Chapter 11

Designing Classes

11.1 Exercises

Exercise 11.1 Define a class named `Lens` that represents a camera lens by its focal length (distance from lens to sensor) and aperture (diameter of lens opening), both measured in millimeters. These will be stored in two `private double` instance variables. Implement the following:

1. A default constructor that sets both attributes to 1.0.
2. A two-argument constructor that sets the attributes as specified by the caller (setting a value to 1.0 if the argument is less than zero).
3. Write getters and setters for both attributes. Again, the mutator sets a value to 1.0 if given an argument less than zero by the caller.
4. Write an instance method named `calcFStop` that returns the lens's f-stop value by dividing the focal length by the aperture.
5. Provide a `toString` method to display the focal length and aperture, properly labeled, with one digit to the right of the decimal point.
6. Implement an `equals` instance method that takes another `Lens` object as its parameter and returns `true` if the two lenses have the same focal length and aperture, `false` otherwise.

7. The `main` program will ask the user for two sets of focal length and aperture and will create corresponding `Lens` object. It will then display the first `Lens`'s attributes and f-stop. If the second `Lens` is not equal to the first one, it will also display the second `Lens`'s attributes and f-stop; otherwise, it will output a message that the two lenses are the same.

Here is some sample output from running the program twice:

```
Enter focal length of first lens in mm: 3.5
Enter aperture of first lens in mm: 2

Enter focal length of second lens in mm: 4.3
Enter aperture of second lens in mm: 3

First lens: focal length: 3.5mm, aperture: 2.0mm; f-stop 1.8
Second lens: focal length: 4.3mm, aperture: 3.0mm; f-stop 1.4

// ===================

Enter focal length of first lens in mm: 3.5
Enter aperture of first lens in mm: 2

Enter focal length of second lens in mm: 3.5
Enter aperture of second lens in mm: 2

First lens: focal length: 3.5mm, aperture: 2.0mm; f-stop 1.8
The second lens is the same as the first one.
```

Exercise 11.2 In an n-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named `RegularPolygon` that contains:

- A private `int` data field named `nSides` that defines the number of sides in the polygon with default value 3.
- A private `double` data field named `sideLength` that stores the length of the side, with default value 1.0
- A private `double` data field named `x` that defines the x-coordinate of the polygon's center with default value 0.0

- A private `double` data field named `y` that defines the y-coordinate of the polygon's center with default value 0.0
- A no-argument constructor that creates a regular polygon with default values
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (0, 0)
- A constructor that creates a regular polygon with the specified number of sides, length of side, and x-and y- coordinates
- The accessor and mutator methods (getters and setters) for all data fields
- The method `getPerimeter` that returns the perimeter of the polygon
- The method `getArea` that returns the area of the polygon. The formula for computing the area of a regular polygon is:

$$\frac{n \times s^2}{4 \times tan(\frac{\pi}{n})}$$

- An `equals` instance method that takes another `RegularPolygon` object as its parameter and returns `true` if the two polygons have the same number of sides and side length, `false` otherwise.

Draw the UML diagram for the class, then implement the class.

Write a test program named *PolygonTest.java*. The test program will create three `RegularPolygon` objects, created using:

- The no-argument constructor
- `RegularPolygon(6, 4.0)`
- `RegularPolygon(10, 4, 5.6, 7.8)`

For each object, display its perimeter and area, properly labeled. Format the values to three decimal places.

Put the `RegularPolygon` class in the *PolygonTest.java* file rather than creating a separate file for the class.

Exercise 11.3 This program will implement a class to represent the data in a bank account; you will also write a class with a `main` method to test this class.

The `Account` class has two `private` instance variables: `acctNumber`, which is an `int`, and `balance`, which is a `double`.

Implement the following methods:

- A two-argument constructor that specifies the account number and starting balance for the account. (In this program, you won't write a no-argument constructor.) If the starting balance is less than zero, leave it unchanged—its value will be zero, because that is the default value for a `double` instance variable.

- A getter method for the account number. It will be called `getAcctNumber`. Do *not* write a setter method—once you establish an account number in the constructor, it should never be changed.

- Both a getter and setter method for the balance. If the setter is given a balance less than zero, leave the current account balance unchanged.

- A `toString` method that returns a `String` with the account number and balance, properly labeled. You must display the balance with a currency symbol and exactly two digits to the right of the decimal point.

- A `void` method named `deposit` that accepts a `double` amount as its single parameter. If the amount is negative, leave the balance untouched. Otherwise, add the amount to the balance.

- A `void` method named `withdraw` that accepts a `double` amount as its single parameter. If the amount is negative or greater than the balance, leave the balance untouched. Otherwise, subtract the amount from the balance.

None of the preceding methods prints anything. If someone gives bad input to `deposit` or `withdraw`, the caller (in this instance, `main`) is responsible for doing appropriate error handling. However, because you can't count on the users to handle errors correctly all the time, it's up to you to make sure that the methods always do something reasonable—in this case, leaving the balance alone rather than changing it to an invalid value.

Draw a UML diagram for the class and implement it.

Finally, write a class named `TestAccount` with a `main` method that does the following:

1. Create an `Account` variable for account number 1047217, with an initial balance of $1732.00.
2. Display the account variable you created in the previous step. *Hint*: use `toString`.
3. Deposit $450.25 to the account and display it.
4. Withdraw $301.75 from the account and display it.
5. Deposit -$22.33 to the account and display it. The balance should not be changed.
6. Withdraw -$44.55 from the account and display it. The balance should not be changed.
7. Withdraw $2000.00 from the account and display it. The balance should not be changed.

Here is what the output might look like:

```
Account 1047217 has balance $1732.00
Deposit $450.25: Account 1047217 has balance $2182.25
Withdraw $301.75: Account 1047217 has balance $1880.50
Deposit -$22.33: Account 1047217 has balance $1880.50
Withdraw -$44.55: Account 1047217 has balance $1880.50
Withdraw $2000: Account 1047217 has balance $1880.50
```

Exercise 11.4 This program will implement two classes, and you will draw a UML diagram for first of these classes.

First, implement a class named `InventoryItem` that contains these instance variables:

- A private `String` data field named `itemName` that gives the name of the item. Its default value is `"TBD"`.

- A private `int` data field named `sku`. An SKU (stock keeping unit) is like an ID number for an item. Its default value is zero.
- A private `double` data field named `price` that stores the price for the item. The default value is 0.0.
- A private `int` data field named `quantity` that tells how many items are in stock. The default value is 0.

Implement the following methods, all of which must be `public`:

- A no-argument constructor that creates an inventory item with default values.
- A three-argument constructor that creates an inventory item with the specified name, SKU, and price (in that order) with the default quantity.
- A four-argument constructor that creates an inventory item with the specified name, SKU, price, and quantity (in that order).
- The accessor and mutator methods (getters and setters) for all the instance fields.
- A method named `getTotalValue()` which returns, as a double, the item price times its quantity.
- A `toString()` method that returns a `String` with the item's name, SKU, price, and quantity, properly labeled. It must *not* include the total value; that is not an attribute of the object.
- A static method named `compare()` which takes as its arguments two `InventoryItem` objects. This method returns:
  - -1 if the total value of the first item is less than the total value of the second item
  - 0 if the total value of the first item equals the total value of the second item
  - 1 if the total value of the first item is greater than the total value of the second item

For example, if `item1` has a price of $2.00 and a quantity of 9, and `item2` has a price of $3.00 and a quantity of 5, `InventoryItem.compare(item1, item2)` would return 1 because the total value of `item1` ($18.00) is greater than the total value of `item2` ($15.00).

This method *must* call the `getTotalValue()` method.

Note: if the constructors and setters are given a negative price or quantity, they must convert it to a positive number. Hint - use `Math.abs()`.

Draw a UML diagram for this class.

Next, implement the `TestInventory` class, which will contain your `main()` method and will do the following:

- Create four `InventoryItems`:
  - `emptyItem`, using the no-argument constructor
  - `staplers`, which has a name of "Stapler, Red", SKU of 91745, and price of $7.89, using the three-argument constructor
  - `pencils`, which has a name "Pencil, #2", SKU of 73105, price of $0.35, and quantity 210
  - `notebooks`, which has a name "Notebook, Spiral", SKU of 68332, price of $2.57, and quantity 38
- Display each of the items (using the `display()` method) and its total value.
- Compare `pencils` to `notebooks` and prints out which one has greater total value. You must call the `compare()` method as part of this step . Output must use the inventory item's `itemName` property.

Here is what output from the program might look like. Your output does not have to look exactly like this, but it must reflect the same information.

```
TBD [SKU 0]: 0 at $0.00 each
Total value: $0.00

Stapler, Red [SKU 91745]: 0 at $7.89 each
Total value: $0.00

Pencil, #2 [SKU 73105]: 210 at $0.35 each
Total value: $73.50

Notebook, Spiral [SKU 68332]: 38 at $2.57 each
Total value: $97.66

Notebook, Spiral has greater value than Pencil, #2
```

Exercise 11.5 One advantage of making instance variables `private` is that they allow you to hide the implementation details from users. You provide an API (application program interface—a set of methods for accessing the data), and users interact with the data through those methods. This frees you to change the underlying implementation at any time.

For example, consider a three-dimensional vector. You might be tempted to create a class with these instance variables `x`, `y`, and `z`, as in the UML diagram of Figure 11.1:

The class has a no-argument constructor (which sets all the coordinates to zero) and a three-argument constructor to set the x, y, and z coordinates explicitly.

These are followed by the getters (accessors) and setters (mutators) for each of the dimensions.

The class specifies an `add`, `dotProduct`, and `distance` instance methods that find the sum, dot product, and distance of the current vector and an "'other" vector.

For the convenience of users, the class also specifies `static` versions of these methods (the convention in UML diagrams is to underline `static` elements) where you specify both vectors you want to manipulate.

But wait—you already have code for doing addition, dot product, and vector distance from Exercise 7.5. Because your instance variables are private, you

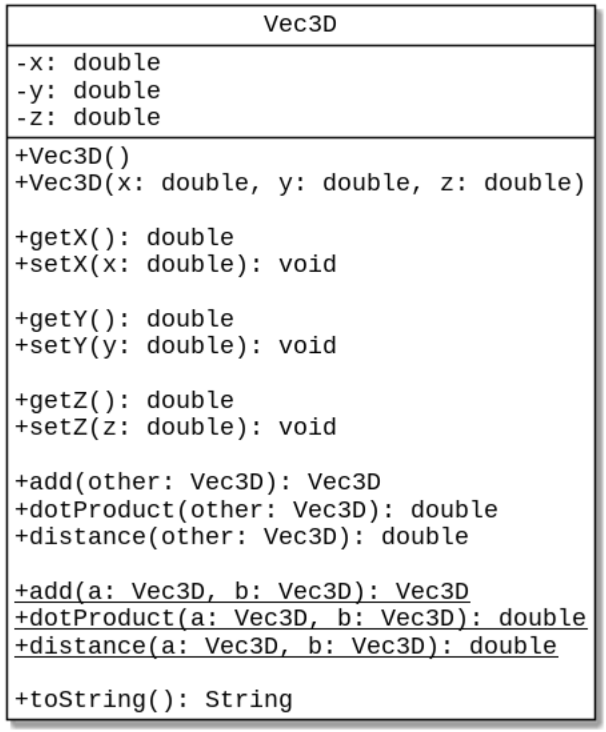

Figure 11.1: UML Diagram with three separate instance variables

can replace `x`, `y`, and `z` with a three-element array of `double` values, as in Figure 11.2, and use the code that you wrote in that exercise.

Notice that the `public` methods have not changed; users will never know that the vector is implemented as an array rather than three individual variables.

And that's your job for this exercise: implement the `Vec3D` class with a `private` array to hold the coordinates. Then, write a `main` method that will ask the user to enter two vectors and then display the sum, dot product, and distance between the vectors. To avoid repetitious code, you might want to write a `getVector` method that has a prompt and a `Scanner` as its parameters. This method will prompt the user for the three vector components and return a `Vec3D` object.

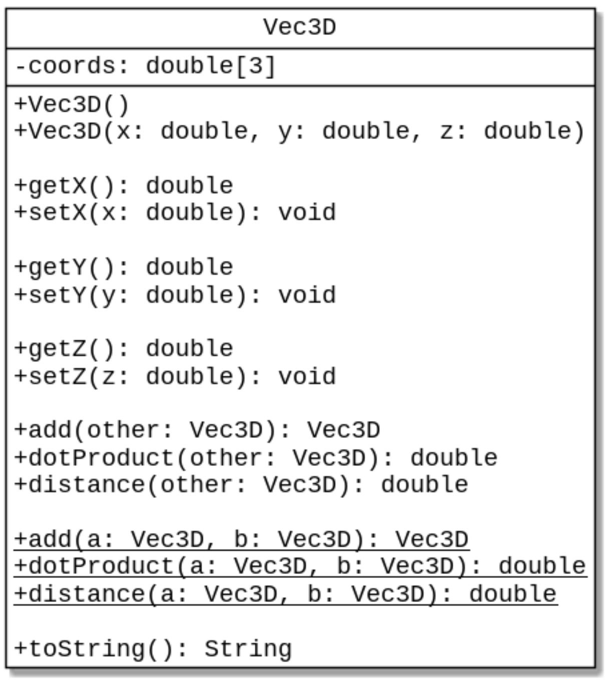

Figure 11.2: UML Diagram with array to hold coordinates