Chapter 14

Virtual Methods and Polymorphism

Virtual Methods and Polymorphism

Objectives

After completing this unit you will be able to:

- **Use polymorphism to simplify your code and enhance maintainability.**
- **Use static and dynamic binding as appropriate in your code.**
- **Describe and use the C# features that support polymorphism.**

Introduction to Polymorphism

- **The fundamentals of inheritance we discussed in the last chapter are extremely important, but they constitute only part of the story of inheritance.**
- **The other part of the story involves the mechanism of *virtual methods*, which are not bound to an object at compile time but are bound dynamically at runtime.**
- **This dynamic behavior enables *polymorphic* code, which is general code that applies to classes in a hierarchy, and the specific class that determines the behavior is determined at runtime.**
- **Polymorphic code can simplify program development and maintenance.**
- **C# provides keywords *virtual* and *override* that precisely specify in base and derived classes, respectively, that the programmer is depending on runtime, dynamic binding.**
- **Specifying polymorphic behavior eliminates the *fragile base class problem*, which can result in unexpected behavior in a program when a base class in a library is modified, but the program itself is unchanged.**

Abstract and Sealed Classes

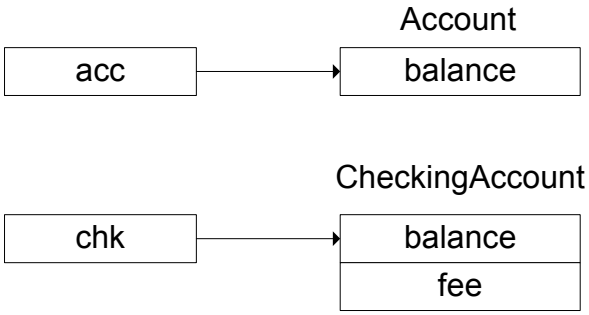
- **Sometimes in an inheritance hierarchy, the base class is never intended to be instantiated.**
- **Such a base class is said to be *abstract*, and *must* be derived from in order to be useful.**
- **At the opposite end of the spectrum, a class is said to be *sealed* if derivation is not allowed.**
- **A class hierarchy can be used to implement heterogeneous collections that can be treated polymorphically. We illustrate the topics of this chapter with the bank case study.**

Virtual Methods And Dynamic Binding

- **In C# the normal way methods are tied to classes is through *static binding*.**
  - That means the type of an object reference is used at compile time to determine the class whose method is called.
- **The program *StaticAccount* illustrates static binding, using a simplified version of our *Account* class and a derived *CheckingAccount* class.**
  - Note the use of the **new** keyword in the **Show** methods of the derived class to specify method hiding.
- **In this program *acc* is an object reference of type *Account*, and calling *Show* through this object reference will always result in *Account.Show* being called, no matter what kind of object acc may actually be referring to.**
  - Notice that the second time we call **Show** through **acc** we are still getting **Account**.

Type Conversions in Inheritance

- **This program also illustrates another feature of inheritance, type conversions.**
  - After the objects **acc** and **chk** have been instantiated, the object references will be referring to different objects, one of type Account and the other of type **CheckingAccount**, as illustrated in the figure.
  - Note that the **CheckingAccount** object has an additional field, **fee**.

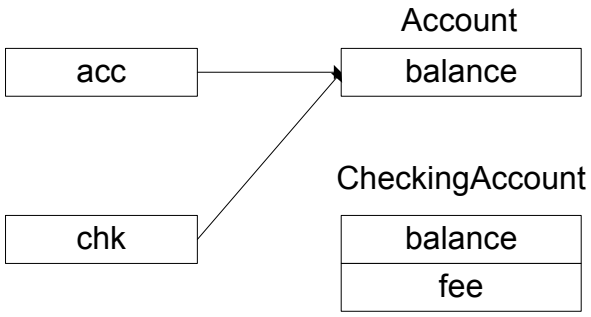


  - The test program tries two type conversions:

```
//chk = acc;      // illegal
acc = chk;
```

Converting Down the Hierarchy

- **The first assignment is illegal (as you can verify by uncommenting and trying to compile).**
  - Suppose the assignment would be allowed.
  - Then you would have an object reference of type **CheckingAccount** referring to an **Account** object, as illustrated in the figure.



  - If the conversion "down the hierarchy" (from a base class to a derived class) were allowed, the program would be open to a bad failure at runtime if code tried to access a nonexistent member, such as **chk** accessing the member **fee**.
- **The program *BadConversion* illustrates this behavior.**
  - The class definition is the same.
  - In the test program an explicit cast operation is performed.
  - There will then be no error messages at compile time, but there will be a runtime failure.

Converting Up the Hierarchy

- **The opposite assignment:**

```
acc = chk;
```

- **is perfectly legal. We are converting "up the hierarchy."**
- **This is okay because of the IS-A relationship of inheritance.**
  - A checking account "is" an account. It is a special kind of account.
  - Everything that applies to an account also applies to a checking account.
  - There can be no "extra field" in the **Account** class that is not also present in the **CheckingAccount** class.

Virtual Methods

- **In C# you can make a small change to specify that a method in C# will be bound *dynamically*.**
- **That means it will be determined at runtime which class's method will be called.**
- **The program *VirtualAccount* illustrates this behavior.**
  - The file **VirtualAccount.cs** contains class definitions for a base class and a derived class, as before.
  - But this time the **Show** method is declared as **virtual** in the base class.
  - In the derived class the **Show** method is declared **override** (in place of **new** that we used before with method hiding).
  - Now the **Show** method in the derived class does not hide the base class method, but *overrides* it.
  - We use the same test program. We just dropped the commented out illegal assignment and changed the comment on invoking the second **acc.Show**.

Virtual Method Example

```
// VirtualAccount.cs

using System;

public class Account
{
   public int balance = 100;
   virtual public void Show()
   {
      Console.WriteLine("I am an Account");
   }
}

public class CheckingAccount : Account
{
   public int fee = 5;
   override public void Show()
   {
      Console.WriteLine(
         "I am a CheckingAccount, fee = {0}", fee);
   }
}

// TestVirtualAccount.cs

public class TestAccount
{
   public static void Main(string[] args)
   {
      Account acc = new Account();
      CheckingAccount chk = new CheckingAccount();
      acc.Show();
      chk.Show();
      acc = chk;
      acc.Show();       // now CheckingAccount.Show
   }
}
```

Virtual Method Cost

- **Virtual method invocation is slightly less efficient than calling an ordinary nonvirtual method.**
  - With a virtual method call, there is some overhead at runtime associated with determining which class's method will be invoked.
  - C# allows you to specify in a base class whether you want the flexibility of a virtual method or the slightly greater efficiency of a nonvirtual method.
  - You simply decide whether or not to use the keyword **virtual**. (In some languages all methods are virtual, and you don't have this choice.)

Method Overriding

- **The *override* keyword in C# is very useful for making programs clearer.**
  - In some languages, such as C++, there is no special notation for overriding a method in a derived class.
  - You simply declare a method with the same signature as a method in the base class.
  - If the base class method is virtual, the behavior is to override.
  - If the base class method is not virtual, the behavior is to hide.
  - In C# this behavior is made explicit.

The Fragile Base Class Problem

- **There is a subtle pitfall in object-oriented programming: the fragile base class problem.**
- **Suppose there is no *override* keyword and you have a method in a class that does not hide or override any method in your base classes.**
- **But assume that you are using a third-party class library, and your class is ultimately derived from a class in this library.**
  - Now suppose a new version of the class library comes out, and the base class you are deriving from has a new virtual method whose signature happens to match one of the methods in your class.
  - Now you can be in trouble!
  - Code in the combined system that consists of your classes and the class library may now behave in unexpected ways.
- **Code that was "expected" to call the new method in the class library—or in code in a derived class that deliberately overrides this method—may now call your method that has nothing whatever to do with the method in the class library.**
  - This situation is rare, but if it occurs it can be extremely vicious.

overrride Keyword

- **Fortunately, C# helps you avoid such situations by requiring you to use the *override* keyword if you are indeed going to perform an override.**
  - If you do not specify either **override** or **new**, you will get a compiler error or warning if a method in your derived class has the same signature as a method in a base class.
  - Thus, if you build against a new version of the class library that introduces an accidental signature match with one of your methods, you will get warned by the compiler.

Polymorphism

- **The machinery of virtual functions makes it easy to write polymorphic code in C#.**
  - As an example of polymorphic code, consider our bank account case study.
  - Imagine a large system with a great many different kinds of accounts.
  - How will you write and maintain code that deals with all these different account types?

Polymorphism Using "Type Tags"

- **A traditional approach is to have a "type field" in an account structure.**
  - Then code that manipulates an account can key off this type field to determine the correct processing to perform, perhaps using a **switch** statement.
  - Although straightforward, this approach can be quite tedious and error-prone.
- **What happens if you have to add a new derived type to a legacy program?**
- **Introducing a new kind of account can require substantial maintenance.**

Polymorphism Using Virtual

- **Polymorphism enables a cleaner solution.**

1. Organize the different kinds of accounts in a class hierarchy, and structure your program so that you write general purpose methods that act upon an object reference whose type is that of the base class.
2. In your code, call virtual methods of the base class.
3. The call will be automatically dispatched to the appropriate class, depending on what kind of account is actually being referenced.

Polymorphism Example

- **The program *PolyAccount* provides an illustration (a more full-blown illustration will be presented later in the chapter with Step 3 of the case study).**
  - This version of the **Account** hierarchy is similar to Step 2 of the case study presented in the previous chapter, only now there are virtual methods in the base class, and methods in the derived class override them.
  - Methods in the derived classes override the virtual methods in the base class.
- **The payoff comes in the client program, which can now call the virtual methods polymorphically.**
  - In this program there is a single **Account** object reference **acc**.
  - At different places in the program, it is assigned to different kinds of account objects: **Account**, **CheckingAccount**, and **SavingsAccount**.
- **The code that gets invoked is determined at runtime based upon the type of account being referenced. This is polymorphism.**
  - In particular, notice that we need only one helper method, **ShowAccount**.

Polymorphism Example (Cont'd)

```
// PolyAccount.cs

public class Account
{
   private int id;
   protected decimal balance;
   private string owner;
   protected int numXact = 0;
                           // number of transactions
   public Account(decimal balance,
      string owner, int id)
   {
      this.balance = balance;
      this.owner = owner;
      this.id = id;
   }
   virtual public void Deposit(decimal amount)
   {
      balance += amount;
      numXact++;
   }
   virtual public void Withdraw(decimal amount)
   {
      balance -= amount;
      numXact++;
   }
   public decimal Balance
   {
      get
      {
         return balance;
      }
   }
```

Polymorhphism Example (Cont'd)

```
   public int Id
   {
      get
      {
         return id;
      }
   }
   public string Owner
   {
      get
      {
         return owner;
      }
      set
      {
         owner = value;
      }
   }
   public int Transactions
   {
      get
      {
         return numXact;
      }
   }
   virtual public string GetStatement()
   {
      string s = "Statement for " + this.Owner +
         " id = " + Id + "\n" + this.Transactions +
         " transactions, balance = " + balance;
      return s;
   }
   virtual public void Post()
   {
   }
}
```

Polymorphism Example (Cont'd)

```
// CheckingAccount.cs

using System;

public class CheckingAccount : Account
{
   private decimal fee = 5.00m;
   private const int FREEXACT = 2;
   public CheckingAccount(decimal balance,
      string owner, int id)
      : base(balance, owner, id)
   {
   }
   public decimal Fee
   {
      get
      {
         if (numXact > FREEXACT)
            return fee;
         else
            return 0.00m;
      }
   }
   override public string GetStatement()
   {
      string s = base.GetStatement();
      s += ", fee = " + Fee;
      return s;
   }
   override public void Post()
   {
      balance -= Fee;
      numXact = 0;
   }
}
```

Abstract Classes

- **Sometimes it does not make sense to instantiate a base class.**
- **Instead, the base class is used to define a standard template to be followed by the various derived classes.**
- **Such a base class is said to be *abstract*, and it cannot be instantiated.**
- **An abstract class may have abstract methods, which are not implemented in the class but only in derived classes.**
- **The purpose of an abstract method is to provide a template for polymorphism.**
  - The method is called through an object reference to the abstract class, but at runtime the object reference will actually be referring to one of the concrete derived classes.
- **The *Account* class in Step 3 of the case study, which we will examine later in the chapter, provides an example of an abstract class.**

Keyword: abstract

- **In C# you can designate a base class as abstract by using the keyword *abstract*.**
- **The compiler will then flag an error if you try to instantiate the class.**
- **The keyword *abstract* is also used to declare abstract methods. In place of curly brackets and implementation code, you simply provide a semicolon after the declaration of the abstract method.**
- **Example:**

```
// Account.cs - Step 3

abstract public class Account
{
   private int id;
   protected decimal balance;
   private string owner;
   protected int numXact = 0;
                          // number of transactions
   ...
   abstract public void Post();
   abstract public string Prompt {get;}
}
```

Sealed Classes

- **At the opposite end of the spectrum from abstract classes are *sealed* classes.**
- **While you *must* derive from an abstract class, you *cannot* derive from a sealed class.**
  - A sealed class provides functionality that you can use as is, but you cannot derive from the class and hide or override some of the methods.
  - An example in the .NET Framework class library of a sealed class is **System.String**.
- **Marking a class as sealed protects against unwarranted class derivations.**
  - It can also make the code a little more efficient, because any virtual functions in the sealed class are automatically treated by the compiler as nonvirtual.
- **In C# you use the *sealed* keyword to mark a class as sealed.**

Heterogeneous Collections

- **A class hierarchy can be used to implement heterogeneous collections that can be treated polymorphically.**
  - For example, you can create an array whose type is that of a base class.
  - Then you can store within this array object references whose type is the base class, but which actually may refer to instances of various derived classes in the hierarchy.
  - You may then iterate through the array and call a virtual method.
  - The appropriate method will be called for each object in the array.
- **The program *HeterogeneousAccount* illustrates a heterogeneous array of three accounts, which are a mixture of checking and savings accounts.**
  - The virtual property **Prompt** returns a prompt string, which is "C: " for a checking account and "S: " for a savings account.

Heterogeneous Collections Example

```
// HeterogeneousAccount.cs

using System;

public class TestAccount
{
   public static void Main(string[] args)
   {
      Account[] list = new Account[3];
      list[0] = new CheckingAccount(100, "Bob", 1);
      list[1] = new SavingsAccount(200, "Mary", 2);
      list[2] = new CheckingAccount(300,
         "Charlie", 3);
      foreach (Account acc in list)
         ShowAccount(acc.Prompt, acc);
   }
   private static void ShowAccount(string caption,
      Account acc)
   {
      Console.Write("{0}: ", caption);
      Console.WriteLine(acc.GetStatement());
   }
}
```

- **Here is the output:**

```
C: : Statement for Bob id = 1
0 transactions, balance = 100, fee = 0
S: : Statement for Mary id = 2
0 transactions, balance = 200, interest = 1
C: : Statement for Charlie id = 3
0 transactions, balance = 300, fee = 0
```

Bank Case Study: Step 3

- **Step 3 of the case study, as usual in the *CaseStudy* directory for this chapter, is an extension of Step 1 from Chapter 12.**
- **The big change is that now we support two kinds of accounts, *CheckingAccount* and *SavingsAccount*, which are derived from the abstract base class *Account*.**
  - Each kind of account has its own characteristics. There are now seven classes in all, each in its own file.
- **Please look at the online files for the code listings.**

Case Study Classes

- ***InputWrapper*. This class simplifies prompting for input and reading in the data. It is identical to the class by this name that we have used previously.**
- ***Account*. This abstract base class provides a template for accounts, specifying an *Id*, an *Owner*, a *Balance*, and the number of *Transactions*. Operations are *Deposit*, *Withdraw*, and *GetStatement*.**
  - There is an abstract method **Post** and an abstract property **Prompt**. There is also a virtual method **MonthEnd**.
- ***CheckingAccount*. This class provides a *Fee* property and overrides *GetStatement*, *Post*, and *Prompt*.**
- ***SavingsAccount*. This class provides an *Interest* property, which is based on a minimum balance.**
  - It overrides **Withdraw**, **GetStatement**, **Post**, **MonthEnd**, and **Prompt**.
- ***Bank*. This class represents a bank, which has several accounts.**
  - Methods are provided to add an account, delete an account, and get a list of accounts.

Case Study Classes (Cont'd)

- ***TestBank*. This class provides an interactive test program for exercising the *Bank* class.**
  - Commands are provided to open an account, close an account, show all the accounts, and start an "ATM" to perform transactions on a particular account.
- ***Atm*. This class provides a user interface for the ATM that allows a user to perform transactions on a particular account.**
  - The operations supported are deposit, withdraw, change owner name, and show account information.
  - A special prompt of "C: " or "S: " is shown, depending on whether the account being operated upon is a checking or a savings account.

Run the Case Study

- **At this point you should try to understand in general terms how the case study works, especially the operation of the virtual functions.**
  - Do not be concerned about every nuance of the business rules of our little bank.
- **The case study is elaborated in greater detail at Step 6 in Chapter 17, where we introduce interfaces.**
  - The use of interfaces will help us view the bank at a higher level of abstraction, and in that chapter we provide additional explanation of the case study and more sample runs.

Account

- **The *Account* class is similar to the version of the class used to illustrate polymorphism earlier in the chapter.**
  - The class is now abstract, with the abstract method **Post** and the abstract property **Prompt**.
  - Notice the syntax used in specifying the signature of a property.
  - In this case the property is read-only, so there is only a **get**.
  - There is a nuance introduced here in the form of another virtual method, **MonthEnd**.
  - This virtual method reinitializes an account for the next month.
  - The basic initialization is to set the transaction count to zero. Additional initialization may be done in derived classes (such as the **SavingsAccount** class, discussed later in this chapter).

CheckingAccount, SavingsAccount

- **Step 3 of the *CheckingAccount* class is similar to the class that was shown in the previous section on polymorphism.**
  - The difference is that **Post** method does not set the number of transactions to zero, as that is now the responsibility of the **MonthEnd** method, which is inherited from the **Account** class.
- **The *SavingsAccount* class has an interest rate, which is expressed as a property.**
  - The interest itself is the monthly interest, based on a minimum balance.
  - To compute the minimum balance, the **Withdraw** method is overridden. The class also overrides **Post**, **GetStatement**, **Prompt**, and **MonthEnd**.

Bank and Atm

- **The class *Bank* maintains an array of *Account* objects.**
  - The code is very similar for Step 1 provided in Chapter 12.
- **The big change is that the *AddAccount* method now takes an account type parameter, which is specified as an *enum* data type, *AccountType*.**
  - Based on the type, either a **CheckingAccount** or a **SavingsAccount** is instantiated.
  - Also, the **GetAccounts** method is modified so that the prompt string for the particular kind of account is returned as part of the string describing the account.
- **The *Atm* class is virtually identical to the Step 1 version.**
  - This is the beauty of polymorphism.
- **The *ProcessAccount* method takes an object reference to the** Account **base class, and it calls virtual methods of this class, which get resolved properly at runtime to the appropriate methods of *CheckingAccount* or *SavingsAccount*.**
- **The only enhancement added to the Step 3 version is using the virtual *Prompt* property of *Account* to tailor the prompt according to the type of account being worked on.**

TestBank

- **The *TestBank* class provides a user interface in the *Main* method to open an account, close an account, and show all the accounts.**
  - The command "account" brings up an ATM user interface to allow the user to perform transactions on a particular account.
  - The only difference between Step 3 and Step 1 is in opening an account, where the user is queried for the type of account (checking or savings) to open.
- **When you run the case study, you should find similar behavior to Step 1.**
  - The difference, of course, is that when you open accounts, you specify checking or savings.

Summary

- **In this chapter we discussed the mechanism of *virtual methods*, which are not bound to an object at compile time but are bound dynamically at runtime.**
- **Polymorphic code can simplify program development and maintenance.**
- **C# provides keywords *virtual* and *override* that precisely specify in base and derived classes, respectively, that the programmer is depending on runtime dynamic binding.**
- **A base class which is not intended to be instantiated is said to be *abstract*, and *must* be derived from in order to be useful.**
- **At the opposite end of the spectrum, a class is said to be *sealed* if derivation is not allowed.**
- **A class hierarchy can be used to implement heterogeneous collections that can be treated polymorphically.**
- **Our bank case study provides a nice example of a heterogeneous collection of savings and checking accounts.**