Exercises:

9. Given a person's year of birth, the Birthday Wizard can compute the year in which the person's *n*th birthday will occur or has occurred. Write statements that can be used in a Java program to perform this computation for the Birthday Wizard.

Solution:

```
System.out.println("Greetings.");

int year, age;

Scanner keyboard = new Scanner(System.in);
System.out.println("What year were you born in?");
year = keyboard.nextInt();
System.out.println("Choose an age in years.");
age = keyboard.nextInt();

System.out.println("You will turn " + age + " in the year ");
System.out.println(year + age);
```

10. Write statements that can be used in a Java program to read two integers and display the number of integers that lie between them, including the integers themselves. For example, four integers are between 3 and 6: 3, 4, 5, and 6.

Solution:

```
System.out.println("Greetings.");

int min, max;

Scanner keyboard = new Scanner(System.in);
System.out.println("Enter a minimum and maximum value");
min = keyboard.nextInt();
max = keyboard.nextInt();

System.out.println("The number of values in the range from " +
      min + " to " + max + " is ");
System.out.println(max-min+1);
```

11. A single bit can represent two values: 0 and 1. Two bits can represent four values: 00, 01, 10, and 11. Three bits can represent eight values: 000, 001, 010, 011, 100, 101, 110, and 111. How many values can be represented by

a. 8 bits? b. 16 bits? c. 32 bits?

Solution:

a) $2^8 = 256$
b) $2^{16} = 65536$
c) $2^{32} = 4294967296$

13. Self-Test Question 27 asked you to think of some attributes for a song object.
What attributes would you want for an object that represents a play list containing many songs?

Solution:

The name of the play list, the songs in the play list, the number of songs, the length of time for the play list

14. What behaviors might a song have? What behaviors might a play list have? Contrast the difference in behavior between the two kinds of objects.

Solution:

A song object would have behaviors for modifying attributes like rating and play count. It might also allow other attributes like the artist and album to be modified. If the object contains music data, it could be able to play the song. In contrast a play list would have behaviors that would allow one to manage the play list. You would be able to add and remove songs. You would be able to change the order of songs in the play list. A play list could also have the ability to play the songs.

A song object would have behaviors for modifying attributes like rating and play count. It might also allow other attributes like the artist and album to be modified. If the object contains music data, it could be able to play the song. In contrast a play list would have behaviors that would allow one to manage the play list. You would be able to add and remove songs. You would be able to change the order of songs in the play list. A play list could also have the ability to play the songs.

15. What attributes and behaviors would an object representing a credit card account have?

Solution:

A credit card account would attributes for personal information (name, address, etc.), account number, credit limit, interest rate, current balance, and charge history. For behaviors we would have actions like charging an item, computing the interest, paying a monthly bill, raising the credit limit, and changing the interest rate.

16. Suppose that you have a number x that is greater than 1. Write an algorithm that computes the largest integer k such that 2^k is less than or equal to x.

Solution:

1. *Let k be 0*
2. *Let check be 1*
3. *While check < x*
   3.1. Let k be k+1
   *3.2. Let check be check*2*

17. Write an algorithm that finds the maximum value in a list of values.

Solution:

1. *Let max be the first value in the list*
2. *For each value x in the list*
   2.1. If x is greater than max
      2.1.1. Let max be x 2

18. Write statements that can be used in a JavaFX application to draw the five interlocking rings that are the symbol of the Olympics. (Don't worry about the color.)

Solution:

```
gc.strokeOval(0, 0, 40, 40);
gc.strokeOval(50, 0, 40, 40);
gc.strokeOval(100, 0, 40, 40);
gc.strokeOval(25, 25, 40, 40);
gc.strokeOval(75, 25, 40, 40);
```

Practice Programs:

1. Obtain a copy of the Java program shown in Listing 1.1 from the Web at the location given in the preface. Name the file FirstProgram.java. Compile the program so that you receive no compiler error messages. Then run the program.

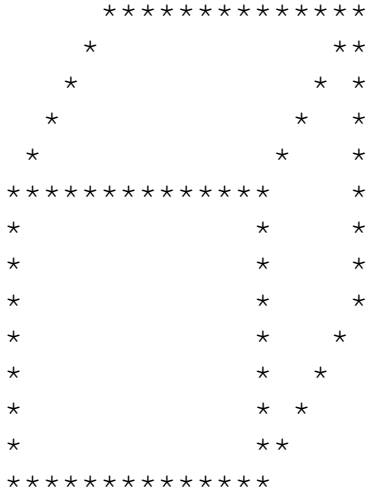
References: *Listing 1.1.*
Solution: *See the code in FirstProgram.java.*

2. Modify the Java program described in Practice Program 1 so that it adds three numbers instead of two. Compile the program so that you receive no compiler error messages. Then run the program.

References: Project 1.1
Solution: *See the code in FirstProgram2.java.*

Programming Projects

1. Write a Java program that displays the following picture. *Hint*: Write a sequence of println statements that display lines of asterisks and blanks.



Solution:

See the code in Simple1.java.

2. Write a complete program for the problem described in Exercise 9.

References:

Exercise 1.9

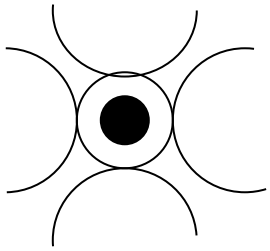
Solution:

See the code in BirthdayWizard.java.

4. Write a JavaFX program similar to the one in Listing 1.2 that displays a picture of a snowman. *Hint*: Draw three circles, one above the other. Make the circles progressively smaller from bottom to top. Make the top circle a happy face.

References: Listing 1.2
Solution: *See the code in Snowman.java*

6. Write a JavaFX program that displays the following pattern:



Solution: *See the code in Icon.java.*

Exercises:

1. Write a program that demonstrates the approximate nature of floating-point values by performing the following tasks:
• Use Scanner to read a floating-point value *x*.
• Compute 1.0 / *x* and store the result in *y*.
• Display *x*, *y*, and the product of *x* and *y*.
• Subtract *1* from the product of *x* and *y* and display the result.
Try your program with values of *x* that range from 2e-11 to 2e11. What can you conclude?

Solution:

See the code in Approximation.java. You cannot count on mathematical identities to always be true. For example, x times 1/x may not be equal to 1. You should avoid testing for equality with floating point numbers.

2. Write a program that demonstrates type casting of double values by performing the following tasks:
• Use Scanner to read a floating-point value *x*.
• Type cast *x* to an int value and store the result in *y*.
• Display *x* and *y* clearly labeled.
• Type cast *x* to a byte value and store the result in *z*.
• Display *x* and *z* clearly labeled.
Try your program with positive and negative values of *x* that range in magnitude from 2e-11 to 2e11. What can you conclude?

Solution:

See the code in TypeCaster.java. As long as the value is not too large, then the type cast acts like truncation. If the value is large enough, however, the result of the type cast is unpredictable.

3. Write a program that demonstrates the operator % by performing the following tasks:

- Use Scanner to read a floating-point value *x*.
- Compute *x* % 2.0 and store the result in *y*.
- Display *x* and *y* clearly labeled.
- Type cast *x* to an int value and store the result in *z*.
- Display *x*, *z*, and *z* % 2 clearly labeled.

Try your program with positive and negative values of *x*. What implications do your results have for deciding whether a negative integer is odd?

Solution:

See the code in ModTester.java. If we compute the mod of a positive number, the result is either zero or negative. If we compute the mod of a negative number, the result is either zero or positive. Mathematically, when we mod an integer by 2, we should get 1. But, because of the way mod works with positive and negative values, we can not just check to see if the value of the mod is 1.

4. If u = 2, v = 3, w = 5, x = 7, and y = 11, what is the value of each of the following expressions, assuming int variables?

- u + v * w + x
- u + y % v * w + x
- u++ / v + u++ * w

Solution:

*u + v * w + x*
*is 2 + 3 * 5 + 7*
is 2 + 15 + 7
is 24

*u + y % v * w + x*
*is 2 + 11 % 3 * 5 + 7*
*is 2 + 2 * 5 + 7*
is 2 + 10 + 7
is 19

*u++ / v + u++ * w*
*is 2 / 3 + 3 * 5*
*is 2 / 3 + 3 * 5*
is 0 + 15
is 15

This code is in Fragments.java.

5. What changes to the ChangeMaker program in Listing 2.3 are necessary if it also accepts coins for one dollar and half a dollar?

Solution:

Add variables.

```
      int dollars, halfDollars;
```

Change the prompt.

```
      System.out.println("Enter a whole number greater than
0.");
```

Compute dollars and halfDollars before the quarters computation.

```
        dollars = amount / 100;
        amount = amount % 100;

        halfDollars = amount / 50;
        amount = amount % 50;
```

Add to the output.

```
      System.out.println(dollars + " dollars");
      System.out.println(halfDollars + " halfDollars");
```

6. If the int variable x contains 10, what will the following Java statements display?
System.out.println("Test 1" + x * 3 * 2.0);
System.out.println("Test 2" + x * 3 + 2.0);
Given these results, explain why the following Java statement will not compile:
System.out.println("Test 3" + x * 3 - 2.0)

Solution:

The first statement prints:
Test 160.0
The second statement prints:
Test 2302.0
In the first print statement, we compute 60.0, convert that into a string and concatenate it with "Test 1".
In the second statement, we compute 30 and convert that into a string and concatenate to "Test 2". Then we convert 2.0 to a string and concatenate. Notice that the plus operator is concatenation here not addition and the result is a string.

The third print statement does not compile, because Java can not apply the minus operator to a string and 2.0.

This code is in Fragments.java.

7. Write some Java statements that use the String methods indexOf and substring to find the first word in a string. We define *word* to be a string of characters that does not include whitespace. For example, the first word of the string " Hello, my good friend!" is the string "Hello," and the second word is the string "my".

Solution:

```
sentence = sentence.trim();
int space = sentence.indexOf(" ");
String word = sentence.substring(0, space);

System.out.println("The first word is: " + word);
```

This code is in Fragments.java.

8. Repeat the previous exercise, but find the second word in the string.

Solution:

```
sentence = sentence.trim();
int space = sentence.indexOf(" ");
String rest = sentence.substring(space).trim();
space = rest.indexOf(" ");
String secondWord = rest.substring(0, space);

System.out.println("The second word is: "
          + secondWord);
```

This code is in Fragments.java.

9. What does the following Java statement display?
System.out.println("\"\tTest\\\\\rIt\'");
Does replacing the r with an n make a difference in what is displayed?

Solution:

The results depend on the environment being used. NetBeans displayed:

```
"    Test\\
It'
```

BlueJ displayed:

```
"    Test\\It'
```

The change made a difference in BlueJ, but not in NetBeans.

This code is in Fragments.java.

10. Write a single Java statement that will display the words *one*, *two*, and *three*, each on its own line.

Solution:

```
System.out.println("one\ntwo\nthree");
```

This code is in Fragments.java.

11. What does the Java code
```
Scanner keyboard = new Scanner(System.in);
System.out.println("Enter a string.");
int n = keyboard.nextInt();
String s = keyboard.next();
System.out.println("n is " + n);
System.out.println("s is " + s);
```
display when the keyboard input is 2istheinput?

Solution:

Exception in thread "main" java.util.InputMismatchException

This code is in Fragments.java.

12. What does the Java code
```
Scanner keyboard = new Scanner(System.in);
keyboard.useDelimiter(“y”);
System.out.println("Enter a string.");
String a = keyboard.next();
String b = keyboard.next();
System.out.println("a is " + a);
System.out.println("b is " + b);
```
display when the keyboard input is
By the
pricking
of my thumbs

Solution:

```
a is B
b is  the
pricking
of m
```

This code is in Fragments.java.

13. Repeat the previous exercise, but change next to nextLine in the statement that assigns a value to b.

Solution:

```
a is b
b is y the
```

This code is in Fragments.java.

14. Many sports have constants embedded in their rules. For example, baseball has 9 innings, 3 outs per inning, 3 strikes in an out, and 4 balls per walk. We might encode the constants for a program involving baseball as follows:

```
public static final int INNINGS = 9;
public static final int OUTS_PER_INNING = 3;
public static final int STRIKES_PER_OUT = 3;
public static final int BALLS_PER_WALK = 4;
```

For each of the following popular sports, give Java named constants that could be used in a program involving that sport:

- Basketball
- American football
- Soccer
- Cricket
- Bowling

Solution:

These are just some of the possible constants we could define.

```
// Basketball
public static final int QUARTERS = 4;
public static final int POINTS_PER_REGULAR_SHOT = 2;
public static final int FOULS_PER_GAME = 5;


// American football
public static final int MINUTES_PER_QUARTER = 15;
public static final int DOWNS = 4;
public static final int YARDS_TO_FIRST_DOWN = 9;
public static final int POINTS_FOR_TD = 6;
public static final int POINTS_FOR_FIELDGOAL = 3;


// Soccer
public static final int PLAYERS_PER_SIDE = 11;
public static final int MINUTES_PER_HALF = 45;
public static final int BREAK_TIME = 15;

// Cricket
public static final int PLAYERS_PER_TEAM = 11;
public static final int PITCH_LENGTH = 66;
public static final int PITCH_WIDTH = 10;


// Bowling
public static final int NUMBER_OF_PINS = 10;
public static final int FRAMES = 10;
public static final int PERFECT_GAME = 300;
```

This code is in Fragments.java.

15. Repeat Exercise 18 in Chapter 1, but define and use named constants.

Solution:

```
    public static final int RING_DIAMETER = 40;
    public static final int RING_X_OFFSET = 25;
    public static final int RING_Y_OFFSET = 25;



    gc.strokeOval(0, 0, RING_DIAMETER, RING_DIAMETER);
    gc.strokeOval(2*RING_X_OFFSET, 0, RING_DIAMETER, RING_DIAMETER);
    gc.strokeOval(4*RING_X_OFFSET, 0, RING_DIAMETER, RING_DIAMETER);
    gc.strokeOval(RING_X_OFFSET, RING_Y_OFFSET, RING_DIAMETER,
RING_DIAMETER);
    gc.strokeOval(3*RING_X_OFFSET, RING_Y_OFFSET, RING_DIAMETER,
RING_DIAMETER);
```

This code is in Rings.java.

16. Define named constants that you could use in Programming Project 6 in Chapter 1.

Solution:

```
public static final int X_CENTER = 120;
public static final int Y_CENTER = 120;
public static final int FILL_DIAMETER = 20;
public static final int CLEAR_DIAMETER = 40;


    gc.strokeOval(X_CENTER - CLEAR_DIAMETER/2,
              Y_CENTER - CLEAR_DIAMETER/2,
              CLEAR_DIAMETER, CLEAR_DIAMETER);
    gc.fillOval(X_CENTER - FILL_DIAMETER/2,
              Y_CENTER - FILL_DIAMETER/2,
              FILL_DIAMETER, FILL_DIAMETER);

    gc.strokeArc(X_CENTER - CLEAR_DIAMETER/2,
              Y_CENTER + CLEAR_DIAMETER/2,
              CLEAR_DIAMETER, CLEAR_DIAMETER, 0, 180,
              ArcType.OPEN);
    gc.strokeArc(X_CENTER - CLEAR_DIAMETER/2,
              Y_CENTER - 3*CLEAR_DIAMETER/2,
              CLEAR_DIAMETER, CLEAR_DIAMETER, 0, -180,
              ArcType.OPEN);
    gc.strokeArc(X_CENTER - 3*CLEAR_DIAMETER/2,
              Y_CENTER - CLEAR_DIAMETER/2,
              CLEAR_DIAMETER, CLEAR_DIAMETER, 90, -180,
              ArcType.OPEN);
    gc.strokeArc(X_CENTER + CLEAR_DIAMETER/2,
              Y_CENTER - CLEAR_DIAMETER/2,
              CLEAR_DIAMETER, CLEAR_DIAMETER, 90, 180,
              ArcType.OPEN);
```

Practice Programs:

1. Write a program that reads three whole numbers and displays the average of the three numbers.

Notes: This project requires careful use of integer and double data types to avoid unwanted truncation when dividing.
Solution: *See the code in Average3.java.*

3. Write a program that reads the amount of a monthly mortgage payment and the amount still owed—the outstanding balance—and then displays the amount of the payment that goes to interest and the amount that goes to principal (i.e., the amount that goes to reducing the debt). Assume that the annual interest rate is 7.49 percent. Use a named constant for the interest rate. Note that payments are made monthly, so the interest is only one twelfth of the annual interest of 7.49 percent.

Notes:

Two solutions are given for this project. Since the `round` method in Java is not discussed in this chapter, the first solution, `Mortgage.java`, takes a simplistic approach in which the payment and principle amounts are entered as whole numbers and the interest calculation truncates, rather than rounds. An alternative approach would be to explain to students how to round numbers by doing the calculation in floating point, adding 0.5, then truncating (by explicitly casting the result to `int`). In addition, you may want to explain a technique for working with money: convert each money value to an integer number of cents. Enter money amounts as a decimal value with two decimal places, multiply by 100, and add 0.5, then truncate by explicitly casting to `int`. Using the resulting integer values for all the money transactions avoids errors in the cents that could occur through truncation or encoding the money in floating point format. `Mortgage2.java`, uses this technique.

Solution:

See the code in Mortgage.java. This program only allows whole dollar input amounts and truncates rather than round the interest calculations.

See the code in Mortgage2.java. This program allows dollar.cents input values. Interest calculations are rounded.

4. Write a program that reads a four-digit integer, such as 1998, and then displays it, one digit per line, like so:

1
9
9
8

Your prompt should tell the user to enter a four-digit integer. You can then assume that the user follows directions. *Hint*: Use the division and remainder operators.

Notes: This project is straightforward since it assumes the user enters correct data.
Solution: *See the code in Vertical4Digits.java.*

Programming Projects

1. Write a program that converts degrees from Fahrenheit to Celsius, using the formula *DegreesC* = 5 (*DegreesF* − 32) / 9. Prompt the user to enter a temperature in degrees Fahrenheit as a whole number without a fractional part. Then have the program display the equivalent Celsius temperature, including the fractional part to at least one decimal point. A possible dialogue with the user might be
Enter a temperature in degrees Fahrenheit: 72
72 degrees Fahrenheit is 22.2 degrees Celsius.

Notes:

This project specifically asks for the Fahrenheit temperature to be an integer so the Fahrenheit variable is declared as an int and the formula has only integer constants. The calculated Celsius value, however, is supposed to include the fractional part, so it must be a floating-point value and a cast to float is used in the calculation to avoid truncation due to integer division.

Solution:

See the code in FtoC.java.

2. Write a program that reads a line of text and then displays the line, but with the first occurrence of *hate* changed to *love*. For example, a possible sample dialogue might be

Enter a line of text.
I hate you.
I have rephrased that line to read:
I love you.
You can assume that the word *hate* occurs in the input. If the word *hate* occurs more than once in the line, your program will replace only its first occurrence.

Notes:

This project gives the student practice using string methods and the backslash character as an escape character to print double quotes.
Solution: *See the code in LoveHate.java.*

3. Write a program that will read a line of text as input and then display the line with the first word moved to the end of the line. For example, a possible sample interaction with the user might be

Enter a line of text. No punctuation please.
Java is the language
I have rephrased that line to read:
Is the language Java

Assume that there is no space before the first word and that the end of the first word is indicated by a blank, not by a comma or other punctuation. Note that the new first word must begin with a capital letter.

Notes: This project is more practice using string methods, including sub-string indexing.
Solution: *See the code in FirstToLastWord.java.*

4. Write a program that asks the user to enter a favorite color, a favorite food, a favorite animal, and the first name of a friend or relative. The program should then print the following two lines, with the user's input replacing the items in italics: I had a dream that *Name* ate a *Color Animal* and said it tasted like *Food*!

For example, if the user entered blue for the color, hamburger for the food, dog for the animal, and Jake for the person's name, the output would be
I had a dream that Jake ate a blue dog and said it tasted like hamburger!
Don't forget to put the exclamation mark at the end.

Notes: This project requires careful attention to spaces in output messages. In particular, a space must be explicitly printed between the variables color and animal.
Solution: *See the code in SillySentence.java.*

5. Write a program that determines the change to be dispensed from a vending machine. An item in the machine can cost between 25 cents and a dollar, in 5-cent increments (25, 30, 35, . . ., 90, 95, or 100), and the machine accepts only a single dollar bill to pay for the item. For example, a possible dialogue with the user might be

Enter price of item
(from 25 cents to a dollar, in 5-cent increments): 45
You bought an item for 45 cents and gave me a dollar,
so your change is
2 quarters,
0 dimes, and
1 nickel.

Notes: This project is a good example to use for explaining code reuse by the simple process of copying a similar program and making modifications to it.
Solution: *See the code in VendingChange.java.*

6. Write a program that reads a four-bit binary number from the keyboard as a string and then converts it into decimal. For example, if the input is 1100, the output should be 12. *Hint*: Break the string into substrings and then convert each substring to a value for a single bit.

Notes: This project gives an opportunity to discuss binary numbers in a simple context. It requires the use of String manipulation and parseInt to convert each character into an integer that can then be used in formula.
Solution: *See the code in FromBinary.java.*

7. Many private water wells produce only 1 or 2 gallons of water per minute. One way to avoid running out of water with these low-yield wells is to use a holding tank. A family of 4 will use about 250 gallons of water per day. However, there is a "natural" water holding tank in the casing (i.e. the hole) of the well itself. The deeper the well, the more water that will be stored that can be pumped out for household use. But how much water will be available?

Write a program that allows the user to input the radius of the well casing in inches (a typical well will have a 3 inch radius) and the depth of the well in feet (assume water will fill this entire depth, although in practice that will not be true since the static water level will generally be 50 feet or more below the ground surface). The program should output the number of gallons stored in the well casing. For your reference:

The volume of a cylinder is $\pi r^2 h$ where *r* is the radius and *h* is the height.
1 cubic foot = 7.48 gallons of water.

For example, a 300 foot well full of water with a radius of 3 inches for the casing holds about 441 gallons of water -- plenty for a family of 4 and no need to install a separate holding tank.

Notes: This project gives the student practice with numerical calculations and simple input/output. Introduces problem solving with relatively straightforward conversion of units. Students are likely to encounter data type conversion issues from double to int and vice versa (e.g. radius / 12 will result in 0 if radius is an int).
Solution: *See the code in WaterWell.java.*

8. The Harris-Benedict equation estimates the number of calories your body needs to maintain your weight if you do no exercise. This is called your basal metabolic rate or BMR.

The calories needed for a woman to maintain her weight is:
BMR = 655 + (4.3 * weight in pounds) + (4.7 * height in inches) - (4.7 * age in years)

The calories needed for a man to maintain his weight is:
BMR = 66 + (6.3 * weight in pounds) + (12.9 * height in inches) - (6.8 * age in years)

A typical chocolate bar will contain around 230 calories. Write a program that allows the user to input their weight in pounds, height in inches, and age in years. The program should then output the number of chocolate bars that should be consumed to maintain one's weight for both a woman and a man of the input weight, height, and age.

Notes: This project gives the student practice with numerical calculations and simple input/output. For a slightly more challenging problem allow the height to be entered in feet and inches and have the program convert to inches.
Solution: *See the code in CandyCalculator.java.*

10. Write a program that reads a string for a date in the format *month* / *day* / *year* and displays it in the format *day* . *month* . *year*, which is a typical format used in Europe. For example, if the input is 06/17/08, the output should be 17.06.08. Your program should use JOptionPane for input and output.

Notes: This project is a simple applet that gives the students more practice with String manipulation.
Solution: *See the code in DateDisplayer.java.*

Exercises:

1. Write a fragment of code that will test whether an integer variable score contains a valid test score. Valid test scores are in the range 0 to 100.

Solution:

```
if( score >= 0 && score <= 100 )
      System.out.println("The score " + score + " is valid.");
else
      System.out.println("The score " + score + " is not valid.");
```

This code is in Fragments.java.

2. Write a fragment of code that will change the integer value stored in x as follows.
If x is even, divide x by 2. If x is odd, multiply x by 3 and subtract 1.

Solution:

```
if(x % 2 == 1)
            x = 3*x - 1;
      else
            x = x/2;
```

This code is in Fragments.java.

3. Suppose you are writing a program that asks the user to give a yes-or-no response. Assume that the program reads the user's response into the String variable response.
a. If response is yes or y, set the boolean variable accept to true otherwise set it to false.
b. How would you change the code so that it will also accept Yes and Y?

Solution:

a)
```
if( response.equals("y")  || response.equals("yes"))
            accept = true;
        else
            accept = false;
```

b) we can add in extra cases:
```
if( response.equals("y")  || response.equals("yes") ||
     response.equals("Y")  || response.equals("Yes"))
            accept = true;
        else
            accept = false;
```
or modify reponse before the if statement.
```
response = response.toLower();
if( response.equals("y")  || response.equals("yes"))
            accept = true;
        else
            accept = false;
```

This code is in Fragments.java.

4. Consider the following fragment of code:

```
if (x > 5)
        System.out.println("A");
else if (x < 10)
        System.out.println("B");
else
        System.out.println("C");
```

What is displayed if x is

a. 4; b. 5; c. 6; d. 9; e. 10; f. 11

Solution:

a) B
b) B
c) A
d) A
e) A
f) A

5. Consider the following fragment of code:

```
if (x > 5)
{
System.out.println("A");
if (x < 10)
System.out.println("B");
}
else
System.out.println("C");
```

What is displayed if x is

a. 4; b. 5; c. 6; d. 9; e. 10; f. 11

Solution:

a) C
b) C
c) A B
d) A B
e) A
f) A

6. We would like to assess a service charge for cashing a check. The service charge depends on the amount of the check. If the check amount is less than $10, we will charge $1. If the amount is greater than $10 but less than $100, we will charge 10% of the amount. If the amount is greater than $100, but less than $1,000, we will charge $5 plus 5% of the amount. If the value is over $1,000, we will charge $40 plus 1% of the amount. Use a multibranch if-else statement in a fragment of code to compute the service charge.

Solution:

```
if(amount < 10.0)
            serviceCharge = 1.0;
        else if(amount < 100.0)
            serviceCharge = 0.1 * amount;
        else if(amount < 1000.0)
            serviceCharge = 5.0 + 0.05 * amount;
        else
            serviceCharge = 40.0 + 0.01 * amount;
```

This code is in Fragments.java.

7. What is the value of each of the following boolean expressions if x is 5, y is 10,
and z is 15?
a. (x < 5 && y > x)
b. (x < 5 || y > x)
c. (x > 3 || y < 10 && z == 15)
d. (!(x > 3) && x != z || x + y == z)

Solution:

a) false
b) true
c) true
d) true

This code is in Fragments.java.

8. The following code fragment will not compile. Why?

```
if !x > x + y
        x = 2 * x;
else
        x = x + 3;
```

Solution:

We need to add a couple pairs of parentheses. The boolean expression for an if statement must be in parentheses. And we must apply the not operator to a boolean value.

if (!(x > x + y))

9. Consider the boolean expression ((x > 10) || (x < 100)). Why is this expression probably not what the programmer intended?

Solution:

This expression will always evaluate to true.

10. Consider the boolean expression ((2 < 5) && (x < 100)). Why is this expression probably not what the programmer intended?

Solution:

This 2<5 is true and this expression is equivalent to just x<100.

11. Write a switch statement to convert a letter grade into an equivalent numeric value on a four-point scale. Set the value of the variable gradeValue to 4.0 for an A, 3.0 for a B, 2.0 for a C, 1.0 for a D, and 0.0 for an F. For any other letter, set the value to 0.0 and display an error message.

Solution:

```
switch(letter){
            case 'A':
                gradeValue = 4.0;
                break;
            case 'B':
                gradeValue = 3.0;
                break;
            case 'C':
                gradeValue = 2.0;
                break;
            case 'D':
                gradeValue = 1.0;
                break;
            case 'F':
                gradeValue = 0.0;
                break;
            default:
                gradeValue = 0.0;
                System.out.println("The grade "
                       + letter + " is not valid");
}
```

This code is in Fragments.java.

12. Consider the previous question, but include + or - letter grades. A+ is 4.25, A- is 3.75, B+ is 3.25, B- is 2.75, and so on.
a. Why can't we use one switch statement with no other conditionals to convert these additional letter grades?
b. Write a fragment of code that will do the conversion using a multibranch ifelse statement.
c. Write a fragment of code that will do the conversion using nested switch statements.

Solution:
a) The grade A+ would be a string, but we cannot use switch on a string.

b)

```
if(enhancedLetterGrade.equals("A+"))
            gradeValue = 4.25;
        else if(enhancedLetterGrade.equals("A"))
            gradeValue = 4.0;
        else if(enhancedLetterGrade.equals("A-"))
            gradeValue = 3.75;
        else if(enhancedLetterGrade.equals("B+"))
            gradeValue = 3.25;
        else if(enhancedLetterGrade.equals("B"))
            gradeValue = 3.0;
        else if(enhancedLetterGrade.equals("B-"))
            gradeValue = 2.75;
        else if(enhancedLetterGrade.equals("C+"))
            gradeValue = 2.25;
        else if(enhancedLetterGrade.equals("C"))
            gradeValue = 2.0;
        else if(enhancedLetterGrade.equals("C-"))
            gradeValue = 1.75;
        else if(enhancedLetterGrade.equals("D+"))
            gradeValue = 1.25;
        else if(enhancedLetterGrade.equals("D"))
            gradeValue = 1.0;
        else if(enhancedLetterGrade.equals("D-"))
            gradeValue = 0.75;
        else if(enhancedLetterGrade.equals("F+"))
            gradeValue = 0.25;
        else if(enhancedLetterGrade.equals("F"))
            gradeValue = 0.0;
        else {
            gradeValue = 0.0;
            System.out.println("The grade "
               + enhancedLetterGrade + " is not valid");
```

```
    }
```

c)

```
char letterPart = enhancedLetterGrade.charAt(0);
char plusPart = '0';

if(enhancedLetterGrade.length()>1)
    plusPart = enhancedLetterGrade.charAt(1);

switch(letterPart){
    case 'A':
        gradeValue = 4.0;
        switch(plusPart){
            case '+':
                gradeValue += 0.25;
                break;
            case '-':
                gradeValue -= 0.25;
                break;
        }
        break;
    case 'B':
        gradeValue = 3.0;
        switch(plusPart){
            case '+':
                gradeValue += 0.25;
                break;
            case '-':
                gradeValue -= 0.25;
                break;
        }
        break;
    case 'C':
        gradeValue = 2.0;
        switch(plusPart){
            case '+':
                gradeValue += 0.25;
                break;
            case '-':
                gradeValue -= 0.25;
                break;
        }
        break;
    case 'D':
        gradeValue = 1.0;
        switch(plusPart){
            case '+':
                gradeValue += 0.25;
                break;
            case '-':
                gradeValue -= 0.25;
                break;
        }
        break;
```

```
            case 'F':
                gradeValue = 0.0;
                switch(plusPart){
                    case '+':
                        gradeValue += 0.25;
                        break;
                }
                break;
            default:
                gradeValue = 0.0;
                System.out.println("The grade " + letter
                           + " is not valid");
}
```

This code is in Fragments.java.

Practice Programs:

3. Write a program that reads three strings from the keyboard. Although the strings are in no particular order, display the string that would be second if they were arranged lexicographically.

Notes: This project does a simple sort. Nested if statements are a natural way to accomplish this task.
Solution: *See the code in MiddleString.java.*

4. Write a program that reads a one-line sentence as input and then displays the following response: If the sentence ends with a question mark (?) and the input contains an even number of characters, display the word Yes. If the sentence ends with a question mark and the input contains an odd number of characters, display the word No. If the sentence ends with an exclamation point (!), display the word Wow. In all other cases, display the words You always say followed by the input string enclosed in quotes. Your output should all be on one line. Be sure to note that in the last case, your output must include quotation marks around the echoed input string. In all other cases, there are no quotes in the output. Your program does not have to check the input to see that the user has entered a legitimate sentence.

Notes: This project requires a three-way selection statement and gives more practice with string methods. The case statement can be used since the control expression is a single character, and, in fact, that is what the solution in this manual uses. It may be instructive to show the code using if/else, instead, and compare them for readability. This project will be extended to use a loop in the next chapter.
Solution: *See the code in AskOrTellMeSelection.java.*

5. Write a program that allows the user to convert a temperature given in

degrees from either Celsius to Fahrenheit or Fahrenheit to Celsius. Use the following formulas:

$Degrees\_C = 5\ (Degrees\_F - 32) / 9$

$Degrees\_F = (9\ (Degrees\_C) / 5) + 32$

Prompt the user to enter a temperature and either a C or c for Celsius or an F or f for Fahrenheit. Convert the temperature to Fahrenheit if Celsius is entered, or to Celsius if Fahrenheit is entered. Display the result in a readable format. If anything other than C, c, F, or f is entered, print an error message and stop.

Notes:

This project uses selection to enhance the program FtoC developed in Chapter 2. The solution includes a default case if an incorrect character (anything other than a 'C' or 'F', either upper or lower case) is entered for the units. A common error is to write the while control expression as an OR instead of an AND, so the loop does not end when either 'Q' or 'q' is entered. With an OR expression one or both sides of the expression will always be true (if 'Q' is entered, the variable `quit` is not equal to 'q', and vice versa); `quit` must be both not equal to 'Q' *and* not equal to 'q' to enter the loop.

Solution:

See the code in TemperatureConversionSelection.java.

Programming Projects

3. Suppose that we are working for an online service that provides a bulletin board for its users. We would like to give our users the option of filtering out profanity. Suppose that we consider the words *cat*, *dog*, and *llama* to be profane. Write a program that reads a string from the keyboard and tests whether the string contains one of our profane words. Your program should find words like *cAt* that differ only in case. *Option*: As an extra challenge, have your program reject only lines that contain a profane word exactly. For example, *Dogmatic concatenation is a small category* should not be considered profane.

Notes: This project provides an opportunity to discuss some of the difficulties of in filtering information. It requires a conversion to lowercase and a compound Boolean expression.
Solution: *See the code in ProfaneFilter.java.*

4. Write a program that reads a string from the keyboard and tests whether it contains a valid date. Display the date and a message that indicates whether it is valid. If it is not valid, also display a message explaining why it is not valid. The input date will have the format *mm*/*dd*/*yyyy*. A valid month value *mm* must be from 1 to 12 (January is 1). The day value *dd* must be from 1 to a value that is appropriate for the given month. September, April, June, and November each have 30 days. February has 28 days except for leap years when it has 29. The remaining months all have 31 days each. A leap year is any year that is divisible by 4 but not divisible by 100 unless it is also divisible by 400.

Notes: This project provides an opportunity to introduce format checking. The solution uses case logic in combination with compound Boolean expressions.
Solution: *See the code in CheckDate.java.*

5. Repeat the calorie counting program described in Programming Project 8 from Chapter 2. This time ask the user to input the string "M" if the user is a man and "W" if the user is a woman. Use only the male formula to calculate calories if "M" is entered and use only the female formula to calculate calories if "W" is entered. Output the number of chocolate bars to consume as before.

Notes: This project is a relatively simple addition using String input.and an if statement.
References: Programming Project 2.8
Solution: *See the code in CandyCalculator2.java.*

6. Repeat Programming Project 10 but in addition ask the user if he or she is:

A. Sedentary
B. Somewhat active (exercise occasionally)
C. Active (exercise 3-4 days per week)
D. Highly active (exercise every day)

If the user answers "Sedentary" then increase the calculated BMR by 20 percent. If the user answers "Somewhat active" then increase the calculated BMR by 30 percent. If the user answers "Active" then increase the calculated BMR by 40 percent. Finally, if the user answers "Highly active" then increase the calculated BMR by 50 percent. Output the number of chocolate bars based on the new BMR value.

Notes: This project requires additional logic and user input from Programming Project 2.8
References: Programming Project 5
Solution: *See the code in CandyCalculator3.java.*

8. Write a program to play the rock-paper-scissor game. Each of two users types in either P, R, or S. The program then announces the winner as well as the basis for determining the winner: paper covers rock, rock breaks scissors, scissors cuts paper, or nobody wins. Your program should allow the users to use lowercase as well as uppercase letters.

Notes: This project is a relatively simple program with a large number of if-else statements inside a loop.
Solution: *See the code in RockPaperScissors.java.*

9. Write a JavaFX program to draw the five interlocking rings that are the symbol of the Olympics. The color of the rings, from left to right, is blue, yellow, black, green, and red.

Notes: Use setStroke(Color) to set the color of the line   
References: Exercise 1.18
Solution: *See the code in Rings.java.*

10. Repeat Programming Project 8 in Chapter 1, but add yes-or-no dialogs to allow the user to make the following color changes:

- Change the color of the solid center circle from black to red.
- Change the color of the outer circle from black to blue.
- Change the color of the spines from black to green.

Notes: This project is an enhancement of a similar project from Chapter 1. This JavaFX program uses dialog windows and selection to change the appearance of the image drawn by the program. Use setStroke to set the line color and setFill to set the fill color.
References: Project 1.8
Solution: *See the code in Icon.java.*

Exercises:

1. Write a fragment of code that will read words from the keyboard until the word done is entered. For each word except done, report whether its first character is equal to its last character. For the required loop, use a
a. while statement
b. do-while statement

Solution:

```
Scanner reader = new Scanner(System.in);
String word = "";

System.out.println("Please enter words ending with done.");

word = reader.next();
while(!word.equals("done")){
    if(word.charAt(0) == word.charAt(word.length()-1)){
        System.out.println("The word " + word
          + " has first and last characters that are the
          same.");
    }
    word = reader.next();
}


System.out.println("Please enter words ending with done.");

boolean finished = false;
do{
    word = reader.next();
    if(word.equals("done"))
        finished = true;
    else

        if(word.charAt(0) == word.charAt(word.length()-1)){
        System.out.println("The word " + word
                + " has first and last characters that are
          the same.");
        }
} while (!finished);
```

This code is in Fragments.java.

2. Develop an algorithm for computing the month-by-month balance in your savings account. You can make one transaction—a deposit or a withdrawal—each month. Interest is added to the account at the beginning of each month. The monthly interest rate is the yearly percentage rate divided by 12.

Solution:

1. *For month goes from 1 to 12*
    - *1.1. Compute the monthly interest*
    - *1.2. Compute the interest and add to the balance*
    - *1.3. Ask if the user is making a deposit or withdrawal.*
    - *1.4. Get the amount from the user*
    - *1.5. If making a deposit*
        - *1.5.1. Add the amount to the balance*
    - *1.6. Else*
        - *1.6.1. Subtract the amount from the balance*
    - *1.7. Display the current balance*

3. Develop an algorithm for a simple game of guessing at a secret five-digit code. When the user enters a guess at the code, the program returns two values: the number of digits in the guess that are in the correct position and the sum of those digits. For example, if the secret code is 53840, and the user guesses 83241, the digits 3 and 4 are in the correct position. Thus, the program should respond with 2 and 7.
Allow the user to guess a fixed number of times.

Solution:

1. *Generate a secret code*
2. *For guess is 1 to max*
   - *2.1. Get the users guess for the 5 digit code*
   - *2.2. Let correct be zero*
   - *2.3. Let sum be zero*
   - *2.4. For each digit*
     - *2.4.1. If the digit in the guess matches the users guess*
       - *2.4.1.1.Add 1 to correct*
       - *2.4.1.2.Add the value of the digit to sum*
   - *2.5. If correct is 5*
     - *2.5.1. Congratulate the user on guessing the code*
     - *2.5.2. Exit the program*
   - *2.6. Else*
     - *2.6.1. Display correct and sum*
   - *2.7. Ask if the user is making a deposit or withdrawal.*
3. *Display a message telling the user they did not guess the code within the maximum number of allowed guesses.*

4. Write a fragment of code that will compute the sum of the first *n* positive odd integers. For example, if *n* is 5, you should compute 1 + 3 + 5 + 7 + 9.

Solution:

```
int sum = 0;
int odd = 1;
for(int i=0; i<n; i++){
     sum += odd;
     odd += 2;
}

System.out.println("The sum of the first " + n
     + " odd numbers is " + sum);
```

This code is in Fragments.java.

5. Convert the following code so that it uses nested while statements instead of for statements:

```
int s = 0;
int t = 1;
for (int i = 0; i < 10; i++)
{
s = s + i;
for (int j = i; j > 0; j--)
{
t = t * (j - i);
}
s = s * t;
System.out.println(“T is “ + t);
}
System.out.println(“S is “ + s);
```

Solution:

```
int s = 0;
int t = 1;
int i = 0;
while(i<10){
        s = s + i;
        int j = i;
        while(j>0){
            t = t * (j - i);
            j--;
        }
        s = s * t;
        System.out.println("T is " + t);
        i++;
}
System.out.println("S is " + s);
```

This code is in Fragments.java.

6. Write a for statement to compute the sum $1 + 2^2 + 3^2 + 4^2 + 5^2 + ... + n^2$.

Solution:

```
sum = 0;
for( i=1; i<=n; i++){
     sum += i*i;
}
System.out.println("The sum is " + sum);
```

This code is in Fragments.java.

7. (*Optional*) Repeat the previous question, but use the comma operator and omit the for statement's body.

Solution:

```
for( sum=0, i=1; i<=n; sum += i*i, i++){}
System.out.println("The sum is " + sum);
```

This code is in Fragments.java.

11. Suppose we attend a party. To be sociable, we will shake hands with everyone else. Write a fragment of code using a for statement that will compute the total number of handshakes that occur. (*Hint*: Upon arrival, each person shakes hands with everyone that is already there. Use the loop to find the total number of handshakes as each person arrives.)

Solution:

```
int numberAttending = 8;
int handShakes = 0;
for( int person=1; person <= numberAttending; person++){
            handShakes += (person - 1);
            // When person k arrives, they will
            //shake hands with the k-1 people already
there
}
System.out.println("The sum is " + sum);
```

This code is in Fragments.java.

12. Define an enumeration for each of the months in the year. Use a for-each statement to display each month.

Solution:

```
enum Month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
DEC}
for (Month nextMonth : Month.values())
            System.out.print(nextMonth + " ");
System.out.println();
```

This code is in Fragments.java.

13. Write a fragment of code that computes the final score of a baseball game. Use a loop to read the number of runs scored by both teams during each of nine innings. Display the final score afterwards.

Solution:

```
        int innings = 9;
        int team1Total = 0;
        int team2Total = 0;

        //Scanner reader = new Scanner(System.in);

        for( int inning=1; inning <= innings; inning++){
            System.out.println("How many runs were scored in inning "
                    + inning + " by each team?");
            team1Total = team1Total + reader.nextInt();
            team2Total = team2Total + reader.nextInt();
        }
        System.out.println("The first team scored: " + team1Total
                + " and the second team scored: " + team2Total);
```

This code is in Fragments.java.

14. Suppose that you work for a beverage company. The company wants to know the optimal cost for a cylindrical container that holds a specified volume. Write a fragment of code that uses an ask-before-iterating loop. During each iteration of the loop, your code will ask the user to enter the volume and the radius of the cylinder. Compute and display the height and cost of the container. Use the following formulas, where V is the volume, r is the radius, h is the height, and C is the cost.

$$h = \frac{V}{\pi r^2}$$

$$C = 2\pi r(r + h)$$

Solution:

```
double volume = 0.0;
double height = 0.0;
double radius = 0.0;
double cost = 0.0;
String answer;

//Scanner reader = new Scanner(System.in);

do
{
    System.out.println("Enter the volume and radius "
            + "of the cylinder ");
    volume = reader.nextDouble();
    radius = reader.nextDouble();

    height = volume / (Math.PI * radius * radius);
    cost = 2 * Math.PI * radius *(radius + height);

    System.out.println("The height required is: "
            + height + " and the cost is " + cost);

    System.out.println("Do you want to compute the "
            + "cost for a different volume & height?");
    System.out.println("Enter yes or no.");
    answer = reader.next();
} while (answer.equalsIgnoreCase("yes"));
```

This code is in Fragments.java.

15. Suppose that we want to compute the geometric mean of a list of positive values. To compute the geometric mean of *k* values, multiply them all together and thenCcompute the *k*th root of the value. For example, the geometric mean of 2, 5, and 7 is $\sqrt[3]{2 \times 5 \times 7}$. Use a loop with a sentinel value to allow a user to enter an arbitrary number of values. Compute and display the geometric mean of all the values, excluding the sentinel. (*Hint*: Math.pow(x, 1.0 / k) will compute the *k*th root of *x*.)

Solution:

```
int count = 0;
double data = 0.0;
double product = 1.0;

//Scanner reader = new Scanner(System.in);

System.out.println("Enter data values for which" +
        " to compute the geometric mean.");
System.out.println("Enter a negative number after");
System.out.println("you have entered all the data values.");
data = reader.nextDouble();
while (data >= 0) {
    product = product * data;
    count++;
    data = reader.nextDouble();
}
System.out.println("The geometric mean is "
        + Math.pow(product, 1.0/count));
```

This code is in Fragments.java.

16. Imagine a program that compresses files by 80 percent and stores them on storage media. Before the compressed file is stored, it must be divided into blocks of 512 bytes each. Develop an algorithm for this program that first reads the number of blocks available on the storage media. Then, in a loop, read the uncompressed size of a file and determine whether the compressed file will fit in the space left on the storage media. If so, the program should compress and save the file. It continues until it encounters a file that will exceed the available space on the media. For example, suppose the media can hold 1000 blocks. A file of size 1100 bytes will compress to size 880 and require 2 blocks. The available space is now 998 blocks. A file of size 20,000 bytes will compress to size 16,000 and require 32 blocks. The available space is now 966.

Solution:

1. *Get the number of free blocks and assign it to the variable free*
2. *Let haveSpace be true*
3. *While haveSpace*
   - *3.1. Get size of the file in bytes*
   - *3.2. Compute the compressed size of the file in bytes*
   - *3.3. Determine the number of blocks needed for the file*
   - *3.4. If the number of blocks needed is less than the free space*
     - *3.4.1. Reduce the number of free blocks*
   - *3.5. Else*
     - *3.5.1. Let haveSpace be false*
     - *3.5.2. guess is 1 to max*

17. Create an applet that draws a pattern of circles whose centers are evenly spaced along a horizontal line. Use six constants to control the pattern: the number of circles to draw, the diameter of the first circle, the *x*- and *y*-coordinates of the center of the first circle, the distance between adjacent centers, and the change in the diameter of each subsequent circle.

Solution:

See the code in MultipleCircles.java.

18. What does the following fragment of code display? What do you think the programmer intended the code to do, and how would you fix it?

```
int product = 1;
int max = 20;
for (int i = 0; i <= max; i++)
        product = product * i;
System.out.println("The product is " + product);
```

Solution:

The product displayed is 0. It is likely that the programmer wanted to compute the product of the first 20 integer values. Change the for loop to be:

```
for (int i = 1; i <= max; i++)
```

19. What does the following fragment of code display? What do you think the programmer intended the code to do, and how would you fix it?

```
int sum = 0;
int product = 1;
int max = 20;
for (int i = 1; i <= max; i++)
sum = sum + i;
product = product * i;
System.out.println("The sum is " + sum +
" and the product is " + product);
```

Solution:

The sum is 210 and the product is 21.0. It is likely that the programmer wanted to compute the product of the first 20 integer values, only the statement

```
sum = sum + i;
```

is inside the body of the for loop. Change the for loop to be:

```
for (int i = 1; i <= max; i++){
      sum = sum + i;
      product = product * i;
}
```

Practice Programs:

1. Repeat Practice Program 4 of Chapter 3, but use a loop that reads and processes sentences until the user says to end the program.

Notes: This project is an extension of a project from the previous chapter to use looping.
References: Project 3.4
Solution: *See the code in AskOrTellMe.java.*

2. Write a program that implements your algorithm from Exercise 2.

Notes: This project uses a loop to simulate a simplified checking account. It is strongly recommended that students develop an algorithm first (Exercise 2) to instill good design practices. This gives them a chance to see how well their design worked.
References: Exercise 4.2
Solution: *See the code in MonthByMonth.java.*

3. Repeat Practice Program 5 of Chapter 3, but use a loop so the user can convert other temperatures. If the user enters a letter other than *C* or *F*—in either uppercase or lowercase—after a temperature, print an error message and ask the user to reenter a valid selection. Do not ask the user to reenter the numeric portion of the temperature again, however. After each conversion, ask the user to type Q or q to quit or to press any other key to repeat the loop and perform another conversion.

Notes:

This project uses loops to further enhance the program TemperatureConversionSelction developed in Chapter 3 (which enhanced FtoC developed in Chapter 2). A common error is to write the while control expression as an OR instead of an AND, so the loop does not end when either 'Q' or 'q' is entered. With an OR expression one or both sides of the expression will always be true (if 'Q' is entered, the variable quit is not equal to 'q', and vice versa); quit must be both not equal to 'Q' *and* not equal to 'q' to enter the loop.

References:

Practice Program 3.5

Solution:

See the code in TemperatureConversion.java.

5. Write a program to read a list of nonnegative integers and to display the largest integer, the smallest integer, and the average of all the integers. The user indicates the end of the input by entering a negative sentinel value that is not used in finding the largest, smallest, and average values. The average should be a value of type double, so that it is computed with a fractional part.

Notes: The solution for this project includes a default case to print a message if no positive integers are entered.
Solution: *See the code in LargeSmallAverage.java.*

6. Write a program to read a list of exam scores given as integer percentages in the range 0 to 100. Display the total number of grades and the number of grades in each letter-grade category as follows: 90 to 100 is an A, 80 to 89 is a B, 70 to 79 is a C, 60 to 69 is a D, and 0 to 59 is an F.
Use a negative score as a sentinel value to indicate the end of the input. (The negative value is used only to end the loop, so do not use it in the calculations.) For example, if the input is

98 87 86 85 85 78 73 72 72 72 70 66 63 50 −1

the output would be

Total number of grades = 14
Number of A's = 1
Number of B's = 4
Number of C's = 6
Number of D's = 2
Number of F's = 1

Notes: This project requires both a sentinel controlled loop, multi-way selection, and running sums that need to be initialized to zero to guarantee correct results.
Solution: *See the code in NumberOfGrades.java.*

7. Combine the programs from Programming Projects 5 and 6 to read integer exam scores in the range 0 to 100 and to display the following statistics:
Total number of scores
Total number of each letter grade
Percentage of total for each letter grade
Range of scores: lowest and highest
Average score
As before, enter a negative score as a sentinel value to end the data input and display the statistics.

Notes: This project combines parts of the previous two projects. There is the potential for incorrect results due to integer division truncation, so a cast to float is used for the percent and average calculations. The solution in this manual also does a check to see if no scores have been entered, and, if so, displays a special message rather than the statistics.
References: Practice Program 4.5, Practice Program 4.6
Solution: *See the code in ExamStatistics.java.*

Programming Projects:

3. Write a program that reads a bank account balance and an interest rate and displays the value of the account in ten years. The output should show the value of the account for three different methods of compounding interest: annually, monthly, and daily. When compounded annually, the interest is added once per year at the end of the year. When compounded monthly, the interest is added 12 times per year. When computed daily, the interest is added 365 times per year. You do not have to worry about leap years; assume that all years have 365 days. For annual interest, you can assume that the interest is posted exactly one year from the date of deposit. In other words, you do not have to worry about interest being posted on a specific day of the year, such as December 31. Similarly, you can assume that monthly interest is posted exactly one month after it is deposited. Since the account earns interest on the interest, it should have a higher balance when interest is posted more frequently. Be sure to adjust the interest rate for the time period of the interest. If the rate is 5 percent, you use 5/12 percent when posting monthly interest and 5/365 percent when posting daily interest. Perform this calculation using a loop that adds in the interest for each time period, that is, do not use some sort of algebraic formula. Your program should have an outer loop that allows the user to repeat this calculation for a new balance and interest rate. The calculation is repeated until the user asks to end the program.

Notes:

This project is a bank account problem, so it requires the same consideration about money calculations as the mortgage problem in Chapter 2. The round function in Java will not be introduced until later, so a discussion of the special problems associated with money calculations may be postponed until then. One solution in this manual, BankAccount.java, takes a simplistic approach, uses floating point values and does not round to the nearest penny. For contrast, a second program, BankAccount2.java, which deals with the rounding problem (without using Java's round function), is also provided.

Solution:

See the code in BankAccount.java and BankAccount2.java.

4. Modify Programming Project 5 from Chapter 2 to check the validity of input data. Valid input is no less than 25 cents, no more than 100 cents, and an integer multiple of 5 cents. Compute the change only if a valid price is entered. Otherwise, print separate error messages for any of the following invalid inputs: a price under 25 cents, a price that is not an integer multiple of 5, and a price that is more than a dollar.

Notes: This project is a simple modification of Project 5 from Chapter 2. Three if statements are added to detect invalid input: less than 25 cents, more than a dollar, and not a multiple of 5 cents.
References: Project 2.5
Solution: *See the code in VendingChangeImproved.java.*

6. Write a program that asks the user to enter the size of a triangle (an integer from 1 to 50). Display the triangle by writing lines of asterisks. The first line will have one asterisk, the next two, and so on, with each line having one more asterisk than the previous line, up to the number entered by the user. On the next line write one fewer asterisk and continue by decreasing the number of asterisks by 1 for each successive line until only one asterisk is displayed. *Hint*: Use nested for loops; the outside loop controls the number of lines to write, and the inside loop controls the number of asterisks to display on a line. For example, if the user enters 3, the output would be

```
*
**
***
**
*
```

Notes:

This project includes input checking so it will not print any lines with asterisks if the user enters a number less than 1 or greater than 50. If a valid number is entered, two pairs of nested for-loops are used to print the triangle of asterisks. The first nested pair prints the lines with an increasing number of asterisks, starting with one and increasing by one per line up to a maximum of the number entered by the user. The second nested pair of for-loops prints the lines with a decreasing number of asterisks, starting with (number –1) down to 1. The outside loops count through the lines printed and the inside loops count through the number of asterisks printed on the line. The number of asterisks to print on any line is its line number set by the outside loop.

Solution:

See the code in TriangleOfAsterisks.java.

7. Write a program that simulates a bouncing ball by computing its height in feet at each second as time passes on a simulated clock. At time zero, the ball begins at height zero and has an initial velocity supplied by the user. (An initial velocity of at least 100 feet per second is a good choice.) After each second, change the height by adding the current velocity; then subtract 32 from the velocity. If the new height is less than zero, multiply both the height and the velocity by -0.5 to simulate the bounce. Stop at the fifth bounce. The output from your program should have the following form:

```
Enter the initial velocity of the ball: 100
Time: 0 Height: 0.0
Time: 1 Height: 100.0
Time: 2 Height: 168.0
Time: 3 Height: 204.0
Time: 4 Height: 208.0
Time: 5 Height: 180.0
Time: 6 Height: 120.0
Time: 7 Height: 28.0
Bounce!
Time: 8 Height: 48.0
```

Notes:

This project is a numerical simulation of a bouncing ball. The simulation of the bounce is not particularly realistic, but it avoids dealing with issues of determining exactly when the ball hits the surface. The quality of the simulation is sensitive to the combination of the input parameters.

Solution:

See the code in Bounce.java.

8. You have three identical prizes to give away and a pool of 10 finalists. The finalists are assigned numbers from 1 to 10. Write a program to randomly select the numbers of 3 finalists to receive a prize. Make sure not to pick the same number twice. For example, picking finalists 3, 6, 2 would be valid but picking 3, 3, 11 would be invalid because finalist number 3 is listed twice and 11 is not a valid finalist number. Random number generation is discussed in Chapter 6, but for this problem you can insert the

following line of code to generate a random number between 1 and 10:

```
int num = (int) (Math.random() * 10) +1;
```

Notes: This project uses random numbers in a loop. You might wish to introduce the Random class instead of Math.random().
Solution: *See the code in RandomWinners.java.*

9. Suppose we can buy a chocolate bar from the vending machine for $1 each. Inside every chocolate bar is a coupon. We can redeem 6 coupons for one chocolate bar from the machine. This means that once you have started buying chocolate bars from the machine, you always have some coupons. We would like to know how many chocolate bars can be eaten if we start with N dollars and always redeem coupons if we have enough for an additional chocolate bar.

For example, with 6 dollars we could consume 7 chocolate bars after purchasing 6 bars giving us 6 coupons and then redeeming the 6 coupons for one bar. This would leave us with one extra coupon. For 11 dollars, we could have consumed 13 chocolate bars and still have one coupon left. For 12 dollars, we could have consumed 14 chocolate bars and have two coupons left.

Write a program that inputs a value for N and outputs how many chocolate bars we can eat and how many coupons we would have leftover. Use a loop that continues to redeem coupons as long as there are enough to get at least one chocolate bar.

Notes: Students often attempt this problem by trying to find a simple formula instead of simulating the process in a loop. This also makes a good problem to re-do later after

covering recursion.
Solution: *See the code in ChocolateCoupons.java.*

10. Holy digits Batman! The Riddler is planning his next caper somewhere on Pennsylvania Avenue. In his usual sporting fashion, he has left the address in the form of a puzzle. The address on Pennsylvania is a four digit number where:

- All four digits are different
- The digit in the thousands place is three times the digit in the tens place
- The number is odd
- The sum of the digits is 27

Write a program that uses a loop (or loops) to find the address where the Riddler plans to strike.

Notes: Loop over all the digits in the address. You can generate an integer that corresponds to the digits and check if it meets the constraints, or check if individual digits meet the constraints.
Solution: *See the code in Riddler.java*

11. Your country is at war and your enemies are using a secret code to communicate with each other. You have managed to intercept a message that reads as follows:

    `:mmZ\dxZmx]Zpgy`

    The message is obviously encrypted using the enemy's secret code. You have just learned that their encryption method is based upon the ASCII code (see Appendix 7). Individual characters in a string are encoded

using this system. For example, the character ‘A’ is encoded using the number 65 and ‘B’ is encoded using the number 66. Your enemy's secret code takes each letter of the message and encrypts it as follows:

```
if (OriginalChar + Key > 126) then
    EncryptedChar = 32 + ((OriginalChar + Key) - 127)
else
    EncryptedChar = (OriginalChar + Key)
```

For example, if the enemy uses Key = 10 then the message "Hey" would initially be represented as:

Character	ASCII code
H	72
e	101
y	121

And "Hey" would be encrypted as:

Encrypted H = (72 + 10) = 82 = R in ASCII
Encrypted e = (101 + 10) = 111 = o in ASCII
Encrypted y = 32 + ((121 + 10) - 127) = 36 = $ in ASCII

Consequently, "Hey" would be transmitted as "Ro$." Write a program Java that decrypts the intercepted message. You only know that the key used is a number between 1 and 100. Your program should try to decode the message using all possible keys between 1 and 100. When you try the valid key, the message will make sense. For all other keys, the message will appear as gibberish. Since there are only 100 keys this would obviously be a pretty crummy encryption system. This Programming Project will require you to explore a bit on your own how to convert from a char to a number, process the number, then convert it back to a char. See Chapter 2 for String methods. You will want to use charAt(). Important: Note that the secret code has a \ so you will need to escape encode it by using \\ if you hard-code it in your program.

Notes:

This program requires some knowledge of how to map from ASCII to integers, perform arithmetic, and map back to ASCII.

Solution:

See the code in Decrypt.java

12. Repeat the Programming Project 7, but write the program as a JavaFX application. Use a constant for the initial velocity of the ball. Draw a circle for the position of the ball at each second. The *y*-coordinate should be proportional to the height of the ball, and the *x*coordinate should change by a small constant amount.

Notes:

This application shows a graphical representation of the previous project. If the time increment used is too large, the resulting picture may look chaotic and not at all like a trajectory.

References:

Project 4.7

Solution:

See the code in BounceJavaFX.java. If the scale factor and initial velocity are chosen badly, your picture may not look very much like a bouncing ball. Instead, it may look chaotic. For example, try initial velocity of 100 and scale of 2.

14. Create a JavaFX application that draws a pattern of evenly spaced circles. Use four constants to control the pattern: the number of circles to draw, the radius of the first circle, the change in the radius of each subsequent circle, and the change in the *x*-coordinate of the circle.

Notes:

This application uses looping to control drawing a number of circles. Changing the parameters results in interesting patterns.

Solution:

See the code in LineCircles.java.

15. (*Challenge*) Repeat the previous project, but position the centers of the circles on a spiral. The center of each circle will depend on both an angle and a distance from the origin. A constant change in both the angle and the distance will result in a spiral pattern.

Notes: This application is an extension of the previous program to draw the circles on a spiral. Knowledge of converting polar coordinates to Cartesian coordinates is helpful.
References: Project 4.14
Solution: *See the code in SpiralCircles.java.*

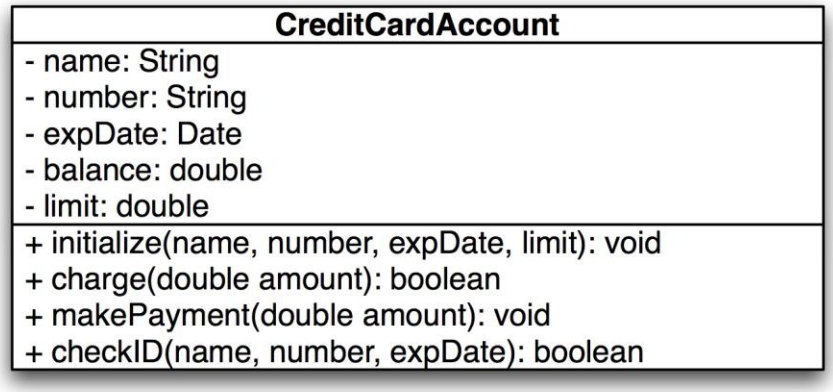
16. Write a JavaFX application that displays a series of pictures of a person with arms, legs, and of course a head. Use a happy face for the head. Use ovals for the body, arms, and legs. Draw a sequence of figures that appear one after the other, as in Listing 4.9. Make the figures assume a running position. Change the color of the person's face in each succeeding figure, going from white to pink to red to yellow to green. Have the smiling face gradually change its mouth shape from a smile on the first person to a frown on the last person. Use a switch statement to choose the color. Embed the switch statement in a loop.

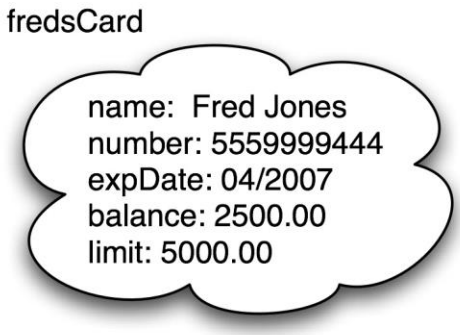
Notes: This program is based on the MultipleFaces.java program in Listing 4.9. Rather than using odd and even counter values to determine color, this program uses a switch to determine the face color and the height adjustment of the mouth. The mouth height adjustment is used to gradually change the mouth from a smile to a frown. Also, a number of constants are added for the position and size of the arms, legs, and body.
References: Listing 4.9
Solution: *See the code in RunningFaces.java*

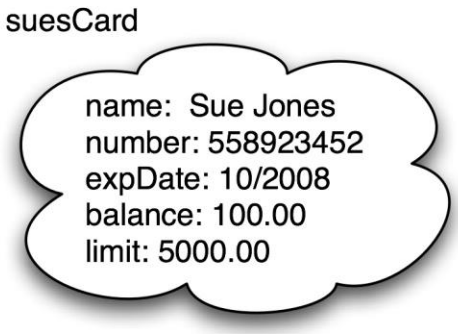
Exercises:

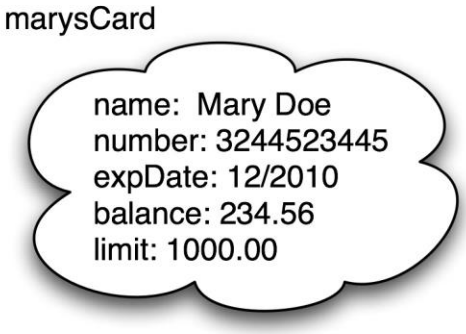
2. Repeat Exercise 1 for a credit card account instead of a credit card. An account represents the charges and payments made using a credit card.

Solution:



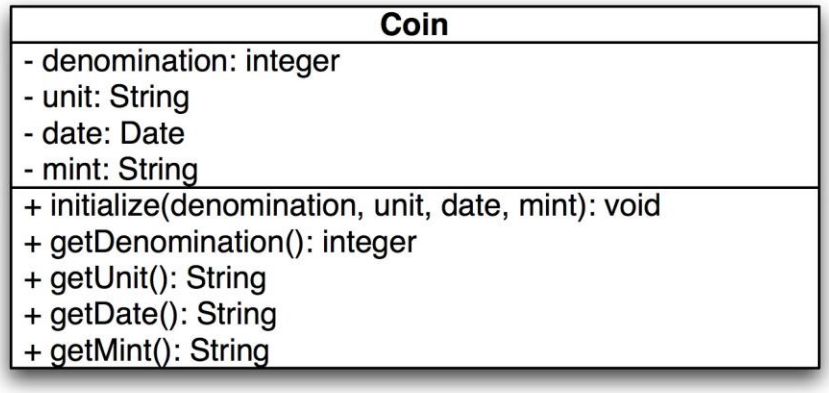


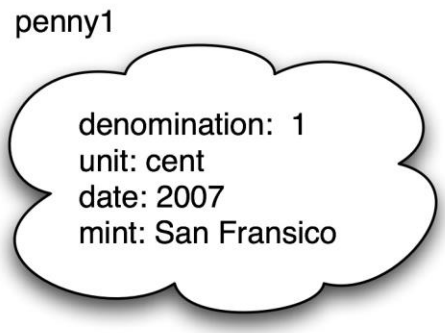


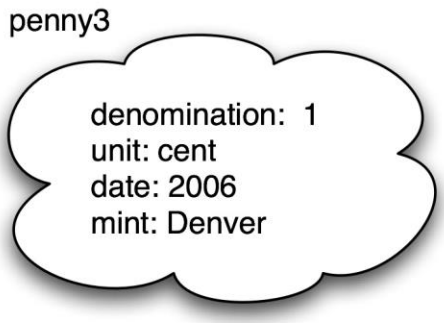


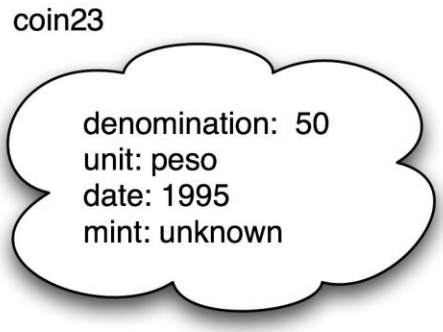
3. Repeat Exercise 1 for a coin instead of a credit card.

Solution:



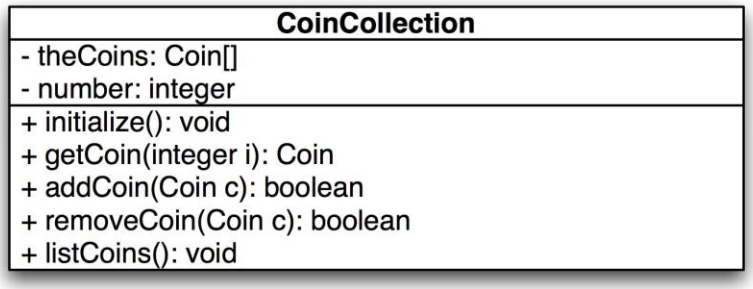


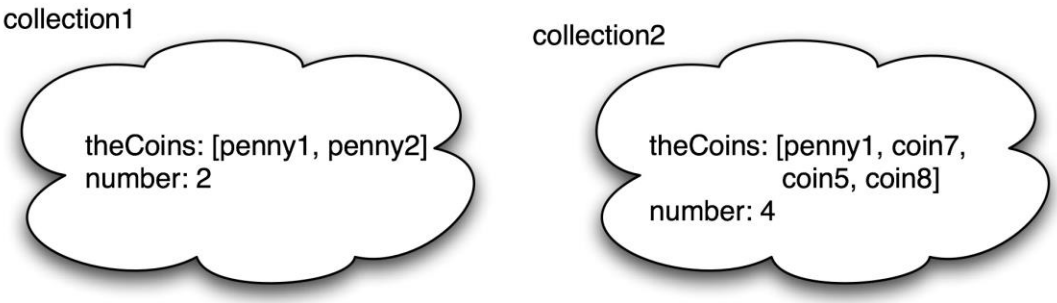


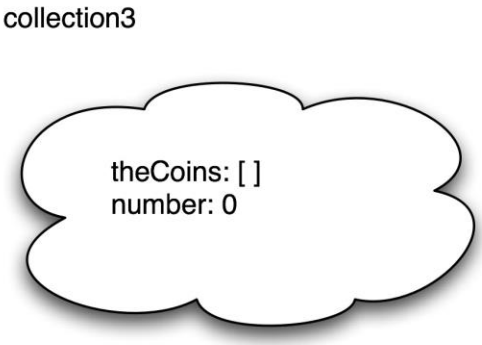


4. Repeat Exercise 1 for a collection of coins instead of a credit card.

Solution:







5. Consider a Java class that you could use to get an acceptable integer value from the user. An object of this class will have the attributes

- Minimum accepted value
- Maximum accepted value
- Prompt string

and the following method:

- getValue displays the prompt and reads a value using the class Scanner. If the value read is not within the allowed range, the method should display an error message and ask the user for a new value, repeating these actions until an acceptable value is entered. The method then returns the value read.

a. Write preconditions and postconditions for the method getValue.
b. Implement the class in Java.
c. Write some Java statements that test the class.

Solution:

Preconditions: minimum value is less than or equal to maximum value
Postconditions: the value returned will be greater than or equal to the minimum and less than or equal to the maximum.

See the code in GetInput.java.

6. Consider a class that keeps track of the sales of an item. An object of this class will have the attributes

- Number sold
- Total sales
- Total discounts
- Cost per item
- Bulk quantity
- Bulk discount percentage

and the following methods:

- registerSale(n) records the sale of *n* items. If *n* is larger than the bulk quantity, the cost per item will be reduced by the bulk discount.
- displaySales displays the number sold, the total sales, and total discount.

a. Implement the class in Java.
b. Write some Java statements that test the class.

Solution:

See the code in ItemSales.java.

7. Consider a class MotorBoat that represents motorboats. A motorboat has attributes for

- The capacity of the fuel tank
- The amount of fuel in the tank
- The maximum speed of the boat
- The current speed of the boat
- The efficiency of the boat's motor
- The distance traveled

The class has methods to

- Change the speed of the boat
- Operate the boat for an amount of time at the current speed
- Refuel the boat with some amount of fuel
- Return the amount of fuel in the tank
- Return the distance traveled so far

If the boat has efficiency *e*, the amount of fuel used when traveling at a speed *s* for time *t* is . The distance traveled in that time is .

a. Write a method heading for each method.
b. Write preconditions and postconditions for each method.
c. Write some Java statements that test the class.
d. Implement the class.

Solution:

a)

```
public void changeSpeed(double newSpeed)
public void operateForTime(double time)
public void refuelBoat(double amount)
public double fuelRemaining()
public double distance()
```

b)

public void changeSpeed(double newSpeed)
Precondition: newSpeed is positive.
Postcondition: The speed of the motor boat has been set to the minimum of newSpeed and the maximum speed.

public void operateForTime(double time)
Precondition: time is positive.
Postcondition: The motor boat operates for the given amount of time or until it runs out of fuel. The fuel and distance traveled will be updated appropriately.

public void refuelBoat(double amount)
Precondition: amount is positive.

Postcondition: The fuel in the motor boat will be set to the minimum of maximum fuel capacity and current fuel plus the given amount.

public double fuelRemaining()
Precondition: none.
Postcondition: The amount fuel in the boat was returned.

public double distance()
Precondition: none.
Postcondition: The distance traveled in boat was returned.

c&d)
See the code in MotorBoat.java.

8. Consider a class PersonAddress that represents an entry in an address book. Its attributes are

- The first name of the person
- The last name of the person
- The e-mail address of the person
- The telephone number of the person

It will have methods to

- Access each attribute
- Change the e-mail address
- Change the telephone number
- Test whether two instances are equal based solely on name

a. Write a method heading for each method.
b. Write preconditions and postconditions for each method.
c. Write some Java statements that test the class.
d. Implement the class.

Solution:

a)

```
public String getFirstName()
public String getLastName()
public String getEmailAddress()
public String getPhoneNumber()
public void updateEmail(String newEmail)
public void updatePhone(String newPhone)
public boolean equal(PersonAddress otherPerson)
```

b)

public String getFirstName()
Precondition: none.
Postcondition: The first name was returned.

public String getLastName()
Precondition: none.
Postcondition: The last name was returned.

public String getEmailAddress()
Precondition: none.
Postcondition: The email address was returned.

public String getPhoneNumber()
Precondition: none.

Postcondition: The phone number was returned.

public void updateEmail(String newEmail)
Precondition: none.
Postcondition: The email address was changed to newEmail.

public void updatePhone(String newPhone)
Precondition: none.
Postcondition: The phone number was changed to newPhone.

public boolean equal(PersonAddress otherPerson)
Precondition: otherPerson is not null.
Postcondition: True was returned if the first and last names match.

c&d)
See the code in PersonAddress.java.

9. Consider a class RatingScore that represents a numeric rating for something such as a movie. Its attributes are

- A description of what is being rated
- The maximum possible rating
- The rating

It will have methods to

- Get the rating from a user
- Return the maximum rating possible
- Return the rating
- Return a string showing the rating in a format suitable for display

a. Write a method heading for each method.
b. Write preconditions and postconditions for each method.
c. Write some Java statements that test the class.
d. Implement the class.

Solution:

a)

```
public void inputRating()
public int getMaxRating()
public int getRating()
public String getRatingString()
```

b)

```
public void inputRating()
```

Precondition: maximum rating is positive.
Postcondition: A value for the rating was obtained from the user. The rating was greater than or equal to zero and less than or equal to the maximum possible rating. The rating attribute was changed to the value obtained from the user.

```
public int getMaxRating()
```

Precondition: none.
Postcondition: The maximum rating was returned.

```
public int getRating()
```

Precondition: none.
Postcondition: The rating was returned.

```
public String getRatingString()
```

Precondition: none.
Postcondition: The rating was returned in some nicely formatted string.

c&d)

See the code in RatingScore.java.

10. Consider a class ScienceFairProjectRating that will be used to help judge a
science fair project. It will use the class RatingScore described in the previous exercise. The attributes for the new class are

- The name of the project
- A unique identification string for the project
- The name of the person
- A rating for the creative ability (max. 30)
- A rating for the scientific thought (max. 30)
- A rating for thoroughness (max. 15)
- A rating for technical skills (max. 15)
- A rating for clarity (max. 10)

It will have methods to

- Get the number of judges
- Get all the ratings for a particular project
- Return the total of the ratings for a particular project
- Return the maximum total rating possible
- Return a string showing a project's rating in a format suitable for display

a. Write a method heading for each method.
b. Write preconditions and postconditions for each method.
c. Write some Java statements that test the class.
d. Implement the class.

Solution:

a)

```
public void rateProject()
public int totalRating()
public int maxRating()
public String getRatingString()
```

b)

```
public void rateProject()
```

Precondition: none.
Postcondition: Ratings were obtained from the user for each category and then set.

```
public int totalRating()
```

Precondition: none.
Postcondition: The total of the ratings in each category was returned.

`public int maxRating()`
Precondition: none.
Postcondition: The maximum possible total rating over all categories was returned.

`public String getRatingString()`
Precondition: none.
Postcondition: A string in a nice format that shows the rating of the project.

c&d)
See the code in ScienceFairProjectRating.java.

Practice Programs:

Documentation and *javadoc*

You may want to have students start using **javadoc** with these problems. Be sure to run it on the solution files, e.g.

javadoc YearsToOvertake.java

to see what files are produced (there are many) and how to interpret them. Even if you do not make **javadoc** part of the assignment, it is still a good idea to run it on this file and look at what it produces, just in case a student tries it and asks questions.

1. Write a program to answer questions like the following: Suppose the species Klingon ox has a population of 100 and a growth rate of 15 percent, and the species elephant has a population of 10 and a growth rate of 35 percent. How many years will it take for the elephant population to exceed the Klingon ox population? Use the class Species in Listing 5.17. Your program will ask for the data on both species and will respond by telling you how many years it will take for the species that starts with the lower population to outnumber the species that starts with the higher population. The two species may be entered in any order. It is possible that the species with the smaller population will never outnumber the other species. In this case, your program should display a suitable message stating this fact.

Notes:

Practice Program 1 is sufficiently complex that it is a good example to show three aspects of program development, design, step-wise refinement, and testing. This manual has four programs that demonstrate incremental development of this Project.

1.1 `YearsToOvertakePhase1` uses comments similar to pseudocode to outline a solution (a simple but useful form of designing a solution before implementing it) and adds just the code to read in two species, determines which has the lower initial population, and display the results. Appropriate test cases would include small values for the initial populations that would make the first entry the lower, another set of values to make the second entry lower, and a set of values to make the initial populations equal. When the two initial populations are equal, the programmer may be surprised that the second one entered will be assigned to `lower`. This last test case should make the programmer aware that this special case needs to be considered *explicitly*. A question worth asking is "Does it make a difference which species is assigned to `lower` if they have equal populations?" Sometimes it will not matter (as in this case), but other times it will. What should occur to the programmer is that even with the same initial populations the two species could have different growth rates, and some decision should be made about which one to assign to

`lower`. Perhaps it makes sense to assign the first one entered to `lower` if the populations are equal, in which case the condition in the first `if` should be changed to "less than or equal to" from just "less than." Although not necessary, this was done in `YearsToOvertakePhase2`.

1.2 `YearsToOvertakePhase2` explicitly assigns the first species entered to `lower` if the two populations are equal (as described above, this is not necessary but demonstrates the idea of explicitly coding for all possibilities). More importantly, this version adds a loop to calculate (and display) the new populations for both species for each year up to 10 years. This version allows the programmer to experiment with population and growth values to develop test cases for the final version. As described in this version's program description, look for values that will result in

- `lower` overtaking `higher` in 1 year,
- `lower` overtaking `higher` in less than 10 years,
- `lower` overtaking `higher` in exactly 10 years, and
- `lower` not overtaking `higher`**,** even after 10 years, but with equal populations.

Some good test cases are shown in the following tables.

Test Case 1	**1st Species' population < 2nd 2nd overtakes 1st in 1 year**	**Predicted results:**
1st Species: Name	foo	1. **foo** is assigned to **higher** 2. **crepek** is assigned to **lower** 3. **crepek** overtakes **foo** after 1 year.
population	2	
growth rate	0	
2nd Species Name	crepek	
population	1	
growth rate	200	

Test Case 2	**1st Species' population < 2nd 2nd overtakes 1st in 7 years**	**Predicted results:**
1st Species: Name	foo	1. **foo** is assigned to **lower** 2. **crepek** is assigned to **higher** 3. **crepek** overtakes **foo** after 7 years.
population	2	
growth rate	0	
2nd Species Name	crepek	
population	1	
growth rate	20	

Test Case 3	**1st Species' population < 2nd 2nd does *not* overtake 1st in 10 yr. (populations are equal)**	**Predicted results:**
1st Species: Name	foo	1. **foo** is assigned to **lower** 2. **crepek** is assigned to **higher** 3. **crepek** does not overtake **foo** after 10 years - they both have final populations of 2.
population	2	
growth rate	0	
2nd Species Name	crepek	
population	1	
growth rate	10	

Test Case 4	**1st Species' population < 2nd 2nd overtakes 1st in exactly 10 yr.**	**Predicted results:**
1st Species: Name	foo	1. **foo** is assigned to **lower** 2. **crepek** is assigned to **higher** 3. **crepek** overtakes **foo** in exactly 10 years.
population	3	
growth rate	5	
2nd Species Name	crepek	
population	2	
growth rate	10	

1.3 `YearsToOvertakePhase3` adds the test condition of the while-loop so that it exits if either the species with the lower initial population overtakes the other species or if the 10-year limit is reached. One run is enough to make the programmer aware, if she/he was not already, that the exit value of `years` will be incremented one time too many, and needs to be adjusted outside the loop by subtracting one. The aware programmer will also note the special condition that if the loop goes to the full ten years, the while-loop will be exited, but you do not know if `lower` overtook `higher`, therefore an additional test must be made.

1.4 The final version, `YearsToOvertake` cleans everything up, removes (or comments out) test lines, and should be tested at least with the test cases listed earlier. Here is another good test case:

Test Case 5	**1st Species' population < 2nd non-zero growth rates, 1st overtakes 2nd in under 10 years**	**Predicted results:**
1st Species: Name	foo	1. **foo** is assigned to **lower** 2. **crepek** is assigned to **higher** 3. **crepek** overtakes **foo** in 9 years.
population	2	
growth rate	10	
2nd Species Name	crepek	
population	1	
growth rate	20	

References:

Listing 5.19

Solution:

The final solution to this code is in YearsToOvertake.java.

Earlier iterations are in YearsToOvertakePhase1.java, YearsToOvertakePhase2.java, and YearsToOvertakePhase3.java,.

2. Define a class called Counter. An object of this class is used to count things, so it records a count that is a nonnegative whole number. Include methods to set the counter to 0, to increase the count by 1, and to decrease the count by 1. Be sure that no method allows the value of the counter to become negative. Also include an accessor method that returns the current count value, as well as a method that displays the count on the screen. Do not define an input method. The only method that can set the counter is the one that sets it to zero. Write a program to test your class definition. (*Hint:* You need only one instance variable.)

Notes:

This project requires a test program in addition to the counter class; both are straightforward

Solution:

See the code in Counter.java and CounterTest.java.

Programming Projects:

1. Write a grading program for an instructor whose course has the following policies:
• Two quizzes, each graded on the basis of 10 points, are given.
• One midterm exam and one final exam, each graded on the basis of 100 points, are given.
• The final exam counts for 50 percent of the grade, the midterm counts for 25 percent, and the two quizzes together count for a total of 25 percent. (Do not forget to normalize the quiz scores. They should be converted to percentages before they are averaged in.)

Any grade of 90 percent or more is an A, any grade between 80 and 89 percent is a B, any grade between 70 and 79 percent is a C, any grade between 60 and 69 percent is a D, and any grade below 60 percent is an F. The program should read in the student's scores and display the student's record, which consists of two quiz scores, two exam scores, the student's total score for the entire course, and the final letter grade. The total score is a number in the range 0 to 100, which represents the weighted average of the student's work.

Define and use a class for the student record. The class should have instance variables for the quizzes, midterm, final, total score for the course, and final letter grade. The class should have input and output methods. The input method should not ask for the final numeric grade, nor should it ask for the final letter grade. The class should have methods to compute the overall numeric grade and the final letter grade. These last two methods will be void methods that set the appropriate instance variables. Remember, one method can call another method. If you prefer, you can define a single method that sets both the overall numeric score and the final letter grade, but if you do this, use a helping method. Your program should use all the methods described here. Your class should have a reasonable set of accessor and mutator methods, whether or not your program uses them.

You may add other methods if you wish.

Notes:

This project has many simple things that need to be coded, so it is another good

opportunity to demonstrate incremental development.
Solution: *See the code in StudentGrade.java and StudentGradeTest.java.*

2. Add methods to the Person class from Self-Test Question 16 to perform the following tasks:
• Set the name attribute of a Person object.
• Set the age attribute of a Person object.
• Test whether two Person objects are equal (have the same name and age).
• Test whether two Person objects have the same name.
• Test whether two Person objects are the same age.
• Test whether one Person object is older than another.
• Test whether one Person object is younger than another.
Write a driver (test) program that demonstrates each method, with at least one true and one false case for each of the methods tested.

Notes: This project requires seven new methods to be added to `Person` class, but most of them are very simple to code. The test program, however, requires significant work to create test cases that cover the various decision paths of the class's methods. Note that the `equalsIgnoreCase` method of the `String` class is used to check for matching names.
References: Self-Test Question 16
Solution: *See the code in PersonImproved.java and PersonImprovedTest.java.*

3. Create a class that represents a grade distribution for a given course. Write methods to perform the following tasks:
• Set the number of each of the letter grades A, B, C, D, and F.
• Read the number of each of the letter grades A, B, C, D, and F.
• Return the total number of grades.
• Return the percentage of each letter grade as a whole number between 0 and 100, inclusive.
• Draw a bar graph of the grade distribution.

The graph will have five bars, one per grade. Each bar can be a horizontal row of asterisks, such that the number of asterisks in a row is proportionate to the percentage of grades in each category. Let one asterisk represent 2 percent, so 50 asterisks correspond to 100 percent. Mark the horizontal axis at 10 percent increments from 0 to 100 percent, and label each line with its letter grade.

For example, if the grades are 1 A, 4 Bs, 6 Cs, 2 Ds, and 1 F, the total number of grades is 14, the percentage of As is 7, the percentage of Bs is 29, the percentage of Cs is 43, the percentage of Ds is 14, and the percentage of Fs is 7. The A row would contain 4 asterisks (7 percent of 50 rounded to the nearest integer), the B row 14, the C row 21, the D row 7, and the F row 4.

Notes:

This project requires several new methods, but most are simple, so it is just a matter of adding and testing a piece at a time until it is complete. The code to draw the graph is based on the project `TriangleOfAsterisks` from Chapter 4.

Solution:

See the code in GradesGraph.java and GradesGraphTest.java.

4. Write a program that uses the Purchase class in Listing 5.13 to set the following prices:
Oranges: 10 for $2.99
Eggs: 12 for $1.69
Apples: 3 for $1.00
Watermelons: $4.39 each
Bagels: 6 for $3.50

Then calculate the cost of each of the following five items and the total bill:
2 dozen oranges
3 dozen eggs
20 apples
2 watermelons
1 dozen bagels

Notes: This project does not specify how to input or display the information, so there are several ways it might be done. The solution in this manual "hard codes" the input data using set methods rather than entering it interactively via `readInput()`. In addition to the total cost of the items, the program outputs the subtotal cost for each item, along with its name and cost information.
References: Listing 5.13
Solution: *See the code in Purchase.java and GroceryBill.java.*

5. Write a program to answer questions like the following: Suppose the species Klingon ox has a population of 100 and a growth rate of 15 percent, and it lives in an area of 1500 square miles. How long would it take for the population density to exceed 1 per square mile? Use the class Species in Listing 5.19 with the addition of the getDensity method from Self-Test Question 10.

Notes:

This project requires a new version of `Species` that includes the `density` method from **Self-Test Question 10**. The main program uses this new version of the class and does the following: asks the user to enter data for the species (name, population and growth rate), the area (in square miles), and the target density. Note that the solution in this manual is generalized so that the user can enter any density; it is not hard-coded for 1 per square mile. Also note that a check is made to see if the density is already at or above the target before it enters the loop to calculate the number of years.

References:

Listing 5.19, Self-Test Question 10

Solution:

See the code in SpeciesWithDensity.java and YearsToDensity.

6. Consider a class that could be used to play a game of hangman. The class has the following attributes:

- The secret word.
- The disguised word, in which each unknown letter in the secret word is replaced with a question mark (?). For example, if the secret word is *abracadabra* and the letters *a*, *b*, and *e* have been guessed, the disguised word would be *ab?a?a?ab?a*.
- The number of guesses made.
- The number of incorrect guesses.

It will have the following methods:

- makeGuess(c) guesses that character c is in the word.
- getDisguisedWord returns a string containing correctly guessed letters in
  their correct positions and unknown letters replaced with *?*.
- getSecretWord returns the secret word.
- getGuessCount returns the number of guesses made.
- isFound returns true if the hidden word has been discovered.

a. Write a method heading for each method.
b. Write preconditions and postconditions for each method.
c. Write some Java statements that test the class.
d. Implement the class.
e. List any additional methods and attributes needed in the implementation that were not listed in the original design. List any other changes made to the original design.
f. Write a program that implements the game of hangman using the class you wrote for Part *d*.

Notes:

This project considers the development of a hangman game. First, we focus on domain methods for the class that encapsulates the operations required by such a game. This class is tested and then is completed by adding in interface methods. To make the code simpler, the class converts all input to lower case. The code takes a particularly straightforward approach to creating the disguised word by just replacing every possible character by '?'. This could be accomplished by using Scanner with patterns but would require a discussion of their use that may not be appropriate for all classes.

Solution:

a)

```
public void makeGuess(Character c)
public String getDisguisedWord()
public String getSecretWord()
public int getGuessCount()
public boolean isFound()
```

b)

```
public void makeGuess(Character c)
```

Precondition: The character is one of the 26 alpha characters.
Postcondition: The number of guesses and incorrect guesses is updated, the disguised word is updated.

```
public String getDisguisedWord()
```

Precondition: none.
Postcondition: The disguised string was returned.

```
public String getSecretWord()
```

Precondition: none.
Postcondition: The secret word was returned.

```
public int getGuessCount()
```

Precondition: none.
Postcondition: The number of correct guesses was returned.

```
public boolean isFound()
```

Precondition: none.
Postcondition: True was returned if the disguised word has become the secret word.

d) We will add an attribute that will be the secret word with letters that have been guessed replaced with #.

We will add the methods

```
public void initialize(String word) {
```

Precondition: none.
Postcondition: All of the attributes were initialized. The secret word was initialized to the argument. The letters remaining attribute was set to the secret word. The disguised word was set to the secret word with all the letters replaced by question mark. The number of guesses made was set to zero. The number of incorrect guesses was set to zero.

```
public String createDisguisedWord(String word)
```
Precondition: none.
Postcondition: Returned a string of the same length as the argument, but with all the alpha characters replaced by ?.

```
public void playGame()
```
Precondition: The word has not already been guessed.
Postcondition: Letters were obtained one at a time until the secret word was guessed. The number of guesses and incorrect guesses where displayed.

See the code in Hangman.java.

8. Consider a class ConcertPromoter that records the tickets sold for a performance. Before the day of the concert, tickets are sold only over the phone. Sales on the day of the performance are made only in person at the concert venue. The class has the following attributes:

- The name of the band
- The capacity of the venue
- The number of tickets sold
- The price of a ticket sold by phone
- The price of a ticket sold at the concert venue
- The total sales amount

It has methods to

- Record the sale of one or more tickets
- Change from phone sales to sales at the concert venue
- Return the number of tickets sold
- Return the number of tickets remaining
- Return the total sales for the concert

a. Write a method heading for each method.
b. Write preconditions and postconditions for each method.
c. Write some Java statements that test the class.
d. Implement the class.
e. List any additional methods and attributes needed in the implementation that were not listed in the original design. List any other changes made to the original design.
f. Write a program using the class you wrote for Part *d* that will be used to record sales for a concert. Your program should record phone sales, then sales at the venue. As tickets are sold, the number of seats remaining should be displayed. At the end of the program, display the number of tickets sold and the total sales amount for the concert.

Notes:

The motivation for this project is to introduce a simplified example of the kind of specialized programs students may encounter. We use a class to encapsulate the domain knowledge. The main method contains a simple text based interface that uses the class.

Solution:

a)

```
public void sellTickets(int number)
public void phoneSalesOver()
public int getTicketsSold()
public int getTicketsLeft()
public double getTotalSales()
```

b)

```
public void sellTickets(int number)
```

Precondition: Then number of tickets requested is positive and less than the number of tickets unsold.
Postcondition: The number of tickets sold and total sales were updated.

```
public void phoneSalesOver()
```

Precondition: none.
Postcondition: Ticket sales can now be made only at the venue.

```
public int getTicketsSold()
```

Precondition: none.
Postcondition: The number of tickets sold was returned.

```
public int getTicketsLeft()
```

Precondition: none.
Postcondition: The number of unsold tickets was returned.

```
public double getTotalSales()
```

Precondition: none.
Postcondition: The total amount of sales was returned.

d) We need to add an attribute that will indicate whether ticket sales are over the phone or at the venue.

We will add the methods

```
public void initialize(String band, int max, double
                costOverPhone, double costAtVenue)
```

Precondition: none.
Postcondition: All the attributes were initialized. The name of the band was set to band. The capacity of the venue was set to max. The number of tickets sold was set to zero. The cost of the tickets when ordered by phone and at the venue were set to the arguments. The sales total was set to zero. The attribute indicating where tickets are being sold was set to indicate sales over the phone.

`public boolean phoneSalesOnly()`
Precondition: none.
Postcondition: Returned true if we are only making sales over the phone, otherwise false.

`public String getSalesReport()`
Precondition: none.
Postcondition: Returned a string with the number of tickets sold and their amount.

`public void doTicketSale()`
Precondition: none.
Postcondition: Obtains a number of tickets to sell. Makes the sale if possible and reports the cost of the tickets.

`public double getSaleCost(int number)`
Precondition: The number of tickets requested is positive and less than the number of tickets unsold.
Postcondition: Returns the cost of the sale of that number of tickets.

We will change sellTickets slightly.
`public boolean sellTickets(int number)`
Precondition: The number of tickets requested is positive and less than the number of tickets unsold.
Postcondition: The number of tickets sold and total sales were updated. True is returned if the sale was finished, false otherwise.

See the code in ConcertPromoter.java.

9. Rewrite the Dog class given in Listing 5.1 by utilizing the information and encapsulation principles described in section 5.2. The new version should include accessor and mutator methods. Also define an equals method for the class that returns true if the dog's name, age, and breed match the same variables for the other object that is being compared. Include a main method to test the functionality of the new Dog class.

Notes:

This project involves a fairly straightforward implementations of encapsulation.

Solution:

See the code in Dog.java and DogDemo.java.

10. Consider a class Movie that contains information about a movie. The class has the following attributes:

- The movie name
- The MPAA rating (e.g. G, PG, PG-13, R)
- The number of people that have rated this movie as a 1 (Terrible)
- The number of people that have rated this movie as a 2 (Bad)
- The number of people that have rated this movie as a 3 (OK)
- The number of people that have rated this movie as a 4 (Good)
- The number of people that have rated this movie as a 5 (Great)

Implement the class with accessors and mutators for the movie name and MPAA rating. Write a method addRating that takes an integer as an input parameter. The method should verify that the parameter is a number between 1 and 5, and if so, increment the number of people rating the movie that matches the input parameter. For example, if 3 is the input parameter, then the number of people that rated the movie as a 3 should be incremented by one. Write another method, getAverage, that returns the average value for all of the movie ratings.

Test the class by writing a main method that creates at least two movie objects, adds at least 5 ratings for each movie, and outputs the movie name, MPAA rating, and average rating for each movie object.

Notes:

This is much more scalable using arrays and this problem is re-visited in Chapter 7.

Solution:

See the code in Movie.java.

11. Repeat Programming Project 18 from Chapter 4, but use a method that displays a circular disk as a subtask.

Notes:

This application draws a bullseye. If the circle-drawing method (suggested in the hint) takes a GraphicsContext object, center x- and y-coordinates, and a Color object as parameters, then both the paint method and the circle-drawing method are quite simple and easy to write.

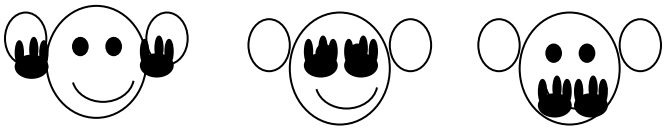
References:

Project 4.18

Solution:

See the code in Bullseye.java.

12. Create a JavaFX application that displays something like the following picture. You should have methods for drawing a monkey face and a hand.



Hear no evil | See no evil | Speak no evil

Notes:

This application draws monkey faces. Getting the proportions and constants correct can be challenging. It is strongly recommended that the methods be developed one at a time and then combined for the final picture.

Solution: *See the code in MonkeyFaces.java.*

Exercises:

1. Create a class that will bundle together several static methods for tax computations. This class should not have a constructor. Its attributes are
• basicRate—the basic tax rate as a static double variable that starts at 4 percent
• luxuryRate—the luxury tax rate as a static double variable that starts at 10 percent
Its methods are
• computeCostBasic(price)—a static method that returns the given price plus the basic tax, rounded to the nearest penny.
• computeCostLuxury(price)—a static method that returns the given price plus the luxury tax, rounded to the nearest penny.
• changeBasicRateTo(newRate)—a static method that changes the basic tax rate.
• changeLuxuryRateTo(newRate)—a static method that changes the luxury tax rate.
• roundToNearestPenny(price)—a private static method that returns the given price rounded to the nearest penny. For example, if the price is 12.567, the method will return 12.57.

Solution:

See the code in TaxComputer.java.

2. Consider a class Time that represents a time of day. It has attributes for the hour and minute. The hour value ranges from 0 to 23, where the range 0 to 11 represents a time before noon. The minute value ranges from 0 to 59.
a. Write a default constructor that initializes the time to 0 hours, 0 minutes.
b. Write a private method isValid(hour, minute) that returns true if the given hour and minute values are in the appropriate range.
c. Write a method setTime(hour, minute) that sets the time if the given values are valid.
d. Write another method setTime(hour, minute, isAM) that sets the time if the given values are valid. The given hour should be in the range 1 to 12. The parameter isAm is true if the time is an a. m. time and false otherwise.

Solution:

See the code in Time.java.

3. Write a default constructor and a second constructor for the class RatingScore, as described in Exercise 9 of the previous chapter.

Solution:

See the code in RatingScore.java.

4. Write a constructor for the class ScienceFairProjectRating, as described in Exercise 10 of the previous chapter. Give this constructor three parameters corresponding to the first three attributes that the exercise describes. The constructor should give default values to the other attributes.

Solution:

See the code in ScienceFairProjectRating.java.

5. Consider a class Characteristic that will be used in an online dating service to assess how compatible two people are. Its attributes are
• description—a string that identifies the characteristic
• rating—an integer between 1 and 10 that indicates a person's desire for this characteristic in another person
a. Write a constructor that sets the description of the characteristic to a given string and sets the rating to zero to indicate that it has not yet been determined.
b. Write a private method isValid(aRating) that returns true if the given rating is valid, that is, is between 1 and 10.
c. Write a method setRating(aRating) that sets the rating to aRating if it is valid.
d. Write a method setRating that reads a rating from the keyboard, insisting that the rating supplied by the user be valid.

Solution:

See the code in Characteristic.java.

6. Create a class RoomOccupancy that can be used to record the number of people in the rooms of a building. The class has the attributes
• numberInRoom—the number of people in a room
• totalNumber—the total number of people in all rooms as a static variable
The class has the following methods:
• addOneToRoom—adds a person to the room and increases the value of totalNumber
• removeOneFromRoom—removes a person from the room, ensuring that numberInRoom does not go below zero, and decreases the value of totalNumber as needed
• getNumber—returns the number of people in the room
• getTotal—a static method that returns the total number of people

Solution:

See the code in RoomOccupancy.java.

7. Write a program that tests the class RoomOccupancy described in the previous exercise.

Solution:

See the code in RoomOccupancy.java.

8. Sometimes we would like a class that has just a single unique instance. Create a class Merlin that has one attribute, theWizard, which is static and of type Merlin. The class has only one constructor and two methods, as follows:

• Merlin—a private constructor. Only this class can invoke this constructor; no other class or program can create an instance of Merlin.

• summon—a static method that returns theWizard if it is not null; if theWizard is null, this method creates an instance of Merlin using the private constructor and assigns it to theWizard before returning it.

• consult—a nonstatic method that returns the string "Pull the sword from the stone".

Solution:

See the code in Merlin.java.

9. Create a program that tests the class Merlin described in the previous exercise. Use the toString method to verify that a unique instance has been created.

Solution:

See the code in MerlinTester.java.

10. In the previous chapter, Self-Test Question 16 described a class Person to represent a person. The class has instance variables for a person's name, which is a string, and an integer age. These variables are name and age, respectively.
a. Write a default constructor for Person that sets name to the string "No name yet" and age to zero.
b. Write a second constructor for Person that sets name to a given string and age to a given age.
c. Write a static method createAdult() for Person that returns a special instance of this class. The instance represents a generic adult and has the name "An adult" and the age 21.

Solution:

See the code in Person.java.

11. Create a class Android whose objects have unique data. The class has the following
attributes:
• tag—a static integer that begins at 1 and changes each time an instance is created
• name—a string that is unique for each instance of this class
Android has the following methods:
• Android—a default constructor that sets the name to "Bob" concatenated with the value of tag. After setting the name, this constructor changes the value of tag by calling the private method changeTag.
• getName—returns the name portion of the invoking object.
• isPrime(n)—a private static method that returns true if n is prime—that is, if it is not divisible by any number from 2 to n - 1.
• changeTag—a private static method that replaces tag with the next prime number larger than the current value of tag.

Solution:

See the code in Android.java.

12. Create a program that tests the class Android described in the previous exercise.

Solution:

See the code in Android.java.

Practice Programs:

1. Modify the definition of the class Species in Listing 5.19 of Chapter 5 by removing the method setSpecies and adding the following methods:
• Five constructors: one for each instance variable, one with three parameters for the three instance variables, and a default constructor. Be sure that each constructor sets all of the instance variables.
• Four methods named set that can reset values: one is the same as the method setSpecies in Listing 5.16, and the other three each reset one of the instance variables.

Then write a test program to test all the methods you have added. Finally, repeat Practice Program 1 in Chapter 5, but be sure to use some constructor other than the default constructor when you define new objects of the class Species.

Notes: This project requires the development of test cases that exercise each of the new constructors and methods at least once.
References: Listing 5.16, Listing 5.19
Solution: *See the code in SpeciesCh6.java, SpeciesCh6Driver.java and YearsToOvertakeCh6.java.*

2. Repeat Programming Project 2 in Chapter 5. This time, add the following four constructor methods: one for each instance variable, one with two parameters for the two instance variables, and a default constructor. Be sure that each constructor sets all of the instance variables. Write a driver program to test each of the methods, including each of the four constructors and at least one true and one false case for each of the test methods.

Notes: The solution to this project sets the name parameter to "No name" and age to 0 if they are not specified in the constructor. In addition, if the age argument is a negative number an error message is printed and the age is set to 0.
References: Project 5.2
Solution: *See the code in PersonCh6.java and PersonCh6Test.java.*

3. Using the class Pet from Listing 6.1, write a program to read data for five pets and display the following data: name of smallest pet, name of largest pet, name of oldest pet, name of youngest pet, average weight of the five pets, and average age of the five pets.

Notes: This project is a little more challenging if it is written to find and display *all* the names if more than one pet weighs the most or least, or is the oldest or youngest, which is how the solution in this manual was written.
References: Listing 6.1
Solution: *See the code in OutputFormat.java, PetRecord.java, and PetStatistics.java.*

Programming Projects:

1. Define a utility class for displaying values of type double. Call the class DoubleOut. Include all the methods from the class DollarFormat in Listing 6.14, all the methods from the class OutputFormat of Self-Test Question 30, and a method called scienceWrite that displays a value of type double using e notation, such as 2.13e−12. (This e notation is also called scientific notation, which explains the method name.) When displayed in e notation, the number should appear with exactly one nonzero digit before the decimal point—unless the number is exactly zero. The method scienceWrite will not advance to the next line. Also add a method called scienceWriteln that is the same as scienceWrite except that it does advance to the next line. All but the last two method definitions can simply be copied from the text (or more easily from the source code for this book that is available on the Web.). Note that you will be overloading the method names
write and writeln. Write a driver program to test your method scienceWriteln. This driver program should use a stub for the method scienceWrite. (Note that this means you can write and test scienceWriteln before you even write scienceWrite.) Then write a driver program to test the method scienceWrite. Finally, write a program that is a sort of super driver program that takes a double value as input and then displays it using the two writeln methods and the scienceWriteln method. Use the number 5 for the number of digits after the decimal point when you need to specify such a number. This super driver program should allow the user to repeat this testing with additional numbers of type double until the user is ready to end the program.

Notes:

The full solution to Project 1, `DoubleOut.java`, requires a little more thought than some of the previous projects. For example, the possibility that the floating-point number may be less than 0 must be taken into account. And, after carefully thinking about how to convert to scientific notation and looking at the code in `OutputFormat`, it should be apparent that `scienceWriteln()` can be written by making a few changes to the method `write(double number, int digitsAfterPoint)`. It is helpful to use step-wise refinement and develop one piece at a time. For example, develop a solution for numbers greater than one first, then make the modifications for values less than one, where, as it happens, a little

pitfall is encountered: The `pow` method does not work with negative exponents, so special provision must be made. One of the difficulties is deciding how to obtain just the digits to print to the right of the decimal place. With a little thought, guidance, or trial and error, students should be able to figure out that the code used in `writePositive` will work if the value in the `allWhole` equation is divided by 10^e (where *e* is the exponent of 10) if *e* is positive. If *e* is negative, the inverse operation is required, so multiply by 10^{-e} (the code is `pow(10, -e)` to make the exponent positive).

References:
Self-Test Question 30, Listing 6.14

Solution:

An intermediate version of the code is given in DoubleOutWithStub.java and DoubleOutWithStubDriver.java.

The final version of the code is given in DoubleOut.java and DoubleOutDriver.java.

2. Write a new class TruncatedDollarFormat that is the same as the class DollarFormat from Listing 6.14, except that it truncates rather than rounds to obtain two digits after the decimal point. When truncating, all digits after the first two are discarded, so 1.229 becomes 1.22, not 1.23. Repeat Programming Project 3 in Chapter 4 using this new class.

Notes:

This project may require a little trial and error to get the code right to truncate past the two digits of the cents and not lose the cents completely. Casting a `double` to an `int` will truncate, but it has to be done *after* the dollars.cents is multiplied by 100, and an explicit cast is required by the java compiler (unlike C or C++):

```
int allCents = amount * 100;
```

gives a compiler error since `amount` is type double.

```
int allCents = (int)amount * 100;
```

loses the cents part of `amount` because the cast operates on `amount` *before* multiplying by 100.

Putting parentheses around the multiplication, however, makes it do the multiplication first:

```
int allCents = (int)(amount * 100);
```

so it does not lose the cents digits.

The only other "tricky" part is using the `write()` method in `TruncatedDollars` along with `System.out.println()` and `System.out.print()` to display money values interspersed with text in sentences.

References:

Listing 6.14, Project 4.3

Solution:

See the code in TruncatedDollarFormat.java and TruncatedBankAccount.java.

3. Complete and fully test the class Time that Exercise 2 describes. Add two more constructors that are analogous to the setTime methods described in Parts *c* and *d* of Exercise 2. Also include the following methods:
• getTime24 returns a string that gives the time in 24-hour notation *hhmm*. For example, if the hour value is 7 and the minute value is 25, return "0725". If the hour value is 0 and the minute value is 5, return "0005". If the hour value is 15 and the minute value is 30, return "1530".
• getTime12 returns a string that gives the time in 12-hour notation *h:mm xx*. For example, if the hour value is 7 and the minute value is 25, return "7:25 am". If the hour value is 0 and the minute value is 5, return "12:05 am". If the hour value is 15 and the minute value is 30, return "3:30 pm".

Notes: This project is a continuation of Exercise 2. The conversion from military time (24 hour format) to civilian time (12 hour format) is not difficult, but it is tricky. Having the students construct a table with corresponding times for both formats may help. The test cases for the class are in the main method.
References: Exercise 6.2
Solution: *See the code in Time.java.*

4. Complete and fully test the class Characteristic that Exercise 5 describes. Include the following methods:

• getDescription—returns the description of this characteristic.

• getRating—returns the rating of this characteristic.

• getCompatability(Characteristic otherRating)—returns the compatibility measure of two matching characteristics, or zero if the descriptions do not match.

• getCompatibilityMeasure(Characteristic otherRating)—a private method that returns a compatibility measure as a double value using the formula $measure = 1 - \frac{(r_1 - r_2)^2}{81}$. when both ratings are nonzero; *m* is zero if either rating is zero. (Recall from Exercise 5 that the constructor sets the rating to zero, indicating that it has not yet been determined.)

• isMatch(Characteristic otherRating)—a private method that returns true if the descriptions match.

Notes: This project is a continuation of Exercise 5. It adds methods that allow one to determine a numeric score for compatibility. Test cases are in the main method.
References: Exercise 6.5
Solution: *See the code in Characteristic.java.*

6. Complete and fully test the class Person that Exercise 10 describes. Include the following additional methods:
• getName—returns the name of the person as a string.
• getAge—returns the age of the person.
• setName(first, last)—sets the name of the person, given a first and last name as strings.
• setName(name)—sets the name of the person, given the entire name as one string.
• setAge(age)—sets the age of the person.
• createToddler—a static method that returns a special instance of the class to represent a toddler. The instance has the name "A toddler" and the age 2.
• createPreschooler—a static method that returns a special instance of the class to represent a preschooler. The instance has the name "A preschooler" and the age 5.
• createAdolescent—a static method that returns a special instance of the class to represent an adolescent. The instance has the name "An adolescent" and the age 9.
• createTeenager—a static method that returns a special instance of the class to represent a teenager. The instance has the name "A teenager" and the age 15.

Notes: This project demonstrates a class that uses static methods to create special instances of the class. Test cases are in the main method
References: Exercise 6.10
Solution: *See the code in Person.java.*

7. Write a Temperature class that represents temperatures in degrees in both Celsius and Fahrenheit. Use a floating-point number for the temperature and a character for the scale: either 'C' for Celsius or 'F' for Fahrenheit. The class should have

• Four constructors: one for the number of degrees, one for the scale, one for both the degrees and the scale, and a default constructor. For each of these constructors, assume zero degrees if no value is specified and Celsius if no scale is given.

• Two accessor methods: one to return the temperature in degrees Celsius, the other to return it in degrees Fahrenheit. Use the formulas from Programming Project 5 of Chapter 3 and round to the nearest tenth of a degree.

• Three set methods: one to set the number of degrees, one to set the scale, and one to set both.

• Three comparison methods: one to test whether two temperatures are equal, one to test whether one temperature is greater than another, and one to test whether one temperature is less than another.

Write a driver program that tests all the methods. Be sure to invoke each of the constructors, to include at least one true and one false case for each comparison method, and to test at least the following three temperature pairs for equality: 0.0 degrees C and 32.0 degrees F, −40.0 degrees C and −40.0 degrees F, and 100.0 degrees C and 212.0 degrees F.

Notes:

This project's requirements include two accessor methods to read the temperature, one in degrees F and the other in degrees C. From this description it is not clear if the "two accessor methods to read the temperature…" should display or return the temperature in the specified units. Also, note the confusing terminology: methods that display values are actually "write" methods. The solution in this manual includes both types of accessor, two `write` methods to display ("read") the temperature and units, and two `get` methods that return just the temperature in either degrees C or degrees F. An added feature of the solution is that it displays or returns temperatures rounded to one decimal place. The expression `Math.round(degrees*10)/10.0` is used, where the divisor is 10.0 (rather than 10) to force the division results to be floating point instead of an integer and not lose the decimal place. Also, as described in the prologue, units is not guaranteed to be a legitimate value (c, C, f, or F). The read methods give an error message if it is not a legitimate value, but the set methods allow any character and the get methods default to a return value of the variable degrees (no conversion is performed) if `units` is anything other than one of the legitimate values.

Getting the equals method to work properly highlights the problem of comparing two floating point values. One consequence of using a fixed number of bits to encode floating point values is that they cannot always be encoded precisely. Two mathematical expressions that have the same result when done by hand may actually have slightly different values (in the last decimal place or so) when stored in memory. For example the calculation

```
double a = 51.8 /1 0;
```

is likely to give a slightly different value than

```
double a = 0.518 * 10;
```

because each number is stored as an approximate value. Because of this comparisons of floating point values in conditional expressions do not always return the expected results. A way to get around it is to decide how many decimal place accuracy we want to compare, multiply the floating point numbers by the appropriate power of 10, then round the results to get integers. This is the approach taken in the comparison methods, `equals`, `isGreaterThan`, and `isLessThan`: First the methods make sure both temperatures are in the same units (degrees C, but degrees F would be equally valid) using the `getC()` method. Since `getC()` returns a value with one decimal place, the temperatures are then multiplied by 10 and rounded using `Math.round()`, which returns an integer value (you can think of it as comparing an integer number of tenths of degrees)

References:

Practice Program 3.5

Solution:

See the code in Temperature.java and TemperatureTest.java.

8. Repeat Programming Project 7 of the previous chapter, but include constructors.

Notes: This project extends the ConcertPromoter class from the previous chapter to use constructors. Instructors may want to point out that often defining a good constructor is preferable to having an initialize method as the previous version did.
References: Project 5.7
Solution: *See the code in ConcertPromoter.java.*

9. Write and fully test a class that represents rational numbers. A rational number can be represented as the ratio of two integer values, *a* and *b*, where *b* is not zero. The class has attributes for the numerator and denominator of this ratio. The ratio should always be stored in its simplest form. That is, any common factor of *a* and *b* should be removed. For example, the rational number 40/12 should be stored as 10/3.
The class has the following constructors and methods:
• A default constructor that sets the rational number to 0/1.
• A constructor that has parameters for the numerator and denominator, and converts the resulting ratio to simplified form.
• simplify—a private method that converts the rational number to simplified form.
• getGCD(x, y)—a private static method that returns the largest common factor of the two positive integers x and y, that is, their greatest common divisor. For example, the greatest common divisor of 40 and 12 is 4.
• value—returns the rational number as a double value.
• toString—returns the rational number as a string in the form *a*/*b*.

Notes: This project demonstrates a class that uses a couple private methods to accomplish some small tasks.
Solution: *See the code in Rational.java.*

10. Write a program that will record the votes for one of two candidates by using the class VoteRecorder, which you will design and create. VoteRecorder will have static variables to keep track of the total votes for candidates and instance variables to keep track of the votes made by a single person. It will have the following attributes:

• nameCandidatePresident1—a static string that holds the name of the first candidate for president
• nameCandidatePresident2—a static string that holds the name of the second candidate for president
• nameCandidateVicePresident1—a static string that holds the name of the first candidate for vice president
• nameCandidateVicePresident2—a static string that holds the name of the second candidate for vice president
• votesCandidatePresident1—a static integer that holds the number of votes for the first candidate for president
• votesCandidatePresident2—a static integer that holds the number of votes for the second candidate for president
• votesCandidateVicePresident1—a static integer that holds the number of votes for the first candidate for vice president
• votesCandidateVicePresident2—a static integer that holds the number of votes for the second candidate for vice president
• myVoteForPresident—an integer that holds the vote of a single individual for president (0 for no choice, 1 for the first candidate, and 2 for the second candidate)
• myVoteForVicePresident—an integer that holds the vote of a single individual for vice president (0 for no choice, 1 for the first candidate, and 2 for the second candidate)

In addition to appropriate constructors, VoteRecorder has the following methods:

• setCandidatesPresident(String name1, String name2)—a static method that sets the names of the two candidates for president
• setCandidatesVicePresident(String name1, String name2)—a static method that sets the names of the two candidates for vice president
• resetVotes—a static method that resets the vote counts to zero
• getCurrentVotePresident—a static method that returns a string with the current total number of votes for both presidential candidates
• getCurrentVoteVicePresident—a static method that returns a string with the current total number of votes for both vice presidential candidates
• getAndConfirmVotes—a nonstatic method that gets an individual's votes, confirms them, and then records them.

• getAVote(String name1, String name2)—a private method that returns a vote choice for a single race from an individual (0 for no choice, 1 for the first candidate, and 2 for the second candidate)
• getVotes—a private method that returns a vote choice for president and vice president from an individual
• confirmVotes—a private method that displays a person's vote for president and vice president, asks whether the voter is happy with these choices, and returns true or false according to a yes or no response
• recordVotes—a private method that will add an individual's votes to the appropriate static variables

Create a program that will conduct an election. The candidates for president are Annie and Bob. The candidates for vice president are John and Susan. Use a loop to record the votes of many voters. Create a new VoteRecorder object for each voter. After all the voters are done, present the results.

Notes:

This project demonstrates how static variables/methods can be used to record common information for a class of objects. In this case, we can have many instances of the vote recorder class getting votes and keep a cumulative total in static variables. While the description looks complicated, many of the methods are similar in nature.

Solution:

See the code in VoteRecorder.java.

11. Repeat Programming Project 10 of the previous chapter, but include constructors.

Notes: This project involves adding constructors to the Movie class.
References: Listing 5.10
Solution: *See the code in Movie.java.*

12. Create a JavaFX application with two buttons and two labels. Add an Image Icon of your choice to the first button and the first label.

Notes: This project uses a VBox layout and the same image used in the text chapter. A different image could be used for either the button or the label.
Solution: *See the code in ButtonLabels.java*

Exercises:

1. Write a program in a class NumberAboveAverage that counts the number of days that the temperature is above average. Read ten temperatures from the keyboard and place them in an array. Compute the average temperature and then count and display the number of days on which the temperature was above average.

Solution:

See the code in NumberAboveAverage.java.

2. Write a program in a class CountFamiles that counts the number of families whose income is below a certain value. Read an integer *k* from the keyboard and then create an array of double values of size *k*. Read *k* values representing family income from the keyboard and place them into the array. Find the maximum income among these values. Then count the families that make less than 10 percent of this maximum income. Display this count and the incomes of these families.

Solution:

See the code in CountFamilies.java.

3. Write a program in a class CountPoor that counts the number of families that are considered poor. Write and use a class Family that has the attributes
• income—a double value that is the income for the family
• size—the number of people in the family and the following methods:
• Family(income, size)—a constructor that sets the attributes
• isPoor(housingCost, foodCost)—a method that returns true if housingCost + foodCost * size is greater than half the family income (foodCost is the average food cost for an individual, while housingCost is for the family)
• toString—a method that returns a string containing the information about the family The program should read an integer *k* from the keyboard and then create an array of size *k* whose base type is Family. It should then create *k* objects of type Family and put them in the array, reading the income and size for each family from the keyboard. After reading an average housing cost and average food cost from the keyboard, it should display the families that are poor.

Solution:

See the code in Family.java and CountPoor.java.

4. Write a program in a class FlowerCounter that computes the cost of flowers sold at a flower stand. Five kinds of flowers—petunia, pansy, rose, violet, and carnation— are stocked and cost, respectively, 50¢, 75¢, $1.50, 50¢, and 80¢ per flower. Create an array of strings that holds the names of these flowers. Create another array that holds the cost of each corresponding flower. Your program should read the name of a flower and the quantity desired by a customer. Locate the flower in the name array and use that index to find the cost per stem in the cost array. Compute and print the total cost of the sale.

Solution:

See the code in FlowerCounter.java.

5. Write a program in a class CharacterFrequency that counts the number of times a digit appears in a telephone number. Your program should create an array of size 10 that will hold the count for each digit from 0 to 9. Read a telephone number from the keyboard as a string. Examine each character in the phone number and increment the appropriate count in the array. Display the contents of the array.

Solution:

See the code in CharacterFrequency.java.

6. Create a class Ledger that will record the sales for a store. It will have the attributes
• sale—an array of double values that are the amounts of all sales
• salesMade—the number of sales so far
• maxSales—the maximum number of sales that can be recorded
and the following methods:
• Ledger(max)—a constructor that sets the maximum number of sales to max
• addSale(d)—adds a sale whose value is d
• getNumberOfSales—returns the number of sales made
• getTotalSales—returns the total value of the sales

Solution:

See the code in Ledger.java.

7. Define the following methods for the class Ledger, as described in the previous exercise:
• getAverageSale()—returns the average value of all the sales
• getCountAbove(v)—returns the number of sales that exceeded v in value

Solution:

See the code in Ledger.java.

8. Write a static method isStrictlyIncreasing(double[] in) that returns true if each value in the given array is greater than the value before it, or false otherwise.

Solution:

```
public static boolean isStrictlyIncreasing(double[] in) {
        boolean result = true;
        for(int i=0; i< (in.length - 1); i++){
            if(in[i+1] <= in[i])
                result = false;
        }
        return result;
}
```

9. Write a static method removeDuplicates(Character[] in) that returns a new array of the characters in the given array, but without any duplicate characters. Always keep the first copy of the character and remove subsequent ones. For example, if in contains b, d, a, b, f, a, g, a, a, and f, the method will return an array containing b, d a, f, and g. *Hint*: One way to solve this problem is to create a boolean array of the same size as the given array in and use it to keep track of which characters to keep. The values in the new boolean array will determine the size of the array to return.

Solution:

See the code in Fragments.java.

10. Write a static method remove(int v, int[] in) that will return a new array of the integers in the given array, but with the value v removed. For example, if v is 3 and in contains 0, 1, 3, 2, 3, 0, 3, and 1, the method will return an array containing 0, 1, 2, 0, and 1.

Solution:

See the code in Fragments.java.

11. Suppose that we are selling boxes of candy for a fund-raiser. We have five kinds of candy to sell: Mints, Chocolates with Nuts, Chewy Chocolates, Dark Chocolate Creams, and Sugar Free Suckers. We will record a customer's order as an array of five integers, representing the number of boxes of each kind of candy. Write a static method combineOrder that takes two orders as its arguments and returns an array that represents the combined orders. For example, if order1 contains 0, 0, 3, 4, and 7, and order2 contains 0, 4, 0, 1, and 2, the method should return an array containing 0, 4, 3, 5, and 9.

Solution:

```
public static int[] combineOrder(int[] order1, int[] order2){

        // Find the number of values that will be in the result
        int[] combinedOrder = new int[5];
        for(int i=0; i<5; i++){
            combinedOrder[i] = order1[i] + order2[i];
        }

        return combinedOrder;
    }
```

12. Create a class Polynomial that is used to evaluate a **polynomial** function of x: $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$ The **coefficients** a_i are floating-point numbers, the exponents of x are integers, and the largest exponent n—called the **degree** of the polynomial—is greater than or equal to zero. The class has the attributes

• degree—the value of the largest exponent *n*
• coefficients—an array of the coefficients *ai*

and the following methods:

• Polynomial(max)—a constructor that creates a polynomial of degree max whose coefficients are all zero
• setConstant(i, value)—sets the coefficient *ai* to value
• evaluate(x)—returns the value of the polynomial for the given value x

For example, the polynomial
$P(x) = 3 + 5\,x + 2\,x^3$ is of degree 3 and has coefficients $a0 = 3$, $a1 = 5$, $a2 = 0$, and $a3 = 2$. The invocation
evaluate(7) computes $3 + 5 \times 7 + 0 \times 7^2 + 2 \times 7^3 = 3 + 35 + 0 + 686 = 724$ and returns the result 724.

Solution:

See the code in Polynomial.java.

15. Write a static method for selection sort that will sort an array of characters.

Solution:

See the code in Fragments.java.

20. Write a static method findFigure(picture, threshold), where picture is a two-dimensional array of double values. The method should return a new twodimensional array whose elements are either 0.0 or 1.0. Each 1.0 in this new array indicates that the corresponding value in picture exceeds threshold times the average of all values in picture. Other elements in the new array are 0.0.
For example, if the values in picture are the average value is 5.55. The resulting array for a threshold of 1.4 would be and the resulting array for a threshold of 0.6 would be

Solution:

See the code in TwoDArrayMethods.java.

21. Write a static method blur(double[][] picture) that you could use on a part of a picture file to obscure a detail such as a person's face or a license plate number. This method computes the weighted averages of the values in picture and returns them in a new two-dimensional array. To find a weighted average of a group of numbers, you count some of them more than others. Thus, you multiply each item by its weight, add these products together, and divide the result by the sum of the weights.

For each element in picture, compute the weighted average of the element and its immediate neighbors. Store the result in a new two-dimensional array in the same position that the element occupies in picture. This new array is the one the method returns. The neighbors of an element in picture can be above, below, to the left of,
and to the right of it, either vertically, horizontally, or diagonally. So each weighted average in the new array will be a combination of up to nine values from the array picture. A corner value will use only four values: itself and three neighbors. An edge value will use only six values: itself and five neighbors. But an interior value will use nine values: itself and eight neighbors. So you will need to treat the corners and edges separately from the other cells.

Solution:

See the code in TwoDArrayMethods.java.

Practice Programs:

1. Write a program that reads integers, one per line, and displays their sum. Also, display all the numbers read, each with an annotation giving its percentage contribution to the sum. Use a method that takes the entire array as one argument and returns the sum of the numbers in the array. *Hint*: Ask the user for the number of integers to be entered, create an array of that length, and then fill the array with the integers read. A possible dialogue between the program and the user follows:

How many numbers will you enter?
4
Enter 4 integers, one per line:
2
1
1
2
The sum is 6.
The numbers are:
2, which is 33.3333% of the sum.
1, which is 16.6666% of the sum.
1, which is 16.6666% of the sum.
2, which is 33.3333% of the sum.

Notes: This project is very much like `ArrayOfTemperatures`, Listing 7.1. Just add code to prompt for and read in the length of the array, and, instead of finding the average, divide each element by the sum to obtain its percent.
References: Listing 7.1
Solution: *See the code in PercentOfSum.java.*

Programming Projects:

1. Write a program that will read a line of text that ends with a period, which serves as a sentinel value. Display all the letters that occur in the text, one per line and in alphabetical order, along with the number of times each letter occurs in the text. Use an array of base type int of length 26, so that the element at index 0 contains the number of *a*'s, the element at index 1 contains the number of *b*'s, and so forth. Allow both uppercase and lowercase letters as input, but treat uppercase and lowercase versions of the same letter as being equal. *Hints*: Use one of the methods toUpperCase or toLowerCase in the wrapper class Character, described in Chapter 6. You will find it helpful to define a method that takes a character as an argument and returns an int value that is the correct index for that character. For example, the argument 'a' results in 0 as the return value, the argument 'b' gives 1 as the return value, and so on. Note that you can use a type cast, such as (int)letter, to change a char to an int. Of course, this will not get the number you want, but if you subtract (int)'a', you will then get the right index. Allow the user to repeat this task until the user says she or he is through.

Notes: This project is a bit challenging to get the loop conditions right. The objective is to keep the array index within bounds and count only letters. Another little problem is how to get the printable character code from the array index after the letter counts have been determined. The "trick" is to know that adding 65 decimal to the array index will produce the ASCII code for the character.
Solution: *See the code in CountLettersInLine.java.*

2. A **palindrome** is a word or phrase that reads the same forward and backward, ignoring blanks and considering uppercase and lowercase versions of the same letter to be equal. For example, the following are palindromes:

- warts n straw
- radar
- Able was I ere I saw Elba
- xyzczyx

Write a program that will accept a sequence of characters terminated by a period and will decide whether the string—without the period—is a palindrome. You may assume that the input contains only letters and blanks and is at most 80 characters long. Include a loop that allows the user to check additional strings until she or he requests that the program end. *Hint*: Define a static method called isPalindrome that begins as follows:

```
/**
Precondition: The array a contains letters and
blanks in positions a[0] through a[used − 1].
Returns true if the string is a palindrome and
false otherwise.
*/
public static boolean isPalindrome(char[] a, int used)
```

Your program should read the input characters into an array whose base type is char and then call the preceding method. The int variable used keeps track of how much of the array is used, as described in the section entitled "Partially Filled Arrays."

Notes:

The solution to this project reads in a line of text using the `String` class, then passes the string to the `palindrome` method which performs the test, returning `true` if the phrase is a palindrome, or `false` if not. A note in the source code's prologue calls attention to its behavior in the degenerate case where the phrase has no letters (either all blanks or a null `String`). Given the requirements in the problem statement that the `palindrome` method is passed the original line of text and returns a Boolean value, it cannot return an indication that the file was empty. As the note in the prologue states, it returns `true` in these situations. The `palindrome` method uses a character array to parse the array, getting rid of white spaces and saving only the letters (actually, it saves anything other than white space characters). Then it analyzes the letters to see if it a palindrome. The parsing technique can be borrowed from Project 2, so the interesting part of this problem is figuring out the algorithm to

get the correct characters to compare. The algorithm is described in comments in the `palindrome` code. An easy way to make the palindrome test insensitive to case (it should ignore whether the characters are upper or lower case) is to make all the characters the same case, either upper or lower. The solution shown here uses the `toUpperCase` method in the `Character` wrapper class, but it would be equally correct to use `toLowerCase`.

Solution:

See the code in Palindrome.java.

3. Add a method bubbleSort to the class ArraySorter, as given in Listing 7.10, that performs a **bubble sort** of an array. The bubble sort algorithm examines all adjacent pairs of elements in the array from the beginning to the end and interchanges any two elements that are out of order. Each interchange makes the array more sorted than it was, until it is entirely sorted. The algorithm in pseudocode follows:

Bubble sort algorithm to sort an array a
Repeat the following until the array a is sorted:
for (index = 0; index < a.length − 1; index++)
    if (a[index] > a[index + 1])
        Interchange the values of a[index] and a[index + 1].

The bubble sort algorithm usually requires more time than other sorting methods.

Notes:

This project requires an extension of the bubble sort algorithm outlined in the problem description because it gives pseudo-code for just one pass through the array, which puts only the highest remaining number in its correct slot. Additional passes (repetitions of the algorithm) are necessary until the array is completely sorted. Playing around with some simple examples can reveal how bubble sort works and the criteria for ending the loop and is a good exercise for students. Best case is when the array is already sorted (nothing needs to be swapped), worst case is when it is exactly backwards (everything needs to be swapped), and other orderings are somewhere in between. An efficient algorithm will detect when the loop is sorted and stop processing it. A flag can be used to detect the situation where the array is already sorted; set the flag to true before executing the swap loop and change it to false if a swap occurs - whenever the array is processed without doing a swap it is obviously sorted. At the other extreme, the worst case ordering shows that there is an upper limit to the number of iterations after which the array is guaranteed sorted. Notice that, for a loop with n elements, only n-1 comparisons of adjacent elements are required to move the largest element into its proper place. The next iteration should process the remaining n-1 elements (after the nth element which is correctly positioned), so it will process n-2 elements and result in the last two elements being properly placed, etc. Following this scenario leads to the conclusion that n-1 passes for an n-element array guarantees the array has been sorted, and each iteration needs to process one less element (the last element in the previous iteration). Combining these two criteria gives an efficient algorithm for sorting an array of n elements: repeat the swap loop until the swap flag stays true for the iteration, up to a maximum of n-1 times. Following the approach in the text, an additional class, `BubbleSortDemo`, is used to demonstrate the bubble sort method with a sample array.

References: Listing 7.10, 7.11
Solution: *See the code in BubbleSort.java and BubbleSortDemo.java.*

4. Add a method insertionSort to the class ArraySorter, as given in Listing 7.10, that performs an **insertion sort** of an array. To simplify this project, our insertion sort algorithm will use an additional array. It copies elements from the original given array to this other array, inserting each element into its correct position in the second array. This will usually require moving a number of elements in the array receiving the new elements. The algorithm in pseudocode is as follows:

Insertion sort algorithm to sort an array a

for (index = 0; index < a.length; index++)

Insert the value of a[index] into its correct position in the array temp, so that all the elements copied into the array temp so far are sorted.
Copy all the elements from temp back to a.

The array temp is partially filled and is a local variable in the method sort.

Notes:

1. This project also requires an extension of the algorithm in the problem statement. In these descriptions the word "up" is used to mean a higher subscript in the array. (It could just as easily be called "down;" the important thing is to use the directions consistently in all descriptions.) The sorted array, `temp`, is created one element at a time, then, when all elements have been inserted correctly, `temp` needs to be copied back into the original array:

```
For each element in the original array:
{
  Get next value.
  Find its insertion point:
  {
    Compare next value to each element of temp, in order, starting at the lowest.
    The insertion point is found either when next value > element in temp,
    or the end of temp is reached.
  }
  Starting at the end of temp and working backward through the insertion point,
     move the elements in temp up one place.  (You need to start at the top and
     work backward to avoid overwriting data in temp, and the value at the
     insertion point needs to be moved so next value can be inserted.)
  Insert next value into temp at the insertion point.
}
```

Copy temp array into original array: the original array is now sorted.

Following the approach in the text, an additional class, `InsertionSortDemo`, is used to demonstrate the bubble sort method with a sample array.

References: Listing 7.10, 7.11
Solution: *See the code in InsertionSort.java and InsertionSortDemo.java.*

5. The class TimeBook in Listing 7.14 is not really finished. Complete the definition of this class in the way described in the text. In particular, be sure to add a default constructor, as well as set and get methods that change or retrieve each of the instance variables and each indexed variable of each array instance variable. Be sure you replace the stub setHours with a method that obtains values from the keyboard.

You should also define a private method having two int parameters that displays the first parameter in the number of spaces given by a second parameter. The extra spaces not filled by the first parameter are to be filled with blanks. This will let you write each array element in exactly four spaces, for example, and so will allow you to display a neat rectangular arrangement of array elements. Be sure that the main method in Listing 7.14 works correctly with these new methods. Also, write a separate test program to test all the new methods. *Hint*: To display an int value *n* in a fixed number of spaces, use Integer.toString(*n*) to convert the number to a string value, and then work with the string value. This method is discussed in Chapter 6 in the section "Wrapper Classes."

Notes: The solution to this project is based on `TimeBook.java`, Listing 7.14. A good approach to this problem is to add one feature at a time to `NewTimeBook`, then add the code to test the feature in `NewTimeBookDemo.java`. The default constructor was written to have just one employee. The features were added in the order that the tests appear in `NewTimeBookDemo.java`, with the method to align the numbers in the table done last.
References: Listing 7.14
Solution: *See the code in NewTimeBook.java and NewTimeBookDemo.java.*

6. Define a class called TicTacToe. An object of type TicTacToe is a single game of tic-tac-toe. Store the game board as a single two-dimensional array of base type char that has three rows and three columns. Include methods to add a move, to display the board, to tell whose turn it is (X or O), to tell whether there is a winner, to say who the winner is, and to reinitialize the game to the beginning. Write a main method for the class that will allow two players to enter their moves in turn at the same keyboard.

Notes:

This project is one of the most sophisticated problems in the text and is complex enough that a good, disciplined step-wise approach is particularly useful. The challenge is to break up the actions into manageable pieces and write methods for them. The goal is to create methods that are easy to use, logically written, and make `main` easy to write and read. It is helpful to write main as a sequence of method calls to do things (like clear the board for a new game, draw the board, enter an X or O on the board, check for a winner, etc.) without actually writing complete code for the methods. Instead, just write stubs with just a line that prints out its method name and, if it has a return type other than void, returns a fixed value. Then proceed to add functionality one method at a time. A good approach is to write the code to display the board first, then the code to clear it to start a new game, then the code to get an entry and insert it into the board, etc. The test for a winning move can be the last part implemented – it is easier work on that algorithm and code if everything is working.

Solution:

See the code in TicTacToe.java.

7. Repeat Programming Project 10 from Chapter 5 but use an array to store the movie ratings instead of separate variables. All changes should be internal to the class so the main method to test the class should run identically with either the old Movie class or the new Movie class using arrays.

Notes:

This project involves re-doing the Movie rating class with arrays instead of hard-coding five separate rating values. An even better version would be to re-do the Programing Project from chapter 6, which is the same Movie class but adds constructors.

References: Programming Project 5.10 (optionally, Programming Project 6.7)
Solution: *See the code in MovieArrays.java.*

8. Traditional password entry schemes are susceptible to "shoulder surfing" in which an attacker watches an unsuspecting user enter their password or PIN number and uses it later to gain access to the account. One way to combat this problem is with a randomized challenge-response system. In these systems the user enters different information every time based on a secret in response to a randomly generated challenge. Consider the following scheme in which the password consists of a five-digit PIN number (00000 to 99999). Each digit is assigned a random number that is 1, 2, or 3. The user enters the random numbers that correspond to their PIN instead of their actual PIN numbers.

For example, consider an actual PIN number of 12345. To authenticate the user would be presented with a screen such as:

```
PIN: 0 1 2 3 4 5 6 7 8 9
NUM: 3 2 3 1 1 3 2 2 1 3
```

The user would enter 23113 instead of 12345. This doesn't divulge the password even if an attacker intercepts the entry because 23113 could correspond to other PIN numbers, such as 69440 or 70439. The next time the user logs in, a different sequence of random numbers would be generated, such as:

```
PIN: 0 1 2 3 4 5 6 7 8 9
NUM: 1 1 2 3 1 2 2 3 3 3
```

Write a program to simulate the authentication process. Store an actual PIN number in your program. The program should use an array to assign random numbers to the digits from 0 to 9. Output the random digits to the screen, input the response from the user, and output whether or not the user's response correctly matches the PIN number.

Notes:

This solution inputs the PIN as a string and extracts the digits using the Unicode/ASCII representation, but a student could also input the number as an integer and extract the digits using division and modulus.

The actual PIN in the solution is 99508.

This project is somewhat difficult as it requires an understanding of arrays storing numbers that are used as an index in another array.

Solution:

See the code in Authenticate.java.

9. Write a JavaFX application that displays a picture of a pine tree formed by drawing a triangle on top of a small rectangle that makes up the visible trunk. The tree should be green and have a gray trunk.

Notes: This project is a short application program that draws a pine tree. It uses four arrays of int values. One array holds the x-coordinates of the branches, the second holds the y-coordinates of the branches, and the last two hold the x- and y-coordinates of the trunk.
Solution: *See the code in PineTree.java*

10. ELIZA was a program written in 1966 that parodied a psychotherapist session. The user typed sentences and the program used those words to compose a question. Create a simple applet based on this idea. The applet will use a label to hold the program's question, a text field into which the user can type an answer, a button for the user to signal that the answer is complete, and a quit button. The initial text for the question label should read: "What would you like to talk about?" When the user presses a button, get the text from the text field. Now extract the words from the text one at a time and find the largest word of length 4 or more. Let's call this largest word X for now. In response, create a question based on the length of the word. If the word is length 4, the new question is: "Tell me more about X." If the word is length 5, the new question is: "Why do you think X is important?" If the word is length 6 or more, the new question is: "Now we are getting somewhere. How does X affect you the most?" If there is no word of length 4, the new question is: "Maybe we should move on. Is there something else you would like to talk about?" *Hint*: You can use the class Scanner to extract the words from a string, assuming blanks separate the words. For example, the following statements

```
String text = " one potato two potato ";
Scanner parser = new Scanner(text);
System.out.println(parser.next());
System.out.println(parser.next());
```

display one and potato on separate lines.

Notes:

This programming project should be moved to Chapter 8 since event handling is not discussed at this point in the text, so processing button clicks hasn't been covered.

This project looks at a tiny piece of computing history and can be used to introduce questions of artificial intelligence. The actual implementation of our application is pretty simple. Extending this applet to do a more complicated parsing of the input would be interesting. It would probably be a good idea as the action becomes more complicated move the parsing code out of `setOnAction` into its own dedicated method or methods.

Solution:

See the code in Eliza.java.

11. Sudoku is a popular logic puzzle that uses a 9 by 9 array of squares that are organized into 3 by 3 subarrays. The puzzle solver must fill in the squares with the digits 1 to 9 such that no digit is repeated in any row, any column, or any of the nine 3 by 3 subgroups of squares. Initially, some squares are filled in already and cannot be changed. For example, the following might be a starting configuration for a sudoku puzzle:

Create a class SudokuPuzzle that has the attributes
• board—a 9 by 9 array of integers that represents the current state of the puzzle, where zero indicates a blank square
• start—a 9 by 9 array of boolean values that indicates which squares in board are given values that cannot be changed and the following methods:
• SudokuPuzzle—a constructor that creates an empty puzzle
• toString—returns a string representation of the puzzle that can be printed
• addInitial(row, col, value)—sets the given square to the given value as an initial value that cannot be changed by the puzzle solver
• addGuess(row, col, value)—sets the given square to the given value; the value can be changed later by another call to addGuess
• checkPuzzle—returns true if the values in the puzzle do not violate the restrictions
• getValueIn(row, col)—returns the value in the given square
• getAllowedValues(row, col)—returns a one-dimensional array of nine booleans, each of which corresponds to a digit and is true if the digit can be placed in the given square without violating the restrictions
• isFull—returns true if every square has a value
• reset—changes all of the nonpermanent squares back to blanks (zeros)

Write a main method in the class Sudoku that creates a SudokuPuzzle object and sets its initial configuration. Then use a loop to allow someone to play sudoku. Display the current configuration and ask for a row, column, and value. Update the game board and display it again. If the configuration does not satisfy the restrictions, let the user know. Indicate when the puzzle has been solved correctly. In that case, both checkPuzzle and isFull would return true. You should also allow options for resetting the puzzle and displaying the values that can be placed in a given square.

Notes:

This project creates a class that could be used to implement Sudoku. It uses three different patterns for accessing values in a two dimensional array. The hardest part of this class is getting the logic for check rows, columns and subsquares correct. It is strongly recommended that the checking logic be implemented in separate methods that are checked one at a time. A text based interface that allows you to play a game of Sudoku is implemented in the main method.

Solution:

See the code in SudokuPuzzle.java.

12. Create a JavaFX application that draws the following picture of a magic wand, using polygons and polylines:

Notes:

This project implements a simple application that draws a picture of a magic wand. It is designed to demonstrate the use of strokePolygon with arrays. The solution also use strokePolyline to outline the figures in black. Creating and testing it iteratively is strongly recommended.

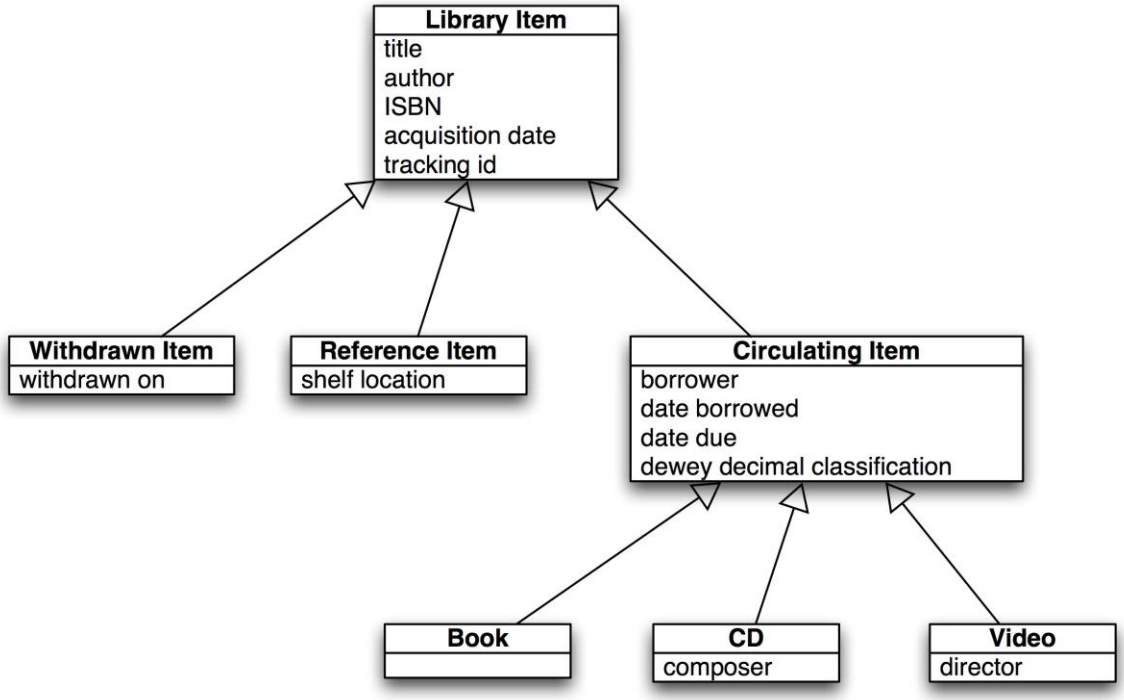
Solution:

See the code in MagicWand.java.

Exercises:

1. Consider a program that will keep track of the items in a school's library. Draw a class hierarchy, including a base class, for the different kinds of items. Be sure to also consider items that cannot be checked out.

Solution:



2. Implement your base class for the hierarchy from the previous exercise.

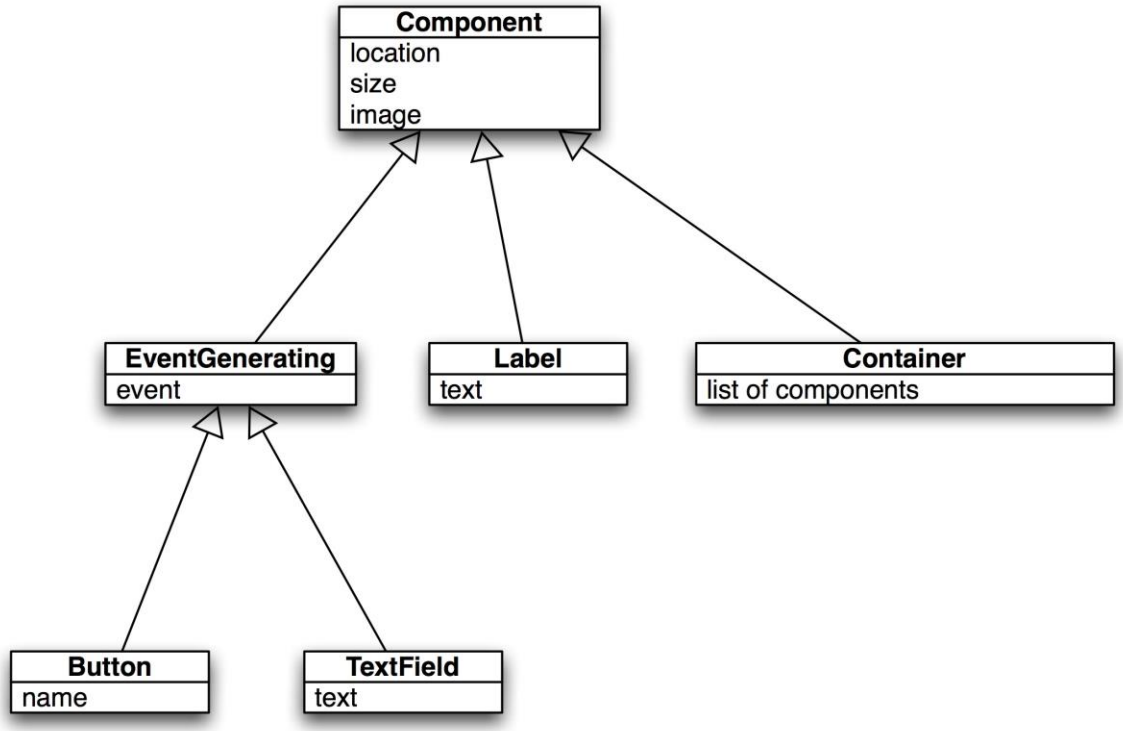
Solution:

See the code in LibraryItem.java.

3. Draw a hierarchy for the components you might find in a graphical user interface. Note that some components can trigger actions. Some components may have graphics associated with them. Some components can hold other components.

Solution:

Here is part of a such a hierarchy. There are many other types of components.



4. Suppose we want to implement a drawing program that creates various shapes using keyboard characters. Implement an abstract base class DrawableShape that knows the center (two integer values) and the color (a string) of the object. Give appropriate accessor methods for the attributes. You should also have a mutator method that moves the object by a given amount.

Solution:

See the code in DrawableShape.java.

5. Create a class Square derived from DrawableShape, as described in the previous exercise. A Square object should know the length of its sides. The class should have an accessor method and a mutator method for this length. It should also have methods for computing the area and perimeter of the square. Although characters are taller than they are wide—so the number of characters in the vertical sides will differ from the number in the horizontal sides—you need not worry about this detail when drawing the square.

Solution:

See the code in Square.java.

6. Create a class SchoolKid that is the base class for children at a school. It should have attributes for the child's name and age, the name of the child's teacher, and a greeting. It should have appropriate accessor and mutator methods for each of the attributes.

Solution:

See the code in SchoolKid.java.

7. Derive a class ExaggeratingKid from SchoolKid, as described in the previous exercise. The new class should override the accessor method for the age, reporting the actual age plus 2. It also should override the accessor for the greeting, returning the child's greeting concatenated with the words "I am the best."

Solution:

See the code in ExaggeratingKid.java.

8. Create an abstract class PayCalculator that has an attribute payRate given in dollars per hour. The class should also have a method computePay(hours) that returns the pay for a given amount of time.

Solution:

See the code in PayCalculator.java.

9. Derive a class RegularPay from PayCalculator, as described in the previous exercise. It should have a constructor that has a parameter for the pay rate. It should not override any of the methods. Then derive a class HazardPay from PayCalculator that overrides the computePay method. The new method should return the amount returned by the base class method multiplied by 1.5.

Solution:

See the code in RegularPay.java.
See the code in HazardPay.java.

10. Create an abstract class DiscountPolicy. It should have a single abstract method computeDiscount that will return the discount for the purchase of a given number of a single item. The method has two parameters, count and itemCost.

Solution:

See the code in DiscountPolicy.java.

11. Derive a class BulkDiscount from DiscountPolicy, as described in the previous exercise. It should have a constructor that has two parameters, minimum and percent. It should define the method computeDiscount so that if the quantity purchased of an item is more than minimum, the discount is percent percent.

Solution:

See the code in BulkDiscount.java.

12. Derive a class BuyNItemsGetOneFree from DiscountPolicy, as described in Exercise 10. The class should have a constructor that has a single parameter n. In addition, the class should define the method computeDiscount so that every *n*th item is free. For example, the following table gives the discount for the purchase of various counts of an item that costs $10, when n is 3:

Solution:

See the code in BuyNItemsGetOneFree.java.

13. Derive a class CombinedDiscount from DiscountPolicy, as described in Exercise 10. It should have a constructor that has two parameters of type DiscountPolicy. It should define the method computeDiscount to return the maximum value returned by computeDiscount for each of its two private discount policies. The two discount policies are described in Exercises 11 and 12.

Solution:

See the code in CombinedDiscount.java.

14. Define DiscountPolicy as an interface instead of the abstract class described in Exercise 10.

Solution:

See the code in DiscountPolicy.java.

15. Create an interface MessageEncoder that has a single abstract method encode(plainText), where plainText is the message to be encoded. The method will return the encoded message.

Solution:

See the code in MessageEncoder.java.

16. Create a class SubstitutionCipher that implements the interface MessageEncoder, as described in the previous exercise. The constructor should have one parameter called shift. Define the method encode so that each letter is shifted by the value in shift. For example, if shift is 3, *a* will be replaced by *d*, *b* will be replaced by *e*, *c* will be replaced by *f*, and so on. *Hint*: You may wish to define a private method that shifts a single character.

Solution:

See the code in SubstitutionCipher.java.

17. Create a class ShuffleCipher that implements the interface MessageEncoder, as described in Exercise 15. The constructor should have one parameter called n. Define the method encode so that the message is shuffled n times. To perform one shuffle, split the message in half and then take characters from each half alternately. For example, if the message is "abcdefghi", the halves are "abcde" and "fghi". The shuffled message is "afbgchdie". *Hint*: You may wish to define a private method that performs one shuffle.

Solution:

See the code in ShuffleCipher.java.

Practice Programs:

1. Define a class named Employee whose objects are records for employees. Derive this class from the class Person given in Listing 8.1. An employee record inherits an employee's name from the class Person. In addition, an employee record contains an annual salary represented as a single value of type double, a hire date that gives the year hired as a single value of type int, and an identification number that is a value of type String. Give your class a reasonable complement of constructors, accessor methods, and mutator methods, as well as an equals method. Write a program to fully test your class definition.

Notes: This project should be pretty straightforward. The solution shown here has nine constructors, including a default that specifies nothing. Since it did not seem reasonable to have salary, hire date or social security numbers without a name, the remaining constructors all require at least the name to be specified. Note the use of methods from the base class, `Person`. For example, `super.writeOutput()` is used to write just the name (and avoid using the local version in `Employee` that writes out all the parameters).
References: Listing 8.1
Solution: *See the code in Employee.java and EmployeeTest.java.*

2. Define a class named Doctor whose objects are records for a clinic's doctors. Derive this class from the class Person given in Listing 8.1. A Doctor record has the doctor's name—defined in the class Person—a specialty as a string (for example Pediatrician, Obstetrician, General Practitioner, and so on), and an office visit fee (use the type double). Give your class a reasonable complement of constructors and accessor methods, and an equals method as well. Write a driver program to test all your methods.

Notes:

This project is a simple variation of Practice Program 1. Note that two of the `Doctor` constructors have the same two parameter types, `String` and `double`, but in opposite order, so they are clearly distinct to the compiler.

References:

Practice Program 8.1

Solution:

See the code in Doctor.java and DoctorTest.java.

3. Create a base class called Vehicle that has the manufacturer's name (type String), number of cylinders in the engine (type int), and owner (type Person given in Listing 8.1). Then create a class called Truck that is derived from Vehicle and has additional properties: the load capacity in tons (type double, since it may contain a fractional part) and towing capacity in tons (type double).
Give your classes a reasonable complement of constructors and accessor methods, and an equals method as well. Write a driver program (no pun intended) that tests all your methods.

Notes:

This project requires most of the same type of thinking as previous problems to write constructors, accessor and mutator methods except that one of `Vehicle`'s data members is an instance of a class rather than a fundamental data type (`owner` is an instance of the class `Person`). Operations involving the owner, therefore, must use the methods of the `Person` class. NOTE: early editions of the text may have an error in the definition of `sameName()` in `Person`. If the method definition is

```
return (this.name.equalsIgnoreCase(otherPerson.name));
```

it should be changed to

```
return
(this.name.equalsIgnoreCase(otherPerson.gettName()));
```

The `sameName()` method is called in the code for the `equals` method of `Vehicle` to test if two vehicles have the same owner.

A methodical approach that makes this problem manageable is to develop one method in `Vehicle`, then write a test for the method in `VehicleTest` and run it. Add the next method to Vehicle only when `VehicleTest` runs successfully. When all the methods for `Vehicle` have been written and tested with `VehicleTest`, then do a similar piece-wise development for `Truck` and `TruckTest`, first testing all the methods inherited from `Vehicle`, then writing one new method at a time and testing it before going on to the next.

References:

Listing 8.1

Solution:

See the code in Vehicle.java, VehicleTest.java, Truck.java, and TruckTest.java.

4. Create a new class called Dog that is derived from the Pet class given in Listing 6.1 of Chapter 6. The new class has the additional attributes of breed (type String) and boosterShot (type boolean), which is true if the pet has had its booster shot and false if not. Give your classes a reasonable complement of constructors and accessor methods. Write a driver program to test all your methods, then write a program that reads in five pets of type Dog and displays the name and breed of all dogs that are over two years old and have not had their booster shots.

Notes:

This project would be easier to do with an array of type `Dog`, but how to write class constructors for arrays has not been covered, so the code to process one dog has to be repeated five times.

References:

Listing 6.1

Solution:

See the code in Dog.java, DogTest.java, and DogBoosterShotList.java.

Programming Projects:

2. Define two derived classes of the abstract class ShapeBase in Listing 8.19. Your two classes will be called RightArrow and LeftArrow. These classes will be like the classes Rectangle and Triangle, but they will draw arrows that point right and left, respectively. For example, the following arrow points to the right:

The size of the arrow is determined by two numbers, one for the length of the "tail" and one for the width of the arrowhead. (The width is the length of the vertical base.) The arrow shown here has a length of 12 and a width of 7. The width of the arrowhead cannot be an even number, so your constructors and mutator methods should check to make sure that it is always odd. Write a test program for each class that tests all the methods in the class. You can assume that the width of the base of the arrowhead is at least 3.

Notes:

The problem description requires that the base width be an odd number so the triangle is symmetrical, but it does not say exactly how to ensure this. One option is to test the user's input and display an error message if it is even, then prompt for re-entry of an odd number. Another approach is to adjust the width by adding or subtracting one if it is even, making sure it does not go below 3 if subtraction is used. The approach taken in the solution shown here is to add 1 if the user inputs an even number and, since it is always a good idea to keep the user informed, a message tells the user that the width was adjusted. Getting the display right on this one is tricky! It may take some trial and error to get it worked out, so it is especially helpful to implement it one step at at time, being patient and using `System.out.println` and `System.out.print` to obtain the right output. Consider doing the implementation in the following steps:

- tail of right arrow
- lines before the tail for a base width of at least 5
- lines after the tail for the same base width
- first line of left arrow for a base width of at least 5
- top lines of left arrow for the same base width
- line including the tail for the same base number
- bottom lines for the same base number

References:

Listing 8.19
Solution: *See the code in LeftArrow.java, LeftArrowTest.java, RightArrow.java, and RightArrowTest.java.*

3. Define two classes, Patient and Billing, whose objects are records for a clinic. Derive Patient from the class Person given in Listing 8.1. A Patient record has the patient's name (defined in the class Person) and identification number (use the type String). A Billing object will contain a Patient object and a Doctor object (from Practice Program 2). Give your classes a reasonable complement of constructors and accessor methods, and an equals method as well. First write a driver program to test all your methods, then write a test program that creates at least two patients, at least two doctors, and at least two Billing records and then displays the total income from the Billing records.

Notes:

This project sounds reasonable until an attempt is made to put `Doctor` and `Patient` objects in the `Billing` class. A `Billing` object actually needs just the doctor's name, patient's name, and doctor's office fee. The approach taken in the solution shown here is to require a `Doctor` and a `Patient` object in the constructor for a `Billing` object, then use `Doctor` and `Patient` accessor methods to set the parameter values for the `Billing` object.

References:

Listing 8.1, Practice Program 1, 2

Solution:

See the code in Patient.java, PatientTest.java, Billing.java, and BillingTest.java.

4. Create the classes RightTriangle and Rectangle, each of which is derived from the abstract class ShapeBase in Listing 8.19. Then derive a class Square
from the class Rectangle. Each of these three derived classes will have two additional methods to calculate area and circumference, as well as the inherited methods. Do not forget to override the method drawHere. Give your classes a reasonable complement of constructors and accessor methods. The Square class should include only one dimension, the side, and should automatically set the height and width to the length of the side. You can use dimensions in terms of the character width and line spacing even though they are undoubtedly unequal, so a square will not look square (just as a Rectangle object, as discussed in this chapter, won't look square.) Write a driver program that tests all your methods.

Notes:

This project is most easily written by modifying the programs in the text. Base `RightTriangle.java` on `Triangle.java` (Listing 8.14), `Rectangle.java` on `Rectangle.java` (Listing 8.13), and `MoreGraphicsDemo.java` on `TreeDemo.java` (Listing 8.15). `Square.java` is derived from `Rectangle`, so it is just a matter of using the parent's methods with both height and width set to the length of the side of the square. Note that special attention is required to draw the correct figures when the base, width, or height values are either 0 or 1.

References:

Listing 8.13, Listing 8.14, Listing 8.15

Solution:

See the code in RightTriangle.java, Rectangle.java, SquarePr7.java, and MoreGraphicsDemo.java.

5. Create an interface MessageDecoder that has a single abstract method decode(cipherText), where cipherText is the message to be decoded. The method will return the decoded message. Modify the classes SubstitutionCipher and ShuffleCipher, as described in Exercises 16 and 17, so that they implement MessageDecoder as well as the interface MessageEncoder that Exercise 15 describes. Finally, write a program that allows a user to encode and decode messages entered on the keyboard.

Notes: This project demonstrates creating multiple concrete classes that implement two related interfaces. The most difficult part of this implementation is handling characters in the substitution cipher.
References: Exercise 8.16, Exercise 8.17
Solution: *See the code in MessageDecoder.java, ShuffleCipher.java, SubstitutionCipher.java, and CodeProgram.java.*

6. For this Programming Project, start with implementations of the Person, Student, and Undergraduate classes as depicted in Figure 8.4 and the polymorphism demo in Listing 8.6. Define the Employee, Faculty, and Staff classes as depicted in Figure 8.2. The Employee class should have instance variables to store the employee ID as an int and the employee's department as a String. The Faculty class should have an instance variable to store the faculty member's title (e.g. "Professor of Computer Science") as a String. The Staff class should have an instance variable to store the staff member's pay grade (a number from 1 to 20) as an int. Every class should have appropriate constructors, accessors, and mutators, along with a writeOutput method that outputs all of the instance variable values.

Modify the program in Listing 8.6 to include at least one Faculty object and at least one Staff object in addition to the Undergraduate and Student objects. Without modification to the for loop, the report should output the

name, employee ID, department, and title for the Faculty objects, and the name, employee ID, department, and pay grade for the Staff objects.

Notes: The solution uses the class Employee2 to distinguish it from the Employee class from Practice Program 1.
References: Listing 8.6, Figure 8.2, Figure 8.4
Solution: *See the code in Employee2.java, Faculty.java, Staff.java, and PolymorphismDemo.java.*

7. Modify the Student class in Listing 8.2 so that it implements the Comparable interface. Define the compareTo method to order Student objects based on the value in studentNumber. In a main method create an array of at least 5 Student objects, sort them using Arrays.sort, and output the students. They should be listed by ascending student number. Next, modify the compareTo method so it orders Student objects based on the lexicographic ordering of the name variable. Without modification to the main method, the program should now output the students ordered by name.

Notes: This solution uses the class Student2 to distinguish it from the listing in the textbook. The solution is similar to Listing 8.18 and Listing 8.19.
References: Listing 8.2, Listing 8.18, Listing 8.19
Solution: *See the code in Student2.java, StudentDemo.java*

8. Create a JavaFX application that uses a TextField to get a message and encode or decode it using the classes described in the previous programming project. Use four buttons to control the kind of cipher used and to specify whether to encode or decode the message. Also, use a TextField to get the number used in the constructor for the ciphers.

Notes: In this project, we create an application that will encode and decode messages. It is relatively straightforward using the classes from the previous project. To do this, we will want variables of MessageEncoder and MessageDecoder interface types. When the cipher button is pressed, create a new cipher instance of the appropriate type and assign it to the variables.
References: Project 8.7
Solution: *See the code in CoderGUI.java*

10. Create an application in a JFrame GUI that will draw a spiral using line segments. The equations for the points on a spiral are
You should draw 150 points. Start θ at 0 and increase it by 0.1 for each new point. Let *k* be 15. Set the size of the window to 500 by 500.

Notes: This project is a simple Swing application that will draw a spiral. It requires familiarity with polar coordinates. To draw the spiral, override the paint method.
Solution: *See the code in Spiral.java.*

Exercises:

1. Write a program that allows students to schedule appointments at either 1, 2, 3, 4, 5, or 6 o'clock p. m. Use an array of six strings to store the names for the time slots. Write a loop that iterates as long as the array has a free space. Within a try block, allow the user to enter a time and a name. If the time is free, put the name in the array. If the time is not free, throw a TimeInUseException. If the time is not valid, throw an InvalidTimeException. Use a catch block for each different kind of exception.

Solution:

See the code in TimeInUseException.java, InvalidTimeException.java, and Scheduler.java.

2. Write a program that allows the user to compute the remainder after the division of two integer values. The remainder of x / y is x % y. Catch any exception thrown and allow the user to enter new values.

Solution:

See the code in ModProgram.java.

3. Write an exception class that is appropriate for indicating that a time entered by a user is not valid. The time will be in the format *hour*:*minute* followed by "am" or "pm."

Solution:

See the code in InvalidTimeFormatException.java.

4. Derive exception classes from the class you wrote in the previous exercise. Each new class should indicate a specific kind of error. For example, InvalidHourException could be used to indicate that the value entered for *hour* was not an integer in the range 1 to 12.

Solution:

See the code in InvalidHourException.java, InvalidMinuteException.java, InvalueFormattingException.java.

5. Write a class TimeOfDay that uses the exception classes defined in the previous exercise. Give it a method setTimeTo(timeString) that changes the time if timeString corresponds to a valid time of day. If not, it should throw an exception of the appropriate type.

Solution:

See the code in TimeOfDay.java.

6. Write code that reads a string from the keyboard and uses it to set the variable myTime of type TimeOfDay from the previous exercise. Use try-catch blocks to guarantee that myTime is set to a valid time.

Solution:

See the code in TimeProgram.java.

7. Create a class SongCard that represents a gift card for the purchase of songs online.
It should have the following private attributes:
• songs—the number of songs on the card
• activated—true if the card has been activated
and the following methods:
• SongCard(*n*)—a constructor for a card with *n* songs.
• activate()—activates the gift card. Throws an exception if the card has already been activated.
• buyASong()—records the purchase of one song by decreasing the number of songs left for purchase using this card. Throws an exception if the gift card is either completely used or not active.
• songsRemaining()—returns the number of songs that can be purchased using the gift card.

Solution:

See the code in SongCard.java, CardNotActivatedException.java,CardEmptyException.java.

8. Create a class Rational that represents a rational number. It should have private attributes for
• The numerator (an integer)
• The denominator (an integer)
and the following methods:
• Rational(*numerator*, *denominator*)—a constructor for a rational number.
• Accessor methods getNumerator and getDenominator and mutator methods setNumerator and setDenominator for the numerator and the denominator.
You should use an exception to guarantee that the denominator is never zero.

Solution:

See the code in Rational.java.

9. Revise the class Rational described in the previous exercise to use an assertion instead of an exception to guarantee that the denominator is never zero.

Solution:

See the code in RationalWithAssert.java.

10. Suppose that you are going to create an object used to count the number of people in a room. We know that the number of people in the room can never be negative.
Create a RoomCounter class having three public methods:
• addPerson—adds one person to the room
• removePerson—removes one person from the room
• getCount —returns the number of people in the room
If removePerson would make the number of people less than zero, throw a NegativeCounterException.

Solution:

See the code in RoomCounter.java.

11. Revise the class RoomCounter described in the previous exercise to use an assertion instead of an exception to prevent the number of people in the room from becoming negative.

Solution:

See the code in RoomCounterWithAssert.java.

12. Show the modifications needed to add exponentiation to the class Calculator in Listing 9.12. Use ^ to indicate the exponentiation operator and the method Math.pow to perform the computation.

Solution:

```
case '^': // THIS IS NEW CODE
                answer = Math.pow(n1, n2);
                break;
```

13. Write a class LapTimer that can be used to time the laps in a race. The class should have the following private attributes:
• running—a boolean indication of whether the timer is running
• startTime—the time when the timer started
• lapStart—the timer's value when the current lap started
• lapTime—the elapsed time for the last lap
• totalTime—the total time from the start of the race through the last completed lap
• lapsCompleted—the number of laps completed so far
• lapsInRace—the number of laps in the race
The class should have the following methods:
• LapTimer(*n*)—a constructor for a race having *n* laps.
• start —starts the timer. Throws an exception if the race has already started.
• markLap—marks the end of the current lap and the start of a new lap. Throws an exception if the race is finished.
• getLapTime—returns the time of the last lap. Throws an exception if the first lap has not yet been completed.
• getTotalTime—returns the total time from the start of the race through the last completed lap. Throws an exception if the first lap has not yet been completed.
• getLapsRemaining—returns the number of laps yet to be completed, including the current one.

Express all times in seconds.
To get the current time in milliseconds from some baseline date, invoke Calendar.getInstance().getTimeInMillis() This invocation returns a primitive value of type long. By taking the difference between the returned values of two invocations at two different times, you will know the elapsed time in milliseconds between the invocations. Note that the class Calendar is in the package java.util.

Solution:

See the code in TimerException.java, LapTimer.java.

Practice Programs:

1. Use the exception class MessageTooLongException of Self-Test Question 16 in a program that asks the user to enter a line of text having no more than 20 characters. If the user enters an acceptable number of characters, the program should display the message, "You entered *x* characters, which is an acceptable length" (with the letter *x* replaced by the actual number of characters). Otherwise, a MessageTooLongException should be thrown and caught. In either case, the program should repeatedly ask whether the user wants to enter another line or quit the program.

Notes: This project has exactly the same organization as `ExceptionDemo`, Listing 9.2
References: Listing 9.2, Self-Test Question 16
Solution: *See the code in MessageTooLongException.java and MessageTooLongExceptionDemo.java.*

2. A method that returns a special error code can sometimes cause problems. The caller might ignore the error code or treat the error code as a valid return value. In this case it is better to throw an exception instead. The following class maintains an account balance and returns a special error code.

```
public class Account
{
        private double balance;
        public Account()
        {
                balance = 0;
        }
        public Account(double initialDeposit)
        {
```

```
            balance = initialDeposit;
        }
        public double getBalance()
        {
            return balance;
        }
        // returns new balance or -1 if error
        public double deposit(double amount)
        {
            if (amount > 0)
                balance += amount;
            else
                return -1;      // Code indicating error
            return balance;
        }
        // returns new balance or -1 if invalid amount
        public double withdraw(double amount)
        {
            if ((amount > balance) || (amount < 0))
                return -1;
            else
                balance -= amount;
            return balance;
        }
}
```

Rewrite the class so that it throws appropriate exceptions instead of returning -1 as an error code. Write test code that attempts to withdraw and deposit invalid amounts and catches the exceptions that are thrown.

Notes: This solution throws a NegativeAmount and InsufficientBalance exception. It is worth pointing out to the students the problems that would occur if negative account balances were desirable.
Solution: *See the code in Account.java.*

Programming Projects:

1. Write a program that converts a time from 24-hour notation to 12-hour notation. The following is a sample interaction between the user and the program:

Enter time in 24-hour notation:
13:07
That is the same as
1:07 PM
Again? (y/n)
y

Enter time in 24-hour notation:
10:15
That is the same as
10:15 AM
Again? (y/n)
y

Enter time in 24-hour notation:
10:65
There is no such time as 10:65
Try Again:
Enter time in 24-hour notation:
16:05
That is the same as
4:05 PM
Again? (y/n)
n
End of program

Define an exception class called TimeFormatException. If the user enters an illegal time, like 10:65, or even gibberish, like 8&*68, your program should throw and handle a TimeFormatException.

Notes:

1. It is helpful to follow the author's suggestion to get the normal case working first, and then add exception handling. There are a number of issues to work out to do the transformation from 24-hour to 12-hour format. The first problem is to decide how to parse the input with a colon separating the hours and minutes integers. All of the methods introduced so far parse text that is delimited with spaces. The solution used here is to read the input one character at a time and use a switch structure to convert the hours and minutes to integers. But this leads to another problem, how to deal with variations on the input format. For example, the number of hours could have zero, one or two digits, e.g. `:10`, `0:10`, or `00:10` for ten minutes after midnight. To make the solution easier, a requirement is imposed on the input: It must be in xx.xx format, i.e. it must have two digits, a semicolon, and then another two digits. Any other format is flagged as a formatting error. So leading zeros are required for times earlier than 10:00. Another problem is obtaining and saving the input so it can be reprinted in the error message. The solution resolves this by reading in five characters individually (note the first one is with `readNonwhiteChar`) and any remaining characters as a string. The next steps check each character to see if they are valid: only 0, 1 or 2 are allowed for the first character, etc. Additional processing of legitimate times is needed to subtract 12 from hours that are over 12, print "noon" for 12:00, change "AM" to "PM" for hours greater than 11, and conditionally print a leading zero for minutes that are less than ten.

   After the code to parse and process the input is done, the easiest way to add exception handling is to modify the code in the `DivideByZeroException.java` and `DivideByZeroExceptionDemo.java` files. Just change the names in the exception definition file, replace the body of the demo file with the code developed above, and add the code in `catch` to get the correct printout.
   This is an excellent example to work on test case development since there are a number of situations that need special attention in the code. Here are some good examples:

- Insufficient number of characters in hh field - should cause an exception.
- 00:00 - Should print "0:00 AM".
- 12:00 - Should print "12:00 noon".
- 12:01 - Should print "12:01 PM".
- 11:59 - Should print "11:59 AM".
- 23:59 - Should print "11:59 PM".
- 24:00 - Should cause an exception.
- 11:60 - Should cause an exception.
- Combinations with all correct values except in one of the five positions (for example, a1:15, 1a:15, 11a15, 11:a5, and 11:1a) - Should cause an exception.
- A completely wrong input (for example f*!bc%) – Should cause and exception.

A correct input with additional characters (for example 11:15%xyz) – Should cause an exception.

Solution:

See the code in TimeFormatException.java and TimeFormatExceptionDemo.java.

2. Write a program that uses the class Calculator in Listing 9.12 to create a more powerful calculator. This calculator will allow you to save one result in memory and call the result back. The commands the calculator takes are

- e for end
- c for clear; sets result to zero
- m for save in memory; sets memory equal to result
- r for recall memory; displays the value of memory but does not change result.

You should define a derived class of the class Calculator that has one more instance variable for the memory, a new main method that runs the improved calculator, a redefinition of the method handleUnknownOpException, and anything else new or redefined that you need. A sample interaction with the user is shown next. Your program need not produce identical output, but it should be similar and just as clear or even clearer.

```
Calculator on:
result = 0.0
+ 4
result + 4.0 = 4.0
updated result = 4.0
/ 2
result / 2.0 = 2.0
updated result = 2.0
m
result saved in memory
c
result = 0.0
+ 99
result + 99.0 = 99.0
updated result = 99.0
/ 3
result / 3.0 = 33.0
updated result = 33.0
r
recalled memory value = 2.0
result = 33.0
+ 2
result + 2.0 = 35.0
updated result = 35.0
e
End of program
```

Notes:

Create a new class that extends `Calculator`, add an instance variable for memory, and add the code to implement the new features. Note that you should use the `setResult()` and `resultValue()` methods to access the instance variable `result` from the parent class. There should be a space between the operator and the number. The program does not catch the exceptions that occur when a non-numeric value is entered instead of a number.

References:

Listing 9.5, Listing 9.10, Listing 9.12

Solution:

See the code in Calculator.java, ImprovedCalculator.java, DivideByZeroException.java, and UnknownOpException.java.

3. Write a program that converts dates from numerical month–day format to alphabetic month–day format. For example, input of 1/31 or 01/31 would produce January 31 as output. The dialogue with the user should be similar to that shown in Programming Project 2. You should define two exception classes, one called MonthException and another called DayException. If the user enters anything other than a legal month number (integers from 1 to 12), your program should throw and catch a MonthException. Similarly, if the user enters anything other than a valid day number (integers from 1 to either 29, 30, or 31, depending on the month), your program should throw and catch a DayException. To keep things simple, assume that February always has 28 days.

Notes:

The big problem that influences the design of this project is that the numbers entered are ASCII strings and not integers. The processing that needs to be done (check for valid month numbers, check for valid day numbers, and translate from month numbers to month names) is much easier if the month and day are integers, so that is the approach taken in the solution shown here. The first problem is to parse the input to get the one or two ASCII character digits for each of the data items, month and day. The program uses `String` methods to find the position of the slash character, `/`, that separates the two, then usees it to obtain the one or two characters before it for the month, and the one or two characters after it for the day. Next it converts the one- or two-character digits to the decimal integers they represent. A relatively "clean" solution is to write a helper method that converts one ASCII digit-character into its integer value, then use the helper method to convert the one- or two-character month and day values to integers. The code to convert an ASCII integer character to a decimal integer value is written as a switch structure. The month must be converted before day because it is used to determine if the day value is valid. Once the month input is converted to a number it is easy to do the remaining processing. The month can be easily checked for validity (only 1 through 12 are valid values), used as an index into a `String` array to get the month's name, and used to check the day number for validity (e.g. if the month number is 1, 3, 5, 7, 8, 10, or 12, the day number must be in the range 1 – 31). The switch structure is also a very good choice for the day-check since it is very compact and readable. Organizing the steps to keep month processing separate from day processing allows the try/catch blocks for the two exceptions to be cleanly and clearly separated. So the solution here is organized as follows:

- Parse the input string to get the month part and the day part.
- Convert the month part to an integer, if possible.
- Check the month for validity: convert day number only if month is valid. It is in this block that a `MonthException` is thrown for any invalid

input for month or if the slash character that separates month and day is missing. Note that any 3-character number for day or month is considered invalid. So, while 01/01 is accepted and converted to January 1, 001/01 and 01/001 are flagged as invalid.

If month is a valid number, convert day to its integer value and check for validity. It is here that a `DayException` is thrown for any invalid day input.

Solution:

See the code in MonthException.java, DayException.java, and DateFormatConverter.java.

5. Define an exception class called DimensionException to use in the driver program from Programming Project 4 in Chapter 8. Modify that driver program to throw and catch a DimensionException if the user enters something less than or equal to zero for a dimension.

Notes:

This project is fairly simple. The `DimensionException` class is written to accept a `String` and an integer so the message can be more helpful (it can display which dimension is invalid and its value). The `MoreGraphicsDemo` program from Chapter 8 Programming Project 4 is modified to create the version using `DimensionException` and to be interactive so the exception code can be exercised.

References:

Project 8.4, Listing 8.15

Solution:

See the code in DimensionException.java, DimensionExceptionDemo.java, SquarePr7.java, Rectangle.java, and RightTriangle.java. Uses ShapeBase.java.

6. Write a program to enter employee data, including social security number and salary, into an array. The maximum number of employees is 100, but your program should also work for any number of employees less than 100. Your program should use two exception classes, one called SSNLengthException for when the social security number entered—without dashes or spaces—is not exactly nine characters and the other called SSNCharacterException for when any character in the social security number is not a digit. When an exception is thrown, the user should be reminded of what she or he entered, told why it is inappropriate, and asked to reenter the data. After all data has been entered, your program should display the records for all employees, with an annotation stating whether the employee's salary is above or below average. You will also need to define the classes Employee, SSNLengthException, and SSNCharacterException. Derive the class Employee from the class Person in Listing 8.4 of Chapter 8. Among other things, the class Employee should have input and output methods, as well as constructors, accessor methods, and mutator methods. Every Employee object should record the employee's name, salary, and social security number, as well as any other data you need or think is appropriate.

Notes:

It may be useful to review arrays of objects before doing this project. For example, even after creating the array of `EmployeeCh8` objects, it is necessary to create each element with a new statement inside the loop that reads in the employee's information. Note that it is useful to use the same variable for the array subscript and the employee's number, except that the subscript begins with 0 and the employee number begins with 1. So, whenever the employee number needs to be displayed, simply use the expression (subscript variable + 1).

References:

Listing 8.4, Practice Program 8.1

Solution:

See the code in EmployeeCh9.java, EmployeeCh9Demo.java, SSNLengthException.java, and SSNCharacterException.java. Uses Person.java.

7. Create a JavaFX application that implements a short survey. The first question should ask the user for his or her favorite color and present the choices “red”, “orange”, “blue”, and “green” in radio buttons. The second question should ask the user for his or her age and present the choice in a spinner with the range 10–100. The third and final question should ask the user to select his or her favorite programming language from the choices “Java”, “C++”, “Python”, and “C#” presented in a choice box. Add a button, that when clicked, summarizes the user’s selections.

Notes: The solution is similar to the Additional Controls Demo given in the text.
References: Listing 9.12
Solution: *See the code in SurveyForm.java*

10. Suppose that you are in charge of customer service for a certain business. As phone calls come in, the name of the caller is recorded and eventually a service representative return the call and handles the request.

Write a class ServiceRequests that keeps track of the names of callers. The class should have the following methods:
• addName(*name*)—adds a name to the list of names. Throws a ServiceBackUpException if there is no free space in the list.
• removeName(*name*)—removes a name from the list. Throws a NoServiceRequestException if the name is not in the list.
• getName(*i*)—returns the *i*th name in the list.
• getNumber—returns the current number of service requests.

Write a program that uses an object of type ServiceRequests to keep track of customers that have called. It should have a loop that, in each iteration, attempts to add a name, remove a name, or print all names. Use an array of size 10 as the list of names.

Notes: This project is fairly straightforward. ServiceRequests is really a queue. RemoveName is the only complicated methods in the class. An iterative approach that leaves it for last would be a good idea. The program is a basic while loop with multiple cases for each of the operations of the Service request class.
Solution: *See the code in ServiceBackUpException.java, NoServiceRequestException.java, ServiceRequests.java, ServiceProgram.java.*

11. Write an application that implements a trip-time calculator. Define and use a class TripComputer to compute the time of a trip. TripComputer should have the private attributes

• totalTime—the total time for the trip

• restStopTaken—a boolean flag that indicates whether a rest stop has been taken at the end of the current leg and the following methods:

• computeLegTime(*distance*, *speed*)—computes the time for a leg of the trip having a given distance in miles and speed in miles per hour. If either the distance or the speed is negative, throws an exception.

• takeRestStop(*time*)—takes a rest stop for the given amount of time. If the time is negative, throws an exception. Also throws an exception if the client code attempts to take two rest stops in a row.

• getTripTime—returns the current total time for the trip.

Notes: This structure of this project is similar to project 9. Even though the class TripComputer is pretty simple, developing and testing it first is advisable.
Solution: *See the code in TripComputerException.java, TripComputer.java, TripComputerApp.java.*

Exercises:

1. Write a program that will write the Gettysburg address to a text file. Place each sentence on a separate line of the file.

Solution:

See the code in Gettysburg.java.

2. Modify the program in the previous exercise so that it reads the name of the file from the keyboard.

Solution:

See the code in Gettysburg2.java.

4. Write a program that will record the purchases made at a store. For each purchase, read from the keyboard an item's name, its price, and the number bought. Compute the cost of the purchase (number bought times price), and write all this data to a text file. Also, display this information and the current total cost on the screen. After all items have been entered, write the total cost to both the screen and the file. Since we want to remember all purchases made, you should append new data to the end of the file.

Solution:

See the code in RecordASale.java.

5. Modify the class LapTimer, as described in Exercise 13 of the previous chapter, as follows:

• Add an attribute for a file stream to which we can write the times

• Add a constructor

LapTimer(n, person, fileName) for a race having *n* laps. The name of the person and the file to record the times are passed to the constructor as strings. The file should be opened and the name of the person should be written to the file. If the file cannot be opened, throw an exception.

Solution:

See the code in LapTimer.java.

6. Write a class TelephoneNumber that will hold a telephone number. An object of this class will have the attributes
• areaCode—a three-digit integer
• exchangeCode—a three-digit integer
• number—a four-digit integer
and the methods
• TelephoneNumber(aString)—a constructor that creates and returns a new instance of its class, given a string in the form xxx-xxx-xxxx or, if the area code is missing, xxx-xxxx. Throw an exception if the format is not valid. *Hint*: To simplify the constructor, you can replace each hyphen in the telephone number with a blank. To accept a telephone number containing hyphens, you could process the string one character at a time or learn how to use Scanner to read words separated by a character—such as a hyphen—other than whitespace.
• toString—returns a string in either of the two formats shown previously for the constructor.

Using a text editor, create a text file of several telephone numbers, using the two formats described previously. Write a program that reads this file, displays the data on the screen, and creates an array whose base type is TelephoneNumber. Allow the user to either add or delete one telephone number. Write the modified data on the text file, replacing its original contents. Then read and display the numbers in
the modified file.

Solution:

See the code in InvalidTelephoneFormatException.java, TelephoneNumber.java, TelephoneProgram.java. Input is in numbers.txt

7. Write a class ContactInfo to store contact information for a person. It should have attributes for a person's name, business phone, home phone, cell phone, email address, and home address. It should have a toString method that returns this data as a string, making appropriate replacements for any attributes that do not have values. It should have a constructor ContactInfo(aString) that creates and returns a new instance of the class, using data in the string aString. The constructor should use a format consistent with what the toString method produces. Using a text editor, create a text file of contact information, as described in the previous paragraph, for several people. Write a program that reads this file, displays the data on the screen, and creates an array whose base type is ContactInfo. Allow the user to do one of the following: change some data in one contact, add a contact, or delete a contact. Finally, write over the file with the modified contacts.

Solution:

See the code in InvalidContactException.java, ContactInfo.java, ContactProgram.java. Input is in contacts.txt

8. Write a program that reads every line in a text file, removes the first word from each line, and then writes the resulting lines to a new text file.

Solution:

See the code in FirstWordRemover.java.

10. Write a program that will make a copy of a text file, line by line. Read the name of the existing file and the name of the new file—the copy—from the keyboard. Use the methods of the class File to test whether the original file exists and can be read. If not, display an error message and abort the program. Similarly, see whether the name of the new file already exists. If so, display a warning message and allow the user to either abort the program, overwrite the existing file, or enter a new name for the file.

Solution:

See the code in FileCopier.java.

11. Suppose you are given a text file that contains the names of people. Every name in the file consists of a first name and last name. Unfortunately, the programmer that created the file of names had a strange sense of humor and did not guarantee that each name was on a single line of the file. Read this file of names and write them to a new text file, one name per line. For example, if the input file contains
Bob Jones Fred
Charles Ed
Marston
Jeff
Williams
the output file should be
Bob Jones
Fred Charles
Ed Marston
Jeff Williams

Solution:

See the code in RecoverNames.java.

12. Suppose that you have a binary file that contains numbers whose type is either int or double. You don't know the order of the numbers in the file, but their order is recorded in a string at the beginning of the file. The string is composed of the letters *i* (for int) and *d* (for double) in the order of the types of the subsequent numbers. The string is written using the method writeUTF. For example, the string "iddiiddd" indicates that the file contains eight values, as follows: one integer, followed by two doubles, followed by two integers, followed by three doubles. Read this binary file and create a new text file of the values, written one to a line.

Solution:

See the code in UTFBinaryFileReader.java. The code in UTFBinaryFileWriter allows one to create a file with the appropriate format.

13. Suppose that we want to store digitized audio information in a binary file. An audio signal typically does not change much from one sample to the next. In this case, less memory is used if we record the change in the data values instead of the actual data values. We will use this idea in the following program. Write a program StoreSignal that will read positive integers, each of which must be within 127 of the previous integer, from the keyboard (or from a text file, if you prefer). Write the first integer to a binary file. For each subsequent integer, compute the difference between it and the integer before it, cast the difference to a byte, and write the result to the binary file. When a negative integer is encountered, stop writing the file.

Solution:

See the code in StoreSignal.java.

14. Write a program RecoverSignal that will read the binary file written by StoreSignal, as described in the previous exercise. Display the integer values that the data represents on the screen.

Solution:

See the code in RecoverSignal.java.

15. Even though a binary file is not a text file, it can contain embedded text. To find out if this is the case, write a program that will open a binary file and read it one byte at a time. Display the integer value of each byte as well as the character, if any, that it represents in ASCII.
Technical details: To convert a byte to a character, use the following code:
char[] charArray = Character.toChars(byteValue);
The argument byteValue of the method toChars is an int whose value equals that of the byte read from the file. The character represented by the byte is charArray[0]. Since an integer is four bytes, byteValue can represent four ASCII characters. The method toChars tries to convert each of the four bytes to a character and places them into a char array. We are interested in just the character at index 0. If a byte in the file does not correspond to a character, the method will throw an IllegalArgumentException. If the exception is thrown, display only the byte value and continue on to the next byte.

Solution:

See the code in ByteReader.java.

Practice Programs:

1. Write a program that searches a file of numbers and displays the largest number, the smallest number, and the average of all the numbers in the file. Do not assume that the numbers in the file are in any special order. Your program should obtain the file name from the user. Use either a text file or a binary file. For the text-file version, assume one number per line. For the binary-file version, use numbers of type double that are written using writeDouble.

Notes:

This project is deceptive: it requires more programs than the ones listed in the problem definition to process binary and text files containing floating point numbers; additional programs are required to create and view the data files. The solution shown here has separate programs to create and view data files, one for text and another for binary files. The program to process text files requires a method to translate the numbers from a `String` to type `double`. The easiest way is to use the wrapper class `Double`'s `parseDouble` method.

1) Develop the class to create and view binary data files. Good models to follow are `Doubler`, Listing 10.8, for the overall program organization, and `BinaryOutputDemo`, Listing 10.5, for showing the file contents.
2) Develop the class to process the data in a binary file (display the high, low and average), again using `Doubler`, Listing 10.8, as the model.

Use the programs from these two steps to develop similar programs for text files, but use `TextFileOutputDemo`, Listing 10.1, and `TextFileInputDemo`, Listing 10.2, as models for writing to and reading from text files.

References:

Listing 10.1, Listing 10.2, Listing 10.5, Listing 10.8

Solution:

See the code in WriteRealNumberBinaryFile.java, RealNumberHighLowAverageBinary.java, WriteRealNumberTextFile.java, and RealNumberHighLowAverageText.java.

2. Write a program that reads a file of numbers of type int and writes all the numbers to another file, but without any duplicate numbers. Assume that the numbers in the input file are already ordered from smallest to largest. After the program is run, the new file will contain all the numbers in the original file, but no number will appear more than once in the file. The numbers in the output file should also be sorted from smallest to largest. Your program should obtain both file names from the user. Use either a text file or a binary file. For the text-file version, assume one number per line. For the binary-file version, use numbers of type int that are written using writeInt.

Notes:

This project is similar to Practice Program 1: data files need to be created before any processing can be done and it is easier to work with binary files than text. So a good approach is start with the binary file classes from Project 1 and modify them so data files can be created and displayed, then write the program to process the binary data files. Note that a separate program to display data files is necessary to view the files created by the program that removes the duplicates. If students do not have the programs from Project 1 to work from, then they could start with files from the text, `BinaryOutputDemo.java` (Listing 10.5) and `Doubler.java` (Listing 10.8). After all the binary file programs are written and tested, it is easier to develop the programs to create, read and process text files. Just as with Project 1, the code for setting up the input and output streams needs to be changed, the while-loop condition needs to be changed to end when a null string is read, and text `String`s in the data file must be changed to `int`s by using the `parseInt` method in the `Integer` class

References:

Practice Program 10.1, Listing 10.1, Listing 10.5, Listing 10.8

Solution:

See the code in WriteIntegerNumberBinaryFile.java, DisplayIntegerNumberBinaryFile.java, SortedIntegerNoDuplicatesBinaryFile.java, WriteIntegerNumberTextFile.java, DisplayIntegerNumberTextFile.java, and SortedIntegerNoDuplicatesTextFile.java.

3. The following is an old word puzzle: “Name a common word, besides tremendous, stupendous and horrendous, that ends in dous.” If you think about this for a while it will probably come to you. However, we can also solve this puzzle by reading a text file of English words and outputting the word if it contains “dous” at the end. The text file “words.txt” contains 87314 English words, including the word that completes the puzzle. This file is available online with the source code for the book. Write a program that reads each word from the text file and outputs only those containing “dous” at the end to solve the puzzle.

Notes: Fairly short text-file program. The dictionary of words can be used for many other puzzle-based programs involving words.
Solution: *See the code in WordPuzzle.java.*

Programming Projects:

1. Write a program that checks a text file for several formatting and punctuation matters. The program asks for the names of both an input file and an output file. It then copies all the text from the input file to the output file, but with the following two changes: (1) Any string of two or more blank characters is replaced by a single blank; (2) all sentences start with an uppercase letter. All sentences after the first one begin after either a period, a question mark, or an exclamation mark that is followed by one or more whitespace characters.

Notes:

This project is deceptively simple. The problem statement is clear enough, with only a couple ambiguities to clarify. The solution shown here assumes that there is at least one line in the file to process and keeps all tabs and newlines unless they precede the first word on the first line. A helper method processes the first part of the first line to remove all leading white space and capitalize the first letter. After that, main simply reads one line of text at a time until it gets a null string, and uses another helper method to process the remaining text. This helper method is where most of the work is done: It processes each line, character by character, and uses flags to control the processing; to print only one space when there more than one in sequence, and to capitalize the first word in each sentence. Notice how the helper method to convert a character to upper case is written. The code first checks to see if the character is a lower case letter, and, if it is, it does integer arithmetic to convert the ASCII lower case code to the ASCII upper case code. A check of an ASCII chart will show that the upper case codes are 32 less than those for lower case. After doing the subtraction it is necessary to cast the integer result back to `char` to match the method's return type, `char`.

References:

Practice Program 10.1, Practice Program 10.2

Solution:

See the code in WriteSentenceTextFile.java, DisplaySentenceTextFile.java, and EditSentenceTextFile.java.

2. Write a program similar to the one in Listing 10.10 that can write an arbitrary number of Species objects to a binary file. (Species appears in Listing 5.19 of Chapter 5.) Read the file name and the data for the objects from a text file that you create by using a text editor. Then write another program that can search a binary file created by your first program and show the user the data for any requested endangered species. The user gives the file name and then enters the name of the species. The program either displays all the data for that species or gives a message if that species is not in the file. Allow the user to either enter additional species' names or quit.

Notes: This project requires careful placement of try/catch blocks to gracefully respond to file names and Species that do not exist, and, at the same time, allow the user to continue looking for Species in a file or choose another file to search.
References: Listing 10.9, Listing 10.10
Solution: *See the code in WriteSpeciesFile.java, DisplaySpeciesFile.java, and FindSpeciesRecords.java. Uses Species.java.*

3. Write a program that reads from a file created by the program in the previous programming project and displays the following information on the screen: the data for the species having the smallest population and the data for the species having the largest population. Do not assume that the objects in the file are in any particular order. The user gives the file name.

Notes:

This project requires only one new file since the classes to build and view Species files are provided in the text. Populations, of course, are not unique, so more than one species may have the same smallest or largest population. If more than Species has the smallest or largest population the solution simply displays the record for the first one it encounters it its sequential search through the records in the file. Test data files should include the following situations:

- multiple records with the smallest and largest populations,
- populations that are out of order numerically,
- records with largest and smallest values in various positions in the file (the first record, the last record, and an intermediate position),
- the record with the smallest population placed before the one with the largest, and
- the record with the largest population placed before the one with the smallest.

References:

Project 10.2, Listing 10.9, Listing 10.10

Solution:

See the code in SpeciesPopulationRange.java. Uses Species.java and files created by WriteSpeciesFile.java from the previous project.

4. Programming Project 2 asks you, among other things, to write a program that creates a binary file of objects of the class Species. Write a program that reads from a file created by that program and writes the objects to another file after modifying their population figures as they would be in 100 years. Use the method predict- Population of the class Species, and assume that you are given each species' growth rate.

Notes: This project requires only one file to be written. This solution is organized as a sequence of just four method calls in main, one to open an input file for reading `Species` records, one to open an output file for writing new records, one to do the processing (read a record from the input file and write a record to the output file with the population changed to the projected population after 100 years), and one to close the input and output files.
References: Project 10.2, Listing 10.9, Listing 10.10
Solution: *See the code in SpeciesPopulationsIn100Years.java. Uses Species.java and files created by WriteSpeciesFile.java from project 2.*

5. Text messaging is a popular means of communication. Many abbreviations are in common use but are not appropriate for formal communication. Suppose the abbreviations are stored, one to a line, in a text file named abbreviations.txt. For example, the file might contain these lines:
lol
:)
iirc
4
u
ttfn

Write a program that will read a message from another text file and surround each occurrence of an abbreviation with <> brackets. Write the marked message to a new text file.
For example, if the message to be scanned is
How are u today? Iirc, this is your first free day. Hope you are having fun! :)
the new text file should contain
How are <u> today? <Iirc>, this is your first free day. Hope you are having fun!
<:)>

Notes:

The solution for this project makes use of a couple methods that break out the processing of a line. The major method processes and marks a line for a single abbreviation. It finds the index of the abbreviation in the line and then breaks the line up into 3 parts. It then gets the character immediately before and after the abbreviation and checks to see if either is a letter or digit. If so, then we assume that the abbreviation is part of a legal word and don't mark it. Otherwise, we splice in the <> marker. This is done in a while loop that processes the remaining part of the line until the abbreviation is not found.

Solution:

See the code in AbbreviationMarker.java.

6. Modify the TelephoneNumber class described in Exercise 6 so that it is serializable. Write a program that creates an array whose base type is TelephoneNumber by reading data from the keyboard. Write the array to a binary file using the method writeObject. Then read the data from the file using the method readObject and display the information to the screen. Allow the user to change, add, or delete any telephone number until he or she indicates that all changes are complete. Then write the modified telephone numbers to the file, replacing its original contents.

Notes:

One difficulty in the solution to this project is that we have not yet seen the collection classes. We will read the telephone numbers into an array. We either need to determine the number of objects in the file or expand the array as we read the objects. This solution reads the file twice, once to determine the number of objects and then to read the objects into an appropriately sized array.

Once this is done, we make a single change and then write the file back out.

References:

Exercise 10.6

Solution:

See the code in SerializedTelephoneNumber.java, SerializedTelephoneProgram.java, MissingTelephoneInputFileException.java and the data file numbers.data.

7. Revise the class Pet, as shown in Listing 6.1 of Chapter 6, so that it is serializable. Write a program that allows you to write and read objects of type Pet to a file. The program should ask the user whether to write to a file or read from a file. In either case, the program next asks for the file name. A user who has asked to write to a file can enter as many records as desired. A user who has asked to read from a file is shown all of the records in the file. Be sure that the records do not scroll by so quickly that the user cannot read them. *Hint*: Think of a way to pause the program after a certain number of lines are displayed.

Notes:

This project can be written by making modifications to `ClassObjectIODemo`, Listing 10.9, however the `PetRecord` class does not have the useful `readInput()` and `toString()` methods as `Species`, so they have been added. Also, note that `PetRecord` must implement Serializable.

References:

Listing 6.1, Listing 10.9, Listing 10.10

Solution:

See the code in PetRecord.java and PetFileReadOrWrite.java

8. Write a program that reads records of type Pet from a file created by the program described in the previous programming project and displays the following information on the screen: the name and weight of the heaviest pet, the name and weight of the lightest pet, the name and age of the youngest pet, and the name and age of the oldest pet.

Notes: This project requires only one file and has a structure similar to Project 6 (in fact it has the same code for `getInputFile()` and `closeFile()` methods.
References: Project 10.9. Listing 6.1. Listing 10.10
Solution: *See the code in PetAgeAndWeightRange.java. Uses PetRecord.*

9. The UC Irvine Machine Learning repository contains many datasets for conducting computer science research. One dataset is the Haberman's Survival dataset, available at http://archive.ics.uci.edu/ml/datasets/Haberman's+Survival and also included online with the source code for the book. The file "haberman.data" contains survival data for breast cancer patients in comma-separated value format. The first field is the patient's age at the time of surgery, the second field is the year of the surgery, the third field is the number of positive axillary nodes detected, and the fourth field is the survival status. The survival status is 1 if the patient survived 5 years or longer and 2 if the patient died within 5 years.

Write a program that reads the CSV file and calculates the average number of positive axillary nodes detected for patients that survived 5 years or longer, and the average number of positive axillary nodes detected for patients that died within 5 years. A significant difference between the two averages suggests whether or not the number of positive axillary nodes

detected can be used to predict survival time. Your program should ignore the age and year fields for each record.

Notes: The case study in Listing 10.4 shows how to process a CSV file and makes a good starting point for this project.
References: Listing 10.4
Solution: *See the code in Haberman.java. Files in haberman.names, haberman.data*

12. Write a Java program that serves as a primitive web browser. For this assignment it merely needs to input a server name and display the HTML that is sent by the webserver. A web server normally listens on port 80. Upon connection the server expects to be sent a string that identifies what webpage to receive (use / for the root) and what protocol is used. The next line is the Host and then a blank line. For example, to get the default page on Wikipedia the Java program would connect to port 80 and send:

```
GET / HTTP/1.1
Host: www.wikipedia.org
(blank line)
```

The Wikipedia server would then send back the HTML for the site which your program should display in text. For a more challenging program, parse and render the HTML in a human-friendly format instead of printing out the raw HTML.

Notes: The code closely follows the example given in the book for the URL class. In the implementation we needed to add try/catch and the appropriate import statements.
Solution: *See the code in Browser.java*

14. Write a JavaFX application uses a text field to get the name of a file, reads the file byte by byte, and displays the bytes as characters. (Exercise 15 describes how to convert a byte value to a character.) Display the first 20 characters in a label. If a byte does not correspond to a legal character, display a space instead.
Clicking the Next button reads and displays the next 20 characters in the file. The GUI might look like the sketch in Figure 10.8.

Notes:

Important note: This program will only read a binary file created using ObjectOutputStream. This is because it sues ObjectInputStream to read the file, which won't work on text files. The chapter doesn't cover material on reading an arbitrary file as bytes.

This project creates a JavaFX application that will let students explore the formatting of various kinds of files. It is very helpful to have a couple methods that process the file. This solution has two such methods. The first method gets 20 bytes and converts them into characters. It uses the method toChars from the Character class to create an array of 8 characters of which the only one we want is the first. The second method opens the file and calls the first method to get the initial 20 bytes for the display.

Solution:

See the code in FileByByte.java.

15. Write a JavaFX application that implements a simple text editor. Use a text field and a button to get the file. Read the entire file as characters and display it in a TextArea. The user will then be able to make changes in the text area. Use a Save button to get the contents of the text area and write that over the original file.
Technical Details: Read each line from the file and then use the method append(aString) to display the line in the text area. The method getText will return all of the text in the text area in a string that then can be written to the file.

Notes:

Surprisingly, using just the built-in text area component of Java a simple editor can be created. The text editing is done via the operating system on the text in the component. What is left for us to do is to read and write the file, which is accomplished with a pair of methods.

Solution:

See the code in SimpleEditor.java.

Exercises:

1. What output will be produced by the following code?

```
public class Demo {
    public static void main(String[] args) {
        System.out.println("The output is:");
        foo(23);
        System.out.println();
    }
    public static void foo(int number) {
        if (number > 0) {
            foo(number / 2);
            System.out.print(number % 2);
        }
    }
}
```

Solution:

The output is:
10111

This code is in Demo1.java.

2. What output will be produced by the following code?

```
public class Demo {
    public static void main(String[] args) {
        System.out.println("The output is:");
        bar(11156);
        System.out.println();
    }
    public static void bar(int number) {
        if (number > 0) {
            int d = number % 10;
            boolean odd = (number / 10) % 2 == 1;
            bar(number / 10);
            if (odd)
                System.out.print(d / 2 + 5);
            else
                System.out.print(d / 2);
        }
    }
}
```

Solution:

The output is:
05578

This code is in Demo2.java.

3. Write a recursive method that will compute the number of odd digits in a number.

Solution:

```
public static long countOdd(long number){
    long result;
    if(number == 0)
        // base case
        result = 0;
    else {
        long digit = number % 10;
        if(digit < 0)
            digit = -1 * digit;
        if(digit % 2 == 1)
            result = countOdd(number/10) + 1;
        else
            result = countOdd(number/10);
    }

    return result;
}
```

This code is in Methods.java.

4. Write a recursive method that will compute the sum of the digits in a positive number.

Solution:

```
public static long sumDigits(long number){
    long result;
    if(number == 0)
        // base case
        result = 0;
    else {
        long digit = number % 10;
        if(digit < 0)
            digit = -1 * digit;
        result = digit + sumDigits(number/10);
    }

    return result;
}
```

This code is in Methods.java.

5. Complete a recursive definition of the following method:

/**

Precondition: n >= 0.

Returns 10 to the power n.

*/

public static int computeTenToThe(int n)

Use the following facts about *xn*:

xn = (*xn*/2)2 when *n* is even and positive

xn = *x* (*x*(*n* - 1)/2)2 when *n* is odd and positive

*x*0 = 1

Solution:

```
/**
 * Precondition: n >= 0.
 * Returns 10 to the power n.
 */
public static int tenToThe(int n){
    int result;

    if(n==0){
        result = 1;
    } else {
        result = tenToThe(n/2);
        result = result * result;
        if(n%2 == 1){
            // n is odd we need to square then multiply by 10
            result = result * 10;
        }
    }
    return result;
}
```

This code is in Methods.java.

6. Write a recursive method that will compute the sum of all the values in an array.

Solution:

```
public static int sumArray(int [] data){
    return sumArray(data, data.length-1);
}

public static int sumArray(int [] data, int last){
    int result;

    if(last < 0)
        result = 0;   // only one value in the subarray
    else{
        result = data[last] + sumArray(data, last-1);
    }

    return result;

}
```

This code is in Methods.java.

7. Write a recursive method that will find and return the largest value in an array of integers. *Hint*: Split the array in half and recursively find the largest value in each half. Return the larger of those two values.

Solution:

```
public static int max(int [] data){
    return max(data, 0, data.length-1);
}

public static int max(int [] data, int first, int last){
    int result;

    if(first == last)
        result = data[first];   // only one value in the subarray
    else{
        int mid = (last + first)/2;
        int max1 = max(data, first, mid);
        int max2 = max(data, mid+1, last);
        if(max1 > max2)
            result = max1;
        else
            result = max2;
    }

    return result;

}
```

This code is in Methods.java..

8. Write a recursive ternary search algorithm that splits the array into three parts instead of the two parts used by a binary search.

Solution:

```
    public static int trinarySearch(int data[], int target){
        return trinarySearch(data, target, 0, data.length-1);
    }

    //Uses trinary search to search for target in a[first] through
    //a[last] inclusive. Returns the index of target if target
    //is found. Returns -1 if target is not found.
    public static int trinarySearch(int data[], int target,
                                    int first, int last) {
        int result;
        if (first > last)
            result = -1;
        else {
            int mid1 = (2*first + last)/3;
            int mid2 = (first + 2*last)/3;
            if (target == data[mid1])
                result = mid1;
            else if (target == data[mid2])
                result = mid2;
            else if (target < data[mid1])
                result = trinarySearch(data, target, first, mid1 - 1);
            else if (target < data[mid2])
                result = trinarySearch(data, target, mid1 + 1, mid2-1);
            else
                result = trinarySearch(data, target, mid2 + 1, last);

        }
        return result;
    }
```

This code is in Methods.java.

9. Write a recursive method that will compute cumulative sums in an array. To find the cumulative sums, add to each value in the array the sum of the values that precede it in the array. For example, if the values in the array are [2, 3, 1, 5, 6, 2, 7], the result will be [2, (2) + 3, (2 + 3) + 1, (2 + 3 + 1) + 5, (2 + 3 + 1 + 5) + 6, (2 + 3 + 1 + 5 + 6) + 2, (2 + 3 + 1 + 5 + 6 + 2) + 7] or [2, 5, 6, 11, 17, 19, 26]. *Hint*: The parenthesized sums in the previous example are the results of a recursive call.

Solution:

```
public static void cumulativeSum(int data[]){
    cumulativeSum(data, 1);
}

public static void cumulativeSum(int data[], int n) {
    if (n == data.length)
        return;
    else {
        data[n] += data[n-1];
        cumulativeSum(data, n+1);
    }
}
```

This code is in Methods.java.

10. Suppose we want to compute the amount of money in a bank account with compound interest. If the amount of money in the account is *m*, the amount in the account at the end of the month will be 1.005*m*. Write a recursive method that will compute the amount of money in an account after *t* months with a starting amount of *m*.

Solution:

```
public static double compoundInterest(double start, int months){
    double result;

    if(months <= 0){
        result = start;
    } else {
        result = 1.005 * compoundInterest(start, months-1);
    }

    return result;

}
```

This code is in Methods.java.

11. Suppose we have a satellite in orbit. To communicate to the satellite, we can send messages composed of two signals: dot and dash. Dot takes 2 microseconds to send, and dash takes 3 microseconds to send. Imagine that we want to know the number of different messages, $M(k)$, that can be sent in k microseconds.

• If k is 0 or 1, we can send 1 message (the empty message).
• If k is 2 or 3, we can send 1 message (dot or dash, respectively).
• If k is larger than 3, we know that the message can start with either dot or dash. If the message starts with dot, the number of possible messages is $M(k - 2)$. If the message starts with dash, the number of possible messages is $M(k - 3)$. Therefore the number of messages that can be sent in k microseconds is $M(k - 2) + M(k - 3)$.

Write a program that reads a value of k from the keyboard and displays the value of $M(k)$, which is computed by a recursive method.

Solution:

```
public static int messages(int time){
     int result;

     if(time <= 3)
         result = 1;
     else
         result = messages(time - 2) + messages(time - 3);

     return result;
  }
```

This code is in Methods.java.

12. Write a recursive method that will count the number of vowels in a string. *Hint*: Each time you make a recursive call, use the String method substring to construct a new string consisting of the second through last characters. The final call will be when the string contains no characters.

Solution:

```
public static int countVowels(String s){
    int result;

    if(s.length() == 0)
        result = 0;
    else {
        if(isVowel(s.charAt(0)))
            result = 1 + countVowels(s.substring(1));
        else
            result = countVowels(s.substring(1));

    }
    return result;
}

public static boolean isVowel(char c){
    return c=='a' || c=='e' || c=='i' || c=='o' || c=='u'
            || c=='A' || c=='E' || c=='I' || c=='O' || c=='U';
}
```

This code is in Methods.java.

13. Write a recursive method that will remove all the vowels from a given string and return what is left as a new string. *Hint*: Use the + operator to perform string concatenation to construct the string that is returned.

Solution:

```
public static String removeVowels(String s){
    String result;

    if(s.length() == 0)
        result = "";
    else {
        if(isVowel(s.charAt(0)))
            result = removeVowels(s.substring(1));
        else
            result = s.charAt(0) + removeVowels(s.substring(1));

    }
    return result;
}
```

This code is in Methods.java.

14. Write a recursive method that will duplicate each character in a string and return the result as a new string. For example, if "book" is the argument, the result would be "bbooookk".

Solution:

```
public static String doubleEachLetter(String s){
        String result;


        if(s.length() == 0)
            result = "";
        else {
            String doubled = "" +  s.charAt(0) + s.charAt(0);
            result = doubled + doubleEachLetter(s.substring(1));
        }
        return result;
    }
```

This code is in Methods.java.

15. Write a recursive method that will reverse the order of the characters in a given string and return the result as a new string. For example, if "book" is the argument, the result would be "koob".

Solution:

```
    public static String reverse(String s){
        String result;


        if(s.length() == 0)
            result = "";
        else {
            result =  reverse(s.substring(1)) + s.charAt(0);
        }
        return result;
    }
```

This code is in Methods.java.

Practice Programs:

1. Write a static recursive method that returns the number of digits in the integer passed to it as an argument of type int. Allow for both positive and negative arguments. For example, −120 has three digits. Do not count leading zeros. Embed the method in a program, and test it.

Notes:

A technique similar to that in `RecursionDemo2`, Listing 11.4, can be used for this Project First change the number to positive if it is negative. The base case is when the number has just one digit, which returns 1 if the result of the truncated division of the number by 10 is zero. If non-zero, a recursive call is made to the method, but with the original number reduced by one digit, and (1 + the value returned by the recursive call) is returned. In this fashion, each recursive call will add 1, but not until the base case is executed. The base case returns a 1 and the stacked calls can now "unwind," each call executing in turn and adding 1 to the total. The first call is the last to execute and, when it does, it returns the number of digits.

References:

Listing 11.4

Solution:

See the code in NumberOfDigitsDemo.java.

2. Write a static recursive method that returns the sum of the integers in the array of int values passed to it as a single argument. You can assume that every indexed variable of the array has a value. Embed the method in a test program.

Notes:

The insight for this problem is to realize that the array passed each iteration must be diminished by one element and the base case is when the passed array has just one element. In order to pass a diminished array, another, temporary, array must be created that is a copy of all but the highest-index value of the passed array. The return value should be the sum of the value at the highest-index of the passed array plus the return value from the call to `sumOfInts`.

Solution:

See the code in SumOfIntsDemo.java.

3. One of the most common examples of recursion is an algorithm to calculate the **factorial** of an integer. The notation *n*! is used for the factorial of the integer *n* and is defined as follows:
0! is equal to 1
1! is equal to 1
2! is equal to 2 _ 1 = 2
3! is equal to 3 _ 2 _ 1 = 6
4! is equal to 4 _ 3 _ 2 _ 1 = 24
. . .
n! is equal to *n* _ (*n* − 1) _ (*n* − 2) _ ... _ 3 _ 2 _ 1

An alternative way to describe the calculation of *n*! is the recursive formula *n* * (*n* − 1)!, plus a base case of 0!, which is 1. Write a static method that implements this recursive formula for factorials. Place the method in a test program that allows the user to enter values for *n* until signaling an end to execution.

Notes:

This problem is very easy to write as a recursive algorithm. The base case returns one for n = 0 or n = 1. All other cases multiply the number passed by the return value of a recursive call for the passed number minus one. Note that the program loops until the user enters a non-negative number. One word of caution: it is easy to enter a number that will result in a calculated value too large to store. An interesting little project would be to have the students find out what the largest integer value is for the platform they are using, then determine which values to enter to bracket the maximum value, and run the program to see what happens when the those values are entered.

Solution:

See the code in Factorial.java.

4. A common example of a recursive formula is one to compute the sum of the first n integers, $1 + 2 + 3 + \ldots + n$. The recursive formula can be expressed as $1+2+3+\ldots+n=n+(1+2+3+\ldots+(n-1))$
Write a static method that implements this recursive formula to compute the sum of the first n integers. Place the method in a test program that allows the user to enter the values of n until signaling an end to execution. Your method definition should not use a loop to add the first n integers.

Notes:

This Project is also very easy to write as a recursive algorithm. The base case returns one and any other case adds the number passed to it to the number returned by a recursive call with the number passed to it reduced by one. Note that the program loops until the user enters a positive integer since the progression is defined only for positive integers.

Solution:

See the code in ArithmeticProgression.java.

Programming Projects:

1. A **palindrome** is a string that reads the same forward and backward, such as "radar". Write a static recursive method that has one parameter of type String and returns true if the argument is a palindrome and false otherwise. Disregard spaces and punctuation marks in the string, and consider upper- and lowercase versions of the same letter to be equal. For example, the following strings should be considered palindromes by your method:
"Straw? No, too stupid a fad, I put soot on warts."
"xyzcZYx?"

Your method need not check that the string is a correct English phrase or word. Embed the method in a program, and test it.

Notes:

The algorithm for this Project is a bit tricky. The recursive algorithm leads to some inefficiency. For example, the problem statement asks for a method that takes a string with spaces and punctuation, but only looks at the letters and returns a Boolean value. So the method must parse the input string and eliminate everything but letters. Although the string needs to be parsed just once, a recursive algorithm must be identical each iteration, so the parsing occurs each iteration. The base case either returns TRUE if the string has zero or one characters, or, if the string has two or more characters, it checks the first and last characters and has a more complex algorithm to determine whether to return TRUE or FALSE. If the two letters (the first and last) are different it returns FALSE. But it the two are the same, it returns the result of a recursive call with a smaller string, with the first and last letters removed. So each iteration reduces the string by a pair of letters until the string is down to zero or one character (even or odd number of letters, respectively, in the original string). The base case returns TRUE and starts unraveling the stacked method calls. The base method always returns TRUE, but each `return` after that ANDs it with the Boolean result of the test for a pair of letters that must be the same if it is a palindrome. If any pair is not equal, a FALSE is ANDed and, by definition of the AND operation, the result will be FALSE. The method, after all iterations are done, will return TRUE only if every iteration returned TRUE (the letters in each pair were the same).

An alternative approach would be to write a second recursive method that looks at each character in the string recursively and appends it to the result if it is not space or punctuation.

Solution:

See the code in PalindromeTestDemo.java.

2. A **geometric progression** is defined as the product of the first *n* integers, and is denoted as
geometric(*n*) = *<formula omitted>*
where this notation means to multiply the integers from 1 to *n*. A **harmonic progression** is defined as the product of the inverses of the first *n* integers, and is denoted as
harmonic(*n*) = *<formula omitted>*
Both types of progression have an equivalent recursive definition:
geometric(*n*) = *<formula omitted>*
harmonic(*n*) = *<formula omitted>*
Write static methods that implement these recursive formulas to compute geometric(*n*) and harmonic(*n*). Do not forget to include a base case, which is not given in these formulas, but which you must determine. Place the methods in a test program that allows the user to compute both geometric(*n*) and harmonic(*n*) for an input integer *n*. Your program should allow the user to enter another value for *n* and repeat the calculation until signaling an end to the program. Neither of your methods should use a loop to multiply *n* numbers.

Notes:

This is another easy program to write as a recursive algorithm. One little detail is to avoid integer division truncation when calculating the harmonic progression by casting the numerator (1) in the division to `double`. Also note that it is easy to enter a value that will cause either an overflow for the geometric progression calculation or underflow for the harmonic progression calculation. Students should be made aware of these common pitfalls, especially because the system does not flag them as errors.

Solution:

See the code in GeometricAndHarmonicProgressions.java.

3. The **Fibonacci sequence** occurs frequently in nature as the growth rate for certain idealized animal populations. The sequence begins with 0 and 1, and each successive Fibonacci number is the sum of the two previous Fibonacci numbers. Hence, the first ten Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. The third number in the series is 0 + 1, which is 1; the fourth number is 1 + 1, which is 2; the fifth number is 1 + 2, which is 3; and so on.

Besides describing population growth, the sequence can be used to define the form of a spiral. In addition, the ratios of successive Fibonacci numbers in the sequence approach a constant, approximately 1.618, called the "golden mean." Humans find this ratio so aesthetically pleasing that it is often used to select the length and width ratios of rooms and postcards.

Use a recursive formula to define a static method to compute the *n*th Fibonacci number, given *n* as an argument. Your method should not use a loop to compute all the Fibonacci numbers up to the desired one, but should be a simple recursive method. Place this static recursive method in a program that demonstrates how the ratio of Fibonacci numbers converges. Your program will ask the user to specify how many Fibonacci numbers it should calculate. It will then display the Fibonacci numbers, one per line. After the first two lines, it will also display the ratio of the current and previous Fibonacci numbers on each line. (The initial ratios do not make sense.) The output should look something like the following if the user enters
5:
Fibonacci #1 = 0
Fibonacci #2 = 1
Fibonacci #3 = 1; 1/1 = 1
Fibonacci #4 = 2; 2/1 = 2
Fibonacci #5 = 3; 3/2 = 1.5

Notes:

The recursive algorithm for Fibonacci numbers is a little more involved than the series calculations in the previous Projects. Base cases for 0, 1 or two numbers simply return a value, and all other numbers make two recursive calls to get the previous two Fibonacci numbers to add together to obtain the current number. The method to calculate a Fibonacci number is recursive, but the code to print the output is not; it uses a for-loop to cycle through the Fibonacci numbers and ratios.

Solution:

See the code in Fibonacci.java.

4. Imagine a candy bar that has k places where it can be cut. You would like to know how many different sequences of cuts are possible to divide the bar into pieces. For example, if k is 3, you could cut the bar at location 1, then location 2, and finally at location 3. We indicate this sequence of cuts by 123. So if k is 3, we have six ways to divide the bar: 123, 132, 213, 231, 312, or 321. Notice that we have k possibilities for making the first cut. Once we make the first cut we have k - 1 places where a cut must be made. Recursively, this can be expressed as

$C(k) = k\ C(k - 1)$

Lets make this a bit more interesting by adding a restriction. You must always cut the leftmost pieces that can be cut. Now if k is 3, we can cut the bar at locations 123, 132, 213, 312, or 321. A cutting sequence of 231 would not be allowed, because after the cut at 2 we would have to make the cut at location 1, since it is the leftmost piece. We still have k possibilities for making the first cut, but now we have to count the number of ways to cut two pieces and multiply. Recursively, this can be expressed as

<formulas omitted>

Develop a program that will read a value of k from the keyboard and then display $C(k)$ and $D(k)$. ($D(k)$ is interesting because it turns out to be the number of ways that we can parenthesize an arithmetic expression that has k binary operators.)

Notes:

The recursive algorithm for C(k) is easy to implement and should be familiar. The recursive algorithm for D(k) is slightly more complicated, but can be based on C(k). Instead of making a single recursive call, loop and make the recursive calls. As with previous projects, this one can experience overflow if the input value k is too large.

Solution:

See the code in Cuts.java.

5. Once upon a time in a kingdom far away, the king hoarded food and the people starved. His adviser recommended that the food stores be used to help the people, but the king refused. One day a small group of rebels attempted to kill the king, but were stopped by the adviser. As a reward, the adviser was granted a gift by the king. The adviser asked for a few grains of wheat from the king's stores to be distributed to the people. The number of

grains were to be determined by placing them on a chessboard. On the first square of the chessboard, he placed one grain of wheat. He then placed two grains on the second square, four grains on the third square, eight grains on the fourth square, and so forth.
Compute the total number of grains of wheat that were placed on *k* squares by writing a recursive method getTotalGrains(k, grains). Each time getTotalGrains is called, it "places" grains on a single square; grains is the number of grains of wheat to place on that square. If *k* is 1, return grains. Otherwise, make a recursive call, where *k* is reduced by 1 and grains is doubled. The recursive call computes the total number of grains placed in the remaining *k* - 1 squares. To find the total number of grains for all *k* squares, add the result of the recursive call to grains and return that sum.

Notes:

This demonstrates a recursion where a partial solution is being built up on the way down the recursion. What makes this interesting is that it is not just a tail recursion (the partial solution becomes the complete solution at the bottom of the recursive chain), but that it computes an answer using each of the partial solutions.

Solution:

See the code in Grain.java.

6. There are n people in a room, where n is an integer greater than or equal to 2. Each person shakes hands once with every other person. What is the total number of handshakes in the room? Write a recursive method to solve this problem with the following header:

public static int handshake(int n)

where handshake(n) returns the total number of handshakes for n people in the room. To get you started, if there are only one or two people in the room, then:

handshake(1) = 0
handshake(2) = 1

Notes:

This is a short and relatively straightforward recursive problem.

Solution:

See the code in Handshake.java.

7. Given the definition of a 2D array such as the following:

```
String[][] data = {
                    {"A","B"},
                    {"1","2"},
                    {"XX","YY","ZZ"}
                  };
```

Write a recursive Java program that outputs all combinations of each subarray in order. In the above example the desired output (although it doesn't have to be in this order) is:

```
A 1 XX
A 1 YY
A 1 ZZ
A 2 XX
A 2 YY
A 2 ZZ
B 1 XX
B 1 YY
B 1 ZZ
B 2 XX
B 2 YY
B 2 ZZ
```

Your program should work with arbitrarily sized arrays in either dimension. For example, the following data:

```
String[][] data = {
                    {"A"},
                    {"1"},
                    {"2"},
                    {"XX","YY"}
                  };
```

Should output:

```
A 1 2 XX
A 1 2 YY
```

Notes:

This is a more complex recursive problem than the others in this chapter and has many possible solutions. The solution given here recursively fill in a oneline[] array that represents one entry of the product of sets. The array is filled in by iterating through each subarray for each recursive call. That is, the first recursive call iterates through the elements in data[0], the second recursive call iterates through the elements in data[1], etc.

Solution:

See the code in ArrayProduct.java.

10. Create an application in a JavaFX GUI that will draw a fractal curve using line segments. Fractals are recursively defined curves. The curve you will draw is based on a line segment between points *p*1 and *p*2: To draw the curve from *p*1 to *p*2, you first split the segment into thirds. Then add two segments and offset the middle segment to form part of a square, as shown in the following picture:
Note that you would not draw the arrowheads, but we use them here to indicate the direction of drawing. If the order of *p*1 and *p*2 were reversed, the square would be below the original line segment. This process is recursive and is applied to each of the five new line segments, resulting in the following curve: The fractal is given by repeating this recursive process an infinite number of times. Of course, we will not want to do that and will stop the process after a certain number of times.
To draw this curve, use a recursive method drawFractal(*p1x*, *p1y*, *p2x*, *p2y*, *k*). If *k* is zero, just draw a line from *p1* to *p2*. Otherwise, split the line segment into five segments, as described previously, and recursively call drawFractal for each of these five segments. Use *k* - 1 for the last argument in the recursive calls.
For convenience, you may assume that the segments are either vertical or horizontal. The initial call should be drawFractal(50, 800, 779, 800, 5). Set the size of the window to 1000 by 1000.

Notes:

The hardest part of this recursive algorithm is getting arguments of recursive calls for each segment correct. Part of the complication is that Java's coordinate system has y positive in the downward direction. One can work on the horizontal case first and then do the vertical. It is recommended that k=1 be used when developing the algorithm as well.

Solution:

See the code in Fractal.java.

Exercises:

1. Repeat Exercise 2 in Chapter 7, but use an instance of ArrayList instead of an array. Do not read the number of values, but continue to read values until the user enters a negative value.

Solution:

See the code in CountFamilies.java.

2. Repeat Exercise 3 in Chapter 7, but use an instance of ArrayList instead of an array. Do not read the number of families, but read data for families until the user enters the word done.

Solution:

See the code in Family.java and CountPoor.java.

3. Repeat Exercise 5 in Chapter 7, but use an instance of ArrayList instead of an array.

Solution:

See the code in CharacterFrequency.java.

4. Repeat Exercises 6 and 7 in Chapter 7, but use an instance of ArrayList instead of an array. We will no longer need to know the maximum number of sales, so the methods will change to reflect this.

Solution:

See the code in Ledger.java.

5. Write a static method removeDuplicates(ArrayList<Character> data) that will remove any duplicates of characters in the object data. Always keep the first copy of a character and remove subsequent ones.

Solution:

```
public static void removeDuplicates(ArrayList<Character> data){
    for(int i=0; i<data.size(); i++){
        int j = i+1;
        while(j<data.size()){
            if(data.get(i) == data.get(j))
                data.remove(j);
            else
                j++;
        }
    }
}
```

This code is in Methods.java.

6. Write a static method getCommonStrings(ArrayList<String> list1, ArrayList<String> list2) that returns a new instance of ArrayList containing all of the strings common to both list1 and list2.

Solution:

```
    public static ArrayList<String>
    getCommonStrings(ArrayList<String> list1,
                     ArrayList<String> list2){

        ArrayList<String> result = new ArrayList<String>();
        for(int i=0; i<list1.size(); i++){
            for(int j=0; j<list2.size(); j++){
                if(list1.get(i).equals(list2.get(j)))
                    result.add(list1.get(i));
            }
        }

        // Remove the duplicates
        for(int i=0; i<result.size(); i++){
            int j = i+1;
            while(j<result.size()){
                if(result.get(i) == result.get(j))
                    result.remove(j);
                else
                    j++;
            }
        }

        return result;
    }
```

This code is in Methods.java. An alternate solution is also given.

7. Write a program that will read sentences from a text file, placing each sentence in its own instance of ArrayList. (You will create a sentence object by adding words to it one at a time as they are read.) When a sentence has been completely read, add the sentence to another instance of ArrayList. Once you have read the entire file, you will have an instance of ArrayList that contains several instances of ArrayList, one for each sentence read. Now ask the user to enter a sentence number and a word number. Display the word that occurs in the given position. If the sentence number or word number is not valid, provide an appropriate error message.

Solution:

See the code in WordInFile.java.

8. Repeat Exercise 12 in Chapter 7, but use an instance of ArrayList instead of an array. Make the following slight changes to the methods to reflect that an ArrayList object can grow in size:

• Change the constructor's parameter from the maximum degree to the desired degree.

• The method setConstant might need to add zero-valued coefficients before a_i. For example, if $a_0 = 3$, $a_1 = 5$, $a_2 = 0$, $a_3 = 2$, $a_4 = 0$, and $a_5 = 0$, the polynomial would be of degree 3, since the last nonzero constant is a_3. The invocation setConstant(8, 15) would need to set a_6 and a_7 to 0 and a_8 to 15.

Solution:

See the code in Polynomial.java.

9. Write a program that will read a text file that contains an unknown number of movie review scores. Read the scores as Double values and put them in an instance of ArrayList. Compute the average score.

Solution:

See the code in MovieScores.java.

10. Revise the class StringLinkedList in Listing 12.5 so that it can add and

remove items from the end of the list.

Solution:

See the code in ListNode.java, StringLinkedList.java, StringLinkedListDemo.java.

11. Suppose we would like to create a data structure for holding numbers that can be accessed either in the order that they were added or in sorted order. We need nodes having two references. If you follow one trail of references, you get the items in the order they were added. If you follow the other trail of references, you get the items in numeric order. Create a class DualNode that would support such a data structure. Do not write the data structure itself.

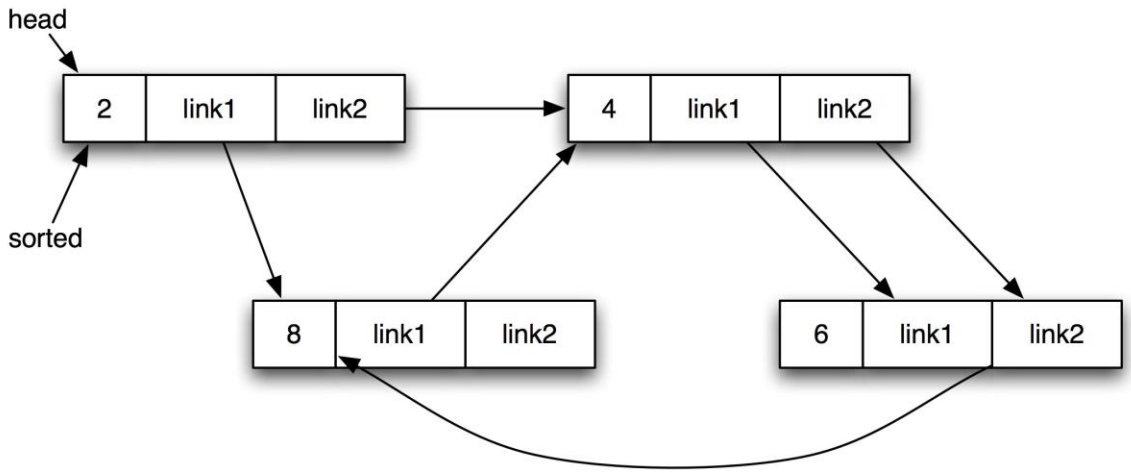
Solution:

See the code in DualNode.java.

12. Draw a picture of an initially empty data structure, as described in the previous exercise, after adding the numbers 2, 8, 4, and 6, in this order.

Solution:

Head and link1 gives the values in the order they were added. Sorted and link2 give the values in sorted order.



13. Write some code that will use an iterator to duplicate every item in an instance of StringLinkedListWithIterator in Listing 12.9. For example, if the list contains "a", "b", and "c", after the code runs, it will contain "a", "a", "b", "b", "c", and “c”.

Solution:

```
// Make sure the iterator is at the front of the list
    list.resetIteration();
    while(list.moreToIterate()){
        String data = list.getDataAtCurrent();
        list.insertNodeAfterCurrent(data);
        // Skip over the data just inserted and the next
        list.goToNext();
        list.goToNext();
    }

    System.out.println("Every value in the list should be”
       + " repeated now");
    list.showList();
```

This code is in StringLinkedIteratorFragments.java.

14. Write some code that will use an iterator to move the first item in an instance of StringLinkedListWithIterator (Listing 12.9) to the end of the list. For example, if the list contains "a", "b", "c", and "d", after the code runs, it will contain "b", "c", "d", and "a".

Solution:

```
    list.resetIteration();
        String data = list.getDataAtCurrent();
        list.deleteCurrentNode();
        // Iterate to the node at the end of the list
        for(int i=0; i<list.length()-1; i++){
            list.goToNext();
        }
        list.insertNodeAfterCurrent(data);

        System.out.println("First value should now be moved to the"
            + " end of the list");
        list.showList();
```

This code is in StringLinkedIteratorFragments.java.

15. Write some code that will use an iterator to interchange the items in every pair of items in an instance of StringLinkedListWithIterator in Listing 12.9. For example, if the list contains "a", "b", "c", "d", "e", and "f", after the code runs, it will contain "b", "a", "d", "c", "f", and "e". You can assume that the list contains an even number of strings.

Solution:

```
list.resetIteration();
    while(list.moreToIterate()){
        String toSwitch = list.getDataAtCurrent();
        list.deleteCurrentNode();
        list.insertNodeAfterCurrent(toSwitch);
        list.goToNext();
        list.goToNext();
    }
    System.out.println("Every pair of strings should have their"
        + " order swapped.");
    list.showList();
```

This code is in StringLinkedIteratorFragments.java.

17. Write a program that creates two instances of the generic class LinkedList given in Listing 12.12. The first instance is stadiumNames and will hold items of type String. The second instance is gameRevenue and will hold items of type Double. Within a loop, read data for the ball games played during a season. The data for a game consists of a stadium name and the amount of money made for that game. Add the game data to stadiumNames and gameRevenue. Since more than one game could be played at a particular stadium, stadiumNames might have duplicate entries. After reading the data for all of the games, read a stadium name and display the total amount of money made for all the games at that stadium.

Solution:

See the code in Revenue.java.

Practice Programs:

2. Repeat the previous practice program, but instead write a method bubbleSort that performs a bubble sort, as described in Programming Project 3 of Chapter 7.

Notes: An easy way to develop this program is to start with the code to bubble sort an array from Chapter 7 Programming Project 3. Change the array references to list syntax. If the code for bubble sorting an array is not available, then it is a matter of making modifications to `StringSelectionSort` to implement the bubble sort algorithm instead of selection sort.
References: Project 7.3
Solution: *See the code in StringBubbleSort.java and StringBubbleSortDemo.java.*

3. Repeat Practice Program 1, but instead write a method insertionSort that performs an insertion sort, as described in Programming Project 4 of Chapter 7.

Notes: As with the previous project, an easy way to develop the program is to start with the insertion sort code for an array, Chapter 7 Programming Project 4. Note, however, that the list code, unlike that for the array, does not need a helper method to move the elements when a new element is added to the temporary list; the `ArrayList` class has a method, `add(int, T)`, that does it for you.
References: Project 7.4
Solution: *See the code in StringInsertionSort.java and StringInsertionSortDemo.java.*

5. Write a program that uses a HashMap to compute a histogram of positive numbers entered by the user. The HashMap's key should be the number that is entered, and the value should be a counter of the number of times the key has been entered so far. Use -1 as a sentinel value to signal the end of user input. For example, if the user inputs:
5
12
3
5
5
3
21
-1
Then the program should output the following (not necessarily in this order):
The number 3 occurs 2 times.
The number 5 occurs 3 times.
The number 12 occurs 1 times.
The number 21 occurs 1 times.

Notes:

The solution is a fairly straightforward implementation of a HashMap<Integer,Integer> to map a number to a counter. It may be worth mentioning that equals and hashcode must be overwritten if the student wants to use HashMap with their own set. Note that Java will automatically unbox/box between an int and Integer. This solution takes advantage of this feature.

Solution:

See the code in HashMapHistogram.java.

Programming Projects:

1. Write a program that creates Pet objects from data read from the keyboard. Store these objects into an instance of ArrayList. Then sort the Pet objects into alphabetic order by pet name, and finally display the data in the sorted Pet objects on the screen. The class Pet is given in Chapter 6, Listing 6.1.

Notes:

This Project has the potential for being more difficult than the Practice Programs since it uses a list of elements other than `Strings`, which the author states can lead to subtle problems. The `PetRecord` variable `nextPet` must be declared inside the while-loop so it is created fresh each iteration. It must be redeclared each time to ensure that `PetRecords` previously added to the list are not overwritten with data entered on the next iteration. If `nextPet` is created only once, outside the while-loop (before entering it), the new values entered are written to same address each time and all the `PetRecords` in the list will end up with identical data, the values entered in the last iteration.

Solution:

See the code in PetRecordsSortedByName.java. Uses PetRecord.java.

2. Repeat the previous programming project, but sort the Pet objects by pet weight instead of by name. After displaying the sorted data on the screen, write the number and percentage of pets that are under 5 pounds, the number and percentage of pets that are 5 to 10 pounds, and the number and percentage of pets that are over 10 pounds.

Notes: This project is related to the next project. It does the same task, but reads/writes pet records from/to a file.

3. Repeat the previous programming project, but read the input data from a file and send the output to another file. If you have covered binary files, use binary files; otherwise, use text files. Read the file names from the user.

Notes: This Project can be done in many different ways. The solution shown here reads records from the user-specified input file into a list, then sorts the elements by the pets' weights using bubble sort, and also sends the output to a file.
References: Listing 6.1, Project 10.2
Solution: *See the code in PetRecordsSortedByName.java. Uses PetRecord.java and PetFileReadOrWrite.java.*

4. Use the class ClassObjectIODemo shown in Listing 10.10 of Chapter 10 to create a file of Species objects. The class Species is given in Chapter 10, Listing 10.9. Then write a program that reads the Species objects from the file you created into an instance of ArrayList, sorts these instances alphabetically by Species name, and then writes the sorted data to both the screen and a file. Read all file names from the user.

Notes:

This Project can be done in many different ways. The solution shown here reads records from the user-specified input file into a list, then sorts the elements alphabetically by name using bubble sort. Note that `Species` has methods specifically for reading records from and writing records to a file, so that part of the program is especially easy to write.

References:

Listing 10.9, Listing 10.10

Solution:

See the code in SpeciesSortedByNameToFile.java. Uses Species.java, WriteSpeciesFile.java, and DisplaySpeciesFile.java.

5. Define a variation on StringLinkedListSelfContained from Listing 12.7 that stores objects of type Species, rather than of type String. Write a program that uses that linked-list class to create a linked list of Species objects, asks the user to enter a Species name, and then searches the linked list and displays one of the following messages, depending on whether the name is or is not on the list:
Species *Species_Name* is one of the
Number_Of_Species_Names_On_List species on the list.
The data for *Species_Name* is as follows:
Data_For_Species_Name
or
Species *Species_Name* is not a species on the list.

The user can enter more Species names until indicating an end to the program. The class Species is given in Listing 5.19 of Chapter 5. (If you prefer, you can use the serialized version of Species in Listing 10.9 of Chapter 10.)

Notes:

This is closely related to the solution of the next project which reads and writes to a file.

6. Repeat the previous programming project, but read the input data from a file and send the output to another file. If you have covered binary files, use binary files; otherwise, use text files. Read the file names from the user.

Notes: The solution shown here gets `Species` records from a file rather than the keyboard.
References: Listing 12.7
Solution: *See the code in SpeciesLinkedListSelfContained.java, and SpeciesLinkedListSearch.java. Uses Species.java, WriteSpeciesFile.java, and DisplaySpeciesFile.java*

7. Define a variation on StringLinkedListSelfContained from Listing 12.7 that stores objects of type Employee, rather than objects of type String. Write a program that uses this linked-list class to create a linked list of Employee objects, asks the user to enter an employee's social security number, and then searches the linked list and displays the data for the corresponding employee. If no such employee exists, display a message that says so. The user can enter more social security numbers until indicating an end to the program. The class Employee is described in Programming Project 6 of Chapter 9. If you have not already done that project, you will need to define the class Employee as described there.

Notes: This project is closely related to the next one, which uses a file.

8. Repeat the previous programming project, but read the input data from a file and send the output to another file. If you have covered binary files, use binary files; otherwise, use text files. Read the file names from the user.

Notes: This Project is complicated by the absence of a program to read and write files with employee records, so one has to be created. Note that the employee class from Chapter 9 must be modified to implement Serializable and the resulting class is renamed EmployeeCh12.java. Also, note that the program gets records from a file and not the keyboard.
References: Project 9.6, Listing 12.7
Solution: *See the code in EmployeeCh12.java, EmployeeCh12FileReadOrWrite,java, EmployeeCh12LinkNode.java, EmployeeCh12LinkedListSelfContained.java, EmployeeCh12LinkedListSearch.java. Uses Person.java,*

9. Write a parameterized class definition for a doubly linked list that has a parameter for the type of data stored in a node. Make the node class an inner class. Choosing which methods to define is part of this project. Also, write a program to thoroughly test your class definition.

Notes:

This program is based on code from Listing 12.9. The solution given here includes support for iteration, both forward and reverse. Both forward and reverse iteration use the `moreToIterate` method, but backward iteration uses the `resetIterationReverse` rather than `resetIteration`. The `ListNode` class includes a `previous` instance variable which makes the `previous` variable in the outer class unnecessary. To make reverse iteration (especially `resetIterationReverse`) easier to write, there is a reference to the tail of the list as well as to the head of the list. An additional method, `findInList`, looks for an element in the list and sets `current` to point to that element if it is found. If it is not found, `current` is set to null. The method `showListState` is for testing and debugging purposes and prints the head of the list, the current element, the tail, and the number of elements in the list.

References:

Listing 12.9

Solution:

See the code in DoublyLinkedList.java, and DoublyLinkedListDemo.java.

10. Create an application that will keep track of several groups of strings. Each string will be a member of exactly one group. We would like to be able to see whether two strings are in the same group as well as perform a union of two groups. Use a linked structure to represent a group of strings. Each node in the structure contains a string and a reference to another node in the group. For example, the group {"a", "b", "d", "e"} is represented by the following structure: One string in each group—"d" in our example—is in a node that has a null reference. That is, it does not reference any other node in the structure. This string is the **representative string** of the group.

Create the class GroupHolder to represent all of the groups and to perform operations on them. It should have the private instance variable items to hold the nodes that belong to all of the groups. The nodes within each group are linked together as described previously. Make items an instance of ArrayList whose base type is GroupNode, where GroupNode is a private inner class of GroupHolder. GroupNode has the following private instance variables:

• data—a string
• link—a reference to another node in the group, or null

Define the following methods in the class GroupHolder:

• addItem(s)— adds a string s to an empty group. First search items for s; if you find s, do nothing; if you do not find s, create a new GroupNode object that has s as its string and null as its link and add it to items. The new group will contain only the item s.
• getRepresentative(s)—returns the representative string for the group containing s. To find the representative string, search items for s. If you do not find s, return null. If you find s, follow links until you find a null reference. The string in that node is the representative string for the group.
• getAllRepresentatives—returns an instance of ArrayList that contains the representative strings of all the groups in this instance of GroupHolder. (A representative string is in an instance of GroupNode that contains a null reference.)
• inSameGroup(s1, s2)—returns true if the representative string for s1 and the representative string for s2 are the same and not null, in which case the strings s1 and s2 are in the same group.
• union(s1, s2)—forms the union of the groups to which s1 and s2 belong.

Hint: Find the representative strings for s1 and s2. If they are different and neither is null, make the link of the node containing s1's representative string reference the node for s2's representative string.

For example, suppose that we call addItem with each of the following strings as an argument: "a", "b", "c", "d", "e", "f", "g", and "h". Next, let's form

groups by using these union operations:
union("a", "d"), union("b", "d"), union("e", "b"),
union("h", "f")
We will have four groups—{"a", "b", "d", "e"}, {"c"}, {"f", "h"}, and {"g"}—represented by the following structure:
The representative strings for these groups are "d", "c", "f", and "g", respectively.

Now the operation inSameSet("a", "e") would return true because both getRepresentative("a") and getRepresentative("e") return d. Also, inSameSet("a", "f") would return false because getRepresentative("a") returns d, and getRepresentative("f") returns f. The operation union("a", "f") would make the node containing the representative string of the group to which "a" belongs—which is "d"—reference the node containing the representative string of the group to which "f" belongs, which is "f". This reference would be represented by an arrow from "d" to "f" in the previous diagram. Your application should create an instance of GroupHolder and allow the user to add an arbitrary number of strings, each to its own group. It should then perform an arbitrary number of union operations to form several groups. Finally, it should demonstrate the other operations.

Notes:

This program implements a simplified version of a linked structure that can be used to detect cycles while building a minimum spanning tree. To really be efficient, we should use a hash table instead of an ArrayList to store the elements. We should also keep track of the height of our trees and always link the shorter tree to the taller tree.

This implementation uses one private method to determine if the element is one that is in our group. Otherwise the implementation is fairly straightforward. The method GetRepresentative follows pointers until it finds null. Union will make a link as long as both arguments are in the group and are not in the same set. Notice that once a link is made, we never need to change it.

Solution:

See the code in GroupHolder.java.

11. For this project, we will create a data structure known as a **queue**. A queue can be thought of as a line. Items are added at the end of the line and are taken from the front of the line. You will create a class LinkedQueue based on one of the linkedlist classes given in this chapter. It should have private attributes for

• front—a reference to the first node in the linked list of queue items
• count—the number of items in the queue

and the following operations:

• addToQueue(item)—adds item to the end of the queue. (Add it at the end of the linked list.)
• removeFromQueue()—removes the first item from the queue and returns it. If the queue is empty, returns null.
• isEmpty—returns true if the queue contains no items; otherwise, returns false.

Create a program that demonstrates the functions of the LinkedQueue class.

Notes: This program is just a simple modification of the existing programs in this chapter. The solution is loosely based on the generic version of the linked list given in Listing 12.12.
References: Listing 12.12
Solution: *See the code in LinkedQueue.java, LinkedQueueDemo.java.*

13. Suppose that we would like to perform a bird survey to count the number of birds of each species in an area. Create a class BirdSurvey that is like one of the linked-list classes given in this chapter. (The linked list you use will affect what your new class can do, so give some thought to your choice.) Modify the inner node class to add room for a count.
BirdSurvey should have the following operations:
• add(bird)—adds the bird species bird to the end of the list, if it is not already there, and sets its count to 1; otherwise, adds 1 to the count for bird.
• getCount(bird)—returns the count associated with the species bird. If bird is not on the list, returns zero.
• getReport—displays the name and count for each bird species on the list.

Write a program that uses BirdSurvey to record the data from a recent bird survey. Use a loop to read bird names until done is entered. Display a report when finished.

Notes: The solution is loosely based on the generic version of the linked list given in Listing 12.12. The definition of node needs to be changed to include an extra piece of data (the count). The add method must be modified to do a search first and increment the count if the bird is already in the list. Notice that once a link is set, it never needs to be changed.
References: Listing 12.12
Solution: *See the code in BirdSurvey.java.*

14. Consider a text file of names, with one name per line, that has been compiled from several different sources. A sample is shown below:

Brooke Trout
Dinah Soars
Jed Dye
Brooke Trout
Jed Dye
Paige Turner

There are duplicate names in the file. We would like to generate an invitation list but don't want to send multiple invitations to the same person. Write a program that eliminates the duplicate names by using a HashSet. Read each name from the file, add it to the HashSet, and then output all names in the HashSet to generate the invitation list without duplicates.

Notes: The solution is a fairly straightforward implementation of a HashSet<String> that holds each name and we loop through the file checking to see if a name is in the set before adding it. It may be worth mentioning that equals and hashcode must be overwritten if the student wants to use HashSet with their own set.
Solution: *See the code in InviteList.java and data in names.txt*

15. You have a list of student ID's followed by the course number (separated by a space) that the student is enrolled in. The listing is in no particular order. For example, if student 1 is in CS100 and CS200 while student 2 is in CS105 and MATH210 then the list might look like this:

1 CS100
2 MATH210
2 CS105
1 CS200

Write a program that reads data in this format from the console. If the ID is -1 then stop inputting data. Use the `HashMap` class to map from an `Integer`

(the student ID) to an `ArrayList` of type `String` that holds each class that the student is enrolled in. The declaration should look like this:

```
HashMap<Integer, ArrayList<String>> students =    new
  HashMap<Integer, ArrayList<String>>();
```

After all data is input, iterate through the map and output the student ID and all classes stored in the vector for that student. The result should be a list of classes organized by student ID.

Notes: This solution uses an ArrayList as the data value for a HashMap.
Solution: *See the code in HashMapStudentIDs.java*

Exercises:

1. Write a GUI application that creates two windows. In the first window, display the label Where is John?, and in the second window, display John is water skiing. *Hint*: Use a single class for both windows, but give an argument to the constructor that determines the string displayed in the label.

Solution:

See the code in Exercise1.java, LabelDisplayWindow.java.

2. Write a GUI application that creates three windows. Each window should be a different color, and each title should match the color. Use the colors MAGENTA, ORANGE, and GREEN.

Solution:

See the code in Exercise2.java, ColorWindow.java.

3. Write a GUI application that creates a single window, using a border layout. Place the following five labels in the window: Northern Location, Southern Location, Western Location, Eastern Location, and Central Location. Put each label in the appropriate place.

Solution:

See the code in Exercise3.java, BorderDisplayWindow.java.

4. Write a GUI application that creates a single window, using a flow layout. Place the following seven labels in the window: Location one, Location two, Location three, Location four, Location five, Location six, and Location seven. Add each label in numeric order.

Solution:

See the code in Exercise4.java, FlowDisplayWindow.java.

5. Repeat the previous exercise, using a 2 by 2 grid layout instead of a flow layout.

Solution:

See the code in Exercise5.java, GridDisplayWindow.java.

6. Write a GUI application that creates a single window containing one button— whose label is Change— in the north position. Set the background color to red initially. Each time a user clicks the button, change the background color from red to white, from white to blue, or from blue to red.

Solution:

See the code in Exercise6.java, ColorChangingWindow.java.

7. Suppose we want to write a stopwatch application that has a GUI. As a start, we will write stubs that create buttons for the application. These buttons will simply indicate which one was pressed, but will not cause any other actions. Create a GUI application that has a single window, three buttons—Start, Stop, and Reset—and one label. When Start is pressed, change the foreground color of the label to green and its text to Start was pressed. When Stop is pressed, change the foreground color of the label to red and its text to Stop was pressed. When Reset is pressed, change the foreground color of the label to orange and its text to Reset was pressed.

Solution:

See the code in Exercise7.java, StopWatchWindow.java.

8. Write an application that models a telephone keypad. Use a JPanel panel to hold twelve buttons—1, 2, 3, 4, 5, 6, 7, 8, 9, *, 0, #—in the center of a grid layout. Place a label in the south locations. As each number is pressed, append that digit to the text of the label.

Solution:

See the code in Exercise8.java, KeypadWindow.java.

9. Write an application that creates a substitution code. You will need a text area in the center of the screen that cannot be edited. The text area will display the code, using a format like A->C, B->Q, C->F, ... This means that *A* would be replaced by *C*, *B* would be replaced by *Q*, *C* would be replaced by *F*, and so on. The code will be generated one letter at a time, starting with *A* and ending with *Z*. You will have 26 buttons, each labeled with a letter of the alphabet. The first button pressed indicates the letter to be substituted for *A*, the second button pressed indicates the letter to be substituted for B, and so forth. For example, to create the previous substitution code, you would press the buttons *C*, *Q*, *F*, and so on. As each button is pressed, add the code to the text area and make the button invisible.

Solution:

See the code in Exercise9.java, CodeWindow.java.

10. Write an application that creates a list of names. You will need a text area in the center of the screen capable of holding ten lines that cannot be edited. Place a text field and an Accept button in the south position. After a user enters a name into the text field and clicks the Accept button, take the name from the text field and add it to the text area. Then clear the text field.

Solution:

See the code in Exercise10.java, NamesWindow.java.

11. Write a small application with a GUI that could be the basis of a larger application. Your application should accept a credit card number entered into a text field. When the user clicks an Accept button, you should check whether the number entered contains exactly 16 digits. If so, display the message Number accepted: as well as the card number in a label, and then clear the text field. If not, display the message Number rejected in the label. (Note: Credit card numbers have fancier format requirements that depend on the issuer of the card and are beyond the scope of this exercise.)

Solution:

See the code in Exercise11.java, CreditCardWindow.java.

12. Write a small application with a GUI that could be the basis of a larger application. Your application will allow someone to enter a user name and password into separate text fields. When the user clicks the Login button, you should check whether the string in the name field matches "John" and the string in the password field matches "myPassword". If both match, put the message You have been logged in. on a result label. If they don't match, put the message Sorry, that password is not valid. on the result label.

Solution:

See the code in Exercise12.java, LoginWindow.java.

13. Write a small application with a GUI that could be the basis of a larger application. Your application will allow someone to change a password. The user will enter the new password into each of two text fields and then press a Change button. You should check whether the strings in the two fields match. If they both match, display the message Your password has been updated. on a result label and make the button invisible. If they don't match, display the message Sorry, the passwords do not match each other. on the result label.

Solution:

See the code in Exercise13.java, PasswordChangeWindow.java.

14. Write a small application with a GUI that could be the basis of a larger application. Your application will ask the user three questions. You will need three labels for the questions and three corresponding text fields for the user's answers. When the user presses an Accept button, you should check whether each text area contains a nonempty string. If each does, display the message Your answers have been recorded. on a result label. If not, change the color of the label holding the question for the blank text field to red and display the message You must answer all questions on the result label.

Solution:

See the code in Exercise14.java, QuestionaireWindow.java.

15. Write a small application with a GUI that could be the basis of a larger application. Your application will accept a string indicating the size of a garment. The valid sizes are S, M, L, XL, and XXL. When the user enters a size into a text field and presses an Accept button, you should check whether the size is one of the valid sizes. If the size is valid, display the message Size accepted: as well as the size in a label and then clear the text field. If not, display the message Size rejected: in the label.

Solution:

See the code in Exercise15.java, SizeWindow.java.

16. Write a small application with a GUI that could be the basis of a larger application. Your application will ask whether the user is over 16 years of age. Display the text Are you at least 16 years old? in a label. If the user clicks the Yes button, display the message The user is 16 years old. in a result label. Otherwise, display The user is not yet 16 years old. in the result label. In both cases, make the two buttons invisible.

Solution:

See the code in Exercise16.java, SizeWindow.java.

Projects:

1. Rewrite the program in Listing 13.9 so that the panel with the buttons changes to pink when the larger panel turns red. Similarly, when the larger panel turns green, the panel with the buttons changes to blue. Also add a label to the larger panel that says Watch this panel!, and add a button to the button panel that is labeled Change. When the Change button is clicked, the colors change (from pink and red to blue and green, respectively, or vice versa). The Change button has no effect on the initial configuration, in which the big panel is blue and the button panel is gray.

Notes: This program can be written by making a few changes to `PanelDemo`, Listing 13.9. One little extra detail is that `buttonPanel` must be visible in the `actionPerformed` method, so it is declared outside the constructor block. A character flag is used to keep track of the background color settings so the click-on-**Change** event can determine the new background colors (and not change them if they are the initial colors).
References: Listing 13.2, Listing 13.9
Solution: *See the code in PanelDemo2.java. Uses WindowDestroyer.java.*

2. Rewrite the program in Listing 13.10 so that it has all of the following changes:

• The class name is MemoSaver2.

• There are six buttons instead of five at the bottom of the GUI. They are arranged as follows*:* <*diagram omitted*>

Hint: Use the GridLayout manager on the button panel.

• When the user saves the text as the first memo, the text area changes so that it says Memo 1 saved, and when the second memo is saved, the text area changes to say Memo 2 saved. (See Self-Test Question 43 for a hint.)

• When the Exit button is clicked, the program ends, and the window goes away. The close-window button also ends the program. So the Exit button and the close-window button perform the same action.

• In addition to the default constructor, another constructor produces the same display, except that the text area can accommodate a given number of lines and characters per line. This constructor has the following form:

public MemoSaver2(int lineCount, int charCount)

• The text area has line wrap, so that if more characters are entered than will fit on the line, the extra characters automatically go on the next line.

• The method main constructs two windows, one using the default constructor and one using the added constructor with arguments 5 and 60, in that order.

Notes:

This program can be created by modifying `MemoSaver.java`, Listing 13.10, but there is one potential problem with the display. Depending on the resolution of the user's screen, the window size may be too small to show the complete text panel. If that happens, the first few characters typed into the text box will not be visible. This is a little disconcerting; unless enough characters are typed it will appear that the text box is not letting the user enter anything. Increasing the width of the window from 600 should fix the problem (changing it to 700 works for a screen resolution of 1024 by 768).

References:

Listing 13.10

Solution:

See the code in MemoSaver2.java. Uses WindowDestroyer.java.

3. (You should do Programming Project 2 before doing this one.) Write a GUI, using Swing, that behaves as follows: When the program is run, a window appears and asks the user for the desired number of lines and characters per line for the memo saver. If the user clicks the close-window button, the program ends. More typically, the user enters these two numbers in two text fields. A Continue button is available that, if clicked, causes the window to disappear and another window to open. This second window is just like the memo saver in Programming Project 2, except that the text area has the number of lines and characters per line specified by the user in the previous window.

Notes:

This Project requires a few new techniques in addition to those in Project 2. `JTextField` is used for the text input boxes (instead of `JTextArea`) since the input will be on a single line, the original window must be hidden when the memo window is created, and consideration must be given to what happens if nothing is entered in the text boxes when "Continue" is clicked. Note that the window size may need to be adjusted, depending on the monitor resolution, to show all the window's objects completely. Also note that the solution does *not* include code to detect errors in data entry by catching `NumberFormatException` exceptions.

References:

Project 13.2, Listing 13.10

Solution:

See the code in MemoSaver3.java. Uses WindowDestroyer.java.

4. (The Swing part of this project is quite straightforward, but you do need to know a little about how to convert numbers from one base to another.) Write a program that converts numbers from base-10 (ordinary decimal) notation to base-2 notation. The program uses Swing to perform input and output via a window interface. The user enters a base-10 number in one text field and clicks a Convert button. The equivalent base-2 number then appears in another text field. Be sure that the two text fields are labeled. Include a Clear button that clears both text fields when clicked. Also be sure that the close-window button works correctly.

Notes:

This Project has some interesting challenges even if the student knows how to convert from decimal to binary. The solution shown here is designed to work for non-negative decimal integers only and uses the successive division by 2 algorithm, which is probably the easiest to program. However, there are several details that need special attention. For example, what if the user clicks the "Convert" button before anything is entered in the decimal text field? And what if the user does not enter a valid non-negative decimal number? The solution shown here displays a message in the binary number field for the first case, but does not check the decimal field input for incorrect input (a better implementation would detect errors in data entry by catching `NumberFormatException` exceptions and would also print an error message if the number entered was not a non-negative integer). The algorithm used to convert decimal to binary by successive division is as follows:

```
quotient = decimal number
while(quotient is not zero)
    next binary digit = remainder(decimal number/2)
    quotient = integer part of (quotient/2) //Throw away the fractional part.
```

Each iteration of the while loop produces one binary digit. The first iteration produces the least significant bit and each successive iteration produces the next higher significant bit, with the most significant bit produced in the last iteration. The loop ends when the quotient is zero, since all divisions after that will just add leading zeros. Note that the last digit is, necessarily, 1. The algorithm is simple to program with the modulo operator (%) and truncating integer division, but, unfortunately, the bits are obtained in the reverse order we need to print (the most significant bit must be printed first but it is the last obtained). The approach taken in the solution shown here puts the bit values in a character array, one at a time, from least significant bit to most significant bit, which makes the conversion process easy to code. Then the character array is read in reverse order, appending each character to a String so it will print in the correct order, with most significant bit on the left

Solution:

See the code in DecimalToBinary.java. Uses WindowDestroyer.java.

5. (It would probably help to do Programming Project 4 before doing this one.) Write a program that converts numbers from base-2 notation to base-10 (ordinary decimal) notation. The program uses Swing to perform input and output via a window interface. The user enters a base-2 number in one text field and clicks a Convert button. The equivalent base-10 number then appears in another text field. Be sure that the two text fields are labeled. Include a Clear button that clears both text fields when clicked. Also, be sure that the close-window button works correctly.
Hint: Include a private method that converts the string for a base-2 number to an equivalent int value.

Notes:

Positional weighting is used to convert from binary to decimal; the binary string is processed one character at a time and the weight for that position added to a running total until all digits have been processed, so the final value is the decimal equivalent of the binary number. Just as with the conversion from decimal to binary in the previous Project, a slight complication arises from the order of the bits. The first binary digit is the most significant digit, so that weight must be determined first and the weight of each successive digit must be reduced by a factor of 2.

Solution:

See the code in BinaryToDecimal.java. Uses WindowDestroyer.java.

6. (It would help to do Programming Projects 4 and 5 before doing this one.) Write a program that converts numbers from base-2 notation to base-10 (ordinary decimal) notation and vice versa. The program uses Swing to perform input and output via a window interface. There are two text fields—one for base-2 numbers and one for base-10 numbers—and three buttons labeled To Base 10, To Base 2, and Clear. If the user enters a base-2 number in the base-2 text field and clicks the To Base 10 button, the equivalent base-10 number appears in the base 10 text field. Similarly, if the user enters a base-10 number in the base-10 text field and clicks the To Base 2 button, the equivalent base-2 number appears in the base-2 text field. Be sure that the two text fields are labeled. The Clear button should clear both text fields when clicked. Also be sure that the close-window button works correctly.

Notes: This program is easily obtained by inserting the action listener code from Project 4 into the program for Project 5 and changing the labels.
References: Project 13.4, Project 13.5
Solution: *See the code in BinaryDecimalConverter.java. Uses WindowDestroyer.java.*

7. Write a program that produces a GUI with the functionality and look of a handheld calculator. Your calculator should allow for addition, subtraction, multiplication, and division. It should allow you to save and later recall two different values. Use the program in Listing 13.13 as a model. If you have not studied Listing 13.13, use Listing 13.12 as a model.

Notes: It may take some trial and error to get the additional buttons to fit into the window and look nice. If all the buttons do not show up, try resizing the window.
References: Listing 13.12
Solution: *See the code in Calculator.java. Uses WindowDestroyer.java.*

9. In Exercise 8, you created an application that modeled a telephone keypad. We would like to improve the operation of the application. Here is a list of the improvements to be made:
• The first number in the number cannot be 0. If the user types 0 as the first number, do nothing with it.
• Format the number using dashes, as follows:
• 1-000-000-0000 if the first digit entered is a 1.
• (000) 000-0000 if ten digits are entered.
• 000-0000 if seven digits are entered.
• Do not accept extra digits.

Notes: This application is pretty simple for the most part. A number of buttons must be created and multi-way switch used to detect which button on the keypad was pressed. But if we set an integer value, the rest of the code is simple. Create a method that checks to see if the digit entered is valid. Create methods that format the number in the different ways.
References: Exercise 13.8
Solution: *See the code in TelephoneKeypad.java..*

10. Write an application called Scramble that has a GUI to play a game of word anagrams. Create two arrays of strings. The first array will hold words, and the second will hold scrambled versions of those words. Your Java code can initialize these arrays directly with the words. Display the scrambled version of the word in a label. The user will enter a guess for the word in a text field and press a Check button, You should see whether the guess is correct. If it is not correct, change the guess in the text field to Sorry, that is incorrect. Please try again. If the guess is correct, change it to "That is correct. Here is a new word to try." And display a new scramble. Also provide a Give Up button. If it is pressed, display the unscrambled word and provide a new scrambled word.

Here are some extensions that can be made to improve this application:
• Read the words from a file.
• Do not use a second array of scrambled words, but instead use Java's random number generator to swap letters in the word just before you display the scrambled word.
• Randomly decide which word to display.
• Keep a score. Award 5 points if the user gets the word on the first guess, 3 points for getting it on the second guess, or 1 point for getting it on the third guess.
Divide the total points scored by the number of words presented.

Notes:

This application keeps the possible words in an array and sets it directly within the code. It also has a couple private variables to remember the word it is on and the number of guesses made. It has a method to scramble the word. There are a number of different ways the scrambling could be done. This implementation repeatedly removes a random character from the word and adds that character to a result string.

Solution:

See the code in Scramble.java.

11. Write an application with a GUI that will convert numbers from binary to octal. Binary numbers are composed of just the digits 0 and 1. Octal numbers use the digits 0, 1, 2, 3, 4, 5, 6, and 7. Note that each octal digit corresponds to a three-bit binary number, as follows:

Binary Octal
000 0
001 1
010 2
011 3
100 4
101 5
110 6
111 7

To convert a number from binary to octal, first group the bits in the binary number into sets of three and then apply the equivalent octal numbers. For example, the binary number 001000101110 would be grouped as 001 000 101 110, which corresponds to the octal digits 1, 0, 5, and 6, respectively, Thus, the octal equivalent of the binary number is 1056. If the number of bits in the binary number is not divisible by 3, add zeros at its beginning until it is. For example, since the binary number 1011011100100 has 13 bits, we would add two zeros before it to get 0010110111100100. We would then group its bits as 001 011 011 100 100 and get 13344 as its octal equivalent.

To convert an octal number to binary, we use our correspondence table in the reverse direction. For example, the octal number 716 is 111 001 110, or 111 001 110 in binary. Your application can omit the spaces we use to show the grouping of the bits.

You will need a text field for the user to enter a number and a label for the results. Provide three buttons: To Octal, To Binary, and Clear. If the user clicks the Clear button, clear any text in the text field and label. If the user clicks one of the two conversion buttons, check whether the number in the text field is in the correct format. If it is not, display the message Bad

Format in the results label. If the format is ok, compute the converted value and display that in the results label.

Notes:

This application does a conversion between binary and octal. Unlike the binary to decimal conversion, this can be done without any mathematics. We just need to have a translation table between octal digits to strings of binary bits. One complication is that when we translate from binary, we want to have groups of 3 bits. If the length of the string of binary bits is not divisible by 3, we will add zeros to the front. A couple methods are used to verify that the input strings are valid. Methods were also created for the conversions.

Solution:

See the code in BinaryOctalConverter.java.

Exercises:

1. Create an applet that will display the following logo in its upper left corner. *Hint*: Draw the logo in the paint method, but remember to call super.paint().

Solution:

See the code in LogoApplet.java.

2. Create an applet that will draw a starlike figure composed of a given number of lines. For example, if the number of lines is 8, you would draw a figure like this: Use a text field to get a number from the user. Then draw the star when the user clicks a Draw button. *Hint*: Draw the star in the paint method, but remember to call super.paint(). You should also call repaint in the actionPerformed method that listens for the button press.

Solution:

See the code in StarFigureApplet.java.

3. Create an applet that will draw a number of strings based upon an initial string specified by the user. If the string is *n* characters long, draw *n* strings as follows: The first string is the original one that the user typed in a text field. Each subsequent string removes the last character from the previous string. For example, if the user enters the string "Charles", the applet would draw
Charles
Charle
Charl
Char
Cha
Ch
C
Use a text field to get a string from the user. Then draw the strings when the user clicks a Draw button. *Hint*: Draw the strings in the paint method, but remember to call super.paint(). You should also call repaint in the actionPerformed method that listens for the button press.

Solution:

See the code in StringReducerApplet.java.

4. Create an applet that will convert lengths given in feet to lengths in meters (1 foot = 0.3048 meters). The applet should have a text field for the user to enter a length in feet. It also should have a Convert button that the user clicks after entering a length. Display the result in a scrollable text area, and put each new result on a separate line.

Solution:

See the code in FeetToMetersApplet.java.

5. Create an applet that will calculate an adult's body mass index (BMI). The formula for BMI is
703 _ *Weight* / *Height*2 where *Weight* is given in pounds and *Height* is given in inches. Your applet should have three text fields, a button, and a label. The user will enter the weight in one field and the height in feet and inches in the other two fields. When the button is clicked, the BMI will be computed and displayed on label.

Solution:

See the code in BMIApplet.java.

6. Create an HTML page to hold the BMI applet you created in the previous exercise. You should include a link to www.cdc.gov/nccdphp/dnpa/bmi/, which is a page about BMI on the Website of the Centers for Disease Control. You should also include the information from the following table on your page: Indicate that this information was current as of May 2007.

Solution:

See the code in Exercise6.html.

7. Create an applet that will implement a timer. Give it two buttons—Start and Stop—and a text area that has three lines. When the applet begins, only the Start button should be visible. When that button is clicked, get the current time and display it on the first line of the text area. Then make the Start button invisible and the Stop button visible. When the Stop button is clicked, get the current time and display it on the second line. Calculate the difference between the two times and display it on the third line. Finally, make the Stop button invisible.

Solution:

See the code in TimerApplet.java.

8. Create an applet that will compute sales tax. Give it two text fields, a Compute tax button, and a label. The user will enter a tax percentage in the first text field and a value in dollars in the second text field. After the user clicks the button, the sales tax should be displayed in the label.

Solution:

See the code in SalesTaxApplet.java.

9. Create an applet that has four buttons and a text field. Attach an image to each of the buttons: a square, a circle, a filled square, and a filled circle, respectively. The user will enter a length in the text field and then click a button. If the square button is clicked, display the circumference of a square whose sides are the given length. If the filled square is clicked, display the area of the square. If the circle is clicked, display the circumference of a circle whose diameter is the given length. If the filled circle is clicked, display the area of the circle.

Solution:

See the code in FigureComputerApplet.java. The associated images are square.jpg, squareFill.jpg, circle.jpg, circleFill.jpg.

10. Create an applet that has four buttons and a label. On each button attach an icon that is the flag of some country. When the user clicks a button, display the name of the country and its capital city in the label.

Solution:

See the code in FlagApplet.java. The associated images are BelgiumFlag.jpg, FranceFlag.jpg, GermanyFlag.jpg, ItalyFlag.jpg.

11. Create an applet for computing the tip at a restaurant. It will have a text field where the amount of the bill will be entered and two buttons: Tip 15% and Tip 20%. When a button is clicked, display the amount of the tip in a label.

Solution:

See the code in TipApplet.java.

12. Create an HTML page that a restaurant could use to display its daily specials. Include a price and description for each special. Include the applet for computing a tip from the previous exercise.

Solution:

See the code in Exercise12.html.

13. Create an applet that computes the user's gasoline cost. You will need text fields where the user will enter the number of miles traveled by highway, the number of miles of city driving, the average fuel consumption of the vehicle for both highway and city driving, given in miles per gallon, and the cost of gasoline per gallon. Initially, fill these fields with the values 2000, 10000, 30, 20, and 3, respectively. Your applet should have a Compute Cost button that, when clicked, causes the cost to the displayed in a label.

Solution:

See the code in GasCostApplet.java.

14. Convert the substitution-code application from Exercise 9 in the previous chapter into an applet.

Solution:

See the code in CodeApplet.java.

15. Convert the list-of-names application from Exercise 10 in the previous chapter into an applet.

Solution:

See the code in NamesApplet.java.

Projects:

2. Convert the Swing application MemoSaver, as given in Listing 13.10 of the previous chapter, to an applet, and place it in an HTML document.

Notes: The applet is easily created by following the directions in the text to convert a Swing application to an applet.
References: Listing 13.10
Solution: *See the code in MemoSaverApplet.java and MemoSaver.html.*

3. Every first-year electrical engineering student learns that two resistors (a resistor is a common type of electrical component) can be connected in either of two configurations— series or parallel—and that there are simple formulas to calculate the equivalent resistance of a single resistor that could replace the two. If $R1$ and $R2$ are the two resistor values, then
Series resistance = $R1 + R2$, and
Parallel resistance = $(R1 * R2) / (R1 + R2)$.

Write an applet that provides a windowing interface to let a user enter two resistor values and choose which configuration to calculate. Include two text fields (label them Resistor 1 and Resistor 2) for the two input values, two buttons (label them Series and Parallel) to select the configuration, and another text field (label it Equivalent Resistance) to display the calculated value and indicate which configuration was selected. For example, if the user enters 100 for $R1$ and 50 for $R2$ and clicks the Series button, the message would read Series Equivalent = 150. If the user enters the same values and clicks the Parallel button, the message would read Parallel Equivalent = 33.3. Put the applet in a Web page that explains the calculations.

Notes: The applet can be created by copying code for any of the previously developed GUIs and may have any number of equally useful layouts. The solution shown here uses a five-row grid layout and two buttons, one to perform the calculations and another to clear all the text fields. The solution also tests for a variety of error conditions (e.g., no value entered for either or both resistors) and prints an appropriate message if the calculation cannot be performed.
References:
Solution: *See the code in ResistanceCalculatorAppet.java and ResistanceCalculator.html.*

4. Repeat Programming Project 7 in the previous chapter, but write the GUI calculator as an applet.

Notes: The applet is easily done if the calculator program from Chapter 13 Programming Project 7 is available. As with Project 2, just follow the directions in the text to convert a Swing application to an applet.
References: Project 13.7
Solution: *See the code in CalculatorApplet.java.*

5. Write an applet that converts numbers from base-10 (ordinary decimal) notation to hexadecimal (base-16) notation. Use Swing for input and output via an applet interface. The user enters a base-10 integer number in one text field and clicks a button labeled Convert. The equivalent hexadecimal number then appears in another text field. Be sure that the two fields are labeled. Include a Clear button that clears both text fields. (The Swing part of this exercise is quite straightforward, but you do need to know a little about how to convert numbers from one base to another.)

Notes:

The applet can be created most easily by modifying the code from Chapter 13 Project 4, `DecimalToBinary`. Change it from a Swing application to an applet (follow the steps described in the text), and, of course, modify the binary conversion code so it converts to hexadecimal, instead. Fortunately, the same successive division algorithm works, regardless of the base: just change the divisor from 2 to 16 and add code to translate each remainder from a decimal *integer* value to a hex *character* code (a switch statement works very nicely, is very readable, and also is usually compiled very efficiently).

References:

Project 13.4

Solution:

See the code in DecimalToHexadecimalApplet.java.

6. Repeat Programming Project 2, but make all of the following changes:
• Name the class MemoApplet.
• Create six buttons instead of five, and arrange them as follows:
<*diagram omitted*>
The buttons are still at the bottom of the GUI (applet), with the text area above them. *Hint*: Use the GridLayout manager on the button panel.
• When the user saves the text as memo 1, the text area should display Memo 1 saved, and when the user saves the text as memo 2, the text area should display Memo 2 saved. *Hint*: See Self-Test Exercise 43 of the previous chapter.
• Give the text area a line wrap, so that if more characters are entered than will fit on the line, the extra characters automatically go on the next line.

Notes: The applet is most easily written by making changes to the solution to Chapter 13 Programming Project 2 (`MemoSaver2`).
References: Project 14.2
Solution: *See the code in MemoAppket.java.*

16. Repeat Programming Project 4 of Chapter 10, but write it as an applet.

Notes:

The first step is to design the user interface with appropriate buttons, labels, and text boxes and a good next step to write and test just the code to create them. After that, the action events such as opening a file to read, reading records, opening a file to write, writing records, etc., can be developed one at a time. Care must be taken to keep track of whether a file is open or not, and, if a file is open, whether it is a read file or a write file, so a character flag variable readOrWrite is used and assigned an appropriate value depending on the action taken. Another design issue to consider is how to display error messages. One alternative is to use pop-up windows, but the solution shown here takes a simpler approach by writing the messages to the text field at the top of the window.

References:

Project 10.4

Solution:

See the code in SpeciesFileReadOrWriteAppket.java.

17. Repeat Programming Project 9 of Chapter 10, but write it as an applet.

Notes:

Similar notes to the previous project apply to this one. Note that the `PetRecord` definition must be the one that is serializable and it must be inside the applet definition since the applet must be self-contained.

References:

Project 10.9

Solution:

See the code in PetRecordFileReadOrWriteApplet.java.

18. Create an applet ScoreKeeper that keeps track of the score in a football game. To display the scores, the applet will have four labels in the north area. Two of the labels will display the name of the team. The other two labels will display the current score of the teams. In the center of the application will be a number of buttons: one for each team, one for each of the three basic ways to score—touchdown, field goal, and safety—and three buttons for the events that can happen after a touchdown— one-point conversion, two-point conversion, and failed to convert. Initially, there will just be two text areas and an Accept Names button in the south area of the applet. The text areas will be used to enter the names of the two teams. When the button is clicked, the team names will appear in the labels in the north area of the applet and on the team buttons in the center. You will then make the components in the south invisible and make the all components in the north visible. Also, make the two team buttons in the center visible. This will be the starting configuration. When one of the team buttons is clicked, make them both invisible and make all three buttons for the ways to score visible. If the field goal button is then clicked, add three points to the appropriate team's score and return to the starting configuration. If the safety button is clicked, add two points to the appropriate score and return to the starting configuration. If the touchdown button is clicked, add six points to the appropriate score and then make the three touchdown conversion event buttons visible. Once one of these buttons is clicked, add zero, one, or two points, respectively, to the appropriate score and then return to the starting configuration.

Notes:

This application looks at how we can change the appearance by making components visible or invisible. The solution uses a number of methods whose job is just to set the visibility of the components. This abstraction helps simplify the actionPerformed method. The solution also uses getSource() to check which component generated the event as it simplifies the code a bit for the buttons that get labeled with the team names. It is relatively easy to change it so that it checks for the string instead.

Solution:

See the code in ScoreKeeper.java.

19. Create an applet Nim that will allow two players to play the game of nim. Initially, there are three rows of coins. During his or her turn, a player can take as many coins as desired from a single row. The rows have 3, 5, and 7 coins each. The last person to take a coin loses. Use buttons with icons for the coins. As a player clicks the button, it will become invisible. There should be a Turn over button that is visible only when a player has taken at least one coin. Use a label to let the players know whose turn it is and who has won the game.

Notes:

This application makes use of icons and visibility. One interesting feature of the solution is that it uses a non-regular two-dimensional array of buttons. Each row in the array corresponds to a row of coins in the game. You will need to have a state variable that records the row that the first coin in a turn was removed from. Subsequent removals must be checked against the row. Determining which coin button was pressed is an interesting challenge. One way to accomplish this is to set the label for each of the coin buttons with its location in the array; Example: "Coin 1,3". You will then need to parse the action command to determine what to do. This solution takes a slightly different and less efficient approach. It scans the array of buttons and checks to see if any of them is the source (uses the getSource method). This gives it the location of the button. Other than that, it is pretty straightforward.

Solution:

See the code in Nim.java. The associated image is coin.jpg

20. Convert the binary to octal translation application from Project 11 in the previous chapter to an applet.

Notes: This is just a simple translation exercise provided that a solution to Project 13.11 is available.
References: Project 13.11
Solution: *See the code in BinaryOctalConverterApplet.java.*

Exercises:

1. Create an application that has a text area with scroll bars. Enable horizontal scrolling as needed. Always provide vertical scroll bars. Create two buttons. If the first button is pressed, append a number to the text area. Start the number at 1, and increase it after the button is pressed. If the second button is pressed, append a new line to the text area. Put a nice border around the scroll pane.

Solution:

See the code in Exercise1.java, ScrollTextWindow.java.

2. Create an application that models a simple sales terminal. You should be able to sell three kinds of items. Have one button for each item, and attach a picture of the item to the button. Each button should have three labels associated with it. These labels will display the price of the item, the number of that item sold in the current transaction, and a subtotal for that item. Each time a button is pressed, increase the count of that item in the current sale by one and update the subtotal. A separate tenth label should show the total cost of the current sale. An "EndSale" menu item ends the current sale and resets the totals to zero.

Solution:

See the code in TerminalWindow.java. Associated images are in Phone.jpg, Printer.jpg, Laptop.jpg.

3. Create an application that overrides the action of the methods windowActivated and windowDeactivated. When the window is activated, change the background color to white. When it is deactivated, change the background to gray. Use at least four components in the GUI.

Solution:

See the code in Exercise3.java, ActivatedWindow.java.

4. Create an application that overrides the action of the method windowIconified. Every time the window is iconified, add one to a counter and display the counter in a label.

Solution:

See the code in Exercise4.java, IconifierWindow.java.

5. Create an application that uses a card layout with three cards. The first card—a login card—should have two text fields, one for a user identification and the other for a password. There are two users—Bob and Fred—whose passwords are "mubby" and "goolag", respectively. If Bob logs in, switch to a card—the bob card— that has a text field, a text area, and two buttons. If the first button is pressed, get the text from the text field and append it to the text area. If the second button is pressed, return to the login card. If Fred logs in, switch to a card—the fred card—that has three buttons. If the first button is pressed, change the background
color to green. If the second button is pressed, change the background color to red. If the third button is pressed, return to the login card.

Solution:

See the code in Exercise5.java, SimpleCardWindow.java.

6. Create a GUI that has two buttons on it. When clicked, each button creates a new window. The first button creates a window that has a single button on it. Pressing that button will change the window's background color back and forth between red and green. The second button creates a window that has two buttons. Pressing the first button of these two buttons will change the window's background color to black. Pressing the second button will change the background color to white. Make sure that your windows will just dispose of themselves when they are closed.

Solution:

See the code in Exercise6.java, MultiWindow.java.

7. Create an application that has a text area, a text field and a button. In the text field you can enter a name. When the button is pressed get the name from the text field and create a new window. Use the name as the title of the new window. Also program the window so that when it is closed, it will append its name to the text area of your application and then dispose of itself.

Solution:

See the code in Exercise7.java, RecorderWindow.java.

8. Create a GUI that has three buttons, each of which contains the name of a song. When a button is pressed, create a new window. Title the window with the name of the song. In a text area in the window, display the lyrics to the song. When the window is closed, it should dispose of itself, but don't quit the application.

Solution:

See the code in Exercise7.java, RecorderWindow.java.

9. Create an application that draws a simple stick figure person that looks something like the following figure:
You should have three buttons: Dress, Hair, and Shoes. Associated with the Dress button is a window that has four buttons, one for each of the colors red, green, blue, and orange. Pressing one of these buttons will change the color of the dress. Similarly, the Hair button is associated with a window that has three buttons, one for each of the colors black, gray, and pink. And the Shoes button is associated with a window that has three buttons, one for each of the colors red, yellow, and blue. When the application starts, create the windows for the Dress, Hair, and Shoes buttons, and make these windows invisible. Pressing a button will make the correct window visible. If the window is closed, just make it invisible.

Solution:

See the code in StickFigureWindow.java.

10. Create an application that will compute the average of a list of numbers. Begin with two windows. The first window, titled "Display," should have a label and a vertically scrollable text area. The second window, titled "Data entry," should have a text field and an Enter button. When a user enters an integer value into the text field and clicks the Enter button, copy the value in the text field to the text area in the display window. Also, update the label in this window to indicate the number of values, their sum, and average. If either window is closed, quit the application.

Solution:

See the code in ComputeAverage.java.

11. Create an application that will display a string of text using different fonts. Your application should have three menus. The font-name menu will have five menu items: Serif, SansSerif, Monospaced, Dialog, and DialogInput. The style menu will have three menu items: Plain, Bold, and Italic. The size menu will have four menu items: 10, 12, 18, and 24. When a menu item is chosen, change the font and then repaint the window. *Hint*: Use the method drawString to display the sample string. Use the expression new Font(type, style, size) to create the font you will display. Refer to the API documentation for the class Font for additional details.

Solution:

See the code in Exercise11.java, FontWindow.java.

12. Modify the application in the previous exercise so that it uses a single nested menu. The single menu, titled "font," should have three submenus, one each for the font-name, style, and size.

Solution:

See the code in Exercise12.java, NestedFontWindow.java.

13. Create an application that uses nested menus to choose an ice cream sundae. The "Choices" menu will have three submenus. The "Flavor" submenu will have three menu items: Chocolate, Strawberry, and Vanilla. The "Toppings" menu will have four menu items: Chocolate chips, Sprinkles, Nuts, and Peppermint. The "Syrup" menu will have three menu items: Chocolate, Butterscotch, and Berry. As the choices are made, update a label in the center of the application. The "Actions" menu will have two items: Clear will clear the current order, and Quit will exit the application.

Solution:

See the code in SundaeCreator.java.

14. Create an application that will draw a figure. Your application will have a Shape menu with three choices: Circle, Square, and Triangle. A Color menu should have four choices: Black, Cyan, Magenta, and Yellow. When one of the menu items is chosen, redraw the figure. In the east area of the application, place three text fields and a button. The text fields will allow you to enter integer values for the *x* and *y* coordinates of the center of the figure and the size of the figure. When the Change button is pressed, get the values from the text fields and redraw the figure.

Solution:

See the code in Exercise14.java, ShapeWindow.java.

15. Repeat Exercise 10 of Chapter 13 to use a menu instead of an Accept button. The menu should contain an "Add Name" item that has the same function as the Accept button.

Solution:

See the code in Exercise15.java, NamesMenuWindow.java.

Projects:

1. Write a GUI that will let the user sample icons. It should have a menu named Icons that offers three choices: Smiley Face, Duke Waving, and Duke Standing. When the user chooses a menu item, the corresponding icon is displayed (and no other icons are displayed). When the GUI is first displayed, no icons are visible. The picture files for these icons are available in the source code for this chapter on the Web.

Notes: The layout of the window, structure of the code, and technique for updating the display based on the user's selection is similar to that for `MemoGUI`, Listing 15.1. As with any of the applications that use icons, you will need to make sure that the image files are in an appropriate folder. This is not always the same as where the source files are located.
References: Listing 15.1
Solution: *See the code in IconMenu.java.*

2. Enhance the memo saver GUI in Listing 15.1 in all of the following ways:
• Add another menu called Scroll Bars that offers the user three choices: Never, Always, and As Needed. When the user makes a choice, the scroll bars are displayed according to the choice.
• When the user clicks the close-window button, another window pops up and asks the user whether to end the program, as was done in Listing 15.10.

Notes:

This program is best done by adding one feature at a time and judiciously copying code from examples on the CD that comes with the text.

To obtain the exit pop-up window, start with the code from `CloseWindowDemo`, Listing 15.10 and add to it. For example, add a call to `ConfirmWindow()` in the `actionPerformed` method if the `ActionEvent` is "Exit". The `setDefaultCloseOperation` code is necessary to get the pop-up window when the exit button (the "X" in the upper left corner of the window) is clicked, and the call in `actionPerformed` is necessary to get the window when "Close" in the "Memos" menu is selected.
Next, add the nested menus and make sure it works before actually coding the events for the options.

Next, add functionality to the "View" menu options.
Unfortunately, there is no program from which to copy the code to make the "Scroll Bars" options functional, but it is mostly a matter of going through "The JScrollPane Class for Scroll Bars" section of Chapter 15 and following the directions. Note that the following line is necessary to make the change visible immediately after selecting any of these options:

```
SwingUtilities.updateComponentTreeUI(this);
```

References:

Listing 15.1, Listing 15.10

Solution:

See the code in MemoGUIEnhanced.java.

3. Enhance the program you wrote for Programming Project 8 in Chapter 8 to give it a GUI that allows the user to write and read objects of type Pet to a file. Include a menu with options to create a new file, read an existing one, or exit. Create two files—named dogRecords and catRecords—and list them in a submenu under the read option. *Hint*: You can use this program to create the files dogRecords and catRecords. This project will require two testing phases, one before and one after creating the files, but that is not a major problem.

Notes:

This is one of the more complicated problems in the text. While it is not difficult to create the menus, building and viewing files is another matter. As is often the case, it is easier to create the interface first, then add functionality a piece at a time. The user interface for the solution shown here a main window to display or enter data, and a pop-up window to enter the file name. The data window does not show text areas until a file has been opened to read or write. One dilemma is that the view option cannot be tested unless files with `PetRecord` records can be built, and, vice versa, the build option cannot be tested unless it is possible to view the files. One way around this dilemma is to use `PetFileReadOrWrite` from Chapter 10 Programming Project 9 to create the files

References:

Project 10.9

Solution:

See the code in PetFilerGUI.java. Uses PetRecord.java.

4. Write a GUI that will let the user sample borders. Include a menu named Borders that offers three options—beveled border, etched border, and line border—as submenus with the following options:
• Beveled-border submenu options: raised or lowered.
• Etched-border submenu options: raised or lowered.
• Line-border submenu options: small, medium, or large.
Each of these options should be a submenu with three color options: black, red, and blue. Put the borders around a label containing text that describes the border, such as Raised Border, Lowered Etched Border, and so forth. Fix the highlight and shadow colors for the etched-border options to whatever colors you like, and make the small line border 5 pixels wide, the medium one 10 pixels wide, and the large one 20 pixels wide.

Notes:

This Programming Project is fairly easy, especially if done a step at a time. First create the menu hierarchy, then add the lines in the `actionPerformed` method to set the border style and change the label's text, depending on the style selected. Note that `setActionCommand` is used to differentiate selections with the same text (e.g. "Raised" could be either Beveled or Etched).

Solution:

See the code in BordersMenu.java.

5. Repeat Programming Project 3 using either the BoxLayout manager or the Box container class to create the GUI. Place two buttons, one to write a file and the other to read a file, vertically on the left.

Notes: While this project is similar to Project 3, it has a significantly revised interface. The interface was designed to display one set of buttons, labels, and text boxes to enter the file name and a different set of buttons, labels, and text boxes for data viewing or entry.
References: Project 15.3
Solution: *See the code in PetFilerBoxLayout.java. Uses PetRecord.java.*

6. Write a “skeleton” GUI program that implements the WindowListener interface. Write code for each WindowListener method that simply displays a message— in a text field—identifying which event occurred. Recall that these methods are the same as the ones listed in Figure 13.3 of Chapter 13 for the class Window- Adapter. Note that your program will not end when the close-window button is clicked, but will instead simply send a message to the text field saying that the windowClosing method has been invoked. Include an Exit button that the user can click to end the program.

Notes: A separate window is used to show the messages each time one of the main window events are clicked. `setDefaultCloseOperation` is used to reprogram the close-window event so that it prints a message instead of closing.
References: Listing 15.9
Solution: *See the code in WindowListenerSkeleton.java.*

7. Create an application that uses a card layout manager to enable two people to play a guessing game. The initial card will have two text fields in which the players will enter their names. The StartGame button will bring up a new card, the numberentry card, that has a text field in which a player can enter a secret integer value.

The PlayGame button will bring up a second card that has three buttons and a text field. The player will enter a guess value in the text field and then press one of the three buttons. The greater button will display true or false depending on whether the player's guess value is greater than the secret value. The less button will display true or false depending on whether the players guess value is less than the secret value. The equal button will display true or false depending on whether the players guess value is equal to the secret value. Once the player correctly guesses the integer value, the number of guesses made is added to that player's total score and displayed in a text field. Pressing the NewNumber button will bring back the number-entry card. Players should alternate between picking the secret number and guessing the value of the secret number. You should have labels that will display each player's current score.

Notes:

This application demonstrates the use of a card layout. Note that in this game, lower scores are better. (This is why the score is shown as blots.) To simplify the actionPerformed method, we break each button's action out into a method. The setup code is long, but uncomplicated.

When constructing this application, it would be a good idea to do it iteratively.

1) Get the card logic to work correctly first (switching between cards)
2) Complete the start card. (Name entry works)
3) Complete the secret number entry card. (A player can enter their secret number)
4) Do the guess card. (Add each button and make it work.)

Solution:

See the code in in GuessingGame.java..

8. Create an application that will play a simple one-player card game. In this game, you get four random cards that have values in the range from 1 to 10 and are face down so their values hidden. As the game progresses, you can press a NewCard button and get a new card with its value shown. Use an icon and a label to display the card. Getting a new card will cost you one point from your total score. You can then choose to replace one of your four cards with the new card. To do so, implement the four cards as buttons with icons and use a button press to indicate which card you want to replace. When the card is replaced, use a text label to inform the player of the value of the card that was replaced. Make sure that only one card can be replaced. At any point, you can press a Stop button to stop that hand.
All four of your cards should then be displayed face up. Total the values of the cards and add that to your total score for all hands. You will need two labels to display the score for the game. The first label will give the total score the player has earned. The second label will give a total that is par for the game. To determine the par value for a given hand, add up the values of the cards in the hand before any replacement is done. Once the hand is over, add the par value of the hand to a total and display that. You will need another button, StartNewHand, that will generate four new random cards for your hand.

Notes:

This application can really look nice with appropriate use of icons. The solution has an icon for the back of the card and one for each of the faces. It then uses the setIcon method for a JButton to change the image as needed. The solution uses getSource to determine which button was pressed, but it could be easily modified to check for an action string instead. The most complicated part of the application is deciding what to do for each click as there are a number of possible states that we can be in and we want to keep track of the score as the player deals out cards.

One interesting thing to do is to analyze this game. Is it advantageous to deal a card? Once the card is dealt, should we replace one of our known or unknown cards with it? This game is simple enough that it can be analyzed, but complicated enough to make it interesting.

Solution:

See the code in CardGame.java. There are a number of jpg files used for the card images.

9. Enhance Programing Project 11 in Chapter 13 that converts between binary and octal numbers by adding conversions to and from the hexadecimal number system and by using menus. A Convert menu should have four items, the first three of which are submenus. The first submenu—FromBinary—will have two menu items: “to octal” and “to hexadecimal.” The second submenu—FromOctal— will have two menu items, “to binary” and “to hexadecimal.” The third submenu— FromHexadecimal—will have two menu items, “to binary” and “to octal.”
The final item in the Convert menu is “Clear.” Hexadecimal numbers use the digits 0 through 9 and the letters A through F. Note that each hexadecimal digit corresponds to a four-bit binary number, as follows:

Binary Hexadecimal

0000 0
0001 1
0010 2
0011 3
0100 4
0101 5
0110 6
0111 7
1000 8
1001 9
1010 A
1011 B
1100 C
1101 D
1110 E
1111 F

To convert a number from binary to hexadecimal, first group the bits in the binary number into sets of four and then apply the correspondences. For example, the binary number 0101110010100110 would be grouped as 0101 1100 1010 0110, which corresponds to the hexadecimal digits 5, C, A, and 6, respectively, Thus, the hexadecimal equivalent of the binary number is 5CA6. If the number of bits in the binary number is not divisible by 4, add 0’s at its beginning until it is. To convert an hexadecimal number to binary, we use our correspondence table in the reverse direction. For example, the hexadecimal number 7F6 is 0111 1111 0110, or 01111111 0110 in binary. Your application can omit the spaces we use to show the grouping of the bits.
To convert between octal and hexadecimal, first convert to binary.

Notes:

This is an extension of the converter project in the previous chapter. Conceptually the changes required to add in hexadecimal numbers are easy, but there are some details that get in the way. Checking the validity of a hexadecimal number is complicated by the fact that the "digits" now include A through F. The groupings are for 4 bits instead of 3 bits, which increases the number of cases to consider. Even using an array to store the mappings is complicated by the alphabetic digits in a hexadecimal representation. One other factor to consider is how to convert between octal and hexadecimal. As directed, this code converts to binary first and then converts from binary to the desired base using existing methods. This makes those two conversions trivial.

Solution:

See the code in BaseConverter.java.