Chapter 13

Inheritance

Inheritance

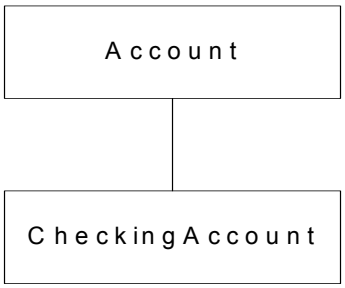
Objectives

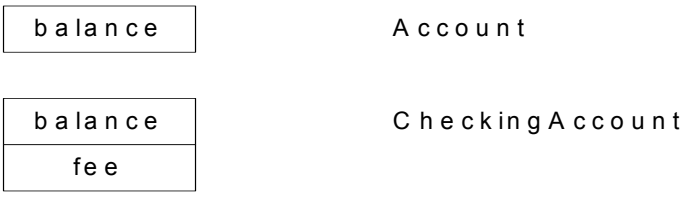
After completing this unit you will be able to:

- **Explain what polymorphism means and how it is implemented in C#.**
- **Use inheritance relationships to produce cleaner designs and re-use code.**
- **Use access qualifiers to encapsulate your implementations and make your code easier to maintain.**

Inheritance Fundamentals

- **Inheritance is a key feature of the object-oriented programming paradigm.**
  - You abstract out common features of your classes and put them in a high-level base class.
  - You can add or change features in more specialized derived classes, which "inherit" the standard behavior from the base class.
- **Consider *Account* as a base class, with derived classes such as *CheckingAccount*.**
  - All accounts share some characteristics, such as a balance.
  - Different kinds of accounts differ in other respects.
  - For example, a checking account has a monthly fee.
  - This figure illustrates the relationship between **Account** and **CheckingAccount**.





Inheritance in C#

- **You implement inheritance in C# by specifying the derived class in the class statement with a colon followed by the base class.**
- **The program *SimpleAccount* illustrates deriving a new class *CheckingAccount* from the class *Account*.**

```
// CheckingAccount.cs

public class CheckingAccount : Account
{
   private decimal fee = 5.00m;
   public void Post()
   {
      balance -= fee;
   }
}
```

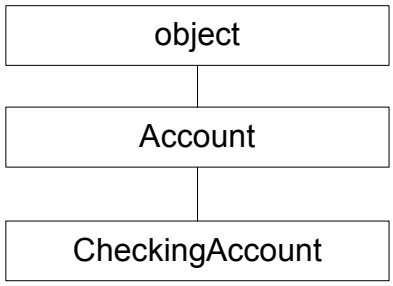
- **The class *CheckingAccount* automatically has all the members that *Account* has, and in addition has the field fee and the method *Post*.**
- **Code for a test program exercises both an *Account* object and a *CheckingAccount* object.**
  - Notice that the **CheckingAccount** object can use the **Deposit** and **Withdraw** methods and the **Balance** property of the base class.
  - No code had to be provided in the derived class for these operations. The derived class can also make use of the new method **Post**.

Single Inheritance

- **It is important to understand that C# supports only *single inheritance*.**
  - In C# a class can derive from only *one* immediate base class.
  - Some languages, such as C++, support multiple inheritance, in which a class can derive from two or more base classes.
  - Multiple inheritance is a powerful feature, but it is also difficult to use correctly.
  - The single inheritance model of C# is simpler.
- **Although multiple inheritance is somewhat problematical, there is great benefit in organizing class behavior into several independent *interfaces*.**
  - The basic idea is to group related methods together into an interface and allow a class to support multiple interfaces.
- **We will discuss interfaces in detail in Chapter 17.**

Root Class – *Object*

- **C# shares a characteristic with some other important object-oriented languages, such as Java and Smalltalk, that have a single inheritance model.**
  - C# has a root class called **object** that is the ultimate base class of every class in C#.
  - You do not need to use the colon notation to show that your class derives from **object**—the compiler does that for you automatically.
  - If your class is derived from another class, it will pick up the methods of its immediate base class plus the methods from classes further up the hierarchy.
  - This figure illustrates the three-level hierarchy of **CheckingAccount** derived from **Account**, which in turn is derived from **object**.



- **The C# keyword object is an alias for *System.Object* in the .NET Framework class library.**
  - All classes in all .NET languages ultimately inherit from **System.Object**.

Access Control

- **C# has two means for controlling accessibility of class members.**
  - Class Accessibility
  - Member Accessibility
- **Access can be controlled at both the class level and the member level.**

Public Class Accessibility

- **An access modifier can be placed in front of the *class* keyword and controls who can get at the class at all.**
  - Access can be further restricted by member accessibility, discussed in the next subsection.
- **There are two class accessibility modifiers, *public* and *internal*.**
- **The most common access modifier of a class is *public*, which makes the class available to everyone.**
- **All of our class examples so far have had *public* accessibility.**
- **Whenever we are implementing a class that anyone can use, we want to make it *public*.**

Internal Class Accessibility

- **The *internal* modifier makes a class available within the current *assembly*, which can be thought of as a logical EXE or DLL.**
- **All of our projects so far have built a single assembly, with both the client test program and the class(es) in this assembly.**
- **That means that if we had used *internal* for the class modifier, the programs would have still worked.**
- **But later, if we put our classes into a DLL and try to access them from a client program in a separate EXE, any *internal* classes would not be accessible.**
  - So using **public** for class accessibility is generally a good idea.
- **A common use of the *internal* modifier is for helper classes that intended for use within the current assembly only, and not generally.**
  - Note that if you omit the access modifier in front of a class, **internal** will be the default used by the compiler.

Member Accessibility

- **Access to individual class members can be controlled by placing an access modifier such as *public* or *private* in front of the member.**
  - Member access can only further restrict access to a class, not widen it.
  - Thus if you have a class with **internal** accessibility, making a member **public** will not make it accessible from outside the assembly.
- **There are five modes of member accessibility**
  - Public
  - Private
  - Protected
  - Internal
  - Internal Protected

Member Accessibility Qualifiers

- **Public**
  - A **public** member can be accessed from outside the class.
- **Private**
  - A **private** member can be accessed only from within the class.
- **Protected**
  - Inheritance introduces a third kind of accessibility, **protected**. A protected member can be accessed from within the class and from within any derived classes.
  - Protected mode access should be used sparingly.
- **Internal**
  - An **internal** member can be accessed from within classes in the same assembly but not from classes outside the assembly.
- **Internal Protected**
  - An **internal protected** member can be accessed either from within the assembly or from outside the assembly by a derived class.

Member Accessibility Example

- **The *SimpleAccount* program that we have already examined illustrates use of protected accessibility.**
  - The field **balance** in the **Account** class is declared as protected, because the **Post** method of the **CheckingAccount** class needs access to **balance**.
  - Note that read access is publicly available through the **Balance** method, but **Post** also updates balance.
  - Here again is the definition of **Account**. Note use of the keyword **protected**.

```
// Account.cs

public class Account
{
   protected decimal balance;
   ...
```

  - And here is the **CheckingAccount** class that makes use of **balance**:

```
// CheckingAccount.cs
public class CheckingAccount : Account
{
   private decimal fee = 5.00m;
   public void Post()
   {
      balance -= fee;
   }
}
```

Method Hiding

- **In our first example of inheritance we added a new method *Post* to our derived class *CheckingAccount*.**
  - The derived class inherited the methods **Deposit** and **Withdraw**, which are automatically available "as is."
- **Sometimes we may want the derived class to do something a little different for some of the methods of the base class.**
  - In this case we will put code for these changed methods in the derived class, and we say the derived class "hides" the corresponding methods in the base class.
  - Note that hiding a method requires that the signatures match exactly.
  - Methods have the same signature if they have the same number of parameters, and these parameters have the same types and modifiers, such as **ref** or **out**. The return type does not contribute to defining the signature of a method. We discussed signatures in Chapter 10.

Method Hiding and Overriding

- **In C#, if you declare a method in a derived class that has the same signature as a method in the base class, you will get a compiler warning message.**
  - In such a circumstance, there are two things you may wish to do.
  - The first is to *hide* the base class method, which is what we discuss in this section.
  - The second is to *override* the base class method, which we will discuss in the next chapter.
- **To hide a base class method, place the keyword *new* in front of the method in the derived class.**
  - When you hide a method of the base class, you may want to call the base class method within your implementation of the new method.
  - You can do this by using the keyword **base**, followed by a period, followed by the method name and actual parameters.

Example: Method Hiding

- **The example program *HideAccount* illustrates method hiding.**
  - This program has the same **Account** base class, as in our previous example **SimpleAccount**.
  - But our derived class **CheckingAccount** is somewhat different.
  - In place of calculating a flat fee, the fee instead is based on the number of transactions.
  - Thus the methods **Deposit** and **Withdraw** now have to increment a count of the number of transactions besides performing the actual operation, which can be delegated to the base class by using the **base** keyword.

Initialization

- **An important issue when working with inheritance is *initialization*.**
- **A common way to initialize a class instance is through a constructor.**
  - When the class is derived from a base class, we may want to invoke a base class constructor to perform further initialization.
  - In this section we will see how to do implement such initializations.
  - We will also review how initialization works without regard to inheritance.

Initialization Fundamentals

- **A classic problem in computer programming is uninitialized variables.**
  - In many programming languages you can get unpredictable results, based on what happens to be in memory when the program is run.
- **C# addresses this issue by either requiring initialization or through assignment of default values.**
  - Local variables must be initialized prior to use; you will get a fatal compiler error if you attempt to use a local variable which has not been initialized.
  - Member variables of a class are initialized to default values.
  - Numeric data types are initialized to zero, and reference data types are initialized to **null**.
- **A member variable can be initialized right where it is defined. This code of both declaring and initializing a variable is called an *initializer*.**
  - An initializer applies to all class instances.
  - A member variable can be initialized in a constructor. The constructor code applies after the value has been set in the initializer, and can be used to initialize each instance individually.

Initialization Fundamentals Example

- **The program *TestInitial* illustrates these fundamentals.**
  - Note: It is illegal to use a local variable without first initializing it. Fields of classes have a default value assigned, but you will get a warning if you do not assign a value in your own code.

```
using System;

public class TestInitial
{
   private static int b;
   private int c = 1;
   private int d;
   public TestInitial()
   {
      Console.WriteLine(
         "In TestInitial constructor");
      Console.WriteLine("c = {0}", c);
      Console.WriteLine("d = {0}", d);
      d = 2;
   }
   public static void Main(string[] args)
   {
      int a;
      //Console.WriteLine("a = {0}", a);
      Console.WriteLine("b = {0}", b);
      TestInitial ti = new TestInitial();
      Console.WriteLine(
  "In Main after TestInitial object constructed");
      Console.WriteLine("c = {0}", ti.c);
      Console.WriteLine("d = {0}", ti.d);
   }
}
```

Default Constructor

- **As discussed in Chapter 8, constructors can be used to assign values to fields and perform other initializations.**
- **If you do not explicitly code an instructor, a default constructor taking no parameters will be provided by the compiler.**
  - The default constructor initializes all fields to their default values.
- **The program *InitialAccount\Step0* illustrates a very simple version of the *Account* class, with reliance on a default constructor.**
  - The four fields of the class do not have initializers, and you will get warning messages when you compile, but the code is legal.
  - The default constructor will assign these fields to their default values, which will be reported when the **GetStatement** method is called.
  - An account object is constructed using new, which invokes the default constructor.

Overloaded Constructors

- **You may overload a constructor just like you overload ordinary methods of a class.**
  - You provide code for several constructors, which must each have a unique signature.
  - As part of one constructor you may invoke another constructor by using a special colon notation, followed by **this** and actual parameters.
  - This code for the other constructor will then be invoked before entering the curly braces.
- **If you explicitly code one or more constructors, you can no longer have a default constructor provided automatically by the compiler.**
  - If you need to be able to create a new object instance without passing any parameters, you must define a constructor without parameters.
  - You do not need to put any code in the curly braces.

Example: Overloaded Constructors

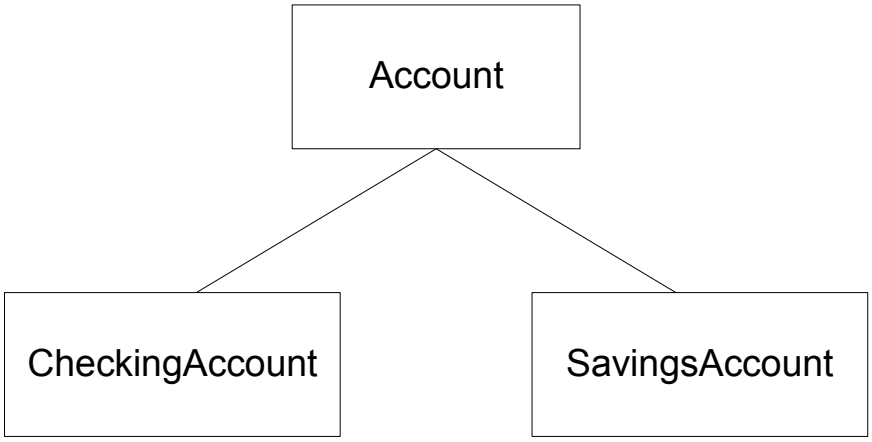
- **The program *InitialAccount\Step1* illustrates several overloaded constructors.**
  - The class is designed so that the user of the class can invoke **new** with three parameters, two parameters, one parameter, or no parameters.
  - Any parameters not explicitly assigned will be given their default values.
  - Write statements are provided inside the various constructors so that you can see the order in which they are invoked.
  - If you like, you may add a write statement to the constructor without parameters.
  - Before running the program, try to figure out the exact order in which the various constructors will be invoked and also what values will be assigned.

Invoking Base Class Constructors

- **If your derived class has a constructor with parameters, you may wish to pass some of these parameters along to a base class constructor.**
- **In C# you can conveniently invoke a base class constructor by using a colon, followed by the base keyword and a parameter list.**
  - This notation is similar to the notation applied for invoking another constructor in the derived class, only the **base** keyword is used in place of **this**.
  - Note that the syntax allows you to explicitly invoke a constructor only of an immediate **base** class. There is no notation that allows you to directly invoke a constructor higher up the inheritance hierarchy.
- **The program *InitialAccount\Step2* illustrates initialization in the *CheckingAccount* class. (The code for *Account* is the same as in *Step 1*.)**
  - An **Account** object is constructed, and then a **CheckingAccount** object is constructed.
  - Again, you should figure out the order of the various constructors before running the program.

Bank Case Study: Step 2

- **We conclude this chapter by giving *Step 2* of our case study.**
  - This step is not a direct extension of **Step 1**, as there is no bank consisting of many accounts.
  - Also, there is no interactive test program.
- **Instead, *Step 2* illustrates an inheritance hierarchy consisting of an *Account* base class and derived classes *CheckingAccount* and *SavingsAccount*, with some simple hardcoded test data.**
  - The figure illustrates this class hierarchy.



Bank Case Study Analysis

- **Our case study at this point consists of four classes, each in its own file.**
  - **Account**. This class encapsulates a single bank account consisting of an **Id**, an **Owner**, and a **Balance**. Operations are **Deposit** and **Withdraw**. There is also a field holding the number of transactions, and a **GetStatement** method is provided to show the current data for an account. The **Account** class counts the number of transactions.
  - **CheckingAccount**. This derived class adds a monthly fee, which is assessed to checking accounts but not to other accounts.
  - **SavingsAccount**. This derived class adds interest, which is paid to savings ac-counts but not to other accounts.
  - **TestAccount**. This class provides a hardcoded test program, which instantiates some classes, performs some transactions, and obtains statements.
- **As usual, the case study code may be found in the *CaseStudy* directory for this chapter.**

Account

- **The *Account* class for *Step 2* is identical to the class for *Step 1*, except two of the fields are protected, because derived classes need to access them.**

```
// Account.cs – Step 2

public class Account
{
   private int id;
   protected decimal balance;
   private string owner;
   protected int numXact = 0;
                          // number of transactions
   ...
```

CheckingAccount

- **The *CheckingAccount* class is new to the case study in *Step 2*.**
  - It is similar to the **CheckingAccount** class in the standalone examples, but a somewhat different algorithm is used for calculating the fee.
  - Some free transactions are allowed.
  - Also, the base class already counts the number of transactions, so the **Deposit** and **Withdraw** methods can be used without change.
  - The **GetStatement** method hides the corresponding method in the base class and adds the functionality of also showing the fee.
  - The **Post** method subtracts the fee from the balance and resets the number of transactions to 0.

Checking Account (Cont'd)

```
// CheckingAccount.cs - Step 2

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;
      }
   }
   new public string GetStatement()
   {
      string s = base.GetStatement();
      s += ", fee = " + Fee;
      return s;
   }
   public void Post()
   {
      balance -= Fee;
      numXact = 0;
   }
}
```

SavingsAccount

- **The *SavingsAccount* class is new to the case study in Step 2.**
  - It adds the feature of interest, which is calculated monthly and based on an annual rate.
  - Interest is paid on the minimum balance.
  - The base class **Withdraw** method is hidden, so that the derived class can also up-date the minimum balance.
  - The **GetStatement** method also hides the base class version, appending information about the interest paid to the statement string.
  - The **Post** method adds the interest and resets the number of transactions and minimum balance.

```
using System;
public class SavingsAccount : Account
{
   private decimal minBalance;
   private decimal rate = 0.06m;
   public SavingsAccount(decimal balance, string
      owner, int id) : base(balance, owner, id)
   {
      minBalance = balance;
   }
   public decimal Interest
   {
      get
      {
         return minBalance * rate/12;
      }
   }
```

SavingsAccount (Cont'd)

```
   new public void Withdraw(decimal amount)
   {
      base.Withdraw(amount);
      if (balance < minBalance)
      {
         minBalance = balance;
      }
   }
   public void Post()
   {
      balance += Interest;
      numXact = 0;
      minBalance = balance;
   }
   new public string GetStatement()
   {
      string s = base.GetStatement();
      s += ", interest = " + Interest;
      return s;
   }
   public decimal Rate
   {
      get
      {
         return rate;
      }
      set
      {
         rate = value;
      }
   }
}
```

TestAccount

- **The test program is hardcoded.**
- **It creates a few account objects, performs a few transactions, and obtains statements.**
- **It also shows the balance after posting.**
  - Notice the three overloaded **ShowAccount** methods that do exactly the same thing!
  - We will see in Chapter 14 that with virtual methods we can handle this kind of situation more simply, with a single method that applies to any **Account** object and does the proper thing, depending on the kind of account.
- **You may look at the straightforward code online.**

Running the Case Study

- **Again, you should both study the code and run the case study.**
  - In this case, running the case study is trivial, as the program is not interactive.
  - You should study the output to make sure you completely understand it. Here is a sample run:

```
Account: Statement for Bob id = 1
0 transactions, balance = 100
Account: Statement for Bob id = 1
3 transactions, balance = 100
CheckingAccount: Statement for Charlie id = 2
0 transactions, balance = 200, fee = 0
CheckingAccount: Statement for Charlie id = 2
3 transactions, balance = 150, fee = 5
After posting, balance = 145
SavingsAccount: Statement for David id = 3
0 transactions, balance = 300, interest = 1.5
SavingsAccount: Statement for David id = 3
3 transactions, balance = 250, interest = 1.25
After posting, balance = 251.25
```

Summary

- **Inheritance is a fundamental part of object-oriented programming.**
- **C# supports single inheritance, and all classes in C# ultimately inherit, or derive, from a common base class, *object*.**
- **Inheritance supports code reuse by automatically making all code in a base class available to the derived classes.**
- **Inheritance gives rise to another option for access control, called *protected*.**
- **Methods in a derived class may hide the corresponding method in the base class, possibly making use of the base class method in their implementation.**
- **C# has a robust set of features for handling initialization issues, including a mechanism for the proper initialization of the base class as well as the current class.**
- **We extended the case study to support an inheritance hierarchy of accounts, including checking and savings accounts.**
- **We will continue our study of inheritance in the next chapter, taking up polymorphism and related topics.**