title	description	ms.date
Object-Oriented Programming (C#)	C# provides full support for object-oriented programming including abstraction, encapsulation, inheritance, and polymorphism.	02/25/2022

Object-Oriented programming (C#)

C# is an object-oriented programming language. The four basic principles of object-oriented programming are:

- *Abstraction* Modeling the relevant attributes and interactions of entities as classes to define an abstract representation of a system.
- *Encapsulation* Hiding the internal state and functionality of an object and only allowing access through a public set of functions.
- *Inheritance* Ability to create new abstractions based on existing abstractions.
- *Polymorphism* Ability to implement inherited properties or methods in different ways across multiple abstractions.

In the preceding tutorial, introduction to classes you saw both *abstraction* and *encapsulation*. The `BankAccount` class provided an abstraction for the concept of a bank account. You could modify its implementation without affecting any of the code that used the `BankAccount` class. Both the `BankAccount` and `Transaction` classes provide encapsulation of the components needed to describe those concepts in code.

In this tutorial, you'll extend that application to make use of *inheritance* and *polymorphism* to add new features. You'll also add features to the `BankAccount` class, taking advantage of the *abstraction* and *encapsulation* techniques you learned in the preceding tutorial.

Create different types of accounts

After building this program, you get requests to add features to it. It works great in the situation where there is only one bank account type. Over time, needs change, and related account types are requested:

- An interest earning account that accrues interest at the end of each month.
- A line of credit that can have a negative balance, but when there's a balance, there's an interest charge each month.
- A pre-paid gift card account that starts with a single deposit, and only can be paid off. It can be refilled once at the start of each month.

All of these different accounts are similar to `BankAccount` class defined in the earlier tutorial. You could copy that code, rename the classes, and make modifications. That technique would work in the short term, but it would be more work over time. Any changes would be copied across all the affected classes.

Instead, you can create new bank account types that inherit methods and data from the `BankAccount` class created in the preceding tutorial. These new classes can extend the `BankAccount` class with the specific behavior needed for each type:

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
```

```
}

public class GiftCardAccount : BankAccount
{
}
```

Each of these classes *inherits* the shared behavior from their shared *base class*, the `BankAccount` class. Write the implementations for new and different functionality in each of the *derived classes*. These derived classes already have all the behavior defined in the `BankAccount` class.

It's a good practice to create each new class in a different source file. In Visual Studio, you can right-click on the project, and select *add class* to add a new class in a new file. In Visual Studio Code, select *File* then *New* to create a new source file. In either tool, name the file to match the class: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs*, and *GiftCardAccount.cs*.

When you create the classes as shown in the preceding sample, you'll find that none of your derived classes compile. A constructor is responsible for initializing an object. A derived class constructor must initialize the derived class, and provide instructions on how to initialize the base class object included in the derived class. The proper initialization normally happens without any extra code. The `BankAccount` class declares one public constructor with the following signature:

```
public BankAccount(string name, decimal initialBalance)
```

The compiler doesn't generate a default constructor when you define a constructor yourself. That means each derived class must explicitly call this constructor. You declare a constructor that can pass arguments to the base class constructor. The following code shows the constructor for the `InterestEarningAccount`:

:::code language="csharp" source="./snippets/object-oriented-programming/InterestEarningAccount.cs" ID="DerivedConstructor":::

The parameters to this new constructor match the parameter type and names of the base class constructor. You use the `: base()` syntax to indicate a call to a base class constructor. Some classes define multiple constructors, and this syntax enables you to pick which base class constructor you call. Once you've updated the constructors, you can develop the code for each of the derived classes. The requirements for the new classes can be stated as follows:

- An interest earning account:
  - Will get a credit of 2% of the month-ending-balance.
- A line of credit:
  - Can have a negative balance, but not be greater in absolute value than the credit limit.
  - Will incur an interest charge each month where the end of month balance isn't 0.
  - Will incur a fee on each withdrawal that goes over the credit limit.
- A gift card account:
  - Can be refilled with a specified amount once each month, on the last day of the month.

You can see that all three of these account types have an action that takes places at the end of each month. However, each account type does different tasks. You use *polymorphism* to implement this code. Create a single `virtual` method in the `BankAccount` class:

:::code language="csharp" source="./snippets/object-oriented-programming/BankAccount.cs" ID="DeclareMonthEndTransactions":::

The preceding code shows how you use the `virtual` keyword to declare a method in the base class that a derived class may provide a different implementation for. A `virtual` method is a method where any derived class may choose to reimplement. The derived classes use the `override` keyword to define the new implementation. Typically you refer to this as "overriding the base class implementation". The `virtual` keyword specifies that derived classes may override the behavior. You can also declare `abstract` methods where derived classes must override the behavior. The base class does not provide an implementation for an `abstract` method. Next, you need to define the implementation for two of the new classes you've created. Start with the `InterestEarningAccount`:

:::code language="csharp" source="./snippets/object-oriented-programming/InterestEarningAccount.cs" ID="ApplyMonthendInterest":::

Add the following code to the `LineOfCreditAccount`. The code negates the balance to compute a positive interest charge that is withdrawn from the account:

:::code language="csharp" source="./snippets/object-oriented-programming/LineOfCreditAccount.cs" ID="ApplyMonthendInterest":::

The `GiftCardAccount` class needs two changes to implement its month-end functionality. First, modify the constructor to include an optional amount to add each month:

:::code language="csharp" source="./snippets/object-oriented-programming/GiftCardAccount.cs" ID="GiftCardAccountConstruction":::

The constructor provides a default value for the `monthlyDeposit` value so callers can omit a `0` for no monthly deposit. Next, override the `PerformMonthEndTransactions` method to add the monthly deposit, if it was set to a non-zero value in the constructor:

:::code language="csharp" source="./snippets/object-oriented-programming/GiftCardAccount.cs" ID="AddMonthlyDeposit":::

The override applies the monthly deposit set in the constructor. Add the following code to the `Main` method to test these changes for the `GiftCardAccount` and the `InterestEarningAccount`:

:::code language="csharp" source="./snippets/object-oriented-programming/Program.cs" ID="FirstTests":::

Verify the results. Now, add a similar set of test code for the `LineOfCreditAccount`:

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

When you add the preceding code and run the program, you'll see something like the following error:

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter 'amount
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in BankAccount.cs:line
```

```
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in LineOfCreditAccount.cs:l
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

> **Note**
>
> The actual output includes the full path to the folder with the project. The folder names were omitted for brevity. Also, depending on your code format, the line numbers may be slightly different.

This code fails because the `BankAccount` assumes that the initial balance must be greater than 0. Another assumption baked into the `BankAccount` class is that the balance can't go negative. Instead, any withdrawal that overdraws the account is rejected. Both of those assumptions need to change. The line of credit account starts at 0, and generally will have a negative balance. Also, if a customer borrows too much money, they incur a fee. The transaction is accepted, it just costs more. The first rule can be implemented by adding an optional argument to the `BankAccount` constructor that specifies the minimum balance. The default is `0`. The second rule requires a mechanism that enables derived classes to modify the default algorithm. In a sense, the base class "asks" the derived type what should happen when there's an overdraft. The default behavior is to reject the transaction by throwing an exception.

Let's start by adding a second constructor that includes an optional `minimumBalance` parameter. This new constructor does all the actions done by the existing constructor. Also, it sets the minimum balance property. You could copy the body of the existing constructor, but that means two locations to change in the future. Instead, you can use *constructor chaining* to have one constructor call another. The following code shows the two constructors and the new additional field:

:::code language="csharp" source="./snippets/object-oriented-programming/BankAccount.cs" ID="ConstructorModifications":::

The preceding code shows two new techniques. First, the `minimumBalance` field is marked as `readonly`. That means the value cannot be changed after the object is constructed. Once a `BankAccount` is created, the `minimumBalance` can't change. Second, the constructor that takes two parameters uses `: this(name, initialBalance, 0) { }` as its implementation. The `: this()` expression calls the other constructor, the one with three parameters. This technique allows you to have a single implementation for initializing an object even though client code can choose one of many constructors.

This implementation calls `MakeDeposit` only if the initial balance is greater than `0`. That preserves the rule that deposits must be positive, yet lets the credit account open with a `0` balance.

Now that the `BankAccount` class has a read-only field for the minimum balance, the final change is to change the hard code `0` to `minimumBalance` in the `MakeWithdrawal` method:

```
if (Balance - amount < _minimumBalance)
```

After extending the `BankAccount` class, you can modify the `LineOfCreditAccount` constructor to call the new base constructor, as shown in the following code:

:::code language="csharp" source="./snippets/object-oriented-programming/LineOfCreditAccount.cs" ID="ConstructLineOfCredit":::

Notice that the `LineOfCreditAccount` constructor changes the sign of the `creditLimit` parameter so it matches the meaning of the `minimumBalance` parameter.

Different overdraft rules

The last feature to add enables the `LineOfCreditAccount` to charge a fee for going over the credit limit instead of refusing the transaction.

One technique is to define a virtual function where you implement the required behavior. The `BankAccount` class refactors the `MakeWithdrawal` method into two methods. The new method does the specified action when the withdrawal takes the balance below the minimum. The existing `MakeWithdrawal` method has the following code:

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
```

```
        if (Balance - amount < _minimumBalance)
        {
            throw new InvalidOperationException("Not sufficient funds for this withdrawal");
        }
        var withdrawal = new Transaction(-amount, date, note);
        _allTransactions.Add(withdrawal);
    }
```

Replace it with the following code:

:::code language="csharp" source="./snippets/object-oriented-programming/BankAccount.cs" ID="RefactoredMakeWithdrawal":::

The added method is `protected`, which means that it can be called only from derived classes. That declaration prevents other clients from calling the method. It's also `virtual` so that derived classes can change the behavior. The return type is a `Transaction?`. The `?` annotation indicates that the method may return `null`. Add the following implementation in the `LineOfCreditAccount` to charge a fee when the withdrawal limit is exceeded:

:::code language="csharp" source="./snippets/object-oriented-programming/LineOfCreditAccount.cs" ID="AddOverdraftFee":::

The override returns a fee transaction when the account is overdrawn. If the withdrawal doesn't go over the limit, the method returns a `null` transaction. That indicates there's no fee. Test these changes by adding the following code to your `Main` method in the `Program` class:

:::code language="csharp" source="./snippets/object-oriented-programming/Program.cs" ID="TestLineOfCredit":::

Run the program, and check the results.

Summary

If you got stuck, you can see the source for this tutorial in our GitHub repo.

This tutorial demonstrated many of the techniques used in Object-Oriented programming:

- You used *Abstraction* when you defined classes for each of the different account types. Those classes described the behavior for that type of account.
- You used *Encapsulation* when you kept many details `private` in each class.
- You used *Inheritance* when you leveraged the implementation already created in the `BankAccount` class to save code.
- You used *Polymorphism* when you created `virtual` methods that derived classes could override to create specific behavior for that account type.