

Module 4: Implementing Object-Oriented Programming Techniques in C#

Contents

Overview	1
Lesson: Designing Objects	2
Lesson: Using Inheritance	12
Lesson: Using Polymorphism	24
Review	36
Lab 4.1: Creating Classes in C#	37



Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2002 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, FrontPage, IntelliSense, JScript, Microsoft Press, MSDN, PowerPoint, Visual Basic, Visual C++, Visual C#, Visual Studio, Win32, Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Overview

- Designing Objects
- Using Inheritance
- Using Polymorphism

Introduction

This module describes the most important principles of object-oriented design: encapsulation, inheritance, and polymorphism. It discusses the benefits of object-oriented programming, and it explains how to design classes so that they encapsulate functionality but limit accessibility to information that the users of your objects do not need.

This module also explains how to create classes that other classes can use through the process of inheritance, so that you can reuse previous work and increase productivity. Finally, this module explains how to override methods that are provided by a base class and how to define abstract classes that specify a set of functionality that a derived class must follow.

Objectives

After completing this module, you will be able to:

- Encapsulate information in an object.
- Create an object that inherits functionality from another object.
- Implement polymorphism to use abstract classes.

Lesson: Designing Objects

- What Are the Benefits of Object-Oriented Programming?
- What Is Encapsulation?
- What Are Properties?

Introduction

This lesson explains how to use the design principals of abstraction and encapsulation to create classes that present a useful programming model to the object user.

Lesson objectives

After completing this lesson, you will be able to:

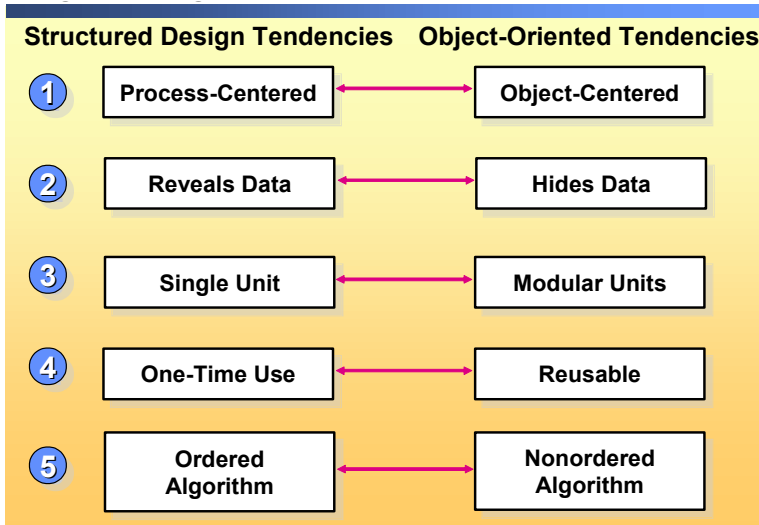
- List the benefits of object oriented programming.
- Encapsulate data in an object.
- Use properties to manage access to encapsulated data.

Lesson agenda

This lesson includes the following topics and activity:

- What Are the Benefits of Object-Oriented Programming?
- What Is Encapsulation?
- What Are Properties?
- Practice: Writing and Using Properties

What Are the Benefits of Object-Oriented Programming?



Introduction

An object-oriented approach to programming provides many benefits over the structured approach.

Process-centered vs. object-centered

The structured approach to programming is process-centered, meaning that it takes a problem and focuses on a hierarchy of processes that must be performed sequentially to arrive at a solution.

Object-oriented analysis and design focuses on objects. The objects have certain behaviors and attributes that determine how they interact and function. No attempt is made to provide an order for those actions at design time because objects function based on the way other objects function.

Object-oriented programming allows developers to create objects that reflect real-world scenarios. Most people find the object-oriented approach a much more natural design model than other methodologies. This is because it meshes well with the way people naturally interpret the world. Human understanding largely rests on identification and generalization (objects and classes), finding relationships between groups, and interacting through the normal interface of an entity (behaviors).

Reveals data vs. hides data

The structured approach packages data and procedures, which are revealed or accessible to the rest of the program. There is little effort to actually hide information from other processes. The structured approach leaves this decision up to the implementer.

The object-oriented implementations hide data, which shows only behaviors to users and hides the underlying code of an object. The behaviors that the programmer exposes are the only items that the user of the object can affect.

Single unit vs. modular unit

The structured approach is based on a single unit of code, where processes call other processes and are dependent on each other.

The object-oriented approach allows objects to be self-contained. Objects stand on their own, with the functionality to call behaviors of other objects. Using the object-oriented approach, developers can create applications that reflect real-world objects such as rectangles, ellipses, and triangles, in addition to money, part numbers, and items in inventory.

One-time use vs. reusable

Structured processes may not be reusable, depending on the implementation.

In the object-oriented approach, objects are by definition modular in their construction. That is, they are complete entities and therefore tend to be highly reusable.

Consider the example of buying a new car. The manufacturer builds a base model car. If you prefer additional features such as air conditioning, power windows, and a sunroof, these items can be added to the car. By adding features, you extend the characteristics of the base model instead of building an entirely new car.

Ordered algorithm vs. nonordered algorithm

Structured approaches with processes tend to result in linear, or top down, algorithm-based implementations.

Object-oriented applications are constructed on a message-based or event-driven paradigm, where objects send messages to other objects, such as the Microsoft® Windows® operating system.

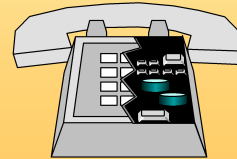
Summary of benefits

In summary, object-oriented programming benefits developers because:

- Programs are easier to design because objects reflect real-world items.
- Applications are easier for users because data they do not need is hidden.
- Objects are self-contained units.
- Productivity increases because you can reuse code.
- Systems are easier to maintain and adapt to changing business needs.

What Is Encapsulation?

- **Grouping related pieces of information and processes into self-contained unit**
 - Makes it easy to change the way things work under the cover without changing the way users interact
- **Hiding internal details**
 - Makes your object easy to use



Introduction

Non-object-oriented programming languages consist of data, either in a database or in computer memory, and separate instructions for manipulating that data. These languages do not usually enforce any sort of relationship between the data and the code that manipulates the data. If any aspect of the data changes—for example, if a year field is changed from 2 digits to 4 digits then all of the code that uses that data must also be changed. Because the code is not closely related to the data, changing the code can be difficult and time-consuming.

Definition

In object-oriented programming, *encapsulation* is the enclosing of both properties and methods (the data and the code that manipulates that data) together in a common structure. Encapsulating both data and the actions that manipulate that data together in this way, and specifying the actions and properties of the object, creates a new data type called a *class*.

Benefit of encapsulation

When data and methods are encapsulated, you can specify methods and properties that define how the external user sees your information and how they can request actions from the object. By hiding information that users do not need, such as implementation information, the user can concentrate on only the useful characteristics of the object.

For example, the internal mechanism of a telephone is hidden from the user. The wires, switches, and other internal parts of a telephone are encapsulated by its cover to allow the user to focus on using the phone and not on the internal operations of the telephone.

This abstraction also enables you to easily change the implementation details of your application without the users of your object experiencing any change in the way they interact with the object.

Design considerations

When you design a class, there are several questions to consider. What will the class represent? What actions should the class provide? What elements does a user of the class need to see?

When you design a class, you should attempt to hide as much of the implementation detail as possible and expose only those actions and values that the users of your class need to know about. This enables you to improve and change the implementation details of your class later, without changing the way users interact with the class. Therefore, do not expose elements just because you think that they might be useful. As you refine your design, you can always add elements to the public definition when you clearly see a need.

Example

At a zoo, visitors can look at a monitor outside an exhibit to learn about the animals. The monitor at the elephant exhibit shows visitors educational information about the elephant, including its size, dietary requirements, reproductive rate, and life span. The application that runs the display uses an **Elephant** object to access information about the elephant. The **Elephant** object hides internal information that the display application does not need to see, such as where the animal-specific data is stored, and how the data is structured in the database.

When you design your application in this way, it is easy for you to change the implementation details without changing the interface. For example, you can move the files to a different database location or even database type without the visitors seeing anything different on the display and without the application that uses your objects changing its implementation.

By preventing access to internal data structures, you also prevent users of your object from accessing information that could corrupt your object.

What Are Properties?

- Properties are methods that protect access to class members

```
private int animalWeight;  
public int Weight {  
    get {  
        return animalWeight;  
    }  
    set {  
        animalWeight = value;  
    }  
}
```

Introduction

To separate the implementation details of your objects from what the user sees, you can define the scope of the class members and thereby control access to the data in your objects.

Although you can control access to class members by using access modifiers, an even more powerful way to manage access is through the use of properties. By using properties, you can manage the access that other objects have to data in your class.

Definition

Properties are class members that provide access to elements of an object or class.

Syntax

The syntax for defining a property consists of an access modifier, such as **public** or **protected**, followed by a type, the property name, the keywords **get** and **set**, and the property code for each in curly braces, as shown in the following code:

```
public int myIntegerProperty {  
    get {  
        // Property get code  
    }  
    set {  
        // Property set code  
    }  
}
```

The **get** and **set** statements are called *accessors*.

The **get** accessor must return a type that is the same as the property type, or one that can be implicitly converted to the property type. The **set** accessor is equivalent to a method that has one implicit parameter, named **value**.

Non-Example 1

You are writing an application to track the amount of food that zoo animals consume so that you can use this value to predict the size of future food purchases. You decide to represent this consumption value as **DailyFoodIntake**, as shown in the following code:

```
class Elephant {  
    // Not a good idea!  
    public decimal DailyFoodIntake;  
}  
  
class Zoo {  
    static void Main(string[] args) {  
        Elephant zooElephant = new Elephant();  
        zooElephant.DailyFoodIntake = 300M;  
    }  
}
```

This code allows the user of the object to directly access the **DailyFoodIntake** value of **zooElephant** and alter it. This is a design flaw because the programmer has no ability to ensure that the change is allowable or that the value is correct.

Non-Example 2

You gain more control over the variable by making it private and using a method to access it, as shown in the following example:

```
class Elephant {  
    private decimal dailyFoodIntake;  
  
    public decimal GetDailyFoodIntake() {  
        return dailyFoodIntake;  
    }  
  
    public void SetDailyFoodIntake(decimal newRate) {  
        if ( newRate < dailyFoodIntake - 25 ) {  
            // call the vet  
        }  
        else {  
            dailyFoodIntake = newRate;  
        }  
    }  
}  
  
class Zoo {  
    static void Main(string[] args) {  
        Elephant e = new Elephant();  
        e.SetDailyFoodIntake( 300M );  
    }  
}
```

However, this approach also has some disadvantages. This code contains two methods: one to set the amount of food that is eaten daily, and one to get the amount. The **dailyFoodIntake** member is private so you control the values that can be set, but this implementation requires the user of the object to remember and use two method names rather than only one element.

Example using properties

Using properties is the best way to declare **DailyFoodIntake**, as shown in the following code:

```
using System;

namespace LearningCSharp {
    class Elephant {
        private decimal dailyConsumptionRate;

        public decimal DailyFoodIntake {
            get {
                return dailyConsumptionRate;
            }
            set {
                if ( value < dailyConsumptionRate - 25 ) {
                    // notify medical center
                }
                else {
                    dailyConsumptionRate = value;
                }
            }
        }
    }
}

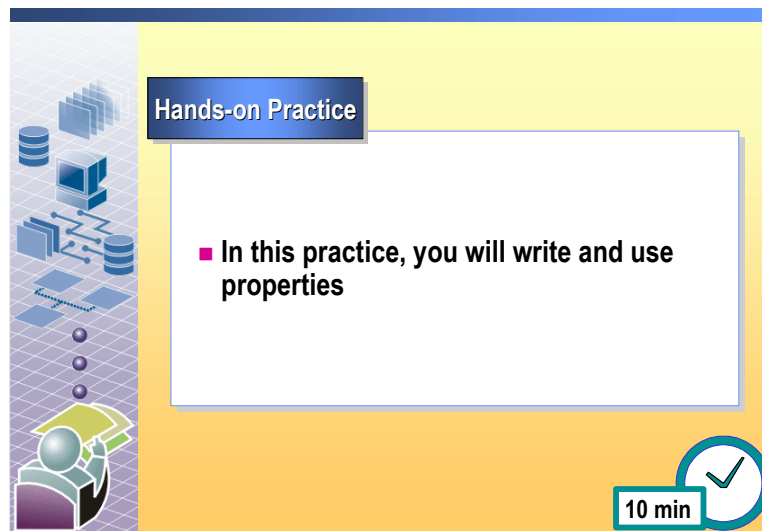
class Zoo {
    static void Main(string[] args) {
        Elephant e = new Elephant();
        e.DailyFoodIntake = 300M;
    }
}
```

In this example, users of the **Elephant** object can access the **DailyFoodIntake** method in the same way that they would access a public member variable in the class. The implementer of the class can separate the interface that it provides, **DailyFoodIntake**, from the member variable that is used internally by the class, **dailyConsumptionRate**, to predict animal food purchases. In future implementations, the programmer can change **dailyConsumptionRate** to another type, but users of the **Elephant** class will not have to modify their code.

Note that the **set** accessor uses the **value** keyword to retrieve the new value.

This code sample is available on the Student Materials compact disc as **Properties.sln** in the folder **Samples\Mod04\Properties**.

Practice: Writing and Using Properties



In this practice, you will write and use properties.

Scenario: the Zoo Medical Center requires a way to represent information about the animals in its care. You have been asked to develop an object that represents information about lions, such as name, age, weight, gender. You are given an example of how the user of this object wants it to behave.

In this practice, you will change the **Gender** member of the **Lion** class from a member variable to a property.

The solution code for this practice located in *install_folder*\Practices\Mod04\Properties_Solution\Properties_Solution.sln. Start a new instance of Microsoft Visual Studio® .NET before opening the solution.

Tasks	Detailed steps
<p>1. Start Visual Studio .NET, and then open <i>install_folder</i>\Practices\Mod04\Properties\Properties.sln.</p> <p>Examine the Lion class, which is located near the top of the code window.</p>	<p>a. Start a new instance of Visual Studio .NET.</p> <p>b. On the Start Page, click Open Project.</p> <p>c. In the Open Project dialog box, browse to <i>install_folder</i>\Practices\Mod04\Properties, click Properties.sln, and then click Open.</p> <p>d. In Solution Explorer, click Form1.cs, and then press F7 to open the Code Editor. Review the Lion class, which is located near the top of the code window.</p>
<p>2. Examine the Task List.</p>	<p>■ On the View menu, point to Show Tasks, and then click All.</p>

Tasks	Detailed steps
3. Implement the rule that the Gender can be Male or Female .	<ul style="list-style-type: none">a. To view the Task List, on the View menu, point to Show Tasks, and then click All.b. In the Task List, double-click TODO: Change Gender member of Lion class to be a property.c. Modify Gender so that it is a property.d. Compile and run your program.e. In your application, click Run, and then check that the output is as follows: Leo: Male age 8; weighs 280kg
4. Save your application, and then quit Visual Studio .NET.	<ul style="list-style-type: none">a. Save your application.b. Quit Visual Studio .NET.

Optional:

- Change the other members of the **Lion** class to be properties.
- Represent the gender as an enumeration.

Lesson: Using Inheritance

- What Is Inheritance?
- How to Create a Derived Class
- How to Call a Base Constructor from a Derived Class
- How to Use a Sealed Class

Introduction

This lesson explains how to implement inheritance by creating base classes from which other classes can be derived.

Lesson objectives

After completing this lesson, you will be able to:

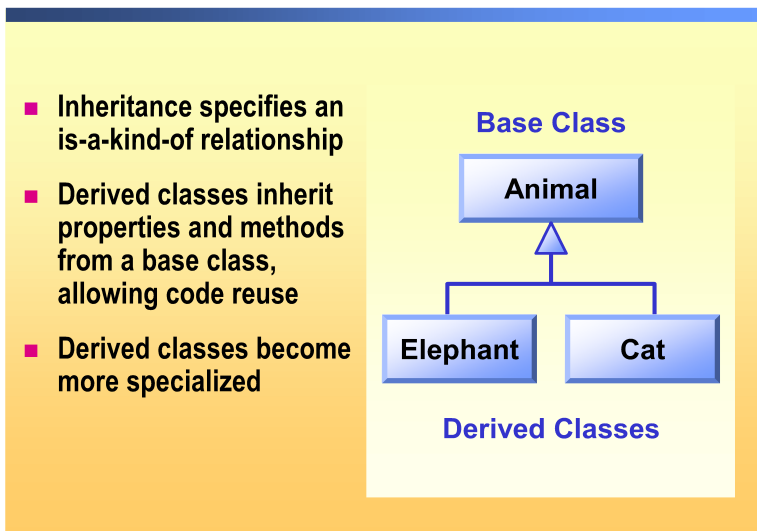
- Design a base class.
- Create a derived class.
- Create a sealed class.

Lesson agenda

This lesson includes the following topics and activity:

- What Is Inheritance?
- How to Create a Derived Class
- How to Call a Base Class Constructor from a Derived Class
- How to Use a Sealed Class
- Practice: Creating a Derived Class

What Is Inheritance?



Introduction

When you design an application, you often must manage similar but not identical items. The object-oriented principle of inheritance enables you to create a generalized class and then derive more specialized classes from it. The general class is referred to as the *base* class. A more specific class is referred to as the *derived* class. Derived classes inherit properties and methods from the base class.

Definition

Inheritance is the ability of a derived class to take on the characteristics of the class or derived class on which it is based. Inheritance allows a common set of behaviors, defined as properties and methods, to be included in a base class and reused in derived classes. It is a means for creating new, more specific types from an existing, more general type. Inheritance also defines one type as a subcategory of another type.

Benefits

The primary benefit of inheritance is code reuse. In a base class, you can write code once that all derived classes will automatically inherit. Inheritance facilitates reusability.

Example of an inheritance hierarchy

In a hierarchy that uses inheritance, you can design several classes, each one deriving from the class above it. When you design classes using inheritance, look for the common features of the objects, and then factor these into a hierarchy of classes with increasingly more specific attributes, as shown in the following example:

Animal	
Mammal	
Monotremes	Duck-billed platypus
Multitubercules	Extinct
Marsupials	Kangaroo
Utherians	Mouse, Bat, Human

The most general features and functions that are common to all animals are defined at the highest level of the hierarchy. In lower levels, previously-defined features and functionality are inherited and new features are added so that each level becomes more specific and more specialized.

Note When you create a derived class in C#, you specify only one base class.

Non-example

In an application that manages animals, you may need to implement objects for a cat, a mouse, and an elephant. You can set up your application by creating three separate classes, one for each type of animal, as shown in the following code.

Note *{ }* indicates where the implementation of the methods would exist.

```
public class Antelope {
    public bool IsSleeping;
    public void Sleep() { }
    public void Eat() { }
}

public class Lion {
    public bool IsSleeping;
    public void Sleep() { }
    public void EatAntelope( ) { }
    public void StalkPrey() { }
}

public class Elephant {
    public bool IsSleeping;
    public void Sleep() { }
    public int CarryCapacity;
    public void Eat() { }
}

Elephant e = new Elephant();
e.Sleep();
```

Why the non-example is poor design

The design in the previous example has two obvious disadvantages:

- *Duplication of code.* For example, the **Sleep** method must be implemented multiple times.
- *User confusion.* The eating methods have different names in different objects.

A better way of designing this application is by using inheritance. You can place the common features of the animals in a base class, derive new classes that inherit these features from the base class, and then refine the derived class to implement any object-specific changes.

How to Create a Derived Class

```
public class Animal {  
    protected bool IsSleeping;  
    public void Sleep() { }  
    public void Eat() { }  
}  
  
public class Lion : Animal {  
    public void StalkPrey() { }  
}  
...  
Lion adoptedLion = new Lion();  
adoptedLion.StalkPrey();  
adoptedLion.Eat();
```

Introduction

You can use many of the classes that are available in the Microsoft .NET class library as base classes from which you can derive new classes. For example, when you create a new Windows application in Visual Studio .NET, the main form that is created by the integrated development environment is a new class that is derived from the **System.Windows.Forms** class.

Syntax

To create a derived class, use the following syntax:

[attributes] [access-modifiers] class identifiers [:base-class] {class-body}

Example

The previous example requires you to create objects that represent types of animals. To use inheritance, first create a base class named **Animal**, and then include any methods that are common to all your derived classes, as shown in the following example:

```
public class Animal {  
    public bool IsSleeping;  
    public void Sleep() { }  
    public void Eat() { }  
}
```

Then, you can create your derived classes. The following code creates the **Antelope**, **Lion**, and **Elephant** classes with animal-specific behavior:

```
public class Animal {  
    public bool IsSleeping;  
    public void Sleep() {  
        Console.WriteLine("Sleeping");  
    }  
    public void Eat() { }  
}  
  
public class Antelope : Animal {  
}  
  
public class Lion : Animal {  
    public void StalkPrey() { }  
}  
  
public class Elephant : Animal {  
    public int CarryCapacity;  
}
```

Note that the **Sleep** and **Eat** methods are defined only once in the **Animal** base class. The derived **Antelope**, **Lion**, and **Elephant** classes each inherit these methods, and they can be invoked as follows:

```
Elephant e = new Elephant();  
e.Sleep();
```

The preceding code produces the following output:

Sleeping

Design considerations

Remember that when you define a new class, you create a new reference type.

Avoid overusing inheritance in your applications. For example, although it is possible to create new versions of Windows user interface components, such as buttons, there is rarely a good reason to do so.

You can inherit from any class that is not sealed, so you can inherit derived classes. The following code declares a **Mammal** class that inherits from the **Animal** class. The code then defines classes for specific animals that inherit the **Mammal** class.

```
public class Animal {
    public bool IsSleeping;
    public void Sleep() {
        Console.WriteLine("Sleeping");
    }
    public void Eat() { }
}

public class Mammal : Animal {
    public MammalGroup PhylogeneticGroup;
}

public class Antelope : Mammal {
}

public class Lion : Mammal {
    public void StalkPrey() { }
}

public class Elephant : Mammal {
    public int CarryCapacity;
}
```

The **MammalGroup** enumeration is defined as follows:

```
public enum MammalGroup {
    Monotremes,
    Multitubercules,
    Marsupials,
    Utherians
}
```

You can also write code that uses the public members of any object in the inherited hierarchy, as shown in the following example:

```
Elephant e = new Elephant();
e.Sleep();
e.PhylogeneticGroup = MammalGroup.Utherians;
```

Protected access modifier

The purpose of the protected access modifier is to limit the scope of the members of the protected class to only that class and those classes that inherit it.

For example, in the following code, the Boolean value **IsSleeping** is declared as protected, so it can be used only by the base **Animal** class and derived classes.

```
public class Animal {  
    protected bool IsSleeping = false;  
    public void Sleep() { }  
    public void Eat() { }  
}
```

How to Call a Base Constructor from a Derived Class

- The **base** keyword is used in derived classes to specify a non-default base class constructor

```
public class Animal {
    public Animal(GenderType gender) {
        // . . .
        Console.WriteLine("Constructing Animal");
    }
}

public class Elephant : Animal {
    public Elephant(GenderType gender): base(gender) {
        //Elephant code
        Console.WriteLine("Constructing Elephant");
    }
}
```

Introduction

When you create an object, the instance constructor is executed. When you create an object from a derived class, the instance constructor for the base class is executed first, and then the instance constructor for the derived class is executed.

Order of execution

Because the derived class uses the base class, the base class must be instantiated before the derived class.

```
public class Animal {
    public Animal() {
        Console.WriteLine("Constructing Animal");
    }
}

public class Elephant : Animal {
    public Elephant() {
        Console.WriteLine("Constructing Elephant");
    }
}
```

When an **Elephant** object is created, as shown in the following code:

```
Elephant e = new Elephant();
```

The following output is produced:

```
Constructing Animal
Constructing Elephant
```

Notice that the constructor for the base class is executed before the constructor for the derived class. The base class of the hierarchy is constructed first, so in a hierarchy that consists of an **Elephant** class that is derived from a **Mammal** class, which in turn is derived from an **Animal** class, the order of constructor execution is **Animal**, followed by **Mammal**, followed by **Elephant**.

Calling a specific constructor

If the base class has a non-default constructor that you want to use, you must use the **base** keyword.

For example, the **Animal** class may allow you to specify the gender of the animal when it is created, as shown in the following code:

```
public enum GenderType {
    Male,
    Female
}

public class Animal {
    public Animal() {
        Console.WriteLine("Constructing Animal");
    }

    public Animal( GenderType gender ) {
        if ( gender == GenderType.Female ) {
            Console.WriteLine("Female ");
        }
        else {
            Console.WriteLine("Male ");
        }
    }
}
```

You use the **base** keyword in the derived class constructor to call the base class constructor with a matching signature.

If your base class does not have a default constructor, you *must* use the **base** keyword to specify which constructor to call when your derived class is instantiated, as shown in the following code:

```
public class Elephant : Animal {
    public Elephant( GenderType gender ) : base( gender ) {
        Console.WriteLine("Elephant");
    }
}
```

You can then create an **Elephant** object as follows:

```
Elephant e = new Elephant(GenderType.Female);
```

The preceding code produces the following output:

```
Female
Elephant
```

This code sample is available on the Student Materials compact disc as BaseConstructor.sln in the folder Samples\Mod04\BaseConstructor.

How to Use a Sealed Class

- You cannot derive from a sealed class
- Prevents the class from being overridden or extended by third parties

```
public sealed class MyClass {  
    // class members  
}
```

Introduction

You cannot inherit from a sealed class. Any attempt to derive a class from a sealed class causes a compile-time error. You can add the **sealed** keyword to any class or method to create a class that cannot be inherited from to prevent the class from being overridden or extended by third parties.

Designing a sealed class

Creating a sealed class is useful when you want to prevent a class or a method from being overridden. For example, you may apply the **sealed** keyword when the class you are writing is crucial to the functionality of your program and any attempts to override it will cause problems. Or, you may also use it to mark certain classes in your program as proprietary to prevent third-party users from extending them.

Example

The .NET Framework class **System.String** is a sealed class because it has a very strict set of conditions under which its internal data structures must operate, and derived classes may break these rules.

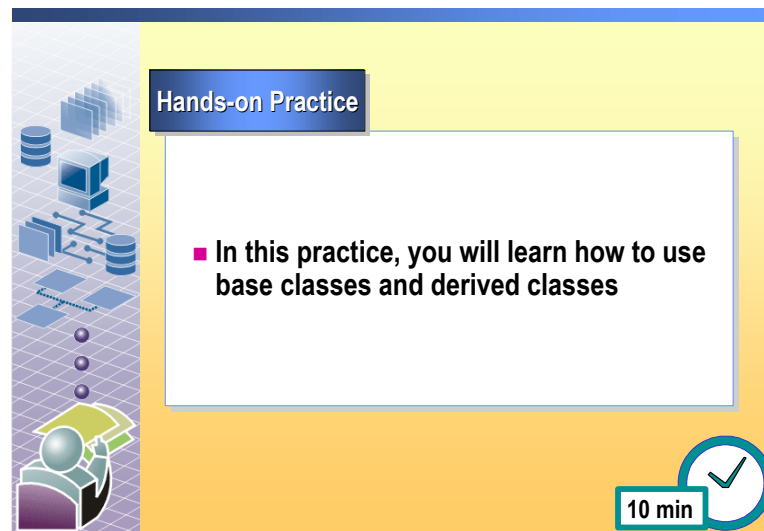
Many of the classes in the **System.Security** and **System.Security.Cryptography** namespaces are sealed to prevent users from overriding their functionality.

Syntax

The syntax for using the **sealed** keyword is as follows:

```
[attributes] [access-modifiers] sealed class identifiers {class-body}  
  
public sealed class Elephant {  
    ...  
}
```

Practice: Creating a Derived Class



In this practice, you will learn how to use base classes and derived classes.

The solution code for this practice is provided in *install_folder*\Practices\Mod04\Inheritance_Solution\Inheritance_Solution.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET and open <i>install_folder</i> \Practices\Mod04\Inheritance\Inheritance.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod04\Inheritance, click Inheritance.sln , and then click Open . d. In Solution Explorer, click Form1.cs , and then press F7 to open the Code Editor, and then review the provided code.
2. Examine the Task List.	a. On the View menu, point to Show Tasks , and then click All .
3. Write a Dolphin class that derives from the Animal class.	a. In the Task List, double-click TODO 1: Create a Dolphin class, derived from Animal . b. Write a class named Dolphin that is derived from the Animal class. You do not need to write any methods or properties in your Dolphin class.

Tasks	Detailed steps
4. Create a Dolphin object.	a. In the Task List, double-click TODO 2: Create a Dolphin object . b. Create an instance of the Dolphin class.
5. Test your code.	a. Build and run your application by pressing F5. b. In your application window, click Run . A message appears informing you that an animal object has been created. Note that this method is defined in the Animal class.
6. Call the Sleep method on the Dolphin object, and then test your code.	a. In the Task List, double-click TODO 3: Call the Sleep method on the dolphin object . b. Add code that calls the Sleep method of the Dolphin object that you created in step 4. c. Test your code by pressing F5.
7. Save your application, and then quit Visual Studio .NET.	a. Save your application. b. Quit Visual Studio .NET.

Lesson: Using Polymorphism

- What Is Polymorphism?
- How to Write Virtual Methods
- How to Use Base Class Members from a Derived Class
- What Are Abstract Methods and Classes?

Introduction

This lesson explains how to use polymorphism in a C# application.

Lesson objectives

After completing this lesson, you will be able to:

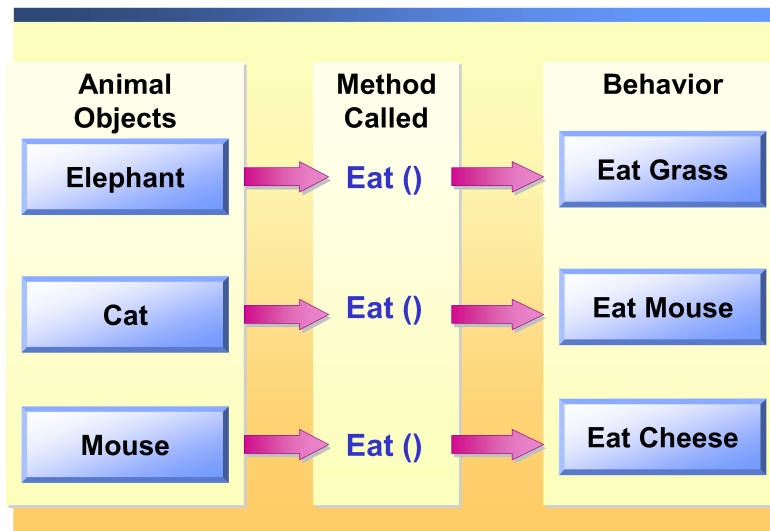
- Implement polymorphism by using virtual methods in base classes.
- Create and use abstract classes.

Lesson agenda

This lesson includes the following topics and activity:

- What Is Polymorphism?
- How to Write Virtual Methods
- How to Use Base Class Members from a Derived Class
- What Are Abstract Methods and Classes
- Practice: Using Polymorphism

What Is Polymorphism?



Introduction

Classes can inherit functionality from a base class, and a derived class can provide new functionality when you change the details of an inherited behavior. In fact, the purpose of creating a derived class is to extend the functionality of the base class and provide multiple ways to accomplish a task.

Definition

Polymorphism is an object-oriented concept that enables you to treat your derived classes in a similar manner, even though they are different. When you create derived classes, you provide more specialized functionality; polymorphism enables you to treat these new objects in a general way.

Example

You are writing an application for a zookeeper to use to manage feeding of the animals. First, you design a base class named **Animal**. In that base class, you include a method named **Eat** that determines how much food the animal will receive. Then, you create the derived classes, such as one for an elephant and one for a cat. Because the dietary requirements of these two types of animals are different, you change the method in each derived class to apply a different set of rules. For example, the **Eat** method for the elephant may calculate a quantity of grass, whereas the **Eat** method for the cat may calculate a quantity of protein.

Although the **Eat** method performs differently in each derived class, polymorphism enables you to simply call **Eat()** on an instance of either derived class without knowing anything about the differences in dietary requirements of these animals.

How to Write Virtual Methods

```
public class Animal {  
    public virtual void Eat() {  
        Console.WriteLine("Eat something");  
    }  
}  
public class Cat : Animal {  
    public override void Eat() {  
        Console.WriteLine("Eat small animals");  
    }  
}
```

Introduction

When you create a method in a base class that you expect to be altered in the derived classes, define the method in your base class as a **virtual** method.

Definition

A **virtual** method is one whose implementation can be replaced by a method in a derived class.

Using the virtual keyword

To define a method in your base class as a **virtual** method, use the keyword **virtual** as shown in the following code:

```
public class Animal {  
    public virtual void Eat() {  
        Console.WriteLine("Eat something");  
    }  
}
```

In the preceding code, any class that is derived from **Animal** can implement a new **Eat** method.

Using the override keyword

To indicate that a method in a derived class is overriding the base class method, you use the **override** keyword, as shown in the following code:

```
public class Cat : Animal {  
    public override void Eat() {  
        Console.WriteLine("Eat small animals");  
    }  
}
```

When you override a virtual method, the overriding method must have the same signature as the virtual method.

Calling virtual methods

You can use polymorphism to treat derived classes in a generalized manner. To achieve this, you can treat derived objects as though they are of their base class type. In the following example, when you use an **Animal** object, you can call the **Eat** method, and the appropriate method for the object will be called.

For example, in the following code, the **FeedingTime** method takes a parameter of **Animal** and calls the **Eat** method to produce the action that is appropriate for that particular animal. The **FeedingTime** method uses polymorphism to invoke the desired action. The **FeedingTime** method can accept as a parameter any object that is derived from **Animal**.

```
public void FeedingTime( Animal someCreature ) {  
    if ( someCreature.IsHungry ) {  
        someCreature.Eat();  
    }  
}
```

The **FeedingTime** method can be passed a **Cat** object, for example:

```
Cat myCat = new Cat();  
FeedingTime(myCat);
```

When you invoke a virtual method in your application, a run-time decision determines which method is actually invoked. The most derived implementation of the method is invoked. The most derived method is the original virtual method if no overriding method is provided (**Animal.Eat** in this example); otherwise, the most derived method is the overriding method in the object for which the method has been invoked (**Cat.Eat** in this example).

```
using System;
namespace LearningCSharp {
    public class Animal { // base class
        public Animal() { }
        public void Sleep() { }
        public bool IsHungry = true;
        public virtual void Eat() {
            Console.WriteLine("Eat something");
        }
    }

    public class Elephant : Animal {
        public int CarryCapacity;
        public override void Eat() {
            Console.WriteLine("Eat grass");
        }
    }

    public class Mouse : Animal {
        public override void Eat() {
            Console.WriteLine("Eat cheese");
        }
    }

    public class Cat : Animal {
        public void StalkPrey() { }
        public override void Eat() {
            Console.WriteLine("Eat mouse");
        }
    }

    public class WildLife {
        public WildLife() {
            Elephant myElephant = new Elephant();
            Mouse myMouse = new Mouse();
            Cat myCat = new Cat();

            FeedingTime(myElephant);
            FeedingTime(myMouse);
            FeedingTime(myCat);
        }

        public void FeedingTime( Animal someCreature ) {
            //Notice use of polymorphism here
            if ( someCreature.IsHungry ) {
                someCreature.Eat();
            }
        }

        static void Main(string[] args) {
            WildLife w = new WildLife();
        }
    }
}
```

When run, the preceding code produces the following result:

```
Eat grass  
Eat cheese  
Eat mouse
```

This code sample is provided on the Student Materials compact disc in Samples\Mod04\Polymorphism\Polymorphism.sln.

How to Use Base Class Members from a Derived Class

- The **base** keyword is used to call a method in the base class from a derived class

```
public class Cat : Animal {  
    public override void Eat() {  
        base.Eat();  
        Console.WriteLine("Eat small animals");  
    }  
}
```

Introduction

The **base** keyword is used in derived classes to access members of the base class.

Example

To call the **Animal.Eat** method from the **Cat.Eat** method, you specify **base.Eat()** in the **Cat** object, as shown in the following code:

```
public class Animal {  
    public virtual void Eat() {  
        Console.WriteLine("Eat something");  
    }  
}  
  
public class Cat : Animal {  
    public void StalkPrey() { }  
    public override void Eat() {  
        base.Eat();  
        Console.WriteLine("Eat small animals");  
    }  
}
```

The following code creates a **Cat** object and calls the **Eat** method:

```
Cat c = new Cat();  
c.Eat();
```

The following output is produced:

```
Eat something  
Eat small animals
```

You will find it useful to call base methods from your derived class when you want to extend the functionality of a method in a base class. You can call the base method from your overriding method, reuse the base method code, and then provide your own extra functionality.

What Are Abstract Methods and Classes?

- **An abstract class is a generic base class**

- Contains an abstract method that must be implemented by a derived class

- **An abstract method has no implementation in the base class**

```
public abstract class Animal {  
    public abstract void Eat();  
    public abstract Group PhylogeneticGroup { get; }  
}
```

- **Can contain non-abstract members**

Introduction

Often, it is useful to create a class that contains methods that must be implemented by all derived classes but not by the base class itself.

Definitions

An *abstract method* is an empty method—one that has no implementation. Instead, derived classes are required to provide an implementation.

An *abstract class* is a class that can contain abstract members, although it is not required to do so. Any class that contains abstract members must be abstract. An abstract class can also contain non-abstract members.

Because the purpose of an abstract class is to act as a base class, it is not possible to instantiate an abstract class directly, nor can an abstract class be sealed.

Syntax

The syntax for creating an abstract method is to use the **abstract** modifier with the name of the method and the parameters, followed by a semicolon instead of a statement block.

```
[access-modifiers] abstract return-type method-name ([parameters]) ;
```

Example

The following example shows how to create an abstract class **Animal** class with an abstract **Eat** method:

```
public abstract class Animal {  
    public abstract void Eat();  
}
```

Benefits

The benefit of creating abstract methods is that it enables you to add methods to your base class that subsequently must be implemented by all derived classes, but the implementation details for these methods do not have to be defined in the base class.

The **Eat** method in the preceding examples is a good use of an abstract method because although all animals eat, the implementation details of **Eat** vary enough between animals that it is not useful to provide a default implementation.

Override

When a derived class inherits an abstract method from an abstract class, it must override the abstract methods. This requirement is enforced at compile time.

The following example shows how a **Mouse** class, which is derived from **Animal**, uses the *override* keyword to implement the **Eat** method:

```
public class Mouse : Animal {  
    public override void Eat() {  
        Console.WriteLine("Eat cheese");  
    }  
}
```

When you call the **Eat** method on the **Mouse** object, the following output is produced:

Eat cheese

Abstract class with virtual method

You can also create an abstract class that contains virtual methods, as shown in the following example:

```
public abstract class Animal {  
    public virtual void Sleep() {  
        Console.WriteLine("Sleeping");  
    }  
    public abstract void Eat();  
}
```

In this case, a derived class does not have to provide an implementation of the **Sleep** method because **Sleep** is defined as virtual. Therefore, if you have a generic method that is common to all derived classes, and you want to force each derived class to implement the method, you must define the method as abstract in the base class.

Example

Because some animals sleep in different ways—some sleep while standing, others sleep for long periods, and so on—you decide that sleep is a good candidate for an abstract method.

To do this, you change the definition of **Sleep** to abstract and remove the implementation, as shown in the following code:

```
public abstract class Animal {  
    public abstract void Sleep();  
    public abstract void Eat();  
}
```

When the application is compiled, because the mouse object does not implement the **Sleep** method, you receive the following error:

LearningCSharp.Mouse' does not implement inherited abstract member 'LearningCSharp.Animal.Sleep()'

Note **LearningCSharp** is the namespace that contains the **Animal** classes.

By changing **Sleep** to an abstract method, you force the derived classes to implement their own version of the method.

The following code shows an implementation of both the **Eat** and the **Sleep** methods in the **Mouse** class:

```
public class Mouse : Animal {
    public override void Eat() {
        Console.WriteLine("Eat cheese");
    }
    public override void Sleep() {
        Console.WriteLine("Mouse sleeping");
    }
}
```

Abstract properties

Properties may also be declared as abstract.

To declare an abstract property, specify the property name and the accessors that the derived property should implement.

Example 1

The following example shows how to create an animal class and declare an abstract property named **PhylogenicGroup** (**Group** is an enumeration that contains phylogenic group names):

```
public abstract class Animal {
    public abstract Group PhylogenicGroup {get; set;}
}
```

To make the property read-only or write-only, you can remove the corresponding accessor.

Example 2

The following code produces a compilation error because the code attempts to set a value while the property is declared as read-only:

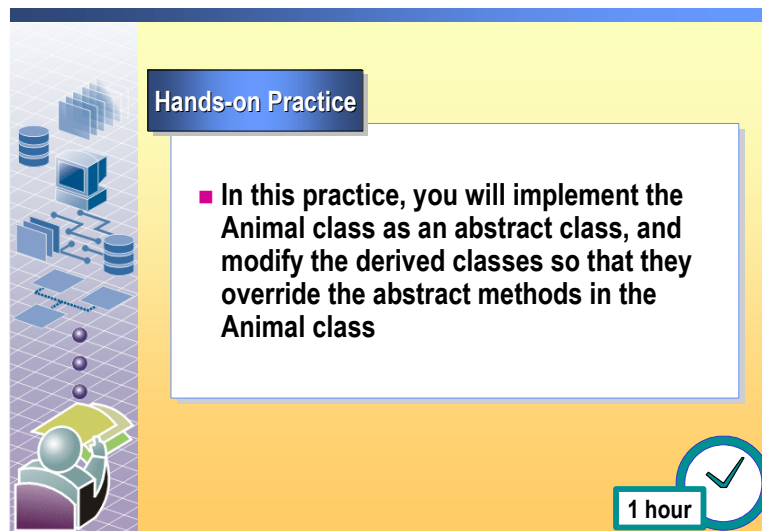
```
public abstract class Animal {
    public abstract Group PhylogenicGroup {get;}
}

public class Cat : Animal {
    public override Group PhylogenicGroup {
        get {
            return Group.Utherians;
        }
    }
}

...

Cat c = new Cat();
//The following line causes a compilation error
c.PhylogenicGroup = Group.Utherians;
```

Practice: Using Polymorphism



Hands-on Practice


- In this practice, you will implement the **Animal** class as an abstract class, and modify the derived classes so that they override the abstract methods in the **Animal** class

1 hour

In this practice, you will implement the **Animal** class as an abstract class, and modify the derived classes so that they override the virtual functions in the **Animal** class.

The solution code for this practice is provided in *install_folder*\Practices\Mod04\Abstract_Solution\Abstract.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod04\Abstract\Abstract.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod04\Abstract, click Abstract.sln , and then click Open .
2. Examine the tasks, and double-click Next button code .	a. In Solution Explorer, click Form1.cs , press F7 to open the Code Editor, and then review the provided code. b. On the View menu, point to Show Tasks , and then click All .

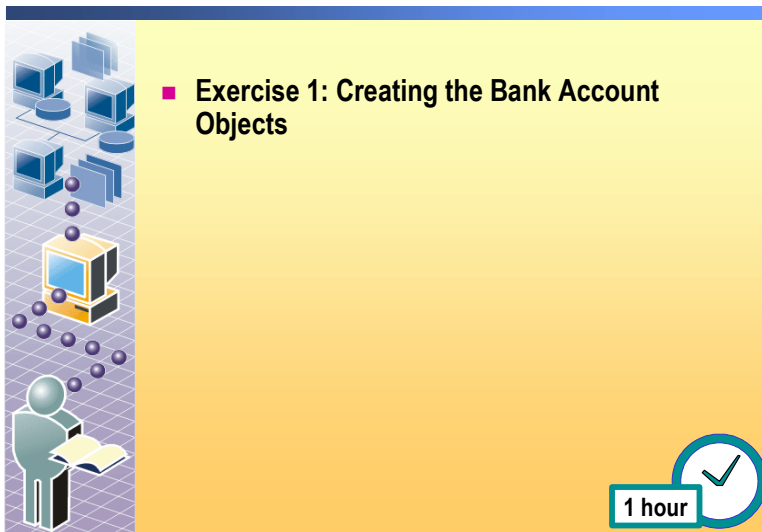
Tasks	Detailed steps
3. Change the methods in the Animal class so that they are abstract methods.	<ul style="list-style-type: none">a. In the Task List, double-click TODO 1: Make the methods in the class abstract.b. Change the methods Sleep and Eat to abstract methods.c. Test your code by pressing F5.  <i>The compiler should list errors indicating that the Lion and Antelope classes do not implement the abstract members of Animal.</i>
4. Change the derived classes to correctly inherit from the Animal class.	<ul style="list-style-type: none">a. Double-click TODO 2: Change Lion and Antelope to work with the abstract Animal class.b. Change the Lion and Antelope classes so that they correctly inherit from the Animal class.c. Test your code by pressing F5.
5. Save your application, and then quit Visual Studio .NET.	<ul style="list-style-type: none">a. Save your application.b. Quit Visual Studio .NET.

Review

- Designing Objects
- Using Inheritance
- Using Polymorphism

1. What keyword do you add to your class definition if you do not want other classes to inherit from it?
2. Should a derived class be more specialized or more generalized than its base class?
3. What are some of the benefits of object-oriented programming?

Lab 4.1: Creating Classes in C#



Objectives

After completing this lab, you will be able to:

- Use properties to provide access to data in a class.
- Create base classes and derive classes from them.
- Create abstract classes and derive classes from them.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Prerequisites

Before working on this lab, you must have:

- Knowledge of how to create derived classes.
- Knowledge of how to write properties.

Scenario

You are a programmer at a bank and have been asked to define an object hierarchy for the types of bank accounts that customers can open. These accounts are:

- Checking account
- Savings account

Note If you have a working solution from module 3, then you can use it in this lab, rather than the starter code.

Checking account

A checking account has the following characteristics:

- The account holder's name can be assigned only when the account is created.
- The opening balance must be specified when the account is created.
- The account number must be assigned when the account is created.
 - Checking account numbers range from 100000 to 499999, and every checking account must have a unique account number. You do not need to check the upper limit of the account number in this lab.

A checking account holder can:

- Order a checkbook.
- Check the account balance.
- Add money to the checking account.
- Withdraw money if the account has sufficient funds.

Savings account

A savings account has the following characteristics:

- The account holder's name that can be assigned only when the account is created.

Saving account numbers range from 500000 to 999999. You do not need to check the upper limit of the account number in this lab.

- The account earns interest.

The interest rate depends on the account balance. If the balance is above 1000, the rate is 6%; otherwise, it is 3%.

A savings account holder can:

- Check the account balance.
- Add money to the account.
- Withdraw money if the account has sufficient balance.

Your bank is likely to add more account types in the future, so it is important to reuse as much code as possible, while ensuring that the different account objects implement a standard set of features.

**Estimated time to
complete this lab:
60 minutes**

Exercise 0

Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
<ul style="list-style-type: none">Log on to Windows as Student with a password of P@ssw0rd.	<ul style="list-style-type: none">Log on to Windows with the following account.<ul style="list-style-type: none">User name: StudentPassword: P@ssw0rd <p>Note that the 0 in the password is a zero.</p>

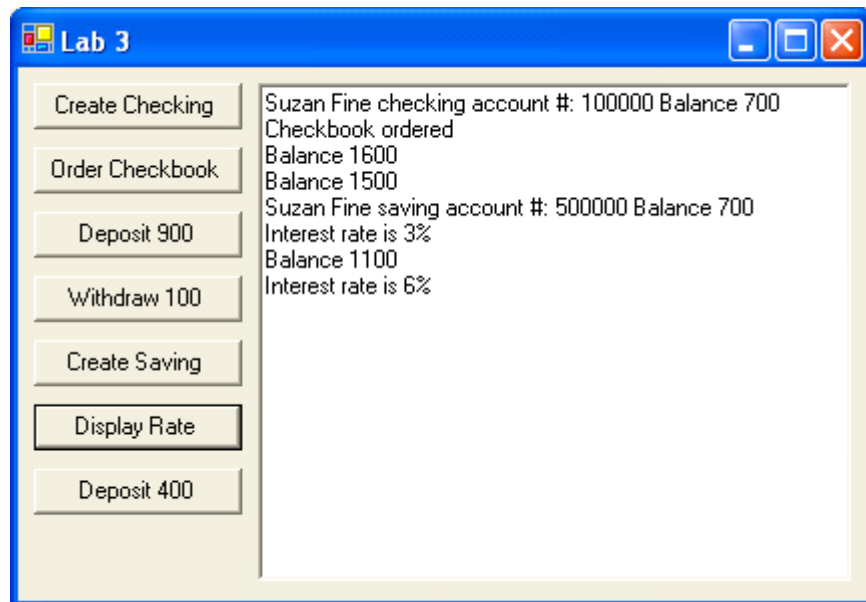
Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

Exercise 1

Creating the Bank Account Objects

In this exercise, you will write the objects that represent the bank account classes that are outlined in the scenario.

A sample application is shown in the following illustration:



The solution code for this lab is provided in *install_folder*\Labfiles\Lab04_1\Exercise1\Solution_Code\Bank.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Labfiles\Lab04_1\Exercise1\Bank.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page , click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Labfiles\Lab04_1\Exercise1, click Bank.sln , and then click Open . Form1.cs provides the user interface. BankAccount.cs is provided as a place for you to implement the bank account class or classes.
2. In Solution Explorer, locate the BankAccount.cs file.	■ In Solution Explorer, click the C# file BankAccount.cs , and then press F7 to open the Code Editor. This file provides a sample implementation of the checking account and saving account classes.

Tasks	Detailed steps
<p>3. Write a base class called BankAccount and modify the CheckingAccount and SavingsAccount classes so that they inherit BankAccount.</p>	<p>The sample solution provides two classes, CheckingAccount and SavingsAccount. These classes duplicate some code, and allow uncontrolled access to information. In this step you will create a base class, and in the next step you will implement the methods and properties of the base class.</p> <ul style="list-style-type: none"> a. At the top of the BankAccount.cs file, locate TODO: Implement BankAccount class here, and declare a base class for CheckingAccount and SavingsAccount. b. Modify CheckingAccount and SavingsAccount so that they inherit BankAccount.
<p>4. Decide which properties should be provided by the base class, add them to the base class, and modify the CheckingAccount and SavingsAccount classes to remove public variables.</p>	<ul style="list-style-type: none"> a. Examine the public variables in the CheckingAccount and SavingsAccount classes, and change them to properties, moving them to the base class where appropriate. b. Use Pascal case for your properties, and change the calling code in Form1.cs appropriately. <ul style="list-style-type: none"> ■ Note that you can use the protected keyword to limit access to a variable to derived classes only. ■ Some properties should be read only: you can make a property read only by omitting the set accessor.
<p>5. Decide which methods should be provided by the base class, add them to the base class, and modify the CheckingAccount and SavingsAccount classes to reuse code.</p>	<ul style="list-style-type: none"> a. Examine the methods in CheckingAccount and SavingsAccount to determine which should be provided by the BankAccount class, and then change the code so that they are provided by BankAccount. b. Use Pascal case for your properties, and change the calling code in Form1.cs appropriately. <ul style="list-style-type: none"> ■ Note that you can use the protected keyword to limit access to a variable to derived classes only.
<p>6. Provide a BankAccount constructor that forces the derived classes to specify the account holder name and initial balance.</p>	<ul style="list-style-type: none"> a. Examine the constructors in CheckingAccount and SavingsAccount to determine what features should be provided by a BankAccount constructor. b. Add a BankAccount constructor that requires the account holder's name and an initial balance. c. Change the CheckingAccount and SavingsAccount constructors so that they pass the correct values to the BankAccount constructor.

Tasks	Detailed steps
7. If any existing methods would be more appropriately implemented as properties, then change them to properties.	<ul style="list-style-type: none">■ Examine the methods in CheckingAccount and SavingsAccount to determine if any should be properties, and change any that you identify.
8. Test your code by running the application, and then clicking the buttons in sequence from top to bottom.	<ul style="list-style-type: none">a. Press F5 to compile and run your application.b. Click each button in sequence and ensure that your application produces the expected output:<ul style="list-style-type: none">• Create Checking: a message stating that Suzan Fine has created a checking account with a balance of 700.• Order Checkbook: a message stating that a checkbook has been ordered.• Deposit 900: a balance of 1600.• Withdraw 100: a balance of 1500.• Create Saving: a message stating that Suzan Fine has created a saving account with a balance of 700.• Display Rate: an interest rate of 3%.• Deposit 400: a balance of 1100.c. Click Display Rate once more to check that the interest rate has increased to 6%.
9. Save your application and quit Visual Studio .NET.	<ul style="list-style-type: none">a. Save your application.b. Quit Visual Studio .NET.