

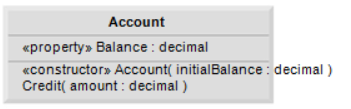
Systems Development: Object Oriented Programming

(H172 35)

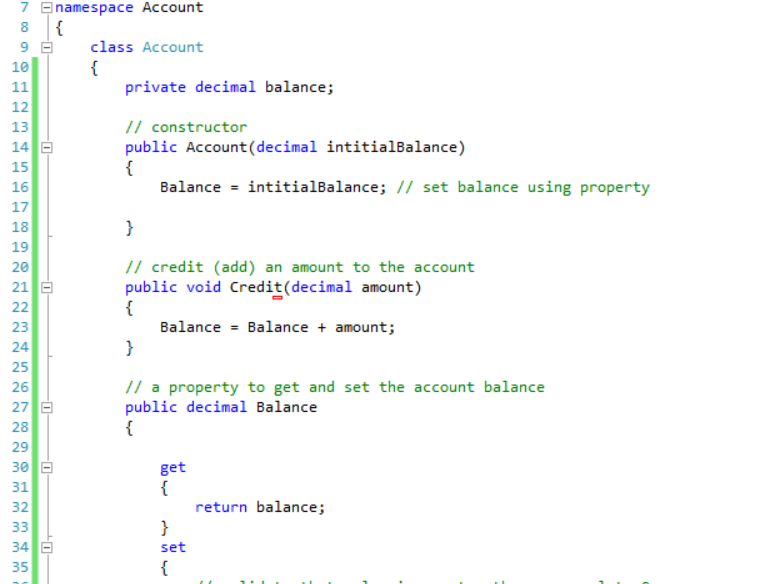
Introducing Classes

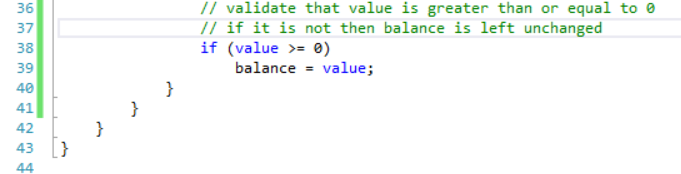
Account Walkthrough

We are going to create a simple class named Account that maintains the balance of a bank account.



Create a new Console Application and add a class called Account.cs which contains the code below. A typical bank services many accounts, each with its own balance, so line 11 declares an instance variable named balance of type decimal. Variable balance is an instance variable because it’s declared in the body of the class (lines 10–42) but outside the class’s method and property declarations (lines 14–18, 21–24 and 27–40). Every instance (i.e., object) of class Account contains its own copy of balance.





Account Class Constructor

Class Account contains a constructor, a method, and a property. Since it’s common for someone opening an account to place money in the account immediately, the constructor (lines 14–18) receives a parameter initialBalance of type decimal that represents the account’s starting balance. Line 16 assigns initialBalance to the property Balance, invoking Balance’s set accessor to initialize the instance variable balance.

Account Method Credit

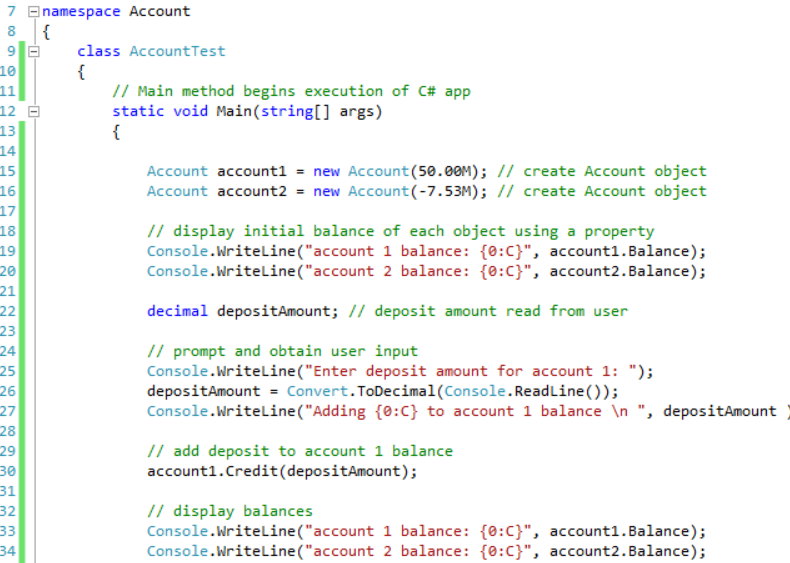
Method Credit (lines 21–24) doesn’t return data when it completes its task, so its return type is void. The method receives one parameter named amount—a decimal value that’s added to the property Balance. Line 23 uses both the get and set accessors of Balance. The expression Balance + amount invokes property Balance’s get accessor to obtain the current value of instance variable balance, then adds amount to it. We then assign the result to instance variable balance by invoking the Balance property’s set accessor (thus replacing the prior balance value).

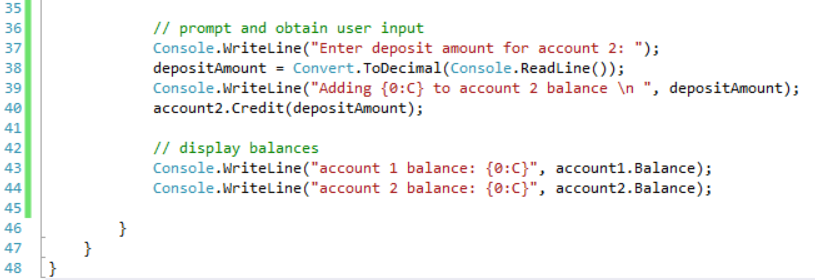
Account Property Balance

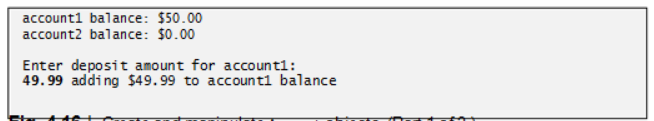
Property Balance (lines 27–41) provides a get accessor, which allows clients of the class (i.e., other classes that use this class) to obtain the value of a particular Account object’s balance. The property has type decimal (line 27). Balance also provides an enhanced set accessor.

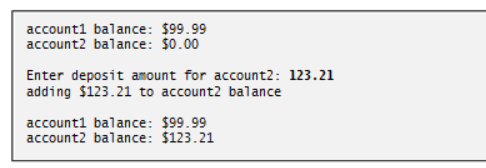
The app enhances the set accessor of class Account’s property Balance to perform this validation (also known as validity checking). Line 38 ensures that value is nonnegative. If the value is greater than or equal to 0, the amount stored in value is assigned to instance variable balance in line 39. Otherwise, balance is left unchanged.

Now that we have implemented the class we can write client code that uses the class. We are now going to write the client application code in the Program.cs file (automatically created as part of a console application project), however we will change the name from Program.cs to AccountTest.cs.



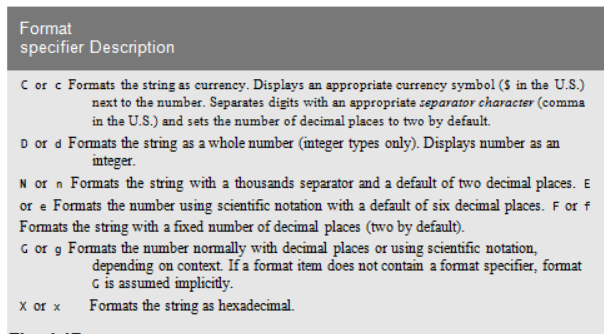






Class AccountTest creates two Account objects (lines 15–16) and initializes them respectively with 50.00M and -7.53M (the decimal literals representing the real num-bers 50.00 and -7.53). The Account constructor (lines 14–18 of **Account.cs**) references property *Balance* to initialise balance. In previous examples, the benefit of referencing the property in the constructor was not evident. Now, however, the constructor takes advantage of the validation provided by the set accessor of the *Balance* property. The constructor simply assigns a value to *Balance* rather than duplicating the set accessor’s validation code. When line 16 of AccountTest.cs passes an initial balance of -7.53 to the Account constructor, the constructor passes this value to the set accessor of property *Balance*, where the actual initialisation occurs. This value is less than 0, so the set accessor does not modify balance, leaving this instance variable with its default value of 0.

Lines 19–20 output the balance in each Account by using the Account’s *Balance* property. When *Balance* is used for account1 (line 19), the value of account1’s *balance* is returned by the get accessor in line 30 of **Account.cs** and displayed by the Console.WriteLine statement.. Similarly, when property *Balance* is called for *account2* from line 20, the value of the account2’s *balance* is returned from line 30 of **Account.cs** and displayed by the Console.WriteLine statement. The balance of account2 is 0 because the constructor ensured that the account could not begin with a negative balance. The value is output by WriteLine with the format item {0:C}, which formats the account balance as a monetary amount. The : after the 0 indi-cates that the next character represents a format specifier, and the C format specifier after the : specifies a monetary amount (C is for currency). The cultural settings on the user’s machine determine the format for displaying monetary amounts. For example, in the United States, 50 displays as $50.00. In Germany, 50 displays as 50,00 €. [Note: Change the Command Prompt’s font to Lucida Console for the € symbol to display correctly.] Below are a few other format specifiers in addition to C.



Line 22 declares local variable *depositAmount* to store each deposit amount entered by the user. Unlike the instance variable balance in class Account, the local variable

*depositAmount* in Main is not initialized to 0 by default. Also, a local variable can be used only in the method in which it’s declared. However, this variable does not need to be initialised here because its value will be determined by the user’s input. The compiler does not allow a local variable’s value to be read until it’s initialised.

Line 25 prompts the user to enter a deposit amount for account1. Line 26 obtains the input from the user by calling the Console class’s ReadLine method, then passing the string entered by the user to the Convert class’s ToDecimal method, which returns the decimal value in this string. Line 27 displays the deposit amount. Line 30 calls object account1’s Credit method and supplies *depositAmount* as the method’s argument. When the method is called, the argument’s value is assigned to parameter *amount* of method Credit (lines 21–24 of **Account.cs**), then method Credit adds that value to the *balance* (line 23 of **Account.cs**). Lines 33–34 output the balances of both Accounts again to show that only account1’s balance changed.

Line 37 prompts the user to enter a deposit amount for account2. Line 38 obtains the input from the user by calling method Console.ReadLine, and passing the return value to the Convert class’s ToDecimal method. Lines 39 display the deposit amount. Line 40 calls object account2’s Credit method and supplies *depositAmount* as the method’s argument, then method Credit adds that value to the *balance*. Finally, lines 43–44 output the balances of both Accounts again to show that only account2’s balance changed.

set and get Accessors with Different Access Modifiers

By default, the get and set accessors of a property have the same access as the property— for example, for a public property, the accessors are public. It’s possible to declare the get and set accessors with different access modifiers. In this case, one of the accessors must implicitly have the same access as the property and the other must be declared with a more restrictive access modifier than the property. For example, in a public property, the get accessor might be public and the set accessor might be private. This will be further discussed in a future walkthrough.

C# type decimal

C# provides three simple types for storing real numbers—float, double, and decimal. Types float and double are called floating-point types. The primary difference between them and decimal is that decimal variables store a limited range of real numbers precisely, whereas floating-point variables store only approximations of real numbers, but across a much greater range of values. Also, double variables can store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number’s precision) than float variables. A key use of type decimal is representing monetary amounts.

Variables of type float represent single-precision floating-point numbers and have seven significant digits. Variables of type double represent double-precision floating-point numbers. These require twice as much storage as float variables and provide 15–16 significant digits—approximately double the precision of float variables. Variables of type decimal require twice as much storage as double variables and provide 28–29 significant digits. In some apps, even variables of type double and decimal will be inadequate.

Most programmers represent floating-point numbers with type double. In fact, C# treats all real numbers you type in an app’s source code (such as 7.33 and 0.0975) as double values by default. Such values in the source code are known as floating-point literals. To type a **decimal** literal, you must type the letter “M” or “m” (which stands for “money”) at the end of a real number (for example, 7.33M is a decimal literal rather than a double). Integer literals are implicitly converted into type float, double or decimal when they’re assigned to a variable of one of these types.

Although floating-point numbers are not always 100% precise, they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Due to the imprecise nature of floating-point numbers, type decimal is preferred over the floating-point types whenever the calculations need to be exact, as with monetary calculations. In cases where approximation is enough, double is preferred over type float because double variables can represent floating-point numbers more accurately. For this reason, we use type decimal for monetary amounts and type double for other real numbers.

Real numbers also arise as a result of division. In conventional arithmetic, for example, when we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.