that initializes the four instance variables. Provide a *set* and a *get* method for each instance variable. In addition, provide a method named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a `double` value. If the quantity is not positive, it should be set to `0`. If the price per item is not positive, it should be set to `0.0`. Write a test app named `InvoiceTest` that demonstrates class `Invoice`'s capabilities.

7.12 ***(`Employee` Class)*** Create a class called `Employee` that includes three instance variables—a first name (type `String`), a last name (type `String`) and a monthly salary (`double`). Provide a constructor that initializes the three instance variables. Provide a *set* and a *get* method for each instance variable. If the monthly salary is not positive, do not set its value. Write a test app named `EmployeeTest` that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10% raise and display each `Employee`'s yearly salary again.

7.13 ***(`Date` Class)*** Create a class called `Date` that includes three instance variables—a month (type `int`), a day (type `int`) and a year (type `int`). Provide a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a *set* and a *get* method for each instance variable. Provide a method `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test app named `DateTest` that demonstrates class `Date`'s capabilities.

7.14 ***(Removing Duplicated Code in Method `main`)*** In the `AccountTest` class of **Fig. 7.9**, method `main` contains six statements (lines 11–12, 13–14, 26–27, 28–29, 38–39 and 40–41) that each display an `Account` object's `name` and `balance`. Study these statements and you'll notice that they differ only in the `Account` object being manipulated—`account1` or `account2`. In this exercise, you'll define a new `displayAccount` method that contains *one* copy of that output statement. The method's parameter will be an `Account` object and the method will output the object's `name` and `balance`. You'll then replace the six duplicated statements in `main` with calls to `displayAccount`, passing as an argument the specific `Account` object to output.

Modify class `AccountTest` of **Fig. 7.9** to declare method `displayAccount` (**Fig. 7.18**) *after* the closing right brace of `main` and *before* the closing right

brace of class `AccountTest`. Replace the comment in the method's body with a statement that displays `accountToDisplay`'s `name` and `balance`.

Fig. 7.18 Method `displayAccount` to add to class `Account`.

```
1   public static void displayAccount(Account
accountToDisplay) {
2      // place the statement that displays
3      // accountToDisplay's name and balance here
4   }
```

Recall that `main` is a `static` method, so it can be called without first creating an object of the class in which `main` is declared. We also declared method `displayAccount` as a `static` method. When `main` needs to call another method in the same class without first creating an object of that class, the other method *also* must be declared `static`.

Once you've completed `displayAccount`'s declaration, modify `main` to replace the statements that display each `Account`'s `name` and `balance` with calls to `displayAccount`—each receiving as its argument the `account1` or `account2` object, as appropriate. Then, test the updated `AccountTest` class to ensure that it produces the same output as shown in Fig. 7.9.

7.15 ***(Enhanced `GradeBook`)*** Modify the `GradeBook` class of Fig. 7.16 so that the constructor accepts as parameters the number of students and the number of exams, then builds an appropriately sized two-dimensional array, rather than receiving a preinitialized two-dimensional array as it does now. Set each element of the new two-dimensional array to `-1` to indicate that no grade has been entered for that element. Add a `setGrade` method that sets one grade for a particular student on a particular exam. Modify class `GradeBookTest` of Fig. 7.17 to input the number of students and number of exams for the `GradeBook` and to allow the instructor to enter one grade at a time.

Exercises 7.16–7.21 are reasonably challenging. Once you've done them, you ought to be able to implement most popular card games easily.

7.16 ***(Card Shuffling and Dealing)*** Modify Fig. 7.13 to deal a five-card poker hand. Then modify class `DeckOfCards` of Fig. 7.12 to include methods that determine whether a hand contains

a. a pair

b. two pairs

c. three of a kind (e.g., three jacks)

d. four of a kind (e.g., four aces)

e. a flush (i.e., all five cards of the same suit)

f. a straight (i.e., five cards of consecutive face values)

g. a full house (i.e., two cards of one face value and three cards of another face value)

[*Hint:* Add methods `getFace` and `getSuit` to class `Card` of Fig. 7.11.]

7.17 ***(Card Shuffling and Dealing)*** Use the methods developed in Exercise 7.16 to write an application that deals two five-card poker hands, evaluates each hand and determines which is better.

7.18 ***(Project: Card Shuffling and Dealing)*** Modify the application developed in Exercise 7.17 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down," so the player cannot see it. The application should then evaluate the dealer's hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The application should then reevaluate the dealer's hand. [*Caution:* This is a difficult problem!]

7.19 ***(Project: Card Shuffling and Dealing)*** Modify the application developed in Exercise 7.18 so that it can handle the dealer's hand automatically, but the player is allowed to decide which cards of the player's hand to replace. The application should then evaluate both hands and determine who wins. Now use this new application to play 20 games against the computer. Who wins more games, you or the computer? Have a friend play 20 games against the computer. Who wins more games? Based on the results of these games, refine your poker-playing application. (This, too, is a difficult problem.) Play 20 more games. Does your modified application play a better game?

7.20 ***(Project: Card Shuffling and Dealing)*** Modify the application of Figs. 7.11–7.13 to use `Face` and `Suit` `enum` types to represent the faces and suits of the cards. Declare each of these `enum` types as a `public` type in its own source-code file. Each `Card` should have a `Face` and a `Suit` instance variable. These should be initialized by the `Card` constructor. In class `DeckOfCards`, create an array of `Faces` that's initialized with the names of the constants in the `Face` `enum` type and an array of `Suits` that's initialized with the names of the constants in the `Suit` `enum` type. [*Note:* When you output an `enum` constant as a `String`, the name of the constant is displayed.]

7.21 ***(Fisher-Yates Shuffling Algorithm)*** Research the Fisher-Yates shuffling algorithm online, then use it to reimplement the `shuffle` method in Fig.

7.12□.

Making a Difference

7.22 ***(Target-Heart-Rate Calculator)*** While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (`http://bit.ly/TargetHeartRates`), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that's 50–85% of your maximum heart rate. [*Note:* These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and