3.4 Account Class with a Balance; Floating-Point Numbers

We now declare an `Account` class that maintains the *balance* of a bank account in addition to the name. Most account balances are not integers. So, class `Account` represents the account balance as a **floating-point number**—a number with a *decimal point*, such as 43.95, 0.0, -129.8873. [In Chapter 8, we'll begin representing monetary amounts precisely with class `BigDecimal` as you should do when writing industrial-strength monetary applications.]

Java provides two primitive types for storing floating-point numbers in memory—`float` and `double`. Variables of type `float` represent **single-precision floating-point numbers** and can hold up to *seven significant digits*. Variables of type `double` represent **double-precision floating-point numbers**. These require *twice* as much memory as `float` variables and can hold up to *15 significant digits*—about *double* the precision of `float` variables.

Most programmers represent floating-point numbers with type `double`. In fact, Java treats all floating-point numbers you type in a program'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**. See Appendix D, Primitive Types, for the precise ranges of values for `float`s and `double`s.

3.4.1 `Account` Class with a `balance` Instance Variable of Type `double`

Our next app contains a version of class `Account` (Fig. 3.8) that maintains as instance variables the `name` *and* the `balance` of a bank account. A typical bank services *many* accounts, each with its *own* balance, so line 7 declares an instance variable `balance` of type `double`. Every instance (i.e., object) of class `Account` contains its *own* copies of *both* the `name` and the `balance`.

```
1   // Fig. 3.8: Account.java
2   // Account class with a double instance variable
3   // and deposit method that perform validation.
4
5   public class Account {
6      private String name; // instance variable
7      private double balance; // instance variable
8
9      // Account constructor that receives two para
10     public Account(String name, double balance) {
11        this.name = name; // assign name to instan
12
13        // validate that the balance is greater th
14        // instance variable balance keeps its def
```

```
        if (balance > 0.0) { // if the balance is
            this.balance = balance; // assign it to
        }
    }

    // method that deposits (adds) only a valid a
    public void deposit(double depositAmount) {
        if (depositAmount > 0.0) { // if the depos
            balance = balance + depositAmount; // a
        }
    }

    // method returns the account balance
    public double getBalance() {
        return balance;
    }

    // method that sets the name
    public void setName(String name) {
        this.name = name;
    }

    // method that returns the name
    public String getName() {
        return name;
    }
}
```

Fig. 3.8

`Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation.

`Account` Class Two-Parameter Constructor

The class has a *constructor* and four *methods*. It's common for someone opening an account to deposit money immediately, so the constructor (lines 10–18) now receives a second parameter—`balance` of type `double` that represents the *starting balance*. Lines 15–17 ensure that `initialBalance` is greater than `0.0`. If so, the `balance` parameter's value is assigned to the instance variable `balance`. Otherwise, the instance variable `balance` remains at `0.0`—its *default initial value*.

`Account` Class `deposit` Method

Method `deposit` (lines 21–25) does *not* return any data when it completes its task, so its return type is `void`. The method receives one parameter named `depositAmount`—a `double` value that's *added* to the instance variable `balance` *only* if the parameter value is *valid* (i.e., greater than zero). Line 23 first adds the current `balance` and `depositAmount`, forming a *temporary* sum which is *then* assigned to `balance`, *replacing* its prior value (recall that addition has a *higher* precedence than assignment). It's important to understand that the calculation on the right side of the assignment operator in line 23 does *not* modify the `balance`—that's why the assignment is necessary.

`Account` Class `getBalance` Method

Method `getBalance` (lines 28–30) allows *clients* of the class (i.e., other classes whose methods call the methods of this class) to obtain the value of a particular `Account` object's `balance`. The method specifies return type `double` and an *empty* parameter list.

`Account`'s Methods Can All Use `balance`

Once again, lines 15, 16, 23 and 29 use the variable `balance` even though it was *not* declared in *any* of the methods. We can use `balance` in these methods because it's an *instance variable* of the class.

3.4.2 `AccountTest` Class to Use Class `Account`

Class `AccountTest` (Fig. 3.9) creates two `Account` objects (lines 7–8) and initializes them with a *valid* balance of

`50.00` and an *invalid* balance of `-7.53`, respectively—for the purpose of our examples, we assume that balances must be greater than or equal to zero. The calls to method `System.out.printf` in lines 11–14 output the account names and balances, which are obtained by calling each `Account`’s `getName` and `getBalance` methods.

```
1  // Fig. 3.9: AccountTest.java
2  // Inputting and outputting floating-point numbe
3  import java.util.Scanner;
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        Account account1 = new Account("Jane Green
8        Account account2 = new Account("John Blue"
9
10       // display initial balance of each object
11       System.out.printf("%s balance: $%.2f%n",
12          account1.getName(), account1.getBalance
13       System.out.printf("%s balance: $%.2f%n%n",
14          account2.getName(), account2.getBalance
15
16       // create a Scanner to obtain input from t
17       Scanner input = new Scanner(System.in);
18
19       System.out.print("Enter deposit amount for
20       double depositAmount = input.nextDouble();
21       System.out.printf("%nadding %.2f to accoun
22          depositAmount);
23       account1.deposit(depositAmount); // add to
24
25       // display balances
26       System.out.printf("%s balance: $%.2f%n",
27          account1.getName(), account1.getBalance
28       System.out.printf("%s balance: $%.2f%n%n",
29          account2.getName(), account2.getBalance
30
```

```
      System.out.print("Enter deposit amount for
      depositAmount = input.nextDouble(); // obt
      System.out.printf("%nadding %.2f to accoun
         depositAmount);
      account2.deposit(depositAmount); // add to

      // display balances
      System.out.printf("%s balance: $%.2f%n",
         account1.getName(), account1.getBalance
      System.out.printf("%s balance: $%.2f%n%n",
         account2.getName(), account2.getBalance
   }
}
```

```
Jane Green balance: $50.00
John Blue balance: $0.00
Enter deposit amount for account1: 25.53
adding 25.53 to account1 balance
Jane Green balance: $75.53
John Blue balance: $0.00
Enter deposit amount for account2: 123.45
adding 123.45 to account2 balance
Jane Green balance: $75.53
John Blue balance: $123.45
```

Fig. 3.9

Inputting and outputting floating-point numbers with `Account` objects.

Displaying the `Account` Objects' Initial Balances

When method `getBalance` is called for `account1` from line 12, the value of `account1`'s `balance` is returned from line 29 of Fig. 3.8 and displayed by the `System.out.printf` statement (Fig. 3.9, lines 11–12). Similarly, when method `getBalance` is called for `account2` from line 14, the value of the `account2`'s `balance` is returned from line 29 of Fig. 3.8 and displayed by the `System.out.printf` statement (Fig. 3.9, lines 13–14). The `balance` of `account2` is initially `0.00`, because the constructor rejected the attempt to start `account2` with a *negative* balance, so the balance retains its default initial value.

Formatting Floating-Point Numbers for Display

Each of the `balance`s is output by `printf` with the format specifier `%.2f`. The `%f` **format specifier** is used to output values of type `float` or `double`. The `.2` between `%` and `f` represents the number of *decimal places* (`2`) that should be output to the *right* of the decimal point in the floating-point number—also known as the number's **precision**. Any floating-point value output with `%.2f` will be *rounded* to the *hundredths position*—for example, 123.457 would be rounded to 123.46 and 27.33379 would be rounded to 27.33.

Reading a Floating-Point Value from the User and Making a Deposit

Line 19 (Fig. 3.9) prompts the user to enter a deposit amount for `account1`. Line 20 declares *local* variable `depositAmount` to store each deposit amount entered by the user. Unlike *instance* variables (such as `name` and `balance` in class `Account`), *local* variables (like `depositAmount` in `main`) are *not* initialized by default, so they normally must be initialized explicitly. As you'll learn momentarily, variable `depositAmount`'s initial value will be determined by the user's input.

Common Programming Error 3.2

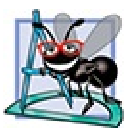
The Java compiler will issue a compilation error if you attempt to use the value of an uninitialized local variable. This helps you avoid dangerous execution-time logic errors. It's always better to get the errors out of your programs at compilation time rather than execution time.

Line 20 obtains the input from the user by calling `Scanner` object `input`'s `nextDouble` method, which returns a `double` value entered by the user. Lines 21–22 display the `depositAmount`. Line 23 calls object `account1`'s `deposit` method with the `depositAmount` as the method's *argument*. When the method is called, the argument's value is assigned to the parameter `depositAmount` of method `deposit` (line 21 of Fig. 3.8); then method `deposit` adds that value to the `balance`. Lines 26–29 (Fig. 3.9) output the `name`s and `balance`s of both `Account`s *again* to show that *only* `account1`'s `balance` has changed.

Line 31 prompts the user to enter a deposit amount for `account2`. Line 32 obtains the input from the user by calling `Scanner` object `input`'s `nextDouble` method. Lines 33–34 display the `depositAmount`. Line 35 calls object `account2`'s `deposit` method with `depositAmount` as the method's *argument*; then method `deposit` adds that value to the `balance`. Finally, lines 38–41 output the `name`s and `balance`s of both `Account`s *again* to show that *only* `account2`'s `balance` has changed.

Duplicated Code in Method `main`

The six statements at lines 11–12, 13–14, 26–27, 28–29, 38–39 and 40–41 are almost *identical*—they each output an `Account`'s `name` and `balance`. They differ only in the name of the `Account` object—`account1` or `account2`. Duplicate code like this can create *code maintenance problems* when that code needs to be updated—if *six* copies of the same code all have the same error or update to be made, you must make that change *six* times, *without making errors*. Exercise 3.15 asks you to modify Fig. 3.9 to include a `displayAccount` method that takes as a parameter an `Account` object and outputs the object's `name` and `balance`. You'll then replace `main`'s duplicated statements with six calls to `displayAccount`, thus reducing the size of your program and improving its maintainability by having *one* copy of the code that displays an `Account`'s `name` and `balance`.



Software Engineering Observation 3.4

Replacing duplicated code with calls to a method that contains one copy of that code can reduce the size of your program and improve its maintainability.

UML Class Diagram for Class `Account`

The UML class diagram in Fig. 3.10 concisely models class `Account` of Fig. 3.8. The diagram models in its *second* compartment the `private` attributes `name` of type `String` and `balance` of type `double`.

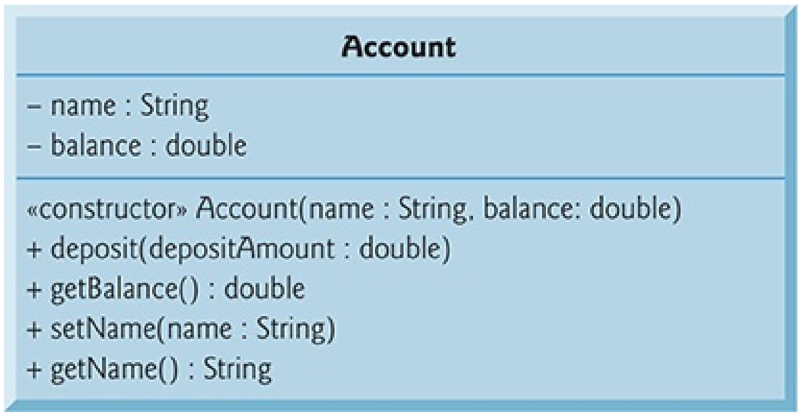


Fig. 3.10

UML class diagram for `Account` class of Fig. 3.8.

Description

Class `Account`’s *constructor* is modeled in the *third* compartment with parameters `name` of type `String` and `initialBalance` of type `double`. The class’s four `public` methods also are modeled in the *third* compartment—operation `deposit` with a `depositAmount` parameter of type `double`, operation `getBalance` with a return type of `double`, operation `setName` with a `name` parameter of type `String` and operation `getName` with a return type of `String`.