Designing Object Classes

16

16.1 Packaging a Group of Data as an Object

16.1.1 The Position of a Game Piece

In Java, object classes are used to assemble a set of data as a unit.

The header of the source code for an object class is the same as the header of the source code for a non-object class. Specifically, the header takes the form of `public class CLASS_NAME`. Declarations about implementation and extension (see Chap. 17) may appear after the class name. The possible components of an object class are: **instance variables** (defined as if they were global variables), **instance methods**, **constructors**, constants, and static methods.

In this chapter, we explore how to write a source code for an object class. We start with a series of programs for manipulating the position of a game piece placed on a game board. We assume that the board has two-dimensional integral coordinates (that is, the position of a piece is represented as a pair of integers), for instance, as shown in Fig. 16.1.

The user specifies the initial position of the piece, and then interacts with the program to change its position. The new position of the piece is specified with the numbers of squares that the piece must move along the x- and y-axes. If the move is possible, the program updates the position. The program then asks whether or not the user wants to move the piece again. If the user answers "yes", the series of actions is repeated; otherwise, the program terminates.

The following program, `UsePositionZero`, is the initial version of the program. The program does not use a custom object class. It uses two `int` variables, `xPos` and `yPos`, for representing the position (Line 7). The program receives the initial values for the two variables in Lines 9–11. Then the program enters a do-while loop (Line 14). In the loop-body, the program first reports the present position of the piece. In reporting the position, the program uses the format `(%d,%d)` with `xPos` and `yPos` supplied to the two placeholders (Line 15). The program then asks the user to enter the amounts of move, and changes the coordinates with the amounts that the user enters (Lines 17–19). The program then reports the new position using the same format (Line 21). The program then asks the user whether or not the program should continue (Line 23). The response is stored in a `String` variable named `answer` (Line 24). The loop is terminated if the response does not start with `"y"` (Line 25).

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_16

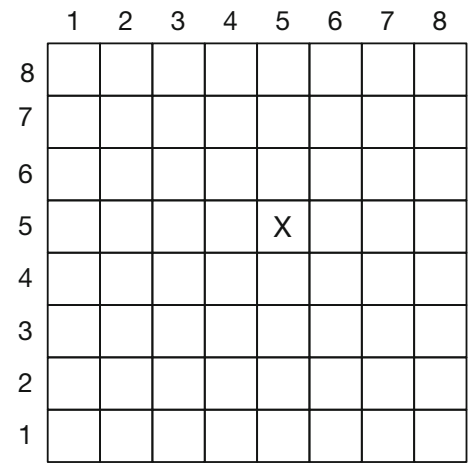

Fig. 16.1 An 8 × 8 game board. The X represents a position of a game piece

```
1  import java.util.*;
2  public class UsePositionZero
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      int xPos, yPos;
8
9      System.out.print( "Enter the initial x y: " );
10     xPos = keyboard.nextInt();
11     yPos = keyboard.nextInt();
12
13     String answer = "";
14     do {
15       System.out.printf( "The current position: (%d,%d)%n", xPos, yPos );
16
17       System.out.print( "By how much do you want to shift x and y? " );
18       xPos += keyboard.nextInt();
19       yPos += keyboard.nextInt();
20
21       System.out.printf( "The new position: (%d,%d)%n", xPos, yPos );
22
23       System.out.print( "Continue (y/n)? " );
24       answer = keyboard.next();
25     } while ( answer.startsWith( "y" ) );
26   }
27 }
```

Listing 16.1 The initial version of the program that manipulates the position of a game piece

Here is an example of executing `UsePositionZero`:

```
Enter the initial x y: 1 3
The current position: (1,3)
By how much do you want to shift x and y? 2 7
The new position: (3,10)
Continue (y/n)? y
The current position: (3,10)
By how much do you want to shift x and y? 10 -8
The new position: (13,2)
Continue (y/n)? y
The current position: (13,2)
By how much do you want to shift x and y? -1 -1
The new position: (12,1)
Continue (y/n)? n
```

We want to write new versions of `UsePositionZero`, where the two coordinates are treated as one unit. To accomplish this, we define an object class `PositionFirst`. An object of `PositionFirst` has two data components, `x` and `y`, representing the x- and y-coordinates of the position. Both `x` and `y` are `int` variables. They are declared as instance variables. Figure 16.2 visualizes the way `PositionFirst` encompasses the two instance variables in it.

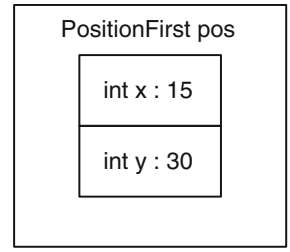

Fig. 16.2 The instance variables of a `PositionFirst` object

Here is the source code of `PositionFirst`. The declarations of `x` and `y` are given in Line 3 of the source

```
3   int x, y;
```

The lack of the `static` attribute signifies that they are instance variables.

The constructor for `PositionFirst` starts at Line 4.

```
4   public PositionFirst( int xValue, int yValue )
5   {
6     x = xValue;
7     y = yValue;
8   }
```

The constructor stores the values `xValue` and `yValue` given as the formal parameters in the instance variables `x` and `y`.

The general format of a constructor is

```
public CLASS_NAME( PARAMETERS )
```

This format is different from the format of method declarations, since there is no return type.

There is only one instance method in PositionFirst. The method is named shift (Lines 9–12). The method shift has two formal parameters, xDiff and yDiff, and adds their values to the instance variables. The header of the method does not have the static attribute. The lack of the attribute signifies that the method is an instance method.

```
1  public class PositionFirst
2  {
3    int x, y;
4    public PositionFirst( int xValue, int yValue )
5    {
6      x = xValue;
7      y = yValue;
8    }
9    public void shift( int xDiff, int yDiff )
10   {
11     x += xDiff;
12     y += yDiff;
13   }
14 }
```

Listing 16.2 Class PositionFirst

Recall that the use of a constructor requires the special keyword of new. For a pair of int data startX and startY,

```
new PositionFirst( startX, startY )
```

produces a new PositionFirst object whose instance variables x and y have values startX and startY. For example,

```
PositionFirst myPos = new PositionFirst( 1, 3 );
```

constructs a PositionFirst object with coordinates (1,3) and assigns it to myPos.

The class PositionFirst does not have the method main. We write a separate class, UsePositionFirst, for handling the interactions with the user. To run UsePositionFirst, both PositionFirst.java and UsePositionFirst.java need to be compiled. The two files can be compiled individually:

```
1  % javac PositionFirst.java
2  % javac UsePositionFirst.java
```

or jointly:

```
% javac PositionFirst.java UsePositionFirst.java
```

The source code of UsePositionFirst is an adaptation of the source code of UsePosition Zero. The program uses a PositionFirst object, pos, to store coordinates (Line 7). To instantiate pos, the constructor of PositionFirst is called (Line 10). The actual parameters of the constructor are two calls of keyboard.nextInt() (Line 10). The return value of the first call keyboard.nextInt() becomes the first actual parameter, ad the return value of the second call keyboard.nextInt() becomes the second actual parameter. The x- and y- coordinates of pos are accessed with pos.x and pos.y (Lines 14 and 18). Moving the position is accomplished by calling the method shift (Line 16). The actual parameters of the method call are two calls to keyboard.nextInt() (Line 16).

```
1  import java.util.*;
2  public class UsePositionFirst
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      PositionFirst pos;
8
9      System.out.print( "Enter the initial x y: " );
10     pos = new PositionFirst( keyboard.nextInt(), keyboard.nextInt() );
11     String answer = "";
12     do {
13       System.out.printf( "The current position: (%d,%d)%n",
14           pos.x, pos.y );
15       System.out.print( "By how much do you want to shift x and y? " );
16       pos.shift( keyboard.nextInt(), keyboard.nextInt() );
17
18       System.out.printf( "The new position: (%d,%d)%n", pos.x, pos.y );
19       System.out.print( "Continue (y/n)? " );
20       answer = keyboard.next();
21     } while ( answer.startsWith( "y" ) );
22   }
23 }
```

Listing 16.3 A program that plays with the position of a game piece using an object of type `PositionFirst`

Here is an execution example of the program:

```
1  Enter the initial x y: -100 -100
2  The current position: (-100,-100)
3  By how much do you want to shift x and y? 1000 1000
4  The new position: (900,900)
5  Continue (y/n)? y
6  The current position: (900,900)
7  By how much do you want to shift x and y? 345 9870
8  The new position: (1245,10770)
9  Continue (y/n)? y
10 The current position: (1245,10770)
11 By how much do you want to shift x and y? -1397 760
12 The new position: (-152,11530)
13 Continue (y/n)? n
```

16.1.2 Private Instance Variables and the `toString` Method

The class `PositionFirst` permits direct access to the instance variables via the attachment of their names, for instance, `pos.x`. This direct access is utilized in Lines 14 and 18 of the source code of `UsePositionFirst`, where the program reports the position of the piece. However, the direct accessibility of instance variables can be problematic for the following reasons:

- If the name and/or type of an instance variable changes, all the direct accesses to the variable appearing in other source codes must be updated accordingly. For instance, if the name `x` in `PositionFirst` changes to `xxx`, the two occurrences of `pos.x` in `UsePositionFirst` must be changed to `pos.xxx`.
- In the case where an instance variable is a number and is directly accessible, an attachment of a short-hand expression to a direct access to the variable may modify the value inadvertently. Such an error is often difficult to identify. For instance, suppose that a programmer intends to store the value of `pos.x + 1` in `r`, but accidentally writes `r = pos.x ++` instead of `r = pos.x + 1`. The execution of the assignments stores the value of `pos.x` in `r` and adds 1 to `pos.x`.

These problems can be avoided by making the instance variables inaccessible from any class other than `PositionFirst`, and providing alternatives for accessing and modifying their values. To make an instance variable inaccessible from other source codes, an attribute of `private` must be attached in its declaration as follows:

```
    private int x, y;
```

Suppose this modification has been applied to the instance variables, `x` and `y`, of `PositionFirst`. Then, `PositionFirst` compiles, but `UsePositionFirst` does not. Here is the compilation error message:

```
1  UsePositionFirst.java:13: error: x has private access in PositionFirst
2        System.out.printf( "The current position: (%d,%d)%n", pos.x, pos.y
             );
3                                                                 ^
4  UsePositionFirst.java:13: error: y has private access in PositionFirst
5        System.out.printf( "The current position: (%d,%d)%n", pos.x, pos.y
             );
6                                                                       ^
7  UsePositionFirst.java:18: error: x has private access in PositionFirst
8        System.out.printf( "The new position: (%d,%d)%n", pos.x, pos.y );
9                                                          ^
10 UsePositionFirst.java:18: error: y has private access in PositionFirst
11       System.out.printf( "The new position: (%d,%d)%n", pos.x, pos.y );
12                                                               ^
13 4 errors
```

The message states that the instance variables of `PositionFirst` are private and so cannot be accessed.

By reading the error message, we learn that the errors originate from the places where the program attempts to access the values of `x` and `y`. To resolve the problem, we introduce instance methods, `getX` and `getY`, that return the values of `x` and `y`.

Here is the class `PositionPrivate` in which the above modifications have been incorporated. The methods `getX` and `getY` appear in Lines 14–17 and Lines 18–21. Both take no parameters and execute just one statement. The actions to be performed by the constructor are the same as before (Lines 6 and 7). The method `shift` is untouched (Lines 9–13).

```
1  public class PositionPrivate
2  {
3    private int x, y;
4    public PositionPrivate( int xValue, int yValue )
5    {
6      x = xValue;
7      y = yValue;
8    }
9    public void shift( int xDiff, int yDiff )
10   {
11     x += xDiff;
12     y += yDiff;
13   }
14   public int getX()
15   {
16     return x;
17   }
18   public int getY()
19   {
20     return y;
21   }
22 }
```

Listing 16.4 Class `PositionPrivate`

We call methods that offer access to the values of instance variables `getters` (or `accessors`), and methods that offer ways to change the values of instance variables `setters` (or `modifers`). In `PositionPrivate`, `getX` and `getY` are getters and `shift` is a setter.

The program `UsePositionPrivate` is derived from `UsePositionFirst` by changing the type of `pos` to `PositionPrivate` (Lines 7 and 10) and changing the direct accesses `pos.x` and `pos.y` to their respective instance methods `pos.getX()` and `pos.getY()` (Lines 14 and 20).

```
1  import java.util.*;
2  public class UsePositionPrivate
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      PositionPrivate pos;
8
9      System.out.print( "Enter the initial x y: " );
10     pos = new PositionPrivate ( keyboard.nextInt(), keyboard.nextInt() );
11     String answer = "";
```

Listing 16.5 Class `UsePositionPrivate` (part 1)

```
12      do
13      {
14        System.out.printf( "The current position: (%d,%d)%n",
15            pos.getX(), pos.getY() );
16        System.out.print( "By how much do you want to shift x and y? " );
17        pos.shift( keyboard.nextInt(), keyboard.nextInt() );
18
19        System.out.printf( "The new position: (%d,%d)%n",
20            pos.getX(), pos.getY() );
21        System.out.print( "Continue (y/n)? " );
22        answer = keyboard.next();
23      } while ( answer.startsWith( "y" ) );
24    }
25  }
```

Listing 16.6 Class `UsePositionPrivate` (part 2)

Again, from the user's point of view, the program works exactly the same way as before.

```
1  Enter the initial x y: -23 54
2  The current position: (-23,54)
3  By how much do you want to shift x and y? 23 -54
4  The new position: (0,0)
5  Continue (y/n)? y
6  The current position: (0,0)
7  By how much do you want to shift x and y? 0 4
8  The new position: (0,4)
9  Continue (y/n)? y
10 The current position: (0,4)
11 By how much do you want to shift x and y? 4 -100
12 The new position: (4,-96)
13 Continue (y/n)? n
```

16.1.3 Using Constants in an Object Class

The position programs we have seen so far allow an arbitrary value for each coordinate. However, in a real game, the area in which the game pieces are placed is finite. For instance, in Chess, the area is 8 by 8, and in the Japanese Chess, Shogi, the area is 9 by 9. Based on this observation, we modify `PositionPrivate` to create a new version, `PositionConfined`, where the coordinates have an upper bound and a lower bound. The constants that define the lower and upper bounds are `MINIMUM` and `MAXIMUM`. They are defined as follows:

```
1   public static final int MINIMUM = 1;
2   public static final int MAXIMUM = 8;
```

These constants are public, so their values can be accessed from outside by attaching their names to `PositionConfined`. For example, an external program can print the values of these constants by:

```
1   System.out.println( "The maximum is " + PositionConfined.MAXIMUM );
2   System.out.println( "The minimum is " + PositionConfined.MINIMUM );
```

The code produces the following output:

```
1 The maximum is 8
2 The minimum is 1
```

Having bounds on the coordinates means that we need to enforcing the bounds. There are two places for the user to suggest an invalid position, when the user specifies the initial position and when the user specifies the amounts of movement. We handle the two situations as follows:

- If the suggested initial position is outside the boundary, we use some default position.
- If the suggested movement will push the position outside the boundary, we do not move the piece.

The default coordinates are defined with two additional constants, `DEFAULT_X` and `DEFAULT_Y`. Both are `int` type and have the value of 1. In the program, they are defined as follows:

```
1   public static final int DEFAULT_X = 1;
2   public static final int DEFAULT_Y = 1;
```

We introduce a new method `inRange` (Line 10) for checking if a suggested value for a coordinate is in the range of valid coordinate values. `inRange` receives an `int` value `z` as its formal parameter, and returns a `boolean` value representing whether or not `z` is in the range. More precisely, the method returns `z >= MINIMUM && z <= MAXIMUM` (Line 12). The variable `z` can be for x or for y. We make `inRange` a public static method. This means that the method is available to other source codes without having to instantiate a `PositionConfined` object. The usage of the message from outside should be like: `PositionConfined.inRange( 10 )`.

Using `inRange`, the constructor now works as follows:

- Tentatively assign the default values to the instance variables (Lines 17 and 18).
- If both parameters are valid (Line 19), assign the values of the parameters to the instance variables.

The new version of `shift` uses `inRange` to check whether or not the suggested coordinates, `x + xDiff` and `y + xDiff`, are valid (Line 28). If both are valid, the method updates the instance variables with the suggested coordinates, and returns `true` (Lines 30–32). Otherwise, the method retains the present values and returns `false` (Line 34). The return type of `shift` has been changed from `void` to `boolean`.

The methods `getX` and `getY` are the same as before (Lines 36–43). A new method has been introduced in `PositionConfined`. The name of the method is `toString`. The method `toString` returns the `String` that presents the two coordinate values in the format we have been using to print the positions (Lines 44–47). The formatted `String` is produced using `String.format`.

```
1  public class PositionConfined
2  {
3    public static final int MINIMUM = 1;
4    public static final int MAXIMUM = 8;
5    public static final int DEFAULT_X = 1;
6    public static final int DEFAULT_Y = 1;
7
8    private int x, y;
9
10   private static boolean inRange( int z )
11   {
12     return z >= MINIMUM && z <= MAXIMUM;
13   }
14
15   public PositionConfined( int xValue, int yValue )
16   {
17     x = DEFAULT_X;
18     y = DEFAULT_Y;
19     if ( inRange( xValue ) && inRange( yValue ) )
20     {
21       x = xValue;
22       y = yValue;
23     }
24   }
25
26   public boolean shift( int xDiff, int yDiff )
27   {
28     if ( inRange( x + xDiff ) && inRange( y + yDiff ) )
29     {
30       x += xDiff;
31       y += yDiff;
32       return true;
33     }
34     return false;
35   }
36   public int getX()
37   {
38     return x;
39   }
40   public int getY()
41   {
42     return y;
43   }
44   public String toString()
45   {
46     return String.format( "(%d,%d)", x, y );
47   }
48 }
```

Listing 16.7 Class `PositionConfined`

Now that we have revised `PositionPrivate` to `PositionConfined`, we revise `UsePositionPrivate` too. The new program is `UsePositionConfined`. The new version uses `PositionConfined` instead of `PositionPrivate`. The program captures the return value of the method `shift`, and stores it in a `boolean` variable, `res` (Line 18). If the value of `res` is `true`, the program prints the new position (Lines 19–22). Otherwise, it prints a special message `"-----UNSUCCESSFUL-----"` (Lines 23–26).

```
import java.util.*;
public class UsePositionConfined
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    PositionConfined pos;

    System.out.print( "Enter the initial x y: " );
    pos = new PositionConfined(
        keyboard.nextInt(), keyboard.nextInt() );
    String answer = "";
    do
    {
      System.out.println( "The current position: " + pos.toString() );
      System.out.print( "By how much do you want to shift x and y? " );

      boolean res = pos.shift( keyboard.nextInt(), keyboard.nextInt() );
      if ( res )
      {
        System.out.println( "The new position: " + pos.toString() );
      }
      else
      {
        System.out.println( "--------UNSUCCESSFUL--------" );
      }
      System.out.print( "Continue (y/n)? " );
      answer = keyboard.next();
    } while ( answer.startsWith( "y" ) );
  }
}
```

Listing 16.8 Class `UsePositionConfined`

Here is an execution example of the program. The error message is generated two times during the execution.

```
Enter the initial x y: 4 4
The current position: (4,4)
By how much do you want to shift x and y? 3 3
The new position: (7,7)
Continue (y/n)? y
The current position: (7,7)
By how much do you want to shift x and y? 10 20
--------UNSUCCESSFUL--------
Continue (y/n)? y
The current position: (7,7)
By how much do you want to shift x and y? -7 -7
--------UNSUCCESSFUL--------
Continue (y/n)? y
The current position: (7,7)
By how much do you want to shift x and y? 1 -2
The new position: (8,5)
Continue (y/n)? n
```

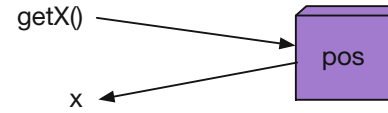

Fig. 16.3 A black box. What is inside the box cannot be seen from the user

16.1.4 Information Hiding

An important concept in the design of object classes is **information hiding** (or **black box**). Information hiding refers to the idea that, by making its instance variables private, the users of an object class can be oblivious to how the information is represented in the object class and how the information is manipulated (see Fig. 16.3). For instance, a programmer of `UsePositionConfined` needs to know how to call the constructor and instance methods of `PositionConfined`, but does not need to know how the two coordinates are represented in the class. Using information hiding, the task of writing a source code for a program can be easily split among multiple programmers.

To demonstrate how information hiding works, we rewrite the class `PositionConfined` into `PositionFancy`. `PositionFancy` combines the two coordinate values, `x` and `y`, into a single `int` variable using the formula `16 * x + y`. The first part of the code defines the same constants and the `inRange` method as `PositionConfined`:

```
1  public class PositionFancy
2  {
3    public static final int MINIMUM = 1;
4    public static final int MAXIMUM = 8;
5    public static final int DEFAULT_X = 1;
6    public static final int DEFAULT_Y = 1;
7
8    private static boolean inRange( int z )
9    {
10     return z >= MINIMUM && z <= MAXIMUM;
11   }
12
```

Listing 16.9 Class `PositionFancy` (part 1). The constants and the method `inRange`

The second part of the source code presents how to go back and forth between two values and on value. The name of our only instance variable is `w` (Line 11). In representing the multiplicative factor of 16, the program uses a constant named `SCALE` (Line 13). Given `x` and `y`, the program combines the two using the formula `x * SCALE + y`. The method `combine` is a private instance method that combines two coordinate values, and stores the result in `w` (Lines 17–20). Because of the change in the encoding, both `getX` and `getY` need different codes. The method `getX` returns `w / SCALE` (Lines 19–22), and the method `getY` returns `w % SCALE` (Lines 23–26).

```
1   private int w;
2
3   private static final int SCALE = 16;
4
5   private void combine( int x, int y )
6   {
7     w = x * SCALE + y;
8   }
9   public int getX()
10  {
11    return w / SCALE;
12  }
13  public int getY()
14  {
15    return w % SCALE;
16  }
17
```

Listing 16.10 Class `PositionFancy` (part 2). The part responsible for combining the two coordinate values and splitting the combined number into two coordinate values

Next we see the constructor (Lines 30–37) and the method `shift` (Lines 39–47). The actions of the form `x = VALUE1` and `y = VALUE2` have been substituted with a method call, `combine( VALUE1, VALUE2 )`.

```
30  public PositionFancy( int xValue, int yValue )
31  {
32    combine( DEFAULT_X, DEFAULT_Y );
33    if ( inRange( xValue ) && inRange( yValue ) )
34    {
35      combine( xValue, yValue );
36    }
37  }
38
39  public boolean shift( int xDiff, int yDiff )
40  {
41    if ( inRange( getX() + xDiff ) && inRange( getY() + yDiff ) )
42    {
43      combine( getX() + xDiff, getY() + yDiff );
44      return true;
45    }
46    return false;
47  }
48
```

Listing 16.11 Class `PositionFancy` (part 3). The constructor and the method `shift`

Next we see the new version of the method `toString`. In building the `String` to be returned, `toString` uses the instance methods `getX` and `getY`.

```
49    public String toString()
50    {
51      return String.format( "(%d,%d)", getX(), getY() );
52    }
53
```

Listing 16.12 Class `PositionFancy` (part 4). The method `toString`

The last part of the code is a new method, `toTable`, that produces the position of the piece using eight lines of `String`, where the position appears as a `'#'` in an 8-by-8 square of `'.'`. The eight lines correspond to the y-coordinates of 8, 7, ..., 1 from top to bottom. For example, `(7,2)` is represented as:

```
........
........
........
........
........
........
......#.
........
```

and `(3,8)` is represented as:

```
..#.....
........
........
........
........
........
........
........
```

To generate this visual encoding, `PositionFancy` uses two `char` constants, `HERE` and `ELSEWHERE` (Lines 54 and 55). `HERE` is the character that appears at the position represented by the `PositionFancy` object `pos` and `ELSEWHERE` is the character that appears at other places in the 8-by-8 diagram. Defining the two types of characters as constants makes it easy to make changes in the future.

To build the output, the program uses a `StringBuilder` object, `builder`. First, the program obtains the coordinate values using the getters and stores them in variables `x` and `y` (Lines 59 and 60). The program uses a double for-loop in building the contents of the `StringBuilder` object. The external loop iterates over the decreasing sequence `MAXIMUM, ..., MINIMUM` with a variable named `i` (Line 62). The internal loop iterates over the increasing sequence `MINIMUM, ..., MAXIMUM` with a variable named `j` (Line 64). In the body of the internal loop, if `( i == y && j == x )` (Line 66), the method appends `HERE` to `builder` (Line 68); otherwise, it appends `ELSEWHERE` (Line 72). Each time the internal loop concludes, the method appends `'\n'` (Line 75). At the conclusion of the external loop, the method returns the `String` represented by `builder`, obtained by calling the method `toString` (Line 77).

```
54    public static final char HERE = '#';
55    public static final char ELSEWHERE = '.';
56
57    public String toTable()
58    {
59      int x = getX();
60      int y = getY();
61      StringBuilder builder = new StringBuilder();
62      for ( int i = MAXIMUM; i >= MINIMUM; i -- )
63      {
64        for ( int j = MINIMUM; j <= MAXIMUM; j ++ )
65        {
66          if ( i == y && j == x )
67          {
68            builder.append( HERE );
69          }
70          else
71          {
72            builder.append( ELSEWHERE );
73          }
74        }
75        builder.append( "\n" );
76      }
77      return builder.toString();
78    }
79  }
```

Listing 16.13 Class `PositionFancy` (part 5). The `toTable` method

The application written with `PositionFancy` is `UsePositionFancy`. In this application, the movement of the position is limited to one space, either only horizontally or only vertically. The four possible movements are thus left, up, down, and right, and they are represented with the letters `'h'`, `'j'`, `'k'`, and `'l'`. In addition, the letter `'q'` represents the termination of the program.

As before, the program asks the initial position from the user (Line 9), and then instantiates a `PositionFancy` object with the input from the user (Line 10). After this, the program enters a do-while loop (Line 13). To report the present position, the program uses the method `toTable` (Line 14). Since the `String` data that the method returns has the newline at the end, the program uses `System.out.print` instead of `System.out.println`. The program then prompts the user to enter the action to be performed (Lines 15 and 16), and receives input from the user (Line 17). The program uses a `switch` statement (Line 18) to perform the action. The method `shift` takes two parameters. The program calls `shift( -1, 0 )` for `'h'`, `shift( 0, -1 )` for `'j'`, `shift( 0, +1 )` for `'k'`, and `shift( +1, 0 )` for `'l'` (Lines 20–23). Since the y-coordinate in the visual encoding decreases from `MAXIMUM` to `MINIMUM`, "down" (the letter `'j'`) must be represented with the decrease in the value of `y`, and "up" (the letter `'k'`) must be represented with the increase in the value of `y`. The return value of the method `shift` is ignored. Since the position is visually presented and the visual presentation takes up many lines, the reporting of the new position immediately after making a move has been eliminated. The program terminates if the command chosen is `'q'` (Line 25).

```
import java.util.*;
public class UsePositionFancy
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    PositionFancy pos;

    System.out.print( "Enter the initial x y: " );
    pos = new PositionFancy( keyboard.nextInt(), keyboard.nextInt() );

    String answer = "";
    do {
      System.out.print( pos.toTable() );
      System.out.print(
          "Enter h,j,k,l to move left, down, up, right; q to quit: " );
      answer = keyboard.next();
      switch( answer.charAt( 0 ) )
      {
        case 'h': pos.shift( -1, 0 ); break;
        case 'j': pos.shift( 0, -1 ); break;
        case 'k': pos.shift( 0, +1 ); break;
        case 'l': pos.shift( +1, 0 ); break;
      }
    } while ( !answer.startsWith( "q" ) );
  }
}
```

Listing 16.14 Class `UsePositionFancy`

Here is an execution example of the code:

```
Enter the initial x y: 1 2
........
........
........
........
........
........
#.......
........
Enter h,j,k,l to move left, down, up, right; q to quit: j
........
........
........
........
........
........
........
#.......
```

```
Enter h,j,k,l to move left, down, up, right; q to quit: j
........
........
........
........
........
........
........
#.......
Enter h,j,k,l to move left, down, up, right; q to quit: k
........
........
........
........
........
........
#.......
........
Enter h,j,k,l to move left, down, up, right; q to quit: k
........
........
........
........
........
#.......
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
.#......
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
..#.....
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
...#....
........
........
```

```
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
....#...
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
.....#..
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
......#.
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
.......#
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: l
........
........
........
........
........
.......#
........
........
Enter h,j,k,l to move left, down, up, right; q to quit: q
```

16.2 An Object Class Representing a Bank Account

Here, we write an application for keeping track of the balances of bank accounts. The application allows the user to create a list of bank accounts, and then execute transactions on individual accounts or between two accounts on the list. An account is represented by the account name and the balance. We select `String` as the data type for the former and `int` as the data type for the latter. We assume that the balance is represented in cents. The largest balance that an `int` can represent is greater than 2 billion cents, or 20 million dollars. The amount is sufficient for our application.

The class `BankAccount` has two instance variables. They are a `String` variable, `name`, and an `int` variable, `balance`. The class has the following five instance methods:

- `getName()` returns a `String` data representing the account name.
- `getBalance()` returns an `int` data representing the balance.
- `getBalanceString()` returns a `String` data representing the balance in dollars, with the dollar sign and the currency punctuation.
- `deposit( int amount )` attempts to deposit the amount represented in the parameter and returns a `boolean` value representing whether or not the deposit was successfully made. The method does not change the balance if the amount is not positive.
- `withdraw( int amount )` attempts to withdraw the amount represented in the parameter and returns a `boolean` value representing whether or not the withdrawal was successfully made. The method does not change the balance if the amount is either nonpositive or greater than the balance.

A source code for the class `BankAccount` appears next. All the instance variables are private. The constructor receives two formal parameters, a `String` value, `name`, and an `int` value, `startBalance`. The latter represents the initial balance of the account. The constructor stores these values in their respective instance variables. However, the value provided for `balance` may be nonpositive. If that happens, the constructor uses 1 cent as the initial balance. This choice is expressed as the assignment `balance = Math.max( 1, startBalance )` appearing in Line 9. As for the account name, since the instance variable and the parameter have the same names, they have to be distinguished. This is accomplished by attaching the prefix `this.` to the instance variable. This appears in Line 8:

```
this.name = name;
```

The assignment without `this.`, i.e.,

```
name = name;
```

stores the value of the formal parameter `name` to itself (this is legitimate, since formal parameters can be treated as variables). This means that the instance variable `name` retains its default value `null`. The prefix `this.` means "the instance method or the instance variable of this class". In the source code of an object class, the prefix `this.` can be attached to any reference to its instance variable and any reference its instance methods.

The five instance methods are written as follows: The method `getName` returns the value of `name` (Lines 12–15) and the method `getBalance` returns the value of `balance` (Line 18). For `getBalanceString`, the program splits the amount into the dollar amount and the residual dollar amount by dividing `balance` by 100. The quotient is used as the former, and the remainder is used as the latter. These two quantities are given to `String.format` with the format of `$%,d.%02d`, and the `String` generated is returned (Line 22). In `deposit` (Line 25), if the value of `amount` at most 0 (Line 27), `false` is immediately returned (Line 28). Otherwise, the method adds the amount to `balance` (Line 30) and returns `true` (Line 31). In `withdraw`, if the amount is strictly negative or the amount exceeds the balance (Line 36), the method returns `false` (Line 38). Otherwise, the method subtracts the amount from the balance, and then returns `true` (Lines 39 and 40).

```
1  public class BankAccount
2  {
3    private String name;
4    private int balance; // in cents
5
6    BankAccount( String name, int startBalance )
7    {
8      this.name = name;
9      balance = Math.max( 1, startBalance );
10     // since the balance have to be positive
11   }
12   public String getName()
13   {
14     return name;
15   }
16   public int getBalance()
17   {
18     return balance;
19   }
20   public String getBalanceString()
21   {
22     return String.format( "$%,d.%02d", balance / 100, balance % 100 );
23   }
24
25   public boolean deposit( int amount )
26   {
27     if ( amount <= 0 ) {
28       return false;
29     }
30     balance += amount;
31     return true;
32   }
33
34   public boolean withdraw( int amount )
35   {
36     if ( amount <= 0 || amount > balance ) {
37       return false;
38     }
39     balance -= amount;
40     return true;
41   }
42 }
```

Listing 16.15 Class `BankAccount`

`SomeBankAccounts` is our application program. `SomeBankAccounts` receives information for some bank accounts from the user, and then interacts with the user to make transactions on the accounts. At the start of the program, the user specifies how many bank accounts are to be handled. The user then provides the names and the initial balances of the accounts. The program enters a loop in which it interacts with the user to perform transactions. The available actions in each round are: (a) making a deposit to one account, (b) making a withdrawal from one account, and (c) transferring money from one account to another. If the requested action fails, the program reports the failure. Here are the methods that perform the three actions.

The method `deposit` attempts to make a deposit of an amount given in the second parameter to the account given in the first parameter (Line 4). To accomplish this, the method calls the instance method `deposit` on the specified account, and then checks its return value (Line 6). If the return value is `false`, the method prints an error message (Line 8).

The method `withdraw` works in a similar manner. The method attempts to make a withdrawal of the amount given in the second parameter from the account given in the first parameter (Line 12). To accomplish this, the method calls the instance method `deposit` on the specified account, and then checks its return value (Line 14). If the return value is `false`, the method prints an error message (Line 16).

The method `transfer` attempts to withdraw the amount given in the third parameter from the account given in the first parameter, and then deposits it to the account given in the second parameter (Line 20). To accomplish the task, the method first attempts the withdrawal (Line 23). If this attempt is successful, the deposit can be made without checking, so the method makes the deposit (Line 25). Otherwise, the method prints an error message (Line 29).

```
1  import java.util.*;
2  public class SomeBankAccounts
3  {
4    public static void deposit( BankAccount acc, int amount )
5    {
6      if ( !acc.deposit( amount ) )
7      {
8        System.out.println( "Deposit unsuccessful" );
9      }
10   }
11
12   public static void withdraw( BankAccount acc, int amount )
13   {
14     if ( !acc.withdraw( amount ) )
15     {
16       System.out.println( "Withdrawal unsuccessful" );
17     }
18   }
19
20   public static void transfer( BankAccount from, BankAccount to,
21       int amount )
22   {
23     if ( from.withdraw( amount ) )
24     {
25       to.deposit( amount );
26     }
27     else
28     {
29       System.out.println( "Transfer unsuccessful" );
30     }
31   }
32
```

Listing 16.16 A program that manipulates with a list of bank accounts (part 1). The methods `deposit`, `withdraw`, and `transfer`

The method `initial` handles the initialization of the individual accounts on the list. The method returns a `BankAccount` object (Line 33). The parameter of the method `index` is an `int` value presumably representing the index to the bank account that is to be generated. The value `index` is used just for presenting a prompt (Line 37). The method receives the name using `nextLine` (Line 38), and the initial balance (Lines 39 and 40). The method uses `nextLine` to receive an input line, and then converts the input line to an `int` value using `Integer.parseInt`. The method then combines the values to instantiate a `BankAccount` object, and then returns this object.

The method `actionPrompt` prints the prompt. The method receives an array of `BankAccount` objects as its formal parameter (Line 44). The method prints the name and the balance of all the accounts (Lines 46–52). To obtain the name and the balance, the method uses `getName` and `getBalanceString` of `BankAccount`. After printing the account information, the method presents the available actions (Lines 53–57). The actions have numbers from 1 to 4, where 4 is for terminating the program.

```
33   public static BankAccount initial( int index )
34   {
35     Scanner keyboard = new Scanner( System.in );
36
37     System.out.printf( "Enter name for account #%d: ", index );
38     String name = keyboard.nextLine();
39     System.out.printf( "Enter amount for account #%d: ", index );
40     int amount = Integer.parseInt( keyboard.nextLine() );
41     return new BankAccount( name, amount );
42   }
43
44   public static void actionPrompt( BankAccount[] allAccounts )
45   {
46     System.out.println( "------" );
47     for ( int i = 0; i < allAccounts.length; i ++ )
48     {
49       System.out.printf( "%2d: %s has the balance of %s%n", i,
50         allAccounts[ i ].getName(),
51         allAccounts[ i ].getBalanceString() );
52     }
53     System.out.println( "-----Select action" );
54     System.out.println( "1. Deposit to an Account" );
55     System.out.println( "2. Withdrawal from an Account" );
56     System.out.println( "3. Transfer Between Accounts" );
57     System.out.println( "4. Quit" );
58   }
59
```

Listing 16.17 A program that manipulates with a list of bank accounts (part 2). The methods `initial` and `actionPrompt`

The final part of the program is the method `main` (Line 60). The method receives the number of accounts, `size`, from the user (Lines 64 and 65), and instantiates an array of `BankAccount` objects, `theAccounts`, having `size` elements (Line 66). The method uses a for-loop that iterates over the

sequence `0, ..., size - 1` with a variable named `i`. For each value of `i`, the method calls `initial( i )`, and stores the returned `BankAccount` object in `theAccounts[ i ]` (Lines 67–70).

```
60    public static void main( String[] args )
61    {
62      Scanner keyboard = new Scanner( System.in );
63      int choice, amount, size, index, index2;
64      System.out.print( "Enter the Number of Accounts: " );
65      size  = Integer.parseInt( keyboard.nextLine() );
66      BankAccount[] theAccounts = new BankAccount[ size ];
67      for ( int i = 0; i < size; i ++ )
68      {
69         theAccounts[ i ] = initial( i );
70      }
```

Listing 16.18 A program that manipulates with a list of bank accounts (part 3). The preparation in the method `main`

The method then commences a do-while loop (Line 72). The action to be performed in the do-while loop is as follows:

Step 1: The method prints the prompt using `actionPrompt` (Line 73).

Step 2: The method prompts the user to enter the choice (Line 75) and obtains the choice of the user. The choice is converted to an `int` using `Integer.parseInt` (Line 75).

Step 3: If the choice is 1, the action to be performed is a deposit (Line 76). The method receives the index to the account (Lines 78 and 79), receives the amount to deposit (Lines 80 and 81), and performs the task by calling the method `deposit` with the element of the `BankAccount` array at the index and the amount specified (Line 82).

Step 4: If the choice is 2, the action to be performed is a withdrawal (Line 84). The method receives the index to the account (Lines 86 and 87), receives the amount to deposit (Lines 88 and 89), and performs the task by calling the method `withdraw` with the element of the `BankAccount` array at the index and the amount specified (Line 90).

Step 5: If the choice is 3, the action to be performed is a transfer (Line 92). The method obtains the index to the destination account of the transfer and the index to the source account of the transfer (Lines 94–97), and then obtains the amount of transfer (Lines 98 and 99). The transaction is performed by calling the method `transfer` with the elements of the `BankAccount` array at the two indexes and the amount specified as the actual parameters (Line 100). The two indexes can be equal to each other. If the two indexes are equal to each other, the money is withdrawn (if the amount specified is valid) from the account specified by the two identical indexes and deposited back to it.

Step 6: The program terminates if the choice is 4 (Line 102).

```
71      do
72      {
73        actionPrompt( theAccounts );
74        System.out.print( "Enter your choice: " );
75        choice = Integer.parseInt( keyboard.nextLine() );
76        if ( choice == 1 )
77        {
78          System.out.print( "Enter account index: " );
79          index = Integer.parseInt( keyboard.nextLine() );
80          System.out.print( "Enter amount: " );
81          amount = Integer.parseInt( keyboard.nextLine() );
82          deposit( theAccounts[ index ], amount );
83        }
84        else if ( choice == 2 )
85        {
86          System.out.print( "Enter account index: " );
87          index = Integer.parseInt( keyboard.nextLine() );
88          System.out.print( "Enter amount: " );
89          amount = Integer.parseInt( keyboard.nextLine() );
90          withdraw( theAccounts[ index ], amount );
91        }
92        else if ( choice == 3 )
93        {
94          System.out.print( "Enter account origination index: " );
95          index = Integer.parseInt( keyboard.nextLine() );
96          System.out.print( "Enter account destination index: " );
97          index2 = Integer.parseInt( keyboard.nextLine() );
98          System.out.print( "Enter amount: " );
99          amount = Integer.parseInt( keyboard.nextLine() );
100         transfer( theAccounts[ index ], theAccounts[ index2 ], amount );
101       }
102     } while ( choice != 4 );
103   }
104 }
```

Listing 16.19 A program that manipulates with a list of bank accounts (part 4). The loop in the method `main`

Here is an execution example of the program, presented in two parts:

```
1  Enter the Number of Accounts: 3
2  Enter name for account #0: My Saving
3  Enter amount for account #0: 10000000
4  Enter name for account #1: My Checking
5  Enter amount for account #1: 100000
6  Enter name for account #2: Her Saving
7  Enter amount for account #2: 20000000
8  ------
9   0: My Saving has the balance of $100,000.00
10  1: My Checking has the balance of $1,000.00
11  2: Her Saving has the balance of $200,000.00
12 -----Select action
13 1. Deposit to an Account
14 2. Withdrawal from an Account
15 3. Transfer Between Accounts
16 4. Quit
```

```
Enter your choice: 1
Enter account index: 0
Enter amount: 5000000
------
 0: My Saving has the balance of $150,000.00
 1: My Checking has the balance of $1,000.00
 2: Her Saving has the balance of $200,000.00
-----Select action
1. Deposit to an Account
2. Withdrawal from an Account
3. Transfer Between Accounts
4. Quit
Enter your choice: 3
Enter account origination index: 2
Enter account destination index: 1
Enter amount: 5000000
------
 0: My Saving has the balance of $150,000.00
 1: My Checking has the balance of $51,000.00
 2: Her Saving has the balance of $150,000.00
-----Select action
1. Deposit to an Account
2. Withdrawal from an Account
3. Transfer Between Accounts
4. Quit
Enter your choice: 2
Enter account index: 2
Enter amount: 999900
------
 0: My Saving has the balance of $150,000.00
 1: My Checking has the balance of $51,000.00
 2: Her Saving has the balance of $140,001.00
-----Select action
1. Deposit to an Account
2. Withdrawal from an Account
3. Transfer Between Accounts
4. Quit
Enter your choice: 4
```

16.3 Array with Capacity and Size (Reprise)

In Sect. 13.6, we studied arrays with capacity and size. We wrote an application using an array with capacity and size to manipulate a list of names `String`, where we enabled the user to add an element, remove an element, search for an element, and see all the elements. In this section, we study how to convert the class to a general purpose object class for maintaining an indefinite number of `String` objects.

To develop the general purpose class, we do the following:

- We will turn the global variables into instance variables.
- We will remove `main` and `action` methods, since they are irrelevant.
- We will convert the method `setup` to a constructor, where the constructor takes `int` data, `capacity`, as its formal parameter, instead of asking for input from the user.
- We will convert the method `add` to an instance method that adds just one element.
- We will convert the remaining methods (`search`, `remove`, and `print`) to instance methods. We will remove the `static` attribute from the header of the methods.

- We will add new methods, `getCapacity` and `getSize`, as getters.

Here is a code for `ArrayWithCapacity`. The global variables are now private instance variables (Lines 4 and 5). They are `int` variables, `size` and `capacity`, and a `String[]` variable, `theNames`. The constructor takes an `int` value, `capacity`, as the formal parameter (Line 7). The constructor saves the value of the parameter to the instance variable, `capacity`. Here we use `this.` for distinction (Line 9). The constructor then instantiates the array using the capacity value as the length (Line 10) and assigns 0 to `size` (Line 11). The getters `getCapacity` (Lines 14–17) and `getSize` (Lines 19–22) return the values of `capacity` and `size`.

```
1  // Using an array of fixed size
2  public class ArrayWithCapacity
3  {
4    private int size, capacity;
5    private String[] theNames;
6
7    public ArrayWithCapacity( int capacity )
8    {
9      this.capacity = capacity;
10     theNames = new String[ capacity ];
11     size = 0;
12   }
13   public int getCapacity()
14   {
15     return capacity;
16   }
17   public int getSize()
18   {
19     return size;
20   }
```

Listing 16.20 The first part of class `ArrayWithCapacity`

The method `search` (Lines 21–36) is the same as before, except that it is now an instance method. Since the previous version used the variables as global variables, just removing the attribute of `static` is enough to make this adjustment.

```
21   public void search( String aName )
22   {
23     boolean found = false;
24     for ( int pos = 0; pos < size; pos ++ )
25     {
26       if ( theNames[ pos ].indexOf( aName ) >= 0 )
27       {
28         found = true;
29         System.out.printf( "Found %s at %d%n", theNames[ pos ], pos );
30       }
31     }
32     if ( !found )
33     {
34       System.out.println( "Not found" );
35     }
36   }
```

Listing 16.21 The search method of the class `ArrayWithCapacity`

The way the method `print` works (Lines 37–43) is the same as before:

```
37  public void print()
38  {
39    for ( int pos = 0; pos < size; pos ++ )
40    {
41      System.out.printf( "%4d:%s%n", pos, theNames[ pos ] );
42    }
43  }
```

Listing 16.22 The method for printing the data in class `ArrayWithCapacity`

The method `add` (Lines 44–54) is now designed to take only one element to add:

```
44  public void add( String aName )
45  {
46    if ( size < capacity )
47    {
48      theNames[ size ++ ] = aName;
49    }
50    else
51    {
52      System.out.println( "The storage is full." );
53    }
54  }
```

Listing 16.23 The method for adding an element in `ArrayWithCapacity`

The method `remove` (Lines 55–65) is the same as before, except that the instance variables appear in place of the global variables:

```
55  public void remove( int index )
56  {
57    if ( index >= 0 && index < size )
58    {
59      theNames[ index ] = theNames[ -- size ];
60    }
61    else
62    {
63      System.out.println( "The specified position does not exist." );
64    }
65  }
```

Listing 16.24 The method for removing an element in class `ArrayWithCapacity`

Now we can use this new class to rewrite the application. Important points to note are:

- This time, we use an `ArrayWithCapacity` object, since we have packed all the relevant components in it (Line 9). We use

  ```
  ArrayWithCapacity array = new ArrayWithCapacity( capacity );
  ```

  for instantiation.

- We use method calls `array.add( name )` (Line 19), `array.remove( index )` (Line 23), `array.search( key )` (Line 27), and `array.print()` (Line 29) for actions.

```
1  import java.util.*;
2  // Using an array of fixed size
3  public class UseArrayWithCapacity {
4    // main
5    public static void main( String[] args ) {
6      Scanner keyboard = new Scanner( System.in );
7      System.out.print( "Enter capacity: " );
8      int capacity = Integer.parseInt( keyboard.nextLine() );
9      ArrayWithCapacity array = new ArrayWithCapacity( capacity );
10     char c;
11     do {
12       System.out.print( "What do you want to do?\n"
13           + "A(dd), R(remove), P(rint), S(earch), Q(uit): " );
14       c = keyboard.nextLine().charAt( 0 );
```

Listing 16.25 A new version of the program that uses an array with capacity and size (part 1)

```
15       switch ( c )
16       {
17         case 'A':
18           System.out.println(
19               "Enter new names, empty line to finish: " );
20           String name;
21           do
22           {
23             System.out.print( "> " );
24             name = keyboard.nextLine();
25             if ( name.length() != 0 )
26             {
27               array.add( name );
28             }
29           } while ( name.length() != 0 );
30           break;
31         case 'R':
32           System.out.print( "Enter index: " );
33           array.remove( Integer.parseInt( keyboard.nextLine() ) );
34           break;
35         case 'S':
36           System.out.print( "Enter a key: " );
37           array.search( keyboard.nextLine() );
38           break;
39         case 'P':
40           array.print();
41           break;
42       }
43     } while ( c != 'Q' );
44     System.out.println( "Closing..." );
45   }
46 }
```

Listing 16.26 A new version of the program that uses an array with capacity and size (part 2)

By appropriately modifying the key matching criterion, `indexOf( key ) >= 0`, it is possible to design a class for maintaining a list of objects of a type other than `String`.

Summary

- If the instance variables of an object class are public, the program of the object class may access the variable by attaching the names of the instance variables to the object name with a comma in between.
- To prohibit access to instance variables from any outside class, the attribute of `private` can be attached.
- To distinguish between an instance variable and a formal parameter having the same name, the attribute of `this.` can be attached to it.
- Information hiding refers to the idea that an object class can be written so that the user of the object class does not need to know what the instance variables of the object class are.
- Methods for accessing values of instance variables (not necessarily as they are) are called getters (or accessors).
- Methods for changing values of instance variables (not necessarily as they are) are called setters (or modifiers).
- It is possible to convert an existing program to a container class that has a certain functionality by changing the global variables to instance variables.

Exercises

1. **Terminology question** State the differences between "private" instance variables and "public" instance variables.
2. **Class Car** Write a class named `Car` for recording information about a car. The class has three getters:
   - `getMake()` returns the make of the car as a `String` value,
   - `getModel()` returns the model of the car as a `String` value, and
   - `getYear()` that returns the year the car was made as an `int` value.

   The class has four setters:
   - `setMake( String make )`,
   - `setModel( Sting model )`,
   - `setYear( String year )`, and
   - `setYear(int year )` (by way of method overloading).

   Write two constructors for the class,
   - `Car( String make, String model, String year )` and
   - `Car( String make, String model, int year )`.

   Choose appropriate instance variables.
3. **An object class DoDo** Suppose `DoDo` is an object class that provides access to two instance variables, a `String` value, `word`, and an `int` value, `quantity`. Write a constructor `public DoDo( String word, int quantity )` that instantiates an object of the class by storing the first value in the instance variable `word` and the second value in the instance variable `quantity`. Also, write getters for `word` and `quantity`.

4. **Using instance methods** Let `Stats` be a class designed for maintaining the statistics of some collection of real numbers. The class does not necessary have to maintain the numbers in the collection, but has to satisfy the following properties:
   - There is one constructor. The constructor has no formal parameters. The constructor initializes the collection of as the empty set.
   - The class has a void method `add( double x )` that incorporates the value `x` into the collection.
   - The class has an `int` method `size()` that returns the number of values that have been incorporated.
   - The class has a `double` method `average()` that returns the average of the numbers that have been incorporated.
   - The class has a `double` method `variance()` that returns the variance of the numbers that have been incorporated, where the variance is the mean of the square of the difference between the individual values from the average. Note that the variance is given as:

     (the sum of the square of the values) − (the square of the average)

   In this exercise, write a Java program `CalcStats` that
   - opens a data file whose name is given as `args[ 0 ]`,
   - uses a `Stats` object to incorporate all the numbers appearing in the file, and then
   - prints the three pieces of information that can be obtained from the object using the accessor methods.

   For example, if the contents of the file `numbers.txt` are:

```
3
4
5
6
7
8
9
10
```

   the program runs as follows

```
% java CalcStats numbers.txt
The count is 8
The average is 6.500000
The variance is 5.250000
```

5. **Writing class `Stats`** Write the class `Stats` whose function was defined in the previous question.
6. **A class for a point in the three-dimensional space** Write a class named `ThreeDPoint` for recording a point in a three-dimensional real space, where the instance variables are in `double`. There should be methods for individually accessing the three coordinates and methods for individually modifying them.

Programming Projects

7. **A class for a linear equation over three unknowns** Write a class named `ThreeDLinear` for recording a linear equation $ax + by + cz = d$, where x, y, z are unknowns and a, b, c, d are coefficients. The class has only one constructor that takes values for a, b, c, and d and stores the values in their respective instance variables. The class has four accessors to obtain the values of the coefficients. The class has three other instance methods:

- `public ThreeDLinear scale( double s )`: This method returns a new `ThreeDLinear` object in which all the coefficients are scaled by `s`.
- `public ThreeDLinear plus( ThreeDLinear o )`: This method returns a new `ThreeDLinear` object that corresponds to the equation whose coefficients are generated by adding the coefficients of `this` and those of `o`.
- `public ThreeDLinear minus( ThreeDLinear o )`: This method returns a new `ThreeDLinear` object that corresponds to the equation whose coefficients are generated by subtracting the coefficients of `o` from the coefficients of `this`.

8. **Class for a complex numbers** Write a class named `ComplexNum` for recording a complex number. The class must have two `double` instance variables, `real` and `imaginary`. An object of this class represents the complex number `real + imaginary * i` (where `i` is the root of $\sqrt{-1}$). Write the instance method `size` that returns, in `double`, the value of $\texttt{real}^2 - \texttt{imaginary}^2$.
9. **Class `MilitaryTime`** Write a class named `MilitaryTime` for representing the time of a day in 24 h. The class must have two accessors, `getHours()` and `getMinutes()`. Both of them must return an integer. The class has two additional instance methods, `advance(MilitaryTime o)` and `rewind(MilitaryTime o)`. The former advances the time by the amount of time represented by `o`. The latter rewinds the time by the amount of time represented by `o`.
10. **Class `TicTacToe`** Write a class named `TicTacToe` for representing a configuration of Tic-Tac-Toe. Tic-Tac-Toe is a game played by two people on a 3×3 grid. The two players take turns and mark one square of the grid with letters assigned to them (O and X). The player who has produced a row, a column, or a diagonal of the same letters wins the game. We view a configuration of the game as a 3×3 two-dimensional array of `char`. We use two characters, `'O'` and `'X'`, for the markings and the whitespace for available squares. The constructor of `TicTacToe` creates the blank configuration (that is, the configuration in which no square has been marked). We use a pair of integers between 0 and 2 to specify a square. The class has the following methods:
    - `boolean isAvailable( int x, int y )`: returns whether or not the row `x` column `y` square of the grid is blank.
    - `boolean isO( int x, int y )`: returns whether or not the row `x` column `y` square of the grid is an `'O'`.
    - `boolean isX( int x, int y )`: returns whether or not the row `x` column `y` square of the grid is an `'X'`.
    - `void setO( int x, int y )`: sets the row `x` column `y` square of the grid to `'O'`.
    - `void setX( int x, int y )`: sets the row `x` column `y` square of the grid to `'X'`.
11. **The 15 puzzle** The 15 Puzzle is a puzzle played with 15 pieces placed on a 4×4 grid. The sizes of the pieces are equal to the size of any square of the grid. The pieces are numbered 1 through 15. At the start of the game the pieces are placed on one of the 16 squares of the grid with no overlap, so there is only one open square. During the course of the puzzle, the player can move a piece from any one of the neighboring squares (from left, right, above, or below) to the open square. The goal of the puzzle is to reorder the pieces so that 1..4 appear in the top row from left to right, 5..8 appear in the next row from left to right, 9..12 appear in the next row, and 13..15 in the bottom row, with the rightmost square open, as shown in the following diagram:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

To play the puzzle, the player herself scrambles the order of the pieces, by randomly moving the pieces according to the rule. Every 15 puzzle configuration generated from the goal configuration in this manner is solvable.

Write a class named `Config15` whose object represents a configuration of the 15 puzzle. The class must have three instance variables. The first is a 4×4 array of `int` values, where the numbers $0, \ldots, 15$ appear exactly once. In this array, 0 represents the open square. The second and the third instance variables are the row and the column indexes of the open square. The constructor takes an array in this format and copies the contents of the array to the instance variable, and then search for an open square to assign values to the row and column instance variables.

There should be the following instance methods.

- `boolean isSolved()`: This method returns whether or not the configuration represented by the object is the final configuration.
- `boolean pullDown()`: This method pulls the piece above the open square into the open square. The move is not possible if the open square is in the first row. The method returns as a `boolean` value indicating whether or not the move was successful.
- `boolean pullUp()`: This method moves the piece in the square immediately above the open square. The move is not possible if the open square is in the last row. The method returns whether or not the move has been successful.
- `boolean pullLeft()`: This method moves the piece in the square immediately to the left of the open square. The move is not possible if the open square is in the first column. The method returns whether or not the move has been successful.
- `boolean pullRight()`: The method moves the piece in the square immediately to the right of the open square. The move is not possible if the open square is in the last column. The method returns whether or not the move has been successful.
- `int get( int i, int j )`: The method returns the value of the array at row `i` and column `j`.
- `int getOpenRow()`: The method returns the value of the row index of the open space.
- `int getOpenCol()`: The method returns the value of the column index of the open space.

12. **File exploration using an array with capacity** Modify the code for `ArrayWithCapacity` to write a code `FileArrayWC` for maintaining a list of `File` objects. The class has the following instance methods:
    - `int getCapacity()`: returns the capacity.
    - `int getSize()`: returns the size.
    - `File get( int index )`: returns the object at position `index`.
    - `void add( File f )`: adds `f`.
    - `void remove( int index )`: removes the file at `index`.
    - `int[] search( String key )`: returns an array of indexes at which the canonical path to the `File` object has `key` appearing in it.

    Using this class, write an application `UseFileArrayWC` that allows the user to do the following:
    (a) Adding one file by specifying its file path.
    (b) Adding all files in a folder by specifying a path to the folder.

(c) Removing a file at a specific index.
(d) Searching for files with a key.
(e) Viewing all files.
(f) Writing the data to a file.

For searching and viewing, write a method named `present` that receives a `FileArrayWC` object named `data` and an `int[]` object named `indexes` as parameters, and then prints all the files in `data` whose positions appear in `indexes`. To print an item, present the index, readability, writability, executability, whether it is a folder or not, along with the canonical path. The four `boolean` properties are represented by the letters `'r'`, `'w'`, `'x'`, and `'d'`. For each of the four properties, if the file does not possess the property, we print `' '`. Here is an example of the output, where the array has only three elements, 0, 1, and 2.

```
1    0 rwxd /Users/ogihara/Documents/CSC120/codeVer1
2    1 rwxd /Users/ogihara/Documents/CSC120/bookDraft
3    2 rw   /Users/ogihara/Documents/CSC120/code/UseFileArrayWC.classVer1
```

To view the data, define a one-parameter version of the method `present` that takes a `FileArrayWC` object `data` and calls the two-parameter version, where the second parameter is an array whose elements are `0, ..., data.getSize() - 1`. That way, all the elements will be printed. Each time the program writes the data into a file, it receives a name of the data file from the user. The very first line of the output is the size. After the first line, the canonical paths of the elements appear, one file per line. A `throws IOException` declaration can be added to each method that makes a call to some `File` method that has the declaration of `throws IOException`.