Programming Exercises

For Chapter 2

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 2 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

<http://www.java2s.clanteam.com>

--------------------------------------------------------------------------------

Exercise 2.1: Write a program that will print your initials to standard output in letters that are nine lines tall. Each big letter should be made up of a bunch of \*'s. For example, if your initials were "DJE", then the output would look something like:

\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*

\*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\*\*\*\*\*\*\*

\*\* \*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\* \*\*

\*\*\*\*\* \*\*\*\* \*\*\*\*\*\*\*\*\*\*

See the solution! : visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 2.2: Write a program that simulates rolling a pair of dice. You can simulate rolling one die by choosing one of the integers 1, 2, 3, 4, 5, or 6 at random. The number you pick represents the number on the die after it is rolled. As pointed out in Section 5, The expression

(int)(Math.random()\*6) + 1

does the computation you need to select a random integer between 1 and 6. You can assign this value to a variable to represent one of the dice that are being rolled. Do this twice and add the results together to get the total roll. Your program should report the number showing on each die as well as the total roll. For example:

The first die comes up 3

The second die comes up 5

Your total roll is 8

(Note: The word "dice" is a plural, as in "two dice." The singular is "die.")

See the solution! :visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 2.3: Write a program that asks the user's name, and then greets the user by name. Before outputting the user's name, convert it to upper case letters. For example, if the user's name is Fred, then the program should respond "Hello, FRED, nice to meet you!".

See the solution! :visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 2.4: Write a program that helps the user count his change. The program should ask how many quarters the user has, then how many dimes, then how many nickels, then how many pennies. Then the program should tell the user how much money he has, expressed in dollars.

See the solution! :visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 2.5: If you have N eggs, then you have N/12 dozen eggs, with N%12 eggs left over. (This is essentially the definition of the / and % operators for integers.) Write a program that asks the user how many eggs she has and then tells the user how many dozen eggs she has and how many extra eggs are left over.

A gross of eggs is equal to 144 eggs. Extend your program so that it will tell the user how many gross, how many dozen, and how many left over eggs she has. For example, if the user says that she has 1342 eggs, then your program would respond with

Your number of eggs is 9 gross, 3 dozen, and 10

since 1342 is equal to 9\*144 + 3\*12 + 10.

See the solution! :visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 3

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 3 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 3.1: How many times do you have to roll a pair of dice before they come up snake eyes? You could do the experiment by rolling the dice by hand. Write a computer program that simulates the experiment. The program should report the number of rolls that it makes before the dice come up snake eyes. (Note: "Snake eyes" means that both dice show a value of 1.) Exercise 2.2 explained how to simulate rolling a pair of dice.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 3.2: Which integer between 1 and 10000 has the largest number of divisors, and how many divisors does it have? Write a program to find the answers and print out the results. It is possible that several integers in this range have the same, maximum number of divisors. Your program only has to print out one of them. One of the examples from Section 3.4 discussed divisors. The source code for that example is CountDivisors.java.

You might need some hints about how to find a maximum value. The basic idea is to go through all the integers, keeping track of the largest number of divisors that you've seen so far. Also, keep track of the integer that had that number of divisors.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 3.3: Write a program that will evaluate simple expressions such as 17 + 3 and 3.14159 \* 4.7. The expressions are to be typed in by the user. The input always consist of a number, followed by an operator, followed by another number. The operators that are allowed are +, -, \*, and /. You can read the numbers with TextIO.getDouble() and the operator with TextIO.getChar(). Your program should read an expression, print its value, read another expression, print its value, and so on. The program should end when the user enters 0 as the first number on the line.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 3.4: Write a program that reads one line of input text and breaks it up into words. The words should be output one per line. A word is defined to be a sequence of letters. Any characters in the input that are not letters should be discarded. For example, if the user inputs the line

He said, "That's not a good idea."

then the output of the program should be

He

said

that

s

not

a

good

idea

(An improved version of the program would list "that's" as a word. An apostrophe can be considered to be part of a word if there is a letter on each side of the apostrophe. But that's not part of the assignment.)

To test whether a character is a letter, you might use (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'). However, this only works in English and similar languages. A better choice is to call the standard function Character.isLetter(ch), which returns a boolean value of true if ch is a letter and false if it is not. This works for any Unicode character. For example, it counts an accented e, é, as a letter.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 3.5: Write an applet that draws a checkerboard. Assume that the size of the applet is 160 by 160 pixels. Each square in the checkerboard is 20 by 20 pixels. The checkerboard contains 8 rows of squares and 8 columns. The squares are red and black. Here is a tricky way to determine whether a given square is red or black: If the row number and the column number are either both even or both odd, then the square is red. Otherwise, it is black. Note that a square is just a rectangle in which the height is equal to the width, so you can use the subroutine g.fillRect() to draw the squares. Here is an image of the checkerboard:

(To run an applet, you need a Web page to display it. A very simple page will do. Assume that your applet class is called Checkerboard, so that when you compile it you get a class file named Checkerboard.class Make a file that contains only the lines:

<applet code="Checkerboard.class" width=160 height=160>

</applet>

Call this file Checkerboard.html. This is the source code for a simple Web page that shows nothing but your applet. You can open the file in a Web browser or with Sun's appletviewer program. The compiled class file, Checkerboard.class, must be in the same directory with the Web-page file, Checkerboard.html.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 3.6: Write an animation applet that shows a checkerboard pattern in which the even numbered rows slide to the left while the odd numbered rows slide to the right. You can assume that the applet is 160 by 160 pixels. Each row should be offset from its usual position by the amount getFrameNumber() % 40. Hints: Anything you draw outside the boundaries of the applet will be invisible, so you can draw more than 8 squares in a row. You can use negative values of x in g.fillRect(x,y,w,h). Here is a working solution to this exercise:

Your applet will extend the non-standard class, SimpleAnimationApplet2, which was introduced in Section 7. When you run your applet, the compiled class files, SimpleAnimationApplet2.class and SimpleAnimationApplet2$1.class, must be in the same directory as your Web-page source file and the compiled class file for your own class. These files are produced when you compile SimpleAnimationApplet2.java. Assuming that the name of your class is SlidingCheckerboard, then the source file for the Web page should contain the lines:

<applet code="SlidingCheckerboard.class" width=160 height=160>

</applet>

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 4

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 4 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 4.1: To "capitalize" a string means to change the first letter of each word in the string to upper case (if it is not already upper case). For example, a capitalized version of "Now is the time to act!" is "Now Is The Time To Act!". Write a subroutine named printCapitalized that will print a capitalized version of a string to standard output. The string to be printed should be a parameter to the subroutine. Test your subroutine with a main() routine that gets a line of input from the user and applies the subroutine to it.

Note that a letter is the first letter of a word if it is not immediately preceded in the string by another letter. Recall that there is a standard boolean-valued function Character.isLetter(char) that can be used to test whether its parameter is a letter. There is another standard char-valued function, Character.toUpperCase(char), that returns a capitalized version of the single character passed to it as a parameter. That is, if the parameter is a letter, it returns the upper-case version. If the parameter is not a letter, it just returns a copy of the parameter.

See the solution! http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 4.2: The hexadecimal digits are the ordinary, base-10 digits '0' through '9' plus the letters 'A' through 'F'. In the hexadecimal system, these digits represent the values 0 through 15, respectively. Write a function named hexValue that uses a switch statement to find the hexadecimal value of a given character. The character is a parameter to the function, and its hexadecimal value is the return value of the function. You should count lower case letters 'a' through 'f' as having the same value as the corresponding upper case letters. If the parameter is not one of the legal hexadecimal digits, return -1 as the value of the function.

A hexadecimal integer is a sequence of hexadecimal digits, such as 34A7, FF8, 174204, or FADE. If str is a string containing a hexadecimal integer, then the corresponding base-10 integer can be computed as follows:

value = 0;

for ( i = 0; i < str.length(); i++ )

value = value\*16 + hexValue( str.charAt(i) );

Of course, this is not valid if str contains any characters that are not hexadecimal digits. Write a program that reads a string from the user. If all the characters in the string are hexadecimal digits, print out the corresponding base-10 value. If not, print out an error message.

See the solution! http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 4.3: Write a function that simulates rolling a pair of dice until the total on the dice comes up to be a given number. The number that you are rolling for is a parameter to the function. The number of times you have to roll the dice is the return value of the function. You can assume that the parameter is one of the possible totals: 2, 3, ..., 12. Use your function in a program that computes and prints the number of rolls it takes to get snake eyes. (Snake eyes means that the total showing on the dice is 2.)

See the solution! http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 4.4: This exercise builds on Exercise 4.3. Every time you roll the dice repeatedly, trying to get a given total, the number of rolls it takes can be different. The question naturally arises, what's the average number of rolls? Write a function that performs the experiment of rolling to get a given total 10000 times. The desired total is a parameter to the subroutine. The average number of rolls is the return value. Each individual experiment should be done by calling the function you wrote for exercise 4.3. Now, write a main program that will call your function once for each of the possible totals (2, 3, ..., 12). It should make a table of the results, something like:

Total On Dice Average Number of Rolls

------------- -----------------------

2 35.8382

3 18.0607

. .

. .

See the solution! http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 5: The sample program RandomMosaicWalk.java from Section 4.6 shows a "disturbance" that wanders around a grid of colored squares. When the disturbance visits a square, the color of that square is changed. The applet at the bottom of Section 4.7 shows a variation on this idea. In this applet, all the squares start out with the default color, black. Every time the disturbance visits a square, a small amount is added to the red component of the color of that square. Write a subroutine that will add 25 to the red component of one of the squares in the mosaic. The row and column numbers of the square should be passed as parameters to the subroutine. Recall that you can discover the current red component of the square in row r and column c with the function call Mosaic.getRed(r,c). Use your subroutine as a substitute for the changeToRandomColor() subroutine in the program RandomMosaicWalk2.java. (This is the improved version of the program from Section 4.7 that uses named constants for the number of rows, number of columns, and square size.) Set the number of rows and the number of columns to 80. Set the square size to 5.

See the solution! http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 6: For this exercise, you will write a program that has the same behavior as the following applet. Your program will be based on the non-standard Mosaic class, which was described in Section 4.6. (Unfortunately, the applet doesn't look too good on many versions of Java.)

The applet shows a rectangle that grows from the center of the applet to the edges, getting brighter as it grows. The rectangle is made up of the little squares of the mosaic. You should first write a subroutine that draws a rectangle on a Mosaic window. More specifically, write a subroutine named rectangle such that the subroutine call statement

rectangle(top,left,height,width,r,g,b);

will call Mosaic.setColor(row,col,r,g,b) for each little square that lies on the outline of a rectangle. The topmost row of the rectangle is specified by top. The number of rows in the rectangle is specified by height (so the bottommost row is top+height-1). The leftmost column of the rectangle is specifed by left. The number of columns in the rectangle is specified by width (so the rightmost column is left+width-1.)

The animation loops through the same sequence of steps over and over. In one step, a rectangle is drawn in gray (that is, with all three color components having the same value). There is a pause of 200 milliseconds so the user can see the rectangle. Then the very same rectangle is drawn in black, effectively erasing the gray rectangle. Finally, the variables giving the top row, left column, size, and color level of the rectangle are adjusted to get ready for the next step. In the applet, the color level starts at 50 and increases by 10 after each step. You might want to make a subroutine that does one loop through all the steps of the animation.

The main() routine simply opens a Mosaic window and then does the animation loop over and over until the user closes the window. There is a 1000 millisecond delay between one animation loop and the next. Use a Mosaic window that has 41 rows and 41 columns. (I advise you not to used named constants for the numbers of rows and columns, since the problem is complicated enough already.)

See the solution! http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 5

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 5 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 5.1: In all versions of the PairOfDice class in Section 2, the instance variables die1 and die2 are declared to be public. They really should be private, so that they are protected from being changed from outside the class. Write another version of the PairOfDice class in which the instance variables die1 and die2 are private. Your class will need methods that can be used to find out the values of die1 and die2. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, if you can think of any. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 5.2: A common programming task is computing statistics of a set of numbers. (A statistic is a number that summarizes some property of a set of data.) Common statistics include the mean (also known as the average) and the standard deviation (which tells how spread out the data are from the mean). I have written a little class called StatCalc that can be used to compute these statistics, as well as the sum of the items in the dataset and the number of items in the dataset. You can read the source code for this class in the file StatCalc.java. If calc is a variable of type StatCalc, then the following methods are defined:

calc.enter(item); where item is a number, adds the item to the dataset.

calc.getCount() is a function that returns the number of items that have been added to the dataset.

calc.getSum() is a function that returns the sum of all the items that have been added to the dataset.

calc.getMean() is a function that returns the average of all the items.

calc.getStandardDeviation() is a function that returns the standard deviation of the items.

Typically, all the data are added one after the other calling the enter() method over and over, as the data become available. After all the data have been entered, any of the other methods can be called to get statistical information about the data. The methods getMean() and getStandardDeviation() should only be called if the number of items is greater than zero.

Modify the current source code, StatCalc.java, to add instance methods getMax() and getMin(). The getMax() method should return the largest of all the items that have been added to the dataset, and getMin() should return the smallest. You will need to add two new instance variables to keep track of the largest and smallest items that have been seen so far.

Test your new class by using it in a program to compute statistics for a set of non-zero numbers entered by the user. Start by creating an object of type StatCalc:

StatCalc calc; // Object to be used to process the data.

calc = new StatCalc();

Read numbers from the user and add them to the dataset. Use 0 as a sentinel value (that is, stop reading numbers when the user enters 0). After all the user's non-zero numbers have been entered, print out each of the six statistics that available from calc.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 5.3: This problem uses the PairOfDice class from Exercise 5.1 and the StatCalc class from Exercise 5.2.

The program in Exercise 4.4 performs the experiment of counting how many times a pair of dice is rolled before a given total comes up. It repeats this experiment 10000 times and then reports the average number of rolls. It does this whole process for each possible total (2, 3, ..., 12).

Redo that exercise. But instead of just reporting the average number of rolls, you should also report the standard deviation and the maximum number of rolls. Use a PairOfDice object to represent the dice. Use a StatCalc object to compute the statistics. (You'll need a new StatCalc object for each possible total, 2, 3, ..., 12. You can use a new pair of dice if you want, but it's not necessary.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 5.4: The BlackjackHand class from Section 5.5 is an extension of the Hand class from Section 5.3. The instance methods in the Hand class are discussed in Section 5.3. In addition to those methods, BlackjackHand includes an instance method, getBlackjackValue(), that returns the value of the hand for the game of Blackjack. For this exercise, you will also need the Deck and Card classes from Section 5.3.

A Blackjack hand typically contains from two to six cards. Write a program to test the BlackjackHand class. You should create a BlackjackHand object and a Deck object. Pick a random number between 2 and 6. Deal that many cards from the deck and add them to the hand. Print out all the cards in the hand, and then print out the value computed for the hand by getBlackjackValue(). Repeat this as long as the user wants to continue.

In addition to TextIO, your program will depend on Card.java, Deck.java, Hand.java, and BlackjackHand.java.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 5.5 Write a program that let's the user play Blackjack. The game will be a simplified version of Blackjack as it is played in a casino. The computer will act as the dealer. As in the previous exercise, your program will need the classes defined in Card.java, Deck.java, Hand.java, and BlackjackHand.java. (This is the longest and most complex program that has come up so far in the exercises.)

You should first write a subroutine in which the user plays one game. The subroutine should return a boolean value to indicate whether the user wins the game or not. Return true if the user wins, false if the dealer wins. The program needs an object of class Deck and two objects of type BlackjackHand, one for the dealer and one for the user. The general object in Blackjack is to get a hand of cards whose value is as close to 21 as possible, without going over. The game goes like this.

First, two cards are dealt into each player's hand. If the dealer's hand has a value of 21 at this point, then the dealer wins. Otherwise, if the user has 21, then the user wins. (This is called a "Blackjack".) Note that the dealer wins on a tie, so if both players have Blackjack, then the dealer wins.

Now, if the game has not ended, the user gets a chance to add some cards to her hand. In this phase, the user sees her own cards and sees one of the dealer's two cards. (In a casino, the dealer deals himself one card face up and one card face down. All the user's cards are dealt face up.) The user makes a decision whether to "Hit", which means to add another card to her hand, or to "Stand", which means to stop taking cards.

If the user Hits, there is a possibility that the user will go over 21. In that case, the game is over and the user loses. If not, then the process continues. The user gets to decide again whether to Hit or Stand.

If the user Stands, the game will end, but first the dealer gets a chance to draw cards. The dealer only follows rules, without any choice. The rule is that as long as the value of the dealer's hand is less than or equal to 16, the dealer Hits (that is, takes another card). The user should see all the dealer's cards at this point. Now, the winner can be determined: If the dealer has gone over 21, the user wins. Otherwise, if the dealer's total is greater than or equal to the user's total, then the dealer wins. Otherwise, the user wins.

Two notes on programming: At any point in the subroutine, as soon as you know who the winner is, you can say "return true;" or "return false;" to end the subroutine and return to the main program. To avoid having an overabundance of variables in your subroutine, remember that a function call such as userHand.getBlackjackValue() can be used anywhere that a number could be used, including in an output statement or in the condition of an if statement.

Write a main program that lets the user play several games of Blackjack. To make things interesting, give the user 100 dollars, and let the user make bets on the game. If the user loses, subtract the bet from the user's money. If the user wins, add an amount equal to the bet to the user's money. End the program when the user wants to quit or when she runs out of money.

Here is an applet that simulates the program you are supposed to write. It would probably be worthwhile to play it for a while to see how it works.

Sorry, your browser doesn't

support Java.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 6

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 6 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 6.1: Write an applet that shows a pair of dice. When the user clicks on the applet, the dice should be rolled (that is, the dice should be assigned newly computed random values). Each die should be drawn as a square showing from 1 to 6 dots. Since you have to draw two dice, its a good idea to write a subroutine, "void drawDie(Graphics g, int val, int x, int y)", to draw a die at the specified (x,y) coordinates. The second parameter, val, specifes the value that is showing on the die. Assume that the size of the applet is 100 by 100 pixels. Here is a working version of the applet. (My applet plays a clicking sound when the dice are rolled. See the solution to see how this is done.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 6.2: Improve your dice applet from the previous exercise so that it also responds to keyboard input. When the applet has the input focus, it should be hilited with a colored border, and the dice should be rolled whenever the user presses a key on the keyboard. This is in addition to rolling them when the user clicks the mouse on the applet. Here is an applet that solves this exercise:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 6.3: In Exercise 6.1, above, you wrote a pair-of-dice applet where the dice are rolled when the clicks on the applet. Now make a pair-of-dice applet that uses the methods discussed in Section 6.6. Draw the dice on a JPanel, and place a "Roll" button at the bottom of the applet. The dice should be rolled when the user clicks the Roll button. Your applet should look and work like this one:

(Note: Since there was only one button in this applet, I added it directly to the applet's content pane, rather than putting it in a "buttonBar" panel and adding the panel to the content pane.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 6.4: In Exercise 3.5, you drew a checkerboard. For this exercise, write a checkerboard applet where the user can select a square by clicking on it. Hilite the selected square by drawing a colored border around it. When the applet is first created, no square is selected. When the user clicks on a square that is not currently selected, it becomes selected. If the user clicks the square that is selected, it becomes unselected. Assume that the size of the applet is 160 by 160 pixels, so that each square on the checkerboard is 20 by 20 pixels. Here is a working version of the applet:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 6.5: Write an applet that shows two squares. The user should be able to drag either square with the mouse. (You'll need an instance variable to remember which square the user is dragging.) The user can drag the square off the applet if she wants; if she does this, it's gone. You can try it here:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 6.6: For this exercise, you should modify the SubKiller game from Section 6.5. You can start with the existing source code, from the file SubKillerGame.java. Modify the game so it keeps track of the number of hits and misses and displays these quantities. That is, every time the depth charge blows up the sub, the number of hits goes up by one. Every time the depth charge falls off the bottom of the screen without hitting the sub, the number of misses goes up by one. There is room at the top of the applet to display these numbers. To do this exercise, you only have to add a half-dozen lines to the source code. But you have to figure out what they are and where to add them. To do this, you'll have to read the source code closely enough to understand how it works.

See the solution! (A working version of the applet can be found here.)

--------------------------------------------------------------------------------

Exercise 6.7: Section 3.7 discussed SimpleAnimationApplet2, a framework for writing simple animations. You can define an animation by writing a subclass and defining a drawFrame() method. It is possible to have the subclass implement the MouseListener interface. Then, you can have an animation that responds to mouse clicks.

Write a game in which the user tries to click on a little square that jumps erratically around the applet. To implement this, use instance variables to keep track of the position of the square. In the drawFrame() method, there should be a certain probability that the square will jump to a new location. (You can experiment to find a probability that makes the game play well.) In your mousePressed method, check whether the user clicked on the square. Keep track of and display the number of times that the user hits the square and the number of times that the user misses it. Don't assume that you know the size of the applet in advance.

See the solution! (A working version of the applet can be found here.)

--------------------------------------------------------------------------------

Exercise 6.8:Write a Blackjack applet that lets the user play a game of Blackjack, with the computer as the dealer. The applet should draw the user's cards and the dealer's cards, just as was done for the graphical HighLow card game in Section 6.6. You can use the source code for that game, HighLowGUI.java, for some ideas about how to write your Blackjack game. The structures of the HighLow applet and the Blackjack applet are very similar. You will certainly want to use the drawCard() method from that applet.

You can find a description of the game of Blackjack in Exercise 5.5. Add the following rule to that description: If a player takes five cards without going over 21, that player wins immediately. This rule is used in some casinos. For your applet, it means that you only have to allow room for five cards. You should assume that your applet is just wide enough to show five cards, and that it is tall enough to show the user's hand and the dealer's hand.

Note that the design of a GUI Blackjack game is very different from the design of the text-oriented program that you wrote for Exercise 5.5. The user should play the game by clicking on "Hit" and "Stand" buttons. There should be a "New Game" button that can be used to start another game after one game ends. You have to decide what happens when each of these buttons is pressed. You don't have much chance of getting this right unless you think in terms of the states that the game can be in and how the state can change.

Your program will need the classes defined in Card.java, Hand.java, BlackjackHand.java, and Deck.java. Here is a working version of the applet:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 7

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 7 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 7.1: Exercise 5.2 involved a class, StatCalc.java, that could compute some statistics of a set of numbers. Write an applet that uses the StatCalc class to compute and display statistics of numbers entered by the user. The applet will have an instance variable of type StatCalc that does the computations. The applet should include a JTextField where the user enters a number. It should have four labels that display four statistics for the numbers that have been entered: the number of numbers, the sum, the mean, and the standard deviation. Every time the user enters a new number, the statistics displayed on the labels should change. The user enters a number by typing it into the JTextField and pressing return. There should be a "Clear" button that clears out all the data. This means creating a new StatCalc object and resetting the displays on the labels. My applet also has an "Enter" button that does the same thing as pressing the return key in the JTextField. (Recall that a JTextField generates an ActionEvent when the user presses return, so your applet should register itself to listen for ActionEvents from the JTextField.) Here is my solution to this problem:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.2: Write an applet with a JTextArea where the user can enter some text. The applet should have a button. When the user clicks on the button, the applet should count the number of lines in the user's input, the number of words in the user's input, and the number of characters in the user's input. This information should be displayed on three labels in the applet. Recall that if textInput is a JTextArea, then you can get the contents of the JTextArea by calling the function textInput.getText(). This function returns a String containing all the text from the JTextArea. The number of characters is just the length of this String. Lines in the String are separated by the new line character, '\n', so the number of lines is just the number of new line characters in the String, plus one. Words are a little harder to count. Exercise 3.4 has some advice about finding the words in a String. Essentially, you want to count the number of characters that are first characters in words. Don't forget to put your JTextArea in a JScrollPane. Scrollbars should appear when the user types more text than will fit in the available area. Here is my applet:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.3: The RGBColorChooser applet lets the user set the red, green, and blue levels in a color by manipulating sliders. Something like this could make a useful custom component. Such a component could be included in a program to allow the user to specify a drawing color, for example. Rewrite the RGBColorChooser as a component. Make it a subclass of JPanel instead of JApplet. Instead of doing the initialization in an init() method, you'll have to do it in a constructor. The component should have a method, getColor(), that returns the color currently displayed on the component. It should also have a method, setColor(Color c), to set the color to a specified value. Both these methods would be useful to a program that uses your component.

In order to write the setColor(Color c) method, you need to know that if c is a variable of type Color, then c.getRed() is a function that returns an integer in the range 0 to 255 that gives the red level of the color. Similarly, the functions c.getGreen() and c.getBlue() return the blue and green components.

Test your component by using it in a simple applet that sets the component to a random color when the user clicks on a button, like this one:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.4: In the Blackjack game BlackjackGUI.java from Exercise 6.8, the user can click on the "Hit", "Stand", and "NewGame" buttons even when it doesn't make sense to do so. It would be better if the buttons were disabled at the appropriate times. The "New Game" button should be disabled when there is a game in progress. The "Hit" and "Stand" buttons should be disabled when there is not a game in progress. The instance variable gameInProgress tells whether or not a game is in progress, so you just have to make sure that the buttons are properly enabled and disabled whenever this variable changes value. Make this change in the Blackjack program. This applet uses a nested class, BlackjackCanvas, to represent the board. You'll have to do most of your work in that class. In order to manipulate the buttons, you will have to use instance variables to refer to the buttons.

I strongly advise writing a subroutine that can be called whenever it is necessary to set the value of the gameInProgress variable. Then the subroutine can take responsibility for enabling and disabling the buttons. Recall that if bttn is a variable of type JButton, then bttn.setEnabled(false) disables the button and bttn.setEnabled(true) enables the button.

See the solution! [A working applet can be found here.]

--------------------------------------------------------------------------------

Exercise 7.5: Building on your solution to the preceding exercise, make it possible for the user to place bets on the Blackjack game. When the applet starts, give the user $100. Add a JTextField to the strip of controls along the bottom of the applet. The user can enter the bet in this JTextField. When the game begins, check the amount of the bet. You should do this when the game begins, not when it ends, because several errors can occur: The contents of the JTextField might not be a legal number. The bet that the user places might be more money than the user has, or it might be <= 0. You should detect these errors and show an error message instead of starting the game. The user's bet should be an integral number of dollars. You can convert the user's input into an integer, and check for illegal, non-numeric input, with a try...catch statement of the form

try {

betAmount = Integer.parseInt( betInput.getText() );

}

catch (NumberFormatException e) {

. . . // The input is not a number.

// Respond by showing an error message and

// exiting from the doNewGame() method.

}

It would be a good idea to make the JTextField uneditable while the game is in progress. If betInput is the JTextField, you can make it editable and uneditable by the user with the commands betAmount.setEditable(true) and betAmount.setEditable(false).

In the paintComponent() method, you should include commands to display the amount of money that the user has left.

There is one other thing to think about: The applet should not start a new game when it is first created. The user should have a chance to set a bet amount before the game starts. So, in the constructor for the canvas class, you should not call doNewGame(). You might want to display a message such as "Welcome to Blackjack" before the first game starts.

See the solution! [A working applet can be found here.]

--------------------------------------------------------------------------------

Exercise 7.6: The StopWatch component from Section 7.4 displays the text "Timing..." when the stop watch is running. It would be nice if it displayed the elapsed time since the stop watch was started. For that, you need to create a Timer. Add a Timer to the original source code, StopWatch.java, to display the elapsed time in seconds. Create the timer in the mousePressed() routine when the stop watch is started. Stop the timer in the mousePressed() routine when the stop watch is stopped. The elapsed time won't be very accurate anyway, so just show the integral number of seconds. You only need to set the text a few times per second. For my Timer method, I use a delay of 100 milliseconds for the timer. Here is an applet that tests my solution to this exercise:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.7: The applet at the end of Section 7.7 shows animations of moving symmetric patterns that look something like the image in a kaleidescope. Symmetric patterns are pretty. Make the SimplePaint3 applet do symmetric, kaleidoscopic patterns. As the user draws a figure, the applet should be able to draw reflected versions of that figure to make symmetric pictures.

The applet will have several options for the type of symmetry that is displayed. The user should be able to choose one of four options from a JComboBox menu. Using the "No symmetry" option, only the figure that the user draws is shown. Using "2-way symmetry", the user's figure and its horizontal reflection are shown. Using "4-way symmetry", the two vertical reflections are added. Finally, using "8-way symmetry", the four diagonal reflections are also added. Formulas for computing the reflections are given below.

The source code SimplePaint3.java already has a drawFigure() subroutine that draws all the figures. You can add a putMultiFigure() routine to draw a figure and some or all of its reflections. putMultiFigure should call the existing drawFigure to draw the figure and any necessary reflections. It decides which reflections to draw based on the setting of the symmetry menu. Where the mousePressed, mouseDragged, and mouseReleased methods call drawFigure, they should call putMultiFigure instead. The source code also has a repaintRect() method that calls repaint() on a rectangle that contains two given points. You can treat this in the same way as drawFigure(), adding a repaintMultiRect() that calls repaintRect() and replacing each call to repaintRect() with a call to repaintMultiRect(). Alternatively, if you are willing to let your applet be a little less efficient about repainting, you could simply replace each call to repaintRect() with a simple call to repaint(), without parameters. This just means that the applet will redraw a larger area than it really needs to.

If (x,y) is a point in a component that is width pixels wide and height pixels high, then the reflections of this point are obtained as follows:

The horizontal reflection is (width - x, y)

The two vertical reflections are (x, height - y) and (width - x, height - y)

To get the four diagonal reflections, first compute the diagonal reflection of (x,y) as

a = (int)( ((double)y / height) \* width );

b = (int)( ((double)x / width) \* height );

Then use the horizontal and vertical reflections of the point (a,b):

(a, b)

(width - a, b)

(a, height - b)

(width - a, height - b)

(The diagonal reflections are harder than they would be if the canvas were square. Then the height would equal the width, and the reflection of (x,y) would just be (y,x).)

To reflect a figure determined by two points, (x1,y1) and (x2,y2), compute the reflections of both points to get the reflected figure.

This is really not so hard. The changes you have to make to the source code are not as long as the explanation I have given here.

Here is my applet. Don't forget to try it with the symmetry menu set to "8-way Symmetry"!

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.8: Turn your applet from the previous exercise into a stand-alone application that runs as a JFrame. (If you didn't do the previous exercise, you can do this exercise with the original SimplePaint3.java.) To make the exercise more interesting, remove the JButtons and JComboBoxes and replace them with a menubar at the top of the frame. You can design the menus any way you like, but you should have at least the same functionality as in the original program.

As an improvement, you might add an "Undo" command. When the user clicks on the "Undo" button, the previous drawing operation will be undone. This just means returning to the image as it was before the drawing operation took place. This is easy to implement, as long as we allow just one operation to be undone. When the off-screen canvas, OSI, is created, make a second off-screen canvas, undoBuffer, of the same size. Before starting any drawing operation, copy the image from OSI to undoBuffer. You can do this with the commands

Graphics undoGr = undoBuffer.getGraphics();

undoGr.drawImage(OSI, 0, 0, null);

When the user clicks "Undo", just swap the values of OSI and undoBuffer and repaint. The previous image will appear on the screen. Clicking on "Undo" again will "undo the undo."

As another improvement, you could make it possible for the user to select a drawing color using a JColorChooser dialog box.

Here is a button that opens my program in its own window. (You don't have to write an applet to launch your frame. Just create the frame in the program's main() routine.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 8

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 8 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 8.1: An example in Section 8.2 tried to answer the question, How many random people do you have to select before you find a duplicate birthday? The source code for that program can be found in the file BirthdayProblemDemo.java. Here are some related questions:

How many random people do you have to select before you find three people who share the same birthday? (That is, all three people were born on the same day in the same month, but not necessarily in the same year.)

Suppose you choose 365 people at random. How many different birthdays will they have? (The number could theoretically be anywhere from 1 to 365).

How many different people do you have to check before you've found at least one person with a birthday on each of the 365 days of the year?

Write three programs to answer these questions. Like the example program, BirthdayProblemDemo, each of your programs should simulate choosing people at random and checking their birthdays. (In each case, ignore the possibility of leap years.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 8.2: Write a program that will read a sequence of positive real numbers entered by the user and will print the same numbers in sorted order from smallest to largest. The user will input a zero to mark the end of the input. Assume that at most 100 positive numbers will be entered.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 8.3: A polygon is a geometric figure made up of a sequence of connected line segments. The points where the line segments meet are called the vertices of the polygon. The Graphics class includes commands for drawing and filling polygons. For these commands, the coordinates of the vertices of the polygon are stored in arrays. If g is a variable of type Graphics then

g.drawPolygon(xCoords, yCoords, pointCt) will draw the outline of the polygon with vertices at (xCoords[0],yCoords[0]), (xCoords[1],yCoords[1]), ..., (xCoords[pointCt-1],yCoords[pointCt-1]). The third parameter, pointCt, is an int that specifies the number of vertices of the polygon. Its value should be 3 or greater. The first two parameters are arrays of type int[]. Note that the polygon automatically includes a line from the last point, (xCoords[pointCt-1],yCoords[pointCt-1]), back to the starting point (xCoords[0],yCoords[0]).

g.fillPolygon(xCoords, yCoords, pointCt) fills the interior of the polygon with the current drawing color. The parameters have the same meaning as in the drawPolygon() method. Note that it is OK for the sides of the polygon to cross each other, but the interior of a polygon with self-intersections might not be exactly what you expect.

Write a little applet that lets the user draw polygons. As the user clicks a sequence of points, count them and store their x- and y-coordinates in two arrays. These points will be the vertices of the polygon. Also, draw a line between each consecutive pair of points to give the user some visual feedback. When the user clicks near the starting point, draw the complete polygon. Draw it with a red interior and a black border. The user should then be able to start drawing a new polygon. When the user shift-clicks on the applet, clear it.

There is no need to store information about the contents of the applet. The paintComponent() method can just draw a border around the applet. The lines and polygons can be drawn using a graphics context, g, obtained with the command "g = getGraphics();".

You can try my solution. Note that as the user is drawing the polygon, lines are drawn between the points that the user clicks. Click within two pixels of the starting point to see a filled polygon.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 8.4: For this problem, you will need to use an array of objects. The objects belong to the class MovingBall, which I have already written. You can find the source code for this class in the file MovingBall.java. A MovingBall represents a circle that has an associated color, radius, direction, and speed. It is restricted to moving in a rectangle in the (x,y) plane. It will "bounce back" when it hits one of the sides of this rectangle. A MovingBall does not actually move by itself. It's just a collection of data. You have to call instance methods to tell it to update its position and to draw itself. The constructor for the MovingBall class takes the form

new MovingBall(xmin, xmax, ymin, ymax)

where the parameters are integers that specify the limits on the x and y coordinates of the ball. In this exercise, you will want balls to bounce off the sides of the applet, so you will create them with the constructor call "new MovingBall(0, getWidth(), 0, getHeight())". The constructor creates a ball that initially is colored red, has a radius of 5 pixels, is located at the center of its range, has a random speed between 4 and 12, and is headed in a random direction. If ball is a variable of type MovingBall, then the following methods are available:

ball.draw(g) -- draw the ball in a graphics context. The parameter, g, must be of type Graphics. (The drawing color in g will be changed to the color of the ball.)

ball.travel() -- change the (x,y)-coordinates of the ball by an amount equal to its speed. The ball has a certain direction of motion, and the ball is moved in that direction. Ordinarily, you will call this once for each frame of an animation, so the speed is given in terms of "pixels per frame". Calling this routine does not move the ball on the screen. It just changes the values of some instance variables in the object. The next time the object's draw() method is called, the ball will be drawn in the new position.

ball.headTowards(x,y) -- change the direction of motion of the ball so that it is headed towards the point (x,y). This does not affect the speed.

These are the methods that you will need for this exercise. There are also methods for setting various properties of the ball, such as ball.setColor(color) for changing the color and ball.setRadius(radius) for changing its size. See the source code for more information.

For this exercise, you should create an applet that shows an animation of 25 balls bouncing around on a black background. Your applet can be defined as a subclass of SimpleAnimationApplet2, which was first introduced in Section 3.7. The drawFrame() method in your applet should move all the balls and draw them. (Alternatively, if you have read Chapter 7, you can program the animation yourself using a Timer.) Use an array of type MovingBall[] to hold the 25 balls.

In addition, your applet should implement the MouseListener and MouseMotionListener interfaces. When the user presses the mouse or drags the mouse, call each of the ball's headTowards() methods to make the balls head towards the mouse's location.

Here is my solution. Try clicking and dragging on the applet:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 8.5: The game of Go Moku (also known as Pente or Five Stones) is similar to Tic-Tac-Toe, except that it played on a much larger board and the object is to get five squares in a row rather than three. Players take turns placing pieces on a board. A piece can be placed in any empty square. The first player to get five pieces in a row -- horizontally, vertically, or diagonally -- wins. If all squares are filled before either player wins, then the game is a draw. Write an applet that lets two players play Go Moku against each other.

Your applet will be simpler than the Checkers applet from Section 8.5. Play alternates strictly between the two players, and there is no need to hilite the legal moves. You will only need two classes, a short applet class to set up the applet and a Board class to draw the board and do all the work of the game. Nevertheless, you will probably want to look at the source code for the checkers applet, Checkers.java, for ideas about the general outline of the program.

The hardest part of the program is checking whether the move that a player makes is a winning move. To do this, you have to look in each of the four possible directions from the square where the user has placed a piece. You have to count how many pieces that player has in a row in that direction. If the number is five or more in any direction, then that player wins. As a hint, here is part of the code from my applet. This code counts the number of pieces that the user has in a row in a specified direction. The direction is specified by two integers, dirX and dirY. The values of these variables are 0, 1, or -1, and at least one of them is non-zero. For example, to look in the horizontal direction, dirX is 1 and dirY is 0.

int ct = 1; // Number of pieces in a row belonging to the player.

int r, c; // A row and column to be examined.

r = row + dirX; // Look at square in specified direction.

c = col + dirY;

while ( r >= 0 && r < 13 && c >= 0 && c < 13

&& board[r][c] == player ) {

// Square is on the board, and it

// contains one of the players's pieces.

ct++;

r += dirX; // Go on to next square in this direction.

c += dirY;

}

r = row - dirX; // Now, look in the opposite direction.

c = col - dirY;

while ( r >= 0 && r < 13 && c >= 0 && c < 13

&& board[r][c] == player ) {

ct++;

r -= dirX; // Go on to next square in this direction.

c -= dirY;

}

Here is my applet. It uses a 13-by-13 board. You can do the same or use a normal 8-by-8 checkerboard.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 7

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 7 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 7.1: Exercise 5.2 involved a class, StatCalc.java, that could compute some statistics of a set of numbers. Write an applet that uses the StatCalc class to compute and display statistics of numbers entered by the user. The applet will have an instance variable of type StatCalc that does the computations. The applet should include a JTextField where the user enters a number. It should have four labels that display four statistics for the numbers that have been entered: the number of numbers, the sum, the mean, and the standard deviation. Every time the user enters a new number, the statistics displayed on the labels should change. The user enters a number by typing it into the JTextField and pressing return. There should be a "Clear" button that clears out all the data. This means creating a new StatCalc object and resetting the displays on the labels. My applet also has an "Enter" button that does the same thing as pressing the return key in the JTextField. (Recall that a JTextField generates an ActionEvent when the user presses return, so your applet should register itself to listen for ActionEvents from the JTextField.) Here is my solution to this problem:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.2: Write an applet with a JTextArea where the user can enter some text. The applet should have a button. When the user clicks on the button, the applet should count the number of lines in the user's input, the number of words in the user's input, and the number of characters in the user's input. This information should be displayed on three labels in the applet. Recall that if textInput is a JTextArea, then you can get the contents of the JTextArea by calling the function textInput.getText(). This function returns a String containing all the text from the JTextArea. The number of characters is just the length of this String. Lines in the String are separated by the new line character, '\n', so the number of lines is just the number of new line characters in the String, plus one. Words are a little harder to count. Exercise 3.4 has some advice about finding the words in a String. Essentially, you want to count the number of characters that are first characters in words. Don't forget to put your JTextArea in a JScrollPane. Scrollbars should appear when the user types more text than will fit in the available area. Here is my applet:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.3: The RGBColorChooser applet lets the user set the red, green, and blue levels in a color by manipulating sliders. Something like this could make a useful custom component. Such a component could be included in a program to allow the user to specify a drawing color, for example. Rewrite the RGBColorChooser as a component. Make it a subclass of JPanel instead of JApplet. Instead of doing the initialization in an init() method, you'll have to do it in a constructor. The component should have a method, getColor(), that returns the color currently displayed on the component. It should also have a method, setColor(Color c), to set the color to a specified value. Both these methods would be useful to a program that uses your component.

In order to write the setColor(Color c) method, you need to know that if c is a variable of type Color, then c.getRed() is a function that returns an integer in the range 0 to 255 that gives the red level of the color. Similarly, the functions c.getGreen() and c.getBlue() return the blue and green components.

Test your component by using it in a simple applet that sets the component to a random color when the user clicks on a button, like this one:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.4: In the Blackjack game BlackjackGUI.java from Exercise 6.8, the user can click on the "Hit", "Stand", and "NewGame" buttons even when it doesn't make sense to do so. It would be better if the buttons were disabled at the appropriate times. The "New Game" button should be disabled when there is a game in progress. The "Hit" and "Stand" buttons should be disabled when there is not a game in progress. The instance variable gameInProgress tells whether or not a game is in progress, so you just have to make sure that the buttons are properly enabled and disabled whenever this variable changes value. Make this change in the Blackjack program. This applet uses a nested class, BlackjackCanvas, to represent the board. You'll have to do most of your work in that class. In order to manipulate the buttons, you will have to use instance variables to refer to the buttons.

I strongly advise writing a subroutine that can be called whenever it is necessary to set the value of the gameInProgress variable. Then the subroutine can take responsibility for enabling and disabling the buttons. Recall that if bttn is a variable of type JButton, then bttn.setEnabled(false) disables the button and bttn.setEnabled(true) enables the button.

See the solution! [A working applet can be found here.]

--------------------------------------------------------------------------------

Exercise 7.5: Building on your solution to the preceding exercise, make it possible for the user to place bets on the Blackjack game. When the applet starts, give the user $100. Add a JTextField to the strip of controls along the bottom of the applet. The user can enter the bet in this JTextField. When the game begins, check the amount of the bet. You should do this when the game begins, not when it ends, because several errors can occur: The contents of the JTextField might not be a legal number. The bet that the user places might be more money than the user has, or it might be <= 0. You should detect these errors and show an error message instead of starting the game. The user's bet should be an integral number of dollars. You can convert the user's input into an integer, and check for illegal, non-numeric input, with a try...catch statement of the form

try {

betAmount = Integer.parseInt( betInput.getText() );

}

catch (NumberFormatException e) {

. . . // The input is not a number.

// Respond by showing an error message and

// exiting from the doNewGame() method.

}

It would be a good idea to make the JTextField uneditable while the game is in progress. If betInput is the JTextField, you can make it editable and uneditable by the user with the commands betAmount.setEditable(true) and betAmount.setEditable(false).

In the paintComponent() method, you should include commands to display the amount of money that the user has left.

There is one other thing to think about: The applet should not start a new game when it is first created. The user should have a chance to set a bet amount before the game starts. So, in the constructor for the canvas class, you should not call doNewGame(). You might want to display a message such as "Welcome to Blackjack" before the first game starts.

See the solution! [A working applet can be found here.]

--------------------------------------------------------------------------------

Exercise 7.6: The StopWatch component from Section 7.4 displays the text "Timing..." when the stop watch is running. It would be nice if it displayed the elapsed time since the stop watch was started. For that, you need to create a Timer. Add a Timer to the original source code, StopWatch.java, to display the elapsed time in seconds. Create the timer in the mousePressed() routine when the stop watch is started. Stop the timer in the mousePressed() routine when the stop watch is stopped. The elapsed time won't be very accurate anyway, so just show the integral number of seconds. You only need to set the text a few times per second. For my Timer method, I use a delay of 100 milliseconds for the timer. Here is an applet that tests my solution to this exercise:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.7: The applet at the end of Section 7.7 shows animations of moving symmetric patterns that look something like the image in a kaleidescope. Symmetric patterns are pretty. Make the SimplePaint3 applet do symmetric, kaleidoscopic patterns. As the user draws a figure, the applet should be able to draw reflected versions of that figure to make symmetric pictures.

The applet will have several options for the type of symmetry that is displayed. The user should be able to choose one of four options from a JComboBox menu. Using the "No symmetry" option, only the figure that the user draws is shown. Using "2-way symmetry", the user's figure and its horizontal reflection are shown. Using "4-way symmetry", the two vertical reflections are added. Finally, using "8-way symmetry", the four diagonal reflections are also added. Formulas for computing the reflections are given below.

The source code SimplePaint3.java already has a drawFigure() subroutine that draws all the figures. You can add a putMultiFigure() routine to draw a figure and some or all of its reflections. putMultiFigure should call the existing drawFigure to draw the figure and any necessary reflections. It decides which reflections to draw based on the setting of the symmetry menu. Where the mousePressed, mouseDragged, and mouseReleased methods call drawFigure, they should call putMultiFigure instead. The source code also has a repaintRect() method that calls repaint() on a rectangle that contains two given points. You can treat this in the same way as drawFigure(), adding a repaintMultiRect() that calls repaintRect() and replacing each call to repaintRect() with a call to repaintMultiRect(). Alternatively, if you are willing to let your applet be a little less efficient about repainting, you could simply replace each call to repaintRect() with a simple call to repaint(), without parameters. This just means that the applet will redraw a larger area than it really needs to.

If (x,y) is a point in a component that is width pixels wide and height pixels high, then the reflections of this point are obtained as follows:

The horizontal reflection is (width - x, y)

The two vertical reflections are (x, height - y) and (width - x, height - y)

To get the four diagonal reflections, first compute the diagonal reflection of (x,y) as

a = (int)( ((double)y / height) \* width );

b = (int)( ((double)x / width) \* height );

Then use the horizontal and vertical reflections of the point (a,b):

(a, b)

(width - a, b)

(a, height - b)

(width - a, height - b)

(The diagonal reflections are harder than they would be if the canvas were square. Then the height would equal the width, and the reflection of (x,y) would just be (y,x).)

To reflect a figure determined by two points, (x1,y1) and (x2,y2), compute the reflections of both points to get the reflected figure.

This is really not so hard. The changes you have to make to the source code are not as long as the explanation I have given here.

Here is my applet. Don't forget to try it with the symmetry menu set to "8-way Symmetry"!

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 7.8: Turn your applet from the previous exercise into a stand-alone application that runs as a JFrame. (If you didn't do the previous exercise, you can do this exercise with the original SimplePaint3.java.) To make the exercise more interesting, remove the JButtons and JComboBoxes and replace them with a menubar at the top of the frame. You can design the menus any way you like, but you should have at least the same functionality as in the original program.

As an improvement, you might add an "Undo" command. When the user clicks on the "Undo" button, the previous drawing operation will be undone. This just means returning to the image as it was before the drawing operation took place. This is easy to implement, as long as we allow just one operation to be undone. When the off-screen canvas, OSI, is created, make a second off-screen canvas, undoBuffer, of the same size. Before starting any drawing operation, copy the image from OSI to undoBuffer. You can do this with the commands

Graphics undoGr = undoBuffer.getGraphics();

undoGr.drawImage(OSI, 0, 0, null);

When the user clicks "Undo", just swap the values of OSI and undoBuffer and repaint. The previous image will appear on the screen. Clicking on "Undo" again will "undo the undo."

As another improvement, you could make it possible for the user to select a drawing color using a JColorChooser dialog box.

Here is a button that opens my program in its own window. (You don't have to write an applet to launch your frame. Just create the frame in the program's main() routine.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 9

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 9 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 9.1: Write a program that uses the following subroutine, from Section 3, to solve equations specified by the user.

static double root(double A, double B, double C)

throws IllegalArgumentException {

// Returns the larger of the two roots of

// the quadratic equation A\*x\*x + B\*x + C = 0.

// (Throws an exception if A == 0 or B\*B-4\*A\*C < 0.)

if (A == 0) {

throw new IllegalArgumentException("A can't be zero.");

}

else {

double disc = B\*B - 4\*A\*C;

if (disc < 0)

throw new IllegalArgumentException("Discriminant < zero.");

return (-B + Math.sqrt(disc)) / (2\*A);

}

}

Your program should allow the user to specify values for A, B, and C. It should call the subroutine to compute a solution of the equation. If no error occurs, it should print the root. However, if an error occurs, your program should catch that error and print an error message. After processing one equation, the program should ask whether the user wants to enter another equation. The program should continue until the user answers no.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 9.2: As discussed in Section 1, values of type int are limited to 32 bits. Integers that are too large to be represented in 32 bits cannot be stored in an int variable. Java has a standard class, java.math.BigInteger, that addresses this problem. An object of type BigInteger is an integer that can be arbitrarily large. (The maximum size is limited only by the amount of memory on your computer.) Since BigIntegers are objects, they must be manipulated using instance methods from the BigInteger class. For example, you can't add two BigIntegers with the + operator. Instead, if N and M are variables that refer to BigIntegers, you can compute the sum of N and M with the function call N.add(M). The value returned by this function is a new BigInteger object that is equal to the sum of N and M.

The BigInteger class has a constructor new BigInteger(str), where str is a string. The string must represent an integer, such as "3" or "39849823783783283733". If the string does not represent a legal integer, then the constructor throws a NumberFormatException.

There are many instance methods in the BigInteger class. Here are a few that you will find useful for this exercise. Assume that N and M are variables of type BigInteger.

N.add(M) -- a function that returns a BigInteger representing the sum of N and M.

N.multiply(M) -- a function that returns a BigInteger representing the result of multiplying N times M.

N.divide(M) -- a function that returns a BigInteger representing the result of dividing N by M.

N.signum() -- a function that returns an ordinary int. The returned value represents the sign of the integer N. The returned value is 1 if N is greater than zero. It is -1 if N is less than zero. And it is 0 if N is zero.

N.equals(M) -- a function that returns a boolean value that is true if N and M have the same integer value.

N.toString() -- a function that returns a String representing the value of N.

N.testBit(k) -- a function that returns a boolean value. The parameter k is an integer. The return value is true if the k-th bit in N is 1, and it is false if the k-th bit is 0. Bits are numbered from right to left, starting with 0. Testing "if (N.testBit(0))" is an easy way to check whether N is even or odd. N.testBit(0) is true if and only if N is an odd number.

For this exercise, you should write a program that prints 3N+1 sequences with starting values specified by the user. In this version of the program, you should use BigIntegers to represent the terms in the sequence. You can read the user's input into a String with the TextIO.getln() function. Use the input value to create the BigInteger object that represents the starting point of the 3N+1 sequence. Don't forget to catch and handle the NumberFormatException that will occur if the user's input is not a legal integer! You should also check that the input number is greater than zero.

If the user's input is legal, print out the 3N+1 sequence. Count the number of terms in the sequence, and print the count at the end of the sequence. Exit the program when the user inputs an empty line.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 9.3: A Roman numeral represents an integer using letters. Examples are XVII to represent 17, MCMLIII for 1953, and MMMCCCIII for 3303. By contrast, ordinary numbers such as 17 or 1953 are called Arabic numerals. The following table shows the Arabic equivalent of all the single-letter Roman numerals:

M 1000 X 10

D 500 V 5

C 100 I 1

L 50

When letters are strung together, the values of the letters are just added up, with the following exception. When a letter of smaller value is followed by a letter of larger value, the smaller value is subtracted from the larger value. For example, IV represents 5 - 1, or 4. And MCMXCV is interpreted as M + CM + XC + V, or 1000 + (1000 - 100) + (100 - 10) + 5, which is 1995. In standard Roman numerals, no more than thee consecutive copies of the same letter are used. Following these rules, every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

M 1000 X 10

CM 900 IX 9

D 500 V 5

CD 400 IV 4

C 100 I 1

XC 90

L 50

XL 40

Write a class to represent Roman numerals. The class should have two constructors. One constructs a Roman numeral from a string such as "XVII" or "MCMXCV". It should throw a NumberFormatException if the string is not a legal Roman numeral. The other constructor constructs a Roman numeral from an int. It should throw a NumberFormatException if the int is outside the range 1 to 3999.

In addition, the class should have two instance methods. The method toString() returns the string that represents the Roman numeral. The method toInt() returns the value of the Roman numeral as an int.

At some point in your class, you will have to convert an int into the string that represents the corresponding Roman numeral. One way to approach this is to gradually "move" value from the Arabic numeral to the Roman numeral. Here is the beginning of a routine that will do this, where number is the int that is to be converted:

String roman = "";

int N = number;

while (N >= 1000) {

// Move 1000 from N to roman.

roman += "M";

N -= 1000;

}

while (N >= 900) {

// Move 900 from N to roman.

roman += "CM";

N -= 900;

}

.

. // Continue with other values from the above table.

.

(You can save yourself a lot of typing in this routine if you use arrays in a clever way to represent the data in the above table.)

Once you've written your class, use it in a main program that will read both Arabic numerals and Roman numerals entered by the user. If the user enters an Arabic numeral, print the corresponding Roman numeral. If the user enters a Roman numeral, print the corresponding Arabic numeral. (You can tell the difference by using TextIO.peek() to peek at the first character in the user's input. If that character is a digit, then the user's input is an Arabic numeral. Otherwise, it's a Roman numeral.) The program should end when the user inputs an empty line. Here is an applet that simulates my solution to this problem:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 9.4: The file Expr.java defines a class, Expr, that can be used to represent mathematical expressions involving the variable x. The expression can use the operators +, -, \*, /, and ^, where ^ represents the operation of raising a number to a power. It can use mathematical functions such as sin, cos, abs, and ln. See the source code file for full details. The Expr class uses some advanced techniques which have not yet been covered in this textbook. However, the interface is easy to understand. It contains only a constructor and two public methods.

The constructor new Expr(def) creates an Expr object defined by a given expression. The parameter, def, is a string that contains the definition. For example, new Expr("x^2") or new Expr("sin(x)+3\*x"). If the parameter in the constructor call does not represent a legal expression, then the constructor throws an IllegalArgumentException. The message in the exception describes the error.

If func is a variable of type Expr and num is of type double, then func.value(num) is a function that returns the value of the expression when the number num is substituted for the variable x in the expression. For example, if Expr represents the expression 3\*x+1, then func.value(5) is 3\*5+1, or 16. If the expression is undefined for the specified value of x, then the special value Double.NaN is returned.

Finally, func.getDefinition() returns the definition of the expression. This is just the string that was used in the constructor that created the expression object.

For this exercise, you should write a program that lets the user enter an expression. If the expression contains an error, print an error message. Otherwise, let the user enter some numerical values for the variable x. Print the value of the expression for each number that the user enters. However, if the expression is undefined for the specified value of x, print a message to that effect. You can use the boolean-valued function Double.isNaN(val) to check whether a number, val, is Double.NaN.

The user should be able to enter as many values of x as desired. After that, the user should be able to enter a new expression. Here is an applet that simulates my solution to this exercise, so that you can see how it works:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 9.5: This exercises uses the class Expr, which was described in Exercise 9.4. For this exercise, you should write an applet that can graph a function, f(x), whose definition is entered by the user. The applet should have a text-input box where the user can enter an expression involving the variable x, such as x^2 or sin(x-3)/x. This expression is the definition of the function. When the user presses return in the text input box, the applet should use the contents of the text input box to construct an object of type Expr. If an error is found in the definition, then the applet should display an error message. Otherwise, it should display a graph of the function. (Note: A JTextField generates an ActionEvent when the user presses return.)

The applet will need a JPanel for displaying the graph. To keep things simple, this panel should represent a fixed region in the xy-plane, defined by -5 <= x <= 5 and -5 <= y <= 5. To draw the graph, compute a large number of points and connect them with line segments. (This method does not handle discontinuous functions properly; doing so is very hard, so you shouldn't try to do it for this exercise.) My applet divides the interval -5 <= x <= 5 into 300 subintervals and uses the 301 endpoints of these subintervals for drawing the graph. Note that the function might be undefined at one of these x-values. In that case, you have to skip that point.

A point on the graph has the form (x,y) where y is obtained by evaluating the user's expression at the given value of x. You will have to convert these real numbers to the integer coordinates of the corresponding pixel on the canvas. The formulas for the conversion are:

a = (int)( (x + 5)/10 \* width );

b = (int)( (5 - y)/10 \* height );

where a and b are the horizontal and vertical coordinates of the pixel, and width and height are the width and height of the canvas.

Here is my solution to this exercise:

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 10

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 10 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 10.1: The WordList program from Section 10.3 reads a text file and makes an alphabetical list of all the words in that file. The list of words is output to another file. Improve the program so that it also keeps track of the number of times that each word occurs in the file. Write two lists to the output file. The first list contains the words in alphabetical order. The number of times that the word occurred in the file should be listed along with the word. Then write a second list to the output file in which the words are sorted according to the number of times that they occurred in the files. The word that occurred most often should be listed first.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 10.2: Write a program that will count the number of lines in each file that is specified on the command line. Assume that the files are text files. Note that multiple files can be specified, as in "java LineCounts file1.txt file2.txt file3.txt". Write each file name, along with the number of lines in that file, to standard output. If an error occurs while trying to read from one of the files, you should print an error message for that file, but you should still process all the remaining files.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 10.3: Section 8.4 presented a PhoneDirectory class as an example. A PhoneDirectory holds a list of names and associated phone numbers. But a phone directory is pretty useless unless the data in the directory can be saved permanently -- that is, in a file. Write a phone directory program that keeps its list of names and phone numbers in a file. The user of the program should be able to look up a name in the directory to find the associated phone number. The user should also be able to make changes to the data in the directory. Every time the program starts up, it should read the data from the file. Before the program terminates, if the data has been changed while the program was running, the file should be re-written with the new data. Designing a user interface for the program is part of the exercise.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 10.4: For this exercise, you will write a network server program. The program is a simple file server that makes a collection of files available for transmission to clients. When the server starts up, it needs to know the name of the directory that contains the collection of files. This information can be provided as a command-line argument. You can assume that the directory contains only regular files (that is, it does not contain any sub-directories). You can also assume that all the files are text files.

When a client connects to the server, the server first reads a one-line command from the client. The command can be the string "index". In this case, the server responds by sending a list of names of all the files that are available on the server. Or the command can be of the form "get <file>", where <file> is a file name. The server checks whether the requested file actually exists. If so, it first sends the word "ok" as a message to the client. Then it sends the contents of the file and closes the connection. Otherwise, it sends the word "error" to the client and closes the connection.

Ideally, your server should start a separate thread to handle each connection request. However, if you don't want to deal with threads you can just call a subroutine to handle the request. See the DirectoryList example in Section 10.2 for help with the problem of getting the list of files in the directory.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 10.5: Write a client program for the server from Exercise 10.4. Design a user interface that will let the user do at least two things: Get a list of files that are available on the server and display the list on standard output. Get a copy of a specified file from the server and save it to a local file (on the computer where the client is running).

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 11

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 11 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 11.1: The DirectoryList program, given as an example at the end of Section 10.2, will print a list of files in a directory specified by the user. But some of the files in that directory might themselves be directories. And the subdirectories can themselves contain directories. And so on. Write a modified version of DirectoryList that will list all the files in a directory and all its subdirectories, to any level of nesting. You will need a recursive subroutine to do the listing. The subroutine should have a parameter of type File. You will need the constructor from the File class that has the form

public File( File dir, String fileName )

// Constructs the File object representing a file

// named fileName in the directory specified by dir.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 11.2: Make a new version of the sample program WordList.java, from Section 10.3, that stores words in a binary sort tree instead of in an array.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 11.3: Suppose that linked lists of integers are made from objects belonging to the class

class ListNode {

int item; // An item in the list.

ListNode next; // Pointer to the next node in the list.

}

Write a subroutine that will make a copy of a list, with the order of the items of the list reversed. The subroutine should have a parameter of type ListNode, and it should return a value of type ListNode. The original list should not be modified.

You should also write a main() routine to test your subroutine.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 11.4: Section 11.4 explains how to use recursion to print out the items in a binary tree in various orders. That section also notes that a non-recursive subroutine can be used to print the items, provided that a stack or queue is used as an auxiliary data structure. Assuming that a queue is used, here is an algorithm for such a subroutine:

Add the root node to an empty queue

while the queue is not empty:

Get a node from the queue

Print the item in the node

if node.left is not null:

add it to the queue

if node.right is not null:

add it to the queue

Write a subroutine that implements this algorithm, and write a program to test the subroutine. Note that you will need a queue of TreeNodes, so you will need to write a class to represent such queues.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 11.5: In Section 11.4, I say that "if the [binary sort] tree is created randomly, there is a high probability that the tree is approximately balanced." For this exercise, you will do an experiment to test whether that is true.

The depth of a node in a binary tree is the length of the path from the root of the tree to that node. That is, the root has depth 0, its children have depth 1, its grandchildren have depth 2, and so on. In a balanced tree, all the leaves in the tree are about the same depth. For example, in a perfectly balanced tree with 1023 nodes, all the leaves are at depth 9. In an approximately balanced tree with 1023 nodes, the average depth of all the leaves should be not too much bigger than 9.

On the other hand, even if the tree is approximately balanced, there might be a few leaves that have much larger depth than the average, so we might also want to look at the maximum depth among all the leaves in a tree.

For this exercise, you should create a random binary sort tree with 1023 nodes. The items in the tree can be real numbers, and you can create the tree by generating 1023 random real numbers and inserting them into the tree, using the usual insert() method for binary sort trees. Once you have the tree, you should compute and output the average depth of all the leaves in the tree and the maximum depth of all the leaves. To do this, you will need three recursive subroutines: one to count the leaves, one to find the sum of the depths of all the leaves, and one to find the maximum depth. The latter two subroutines should have an int-valued parameter, depth, that tells how deep in the tree you've gone. When you call the routine recursively, the parameter increases by 1.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 11.6: The parsing programs in Section 11.5 work with expressions made up of numbers and operators. We can make things a little more interesting by allowing the variable "x" to occur. This would allow expression such as "3\*(x-1)\*(x+1)", for example. Make a new version of the sample program SimpleParser3.java that can work with such expressions. In your program, the main() routine can't simply print the value of the expression, since the value of the expression now depends on the value of x. Instead, it should print the value of the expression for x=0, x=1, x=2, and x=3.

The original program will have to be modified in several other ways. Currently, the program uses classes ConstNode, BinOpNode, and UnaryMinusNode to represent nodes in an expression tree. Since expressions can now include x, you will need a new class, VariableNode, to represent an occurrence of x in the expression.

In the original program, each of the node classes has an instance method, "double value()", which returns the value of the node. But in your program, the value can depend on x, so you should replace this method with one of the form "double value(double xValue)", where the parameter xValue is the value of x.

Finally, the parsing subroutines in your program will have to take into account the fact that expressions can contain x. There is just one small change in the BNF rules for the expressions: A <factor> is allowed to be the variable x:

<factor> ::= <number> | <x-variable> | "(" <expression> ")"

where <x-variable> can be either a lower case or an upper case "X". This change in the BNF requires a change in the factorTree() subroutine.

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 11.7: This exercise builds on the previous exercise, Exercise 11.6. To understand it, you should have some background in Calculus. The derivative of an expression that involves the variable x can be defined by a few recursive rules:

The derivative of a constant is 0.

The derivative of x is 1.

If A is an expression, let dA be the derivative of A. Then the derivative of -A is -dA.

If A and B are expressions, let dA be the derivative of A and let dB be the derivative of B. Then

The derivative of A+B is dA+dB.

The derivative of A-B is dA-dB.

The derivative of A\*B is A\*dB + B\*dA.

The derivative of A/B is (B\*dA - A\*dB) / (B\*B).

For this exercise, you should modify your program from the previous exercise so that it can compute the derivative of an expression. You can do this by adding a derivative-computing method to each of the node classes. First, add another abstract method to the ExpNode class:

abstract ExpNode derivative();

Then implement this method in each of the four subclasses of ExpNode. All the information that you need is in the rules given above. In your main program, you should print out the stack operations that define the derivative, instead of the operations for the original expression. Note that the formula that you get for the derivative can be much more complicated than it needs to be. For example, the derivative of 3\*x+1 will be computed as (3\*1+0\*x)+0. This is correct, even though it's kind of ugly.

As an alternative to printing out stack operations, you might want to print the derivative as a fully parenthesized expression. You can do this by adding a printInfix() routine to each node class. The problem of deciding which parentheses can be left out without altering the meaning of the expression is a fairly difficult one, which I don't advise you to attempt.

(There is one curious thing that happens here: If you apply the rules, as given, to an expression tree, the result is no longer a tree, since the same subexpression can occur at multiple points in the derivative. For example, if you build a node to represent B\*B by saying "new BinOpNode('\*',B,B)", then the left and right children of the new node are actually the same node! This is not allowed in a tree. However, the difference is harmless in this case since, like a tree, the structure that you get has no loops in it. Loops, on the other hand, would be a disaster in most of the recursive subroutines that we have written to process trees, since it would lead to infinite recursion.)

See the solution! visit this website <http://www.java2s.clanteam.com>

--------------------------------------------------------------------------------

Programming Exercises

For Chapter 12

--------------------------------------------------------------------------------

THIS PAGE CONTAINS programming exercises based on material from Chapter 12 of this on-line Java textbook. Each exercise has a link to a discussion of one possible solution of that exercise.

--------------------------------------------------------------------------------

Exercise 12.1: In Section 12.2, I mentioned that a LinkedList can be used as a queue by using the addLast() and removeFirst() methods to enqueue and dequeue items. But, if we are going to work with queues, it's better to have a Queue class. The data for the queue could still be represented as a LinkedList, but the LinkedList object would be hidden as a private instance variable in the Queue object. Use this idea to write a generic Queue class for representing queues of Objects. Also write a generic Stack class that uses either a LinkedList or an ArrayList to store its data. (Stacks and queues were introduced in Section 11.3.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 12.2: In mathematics, several operations are defined on sets. The union of two sets A and B is a set that contains all the elements that are in A together with all the elements that are in B. The intersection of A and B is the set that contains elements that are in both A and B. The difference of A and B is the set that contains all the elements of A except for those elements that are also in B.

Suppose that A and B are variables of type set in Java. The mathematical operations on A and B can be computed using methods from the Set interface. In particular: The set A.addAll(B) is the union of A and B; A.retainAll(B) is the intersection of A and B; and A.removeAll(B) is the difference of A and B. (These operations change the contents of the set A, while the mathematical operations create a new set without changing A, but that difference is not relevant to this exercise.)

For this exercise, you should write a program that can be used as a "set calculator" for simple operations on sets of non-negative integers. (Negative integers are not allowed.) A set of such integers will be represented as a list of integers, separated by commas and, optionally, spaces and enclosed in square brackets. For example: [1,2,3] or [17, 42, 9, 53,108]. The characters +, \*, and - will be used for the union, intersection, and difference operations. The user of the program will type in lines of input containing two sets, separated by an operator. The program should perform the operation and print the resulting set. Here are some examples:

Input Output

------------------------- -------------------

[1, 2, 3] + [3, 5, 7] [1, 2, 3, 5, 7]

[10,9,8,7] \* [2,4,6,8] [8]

[ 5, 10, 15, 20 ] - [ 0, 10, 20 ] [5, 15]

To represent sets of non-negative integers, use TreeSets containing objects belonging to the wrapper class Integer. Read the user's input, create two TreeSets, and use the appropriate TreeSet method to perform the requested operation on the two sets. Your program should be able to read and process any number of lines of input. If a line contains a syntax error, your program should not crash. It should report the error and move on to the next line of input. (Note: To print out a Set, A, of Integers, you can just say System.out.println(A). I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

See the solution! visit this website http://java2s.clanteam.com/

--------------------------------------------------------------------------------

Exercise 12.3: The fact that Java has a HashMap class means that no Java programmer has to write an implementation of hash tables from scratch -- unless, of course, you are a computer science student.

Write an implementation of hash tables from scratch. Define the following methods: get(key), put(key,value), remove(key), containsKey(key), and size(). Do not use any of Java's generic data structures. Assume that both keys and values are of type Object, just as for HashMaps. Every Object has a hash code, so at least you don't have to define your own hash functions. Also, you do not have to worry about increasing the size of the table when it becomes too full.

You should also write a short program to test your solution.

See the solution! visit this website <http://www.java2s.clanteam.com>

--------------------------------------------------------------------------------

Exercise 12.4: A predicate is a boolean-valued function with one parameter. Some languages use predicates in generic programming. Java doesn't, but this exercise looks at how predicates might work in Java.

In Java, we could use "predicate objects" by defining an interface:

public interface Predicate {

public boolean test(Object obj);

}

The idea is that an object that implements this interface knows how to "test" objects in some way. Define a class Predicates that contains the following generic methods for working with predicate objects:

public static void remove(Collection coll, Predicate pred)

// Remove every object, obj, from coll for which

// pred.test(obj) is true.

public static void retain(Collection coll, Predicate pred)

// Remove every object, obj, from coll for which

// pred.test(obj) is false. (That is, retain the

// objects for which the predicate is true.)

public static List collect(Collection coll, Predicate pred)

// Return a List that contains all the objects, obj,

// from the collection, coll, such that pred.test(obj)

// is true.

public static int find(ArrayList list, Predicate pred)

// Return the index of the first item in list

// for which the predicate is true, if any.

// If there is no such item, return -1.

(In C++, methods similar to these are included as a standard part of the generic programming framework.)

See the solution! visit this website <http://www.java2s.clanteam.com>

--------------------------------------------------------------------------------

Exercise 12.5: One of the examples in Section 12.4 concerns the problem of making an index for a book. A related problem is making a concordance for a document. A concordance lists every word that occurs in the document, and for each word it gives the line number of every line in the document where the word occurs. All the subroutines for creating an index that were presented in Section 12.4 can also be used to create a concordance. The only real difference is that the integers in a concordance are line numbers rather than page numbers.

Write a program that can create a concordance. The document should be read from an input file, and the concordance data should be written to an output file. The names of the input file and output file should be specified as command line arguments when the program is run. You can use the indexing subroutines from Section 12.4, modified to write the data to a file instead of to System.out. You will also need to make the subroutines static. If you need some help with using files and command-line arguments, you can find an example in the sample program WordCount.java, which was also discussed in Section 12.4.

As you read the file, you want to take each word that you encounter and add it to the concordance along with the current line number. Your program will need to keep track of the line number. The end of each line in the file is marked by the newline character, '\n'. Every time you encounter this character, add one to the line number. One warning: The method in.eof(), which is defined in the TextReader, skips over end-of-lines. Since you don't want to skip end-of-line characters, you should not use in.eof() -- at least, you should not use it in the same way that it is used in the program WordCount.java. The function in.peek() from the TextReader class returns the nul character '\0' at the end of the file. Use this function instead of in.eof() to test for end-of-file.

Because it is so common, don't include the word "the" in your concordance. Also, do not include words that have length less than 3.

See the solution! visit this website <http://www.java2s.clanteam.com>

--------------------------------------------------------------------------------

Exercise 12.6: The sample program SimpleParser5.java from Section 12.4 can handle expressions that contain variables, numbers, operators, and parentheses. Extend this program so that it can also handle the standard mathematical functions sin, cos, tan, abs, sqrt, and log. For example, the program should be able to evaluate an expression such as sin(3\*x-7)+log(sqrt(y)), assuming that the variables x and y have been given values.

In the original program, a symbol table holds a value for each variable that has been defined. In your program, you should add another type of symbol to the table to represent standard functions. You can use objects belonging to the following class:

class StandardFunction {

// An object of this class represents

// one of the standard functions.

static final int SIN = 0, COS = 1, // Code numbers for each

TAN = 2, ABS = 3, // of the functions.

SQRT = 4, LOG = 5;

int functionCode; // Tells which function this is.

// The value is one of the above codes.

StandardFunction(int code) {

// Constructor creates the standard function specified

// by the given code, which should be one of the

// above code numbers.

functionCode = code;

}

double evaluate(double x) {

// Finds the value of this function for the

// specified parameter value, x.

switch(functionCode) {

case SIN:

return Math.sin(x);

case COS:

return Math.cos(x);

case TAN:

return Math.tan(x);

case ABS:

return Math.abs(x);

case SQRT:

return Math.sqrt(x);

default:

return Math.log(x);

}

}

} // end class StandardFunction

Add a symbol to the symbol table to represent each function. The key is the name of the function and the value is an object of type StandardFunction that represents the function. For example:

symbolTable.put("sin", new StandardFunction(StandardFunction.SIN));

In your parser, when you encounter a word, you have to be able to tell whether it's a variable or a standard function. Look up the word in the symbol table. If the associated value is non-null and is of type Double, then the word is a variable. If it is of type StandardFunction, then the word is a function. Remember that you can test the type of an object using the instanceof operator. For example: if (obj instanceof Double)

See the solution! visit this website

<http://www.java2s.clanteam.com>

--------------------------------------------------------------------------------