 1;
        Width = 1;
        Depth = 1;
    }
}
```

Figure 9-31 The Box class

Using Object Initializers (cont'd.)

```
using System;
class DemoObjectInitializer2
{
   static void Main()
   {
      Box box1 = new Box {Height = 3};
      Box box2 = new Box {Width = 15};
      Box box3 = new Box {Depth = 268};
      DisplayDimensions(1, box1);
      DisplayDimensions(2, box2);
      DisplayDimensions(3, box3);
   }
   static void DisplayDimensions(int num, Box box)
   {
      Console.WriteLine("Box {0}: Height: {1} Width: {2} Depth: {3}",
        num, box.Height, box.Width, box.Depth);
   }
}
```

Figure 9-32 The DemoObjectInitializer2 program

Using Object Initializers (cont'd.)

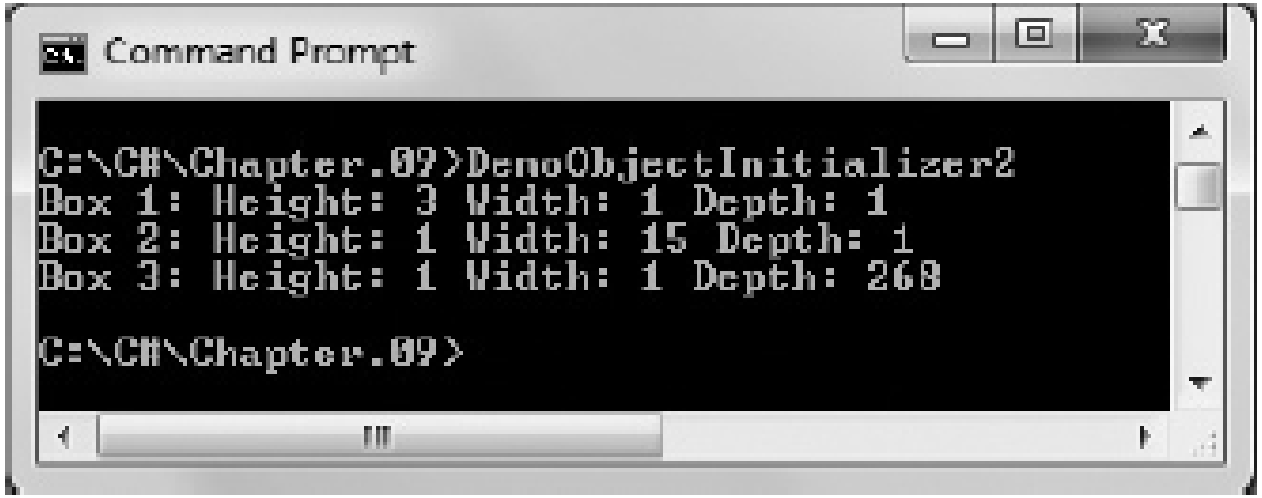

Figure 9-33 Output of the `DemoObjectInitializer2` program

Overloading Operators

- Overloading operators
  - Enables you to use arithmetic symbols with your own objects
- Overloadable unary operators:

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

- Overloadable binary operators:

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

- You cannot overload the following operators:

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

- You cannot overload an operator for a built-in data type

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*)

```
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 9-34 Book class with overloaded + operator

Overloading Operators (cont'd.)

```
using System;
class AddBooks
{
   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 9-35 The AddBooks program

Overloading Operators (cont'd.)

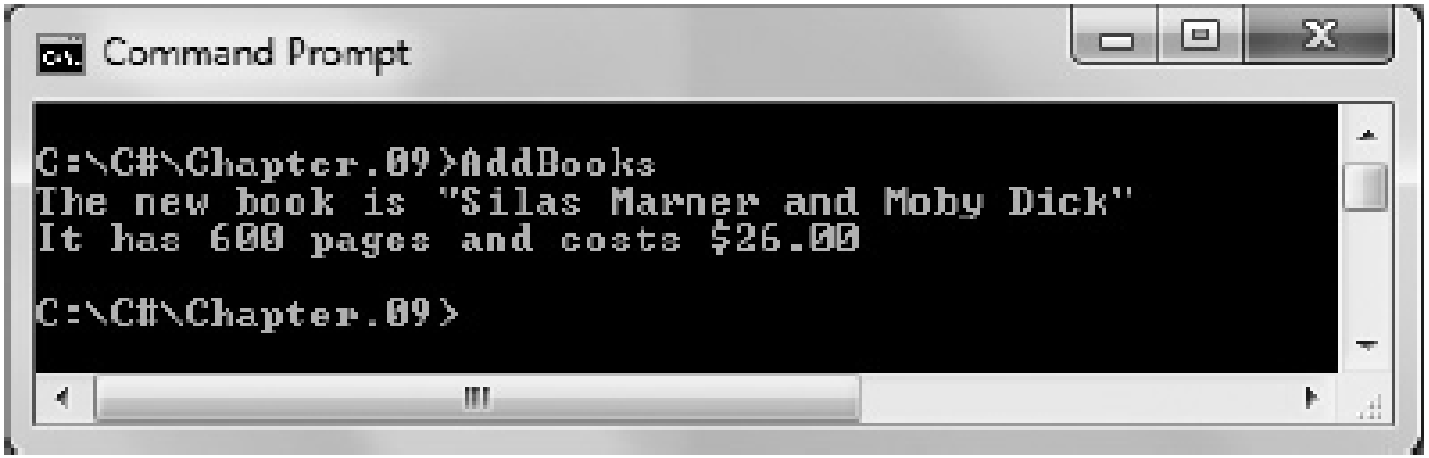

Figure 9-36 Output of the AddBooks program

Overloading Operators (cont'd.)

- Overloaded unary operators take a single argument

```
public static Book operator-(Book aBook)
{
    aBook.Price = -aBookPrice;
    return aBook;
}
```

Figure 9-37 An operator-() method for a Book

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();
```

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects

- **`CompareTo()` method**
  - Provides the details of how the basic data types compare to each other
  - Used by the `Sort()` and `BinarySearch()` methods
- When you create a class that contains many fields, tell the compiler which field to use when making comparisons
  - Use an interface

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects (cont'd.)

- **Interface**
  - A collection of methods that can be used by any class as long as the class provides a definition to override the interface's do-nothing, or abstract, method definitions
- When a method **overrides** another, it takes precedence, hiding the original version
- **`IComparable` interface**
  - Contains the definition for the `CompareTo()` method
  - Compares one object to another and returns an integer

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects (cont'd.)

```
interface IComparable
{
    int CompareTo(Object o);
}
```

Figure 9-38 The IComparable interface

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects (cont'd.)

Return Value	Meaning
Negative	This instance is less than the compared object.
Zero	This instance is equal to the compared object.
Positive	This instance is greater than the compared object.

Table 9-2 Return values of `IComparable.CompareTo()` method

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects (cont'd.)

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

    int IComparable.CompareTo(Object o)
    {
        int returnVal;
        Employee temp = (Employee)o;
        if(this.IdNumber > temp.IdNumber)
            returnVal = 1;
        else
            if(this.IdNumber < temp.IdNumber)
                returnVal = -1;
            else
                returnVal = 0;
        return returnVal;
    }
}
```

Figure 9-39 Employee class using IComparable interface

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects (cont'd.)

```
using System;
class ComparableEmployeeArray
{
   static void Main()
   {
      Employee[] empArray = new Employee[5];
      int x;
      for(x = 0; x < empArray.Length; ++x)
         empArray[x] = new Employee();
      empArray[0].IdNumber = 333;
      empArray[1].IdNumber = 444;
      empArray[2].IdNumber = 555;
      empArray[3].IdNumber = 111;
      empArray[4].IdNumber = 222;
      Employee seekEmp = new Employee();
      seekEmp.IdNumber = 222;
      Array.Sort(empArray);
      Console.WriteLine("Sorted employees:");
      for(x = 0; x < empArray.Length; ++x)
        Console.WriteLine("Employee #{0}: {1} {2}",
           x, empArray[x].IdNumber,
           empArray[x].Salary.ToString("C"));
      x = Array.BinarySearch(empArray, seekEmp);
      Console.WriteLine("Employee #{0} was found at position {1}",
         seekEmp.IdNumber, x);
   }
}
```

Figure 9-40 ComparableEmployeeArray program

Using the `Sort()` and `BinarySearch()` Methods with Arrays of Objects (cont'd.)

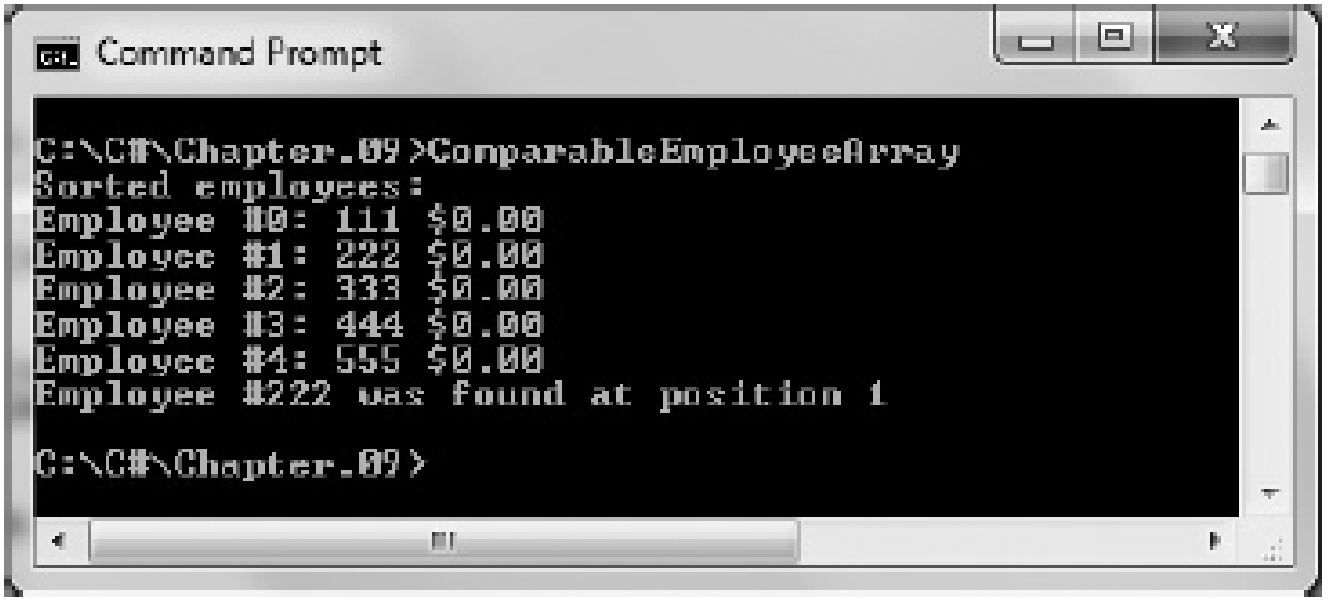

Figure 9-41 Output of the ComparableEmployeeArray program

Understanding Destructors

- **Destructor**
  - Contains the actions you require when an instance of a class is destroyed
- Most often, an instance of a class is destroyed when it goes out of scope
- Explicitly declare a destructor
  - The identifier consists of a tilde (~) followed by the class name

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);
    }
}
```

Figure 9-43 Employee class with destructor

Understanding Destructors (cont'd.)

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

Figure 9-44 DemoEmployeeDestructor program

Understanding Destructors (cont'd.)

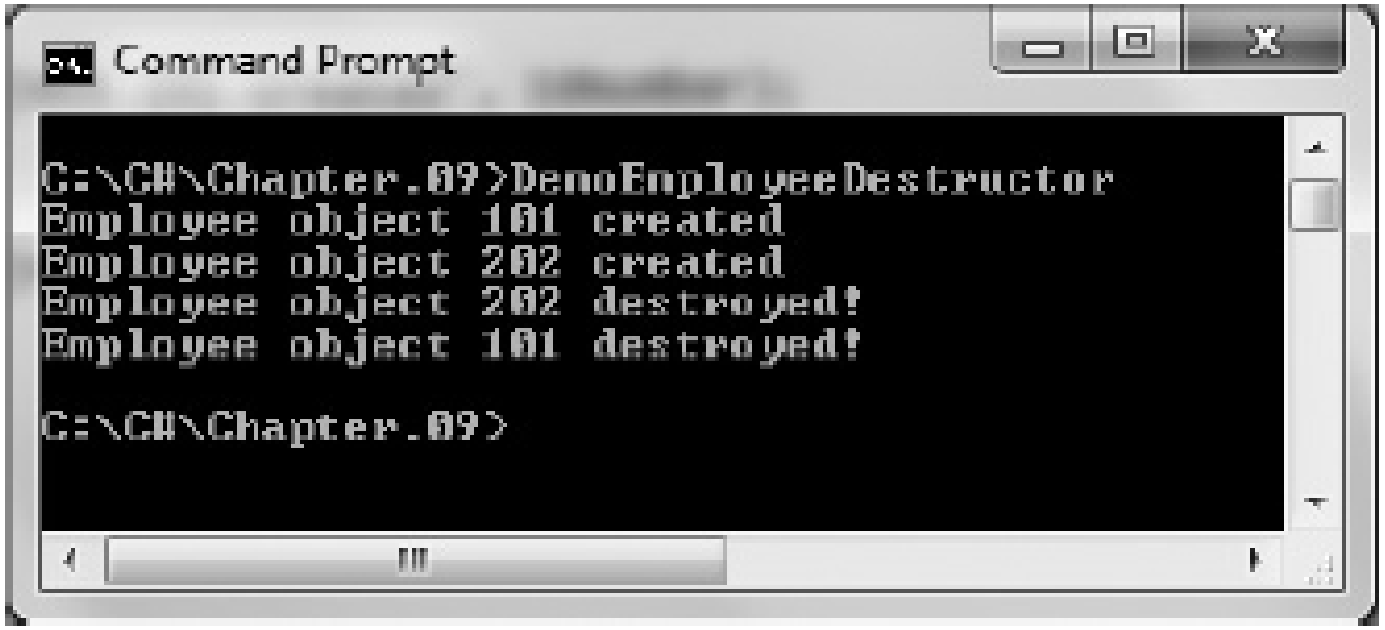

Figure 9-45 Output of DemoEmployeeDestructor program

You Do It

- Creating a Class and Objects
- Using Auto-Implemented Properties
- Adding Overloaded Constructors to a Class
- Creating an Array of Objects

Summary

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

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

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