**Chapter 6**

**6.10 (Sales Commissions)** Use a one-dimensional array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who grosses $5,000 in sales in a week receives $200 plus 9% of $5,000, or a total of $650. Write an application (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson’s salary is truncated to an integer amount):

a) $200–299

b) $300–399

c) $400–499

d) $500–599

e) $600–699

f) $700–799

g) $800–899

h) $900–999

i) $1,000

and over Summarize the results in tabular format.

**6.11** Write statements that perform the following one-dimensional-array operations:

a) Set the 10 elements of integer array counts to zero.

b) Add one to each of the 15 elements of integer array bonus.

c) Display the five values of integer array bestScores in column format.

**6.12 (Duplicate Elimination)** Use a one-dimensional array to solve the following problem: Write an application that inputs five numbers, each between 10 and 100, inclusive. As each number is read, display it only if it’s not a duplicate of a number already read. Provide for the “worst case,” in which all five numbers are different. Use the smallest possible array to solve this problem. Display the complete set of unique values input after the user enters each new value.

**6.13** Label the elements of three-by-five two-dimensional array sales to indicate the order in which they’re set to zero by the following program segment:

for (int row = 0; row < sales.length; row++)

{

for (int col = 0; col < sales[row].length; col++)

{

sales[row][col] = 0;

}

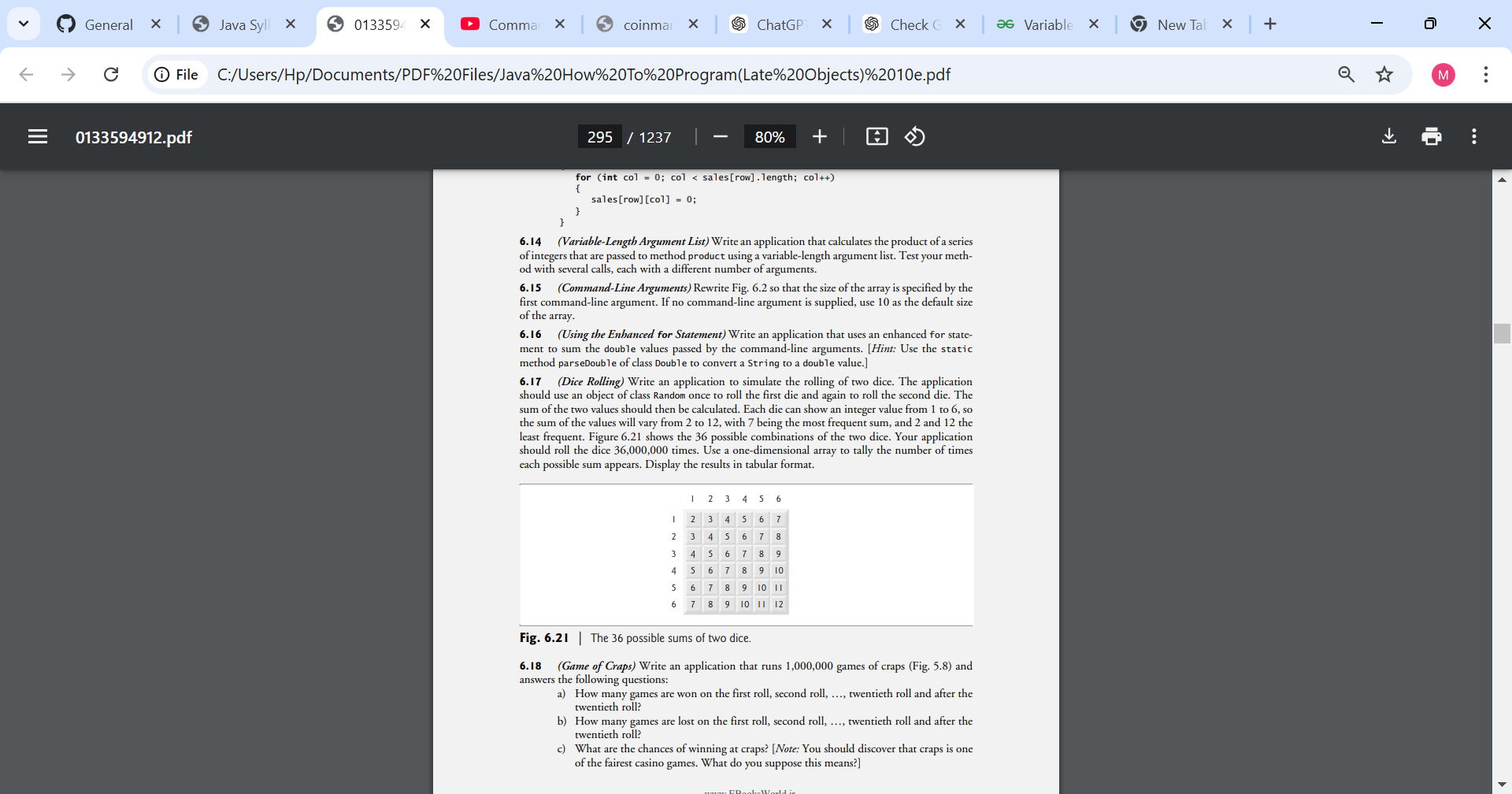
}

**6.14** (Variable-Length Argument List) Write an application that calculates the product of a series of integers that are passed to method product using a variable-length argument list. Test your method with several calls, each with a different number of arguments.

**6.15** (Command-Line Arguments) Rewrite Fig. 6.2 so that the size of the array is specified by the first command-line argument. If no command-line argument is supplied, use 10 as the default size of the array.

**6.16** (Using the Enhanced for Statement) Write an application that uses an enhanced for statement to sum the double values passed by the command-line arguments. [Hint: Use the static method parseDouble of class Double to convert a String to a double value.]

**6.17** (Dice Rolling) Write an application to simulate the rolling of two dice. The application should use an object of class Random once to roll the first die and again to roll the second die. The sum of the two values should then be calculated. Each die can show an integer value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent. Figure 6.21 shows the 36 possible combinations of the two dice. Your application should roll the dice 36,000,000 times. Use a one-dimensional array to tally the number of times each possible sum appears. Display the results in tabular format.



**6.18 (Airline Reservations System)** A small airline has just purchased a computer for its new automated reservations system. You’ve been asked to develop the new system. You’re to write an application to assign seats on each flight of the airline’s only plane (capacity: 10 seats).

Your application should display the following alternatives: Please type 1 for First Class and Please type 2 for Economy. If the user types 1, your application should assign a seat in the firstclass section (seats 1–5). If the user types 2, your application should assign a seat in the economy section (seats 6–10). Your application should then display a boarding pass indicating the person’s seat number and whether it’s in the first-class or economy section of the plane.

Use a one-dimensional array of primitive type boolean to represent the seating chart of the plane. Initialize all the elements of the array to false to indicate that all the seats are empty. As each seat is assigned, set the corresponding element of the array to true to indicate that the seat is no longer available.

Your application should never assign a seat that has already been assigned. When the economy section is full, your application should ask the person if it’s acceptable to be placed in the first-class section (and vice versa). If yes, make the appropriate seat assignment. If no, display the message "Next flight leaves in 3 hours."

**6.19 (Total Sales)** Use a two-dimensional array to solve the following problem: A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each type of product sold. Each slip contains the following:

a) The salesperson number

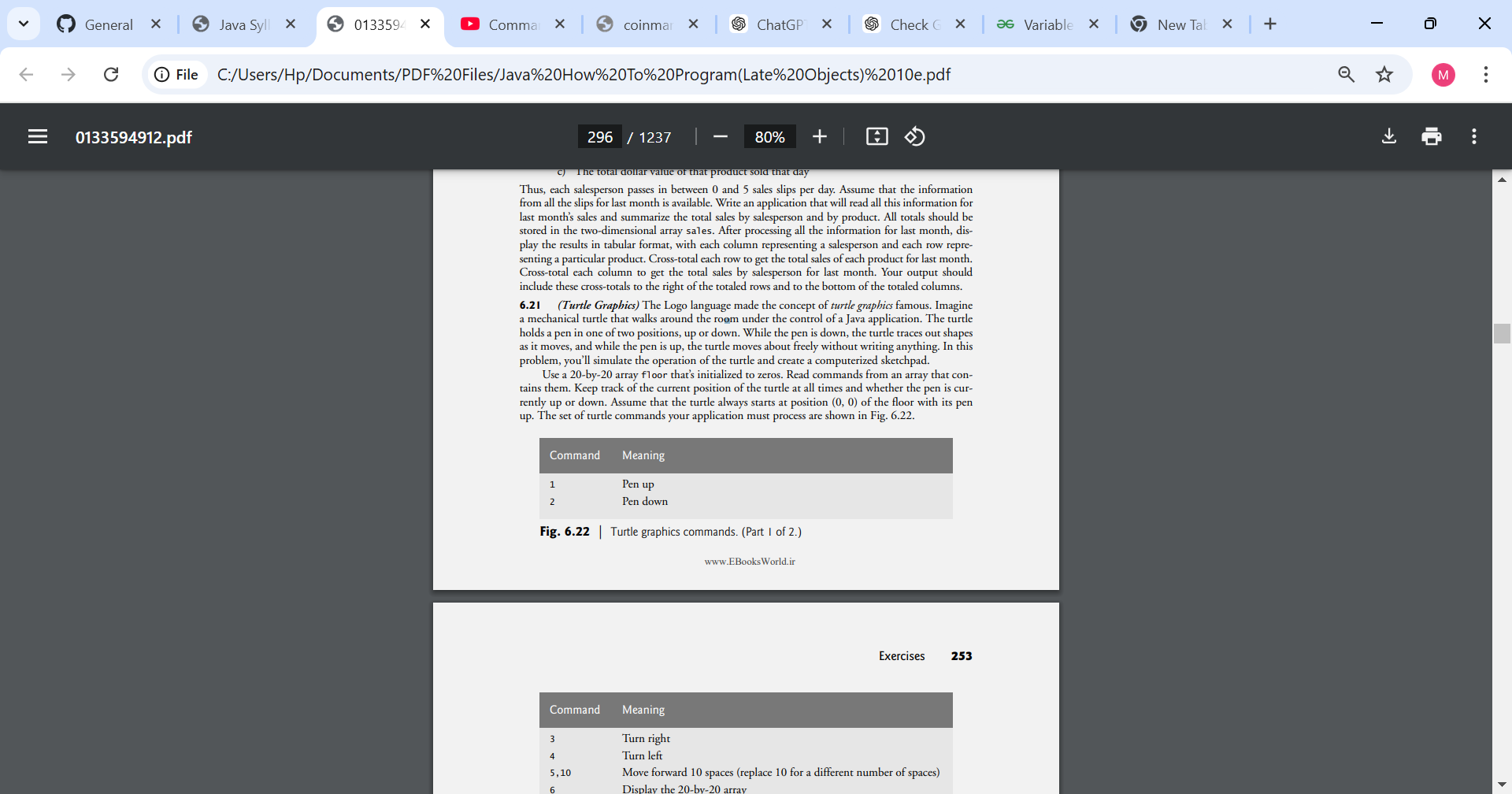
b) The product number

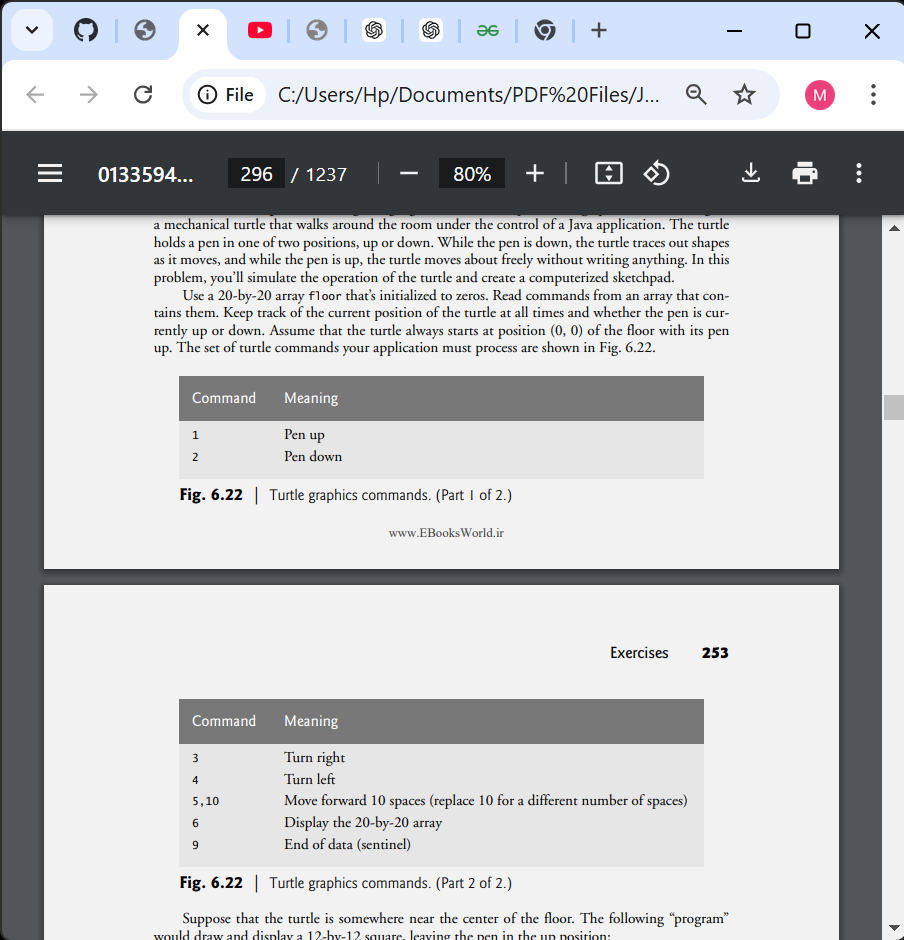
c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all the slips for last month is available. Write an application that will read all this information for last month’s sales and summarize the total sales by salesperson and by product. All totals should be stored in the two-dimensional array sales. After processing all the information for last month, display the results in tabular format, with each column representing a salesperson and each row representing a particular product. Cross-total each row to get the total sales of each product for last month. Cross-total each column to get the total sales by salesperson for last month. Your output should include these cross-totals to the right of the totaled rows and to the bottom of the totaled columns.

**6.20 (Turtle Graphics**) The Logo language made the concept of turtle graphics famous. Imagine a mechanical turtle that walks around the room under the control of a Java application. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves, and while the pen is up, the turtle moves about freely without writing anything. In this problem, you’ll simulate the operation of the turtle and create a computerized sketchpad.

Use a 20-by-20 array floor that’s initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor with its pen up. The set of turtle commands your application must process are shown in Fig. 6.22.





Suppose that the turtle is somewhere near the center of the floor. The following “program” would draw and display a 12-by-12 square, leaving the pen in the up position:

2

5,12

3

5,12

3

5,12

3

5,12

1

6

9

As the turtle moves with the pen down, set the appropriate elements of array floor to 1s. When the 6 command (display the array) is given, wherever there’s a 1 in the array, display an asterisk or any character you choose. Wherever there’s a 0, display a blank.

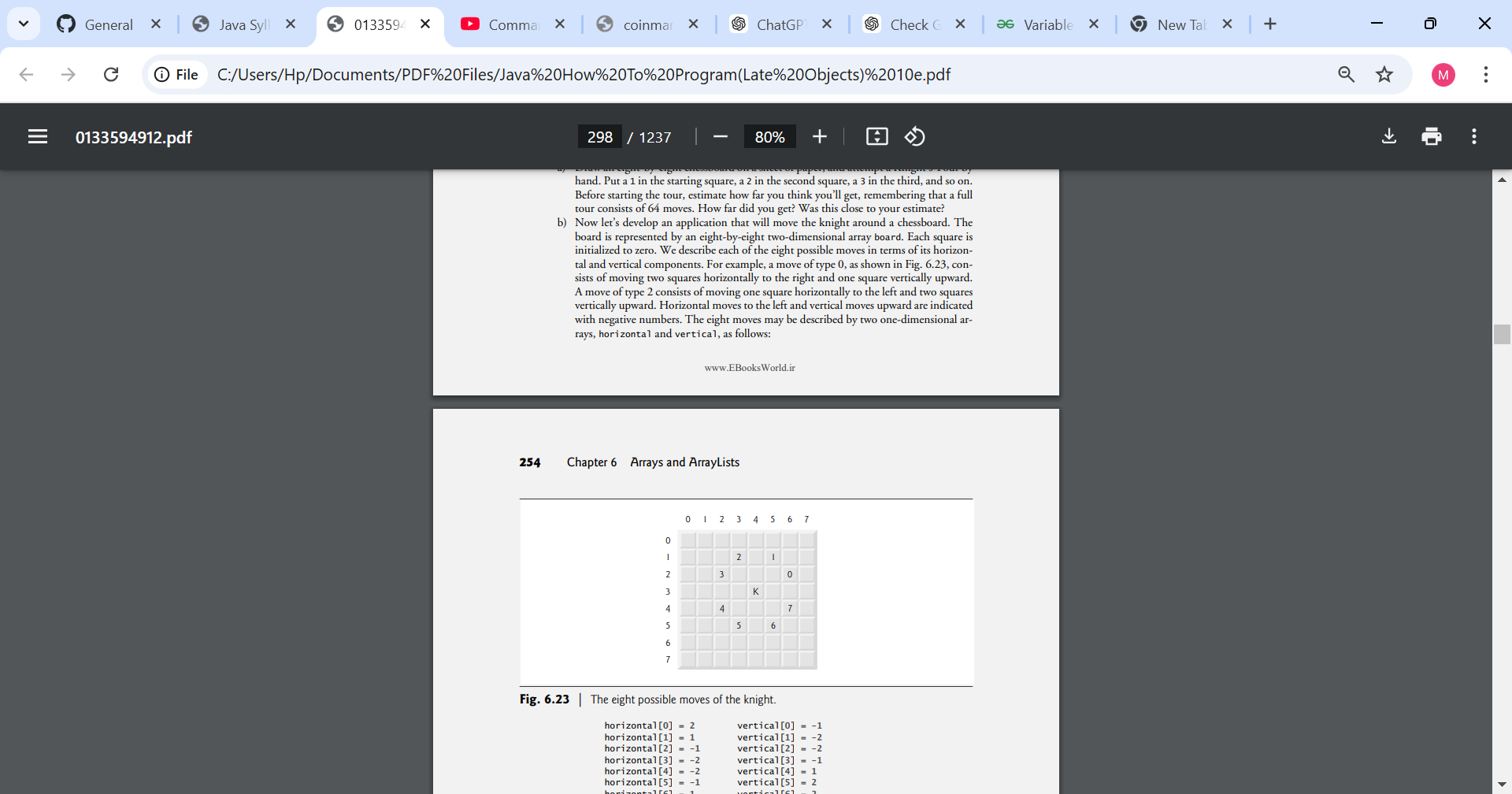
Write an application to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

**6.21 (Knight’s Tour)** An interesting puzzler for chess buffs is the Knight’s Tour problem, originally proposed by the mathematician Euler. Can the knight piece move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). Thus, as shown in Fig. 6.23, from a square near the middle of an empty chessboard, the knight (labeled K) can make eight different moves (numbered 0 through 7).

a) Draw an eight-by-eight chessboard on a sheet of paper, and attempt a Knight’s Tour by hand. Put a 1 in the starting square, a 2 in the second square, a 3 in the third, and so on. Before starting the tour, estimate how far you think you’ll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?

b) Now let’s develop an application that will move the knight around a chessboard. The board is represented by an eight-by-eight two-dimensional array board. Each square is initialized to zero. We describe each of the eight possible moves in terms of its horizontal and vertical components. For example, a move of type 0, as shown in Fig. 6.23, consists of moving two squares horizontally to the right and one square vertically upward. A move of type 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, horizontal and vertical, as follows:



horizontal[0] = 2 vertical[0] = -1

horizontal[1] = 1 vertical[1] = -2

horizontal[2] = -1 vertical[2] = -2

horizontal[3] = -2 vertical[3] = -1

horizontal[4] = -2 vertical[4] = 1

horizontal[5] = -1 vertical[5] = 2

horizontal[6] = 1 vertical[6] = 2

horizontal[7] = 2 vertical[7] = 1

Let the variables currentRow and currentColumn indicate the row and column, respectively, of the knight’s current position. To make a move of type moveNumber, where moveNumber is between 0 and 7, your application should use the statements

currentRow += vertical[moveNumber]; currentColumn += horizontal[moveNumber];

Write an application to move the knight around the chessboard. Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Test each potential move to see if the knight has already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Run the application. How many moves did the knight make?

c) After attempting to write and run a Knight’s Tour application, you’ve probably developed some valuable insights. We’ll use these insights to develop a heuristic (i.e., a common-sense rule) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome or inaccessible squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so that when the board gets congested near the end of the tour, there will be a greater chance of success.

We could develop an “accessibility heuristic” by classifying each of the squares according to how accessible it is and always moving the knight (using the knight’s Lshaped moves) to the most inaccessible square. We label a two-dimensional array accessibility with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each of the 16 squares nearest the center is rated as 8, each corner square is rated as 2, and the other squares have accessibility numbers of 3, 4 or 6 as follows:

2 3 4 4 4 4 3 2

3 4 6 6 6 6 4 3

4 6 8 8 8 8 6 4

4 6 8 8 8 8 6 4

4 6 8 8 8 8 6 4

4 6 8 8 8 8 6 4

3 4 6 6 6 6 4 3

2 3 4 4 4 4 3 2

Write a new version of the Knight’s Tour, using the accessibility heuristic. The knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your application should reduce the accessibility numbers as more squares become occupied. In this way, at any given time during the tour, each available square’s accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your application. Did you get a full tour? Modify the application to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

d) Write a version of the Knight’s Tour application that, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the “tied” squares. Your application should move to the tied square for which the next move would arrive at a square with the lowest accessibility number.

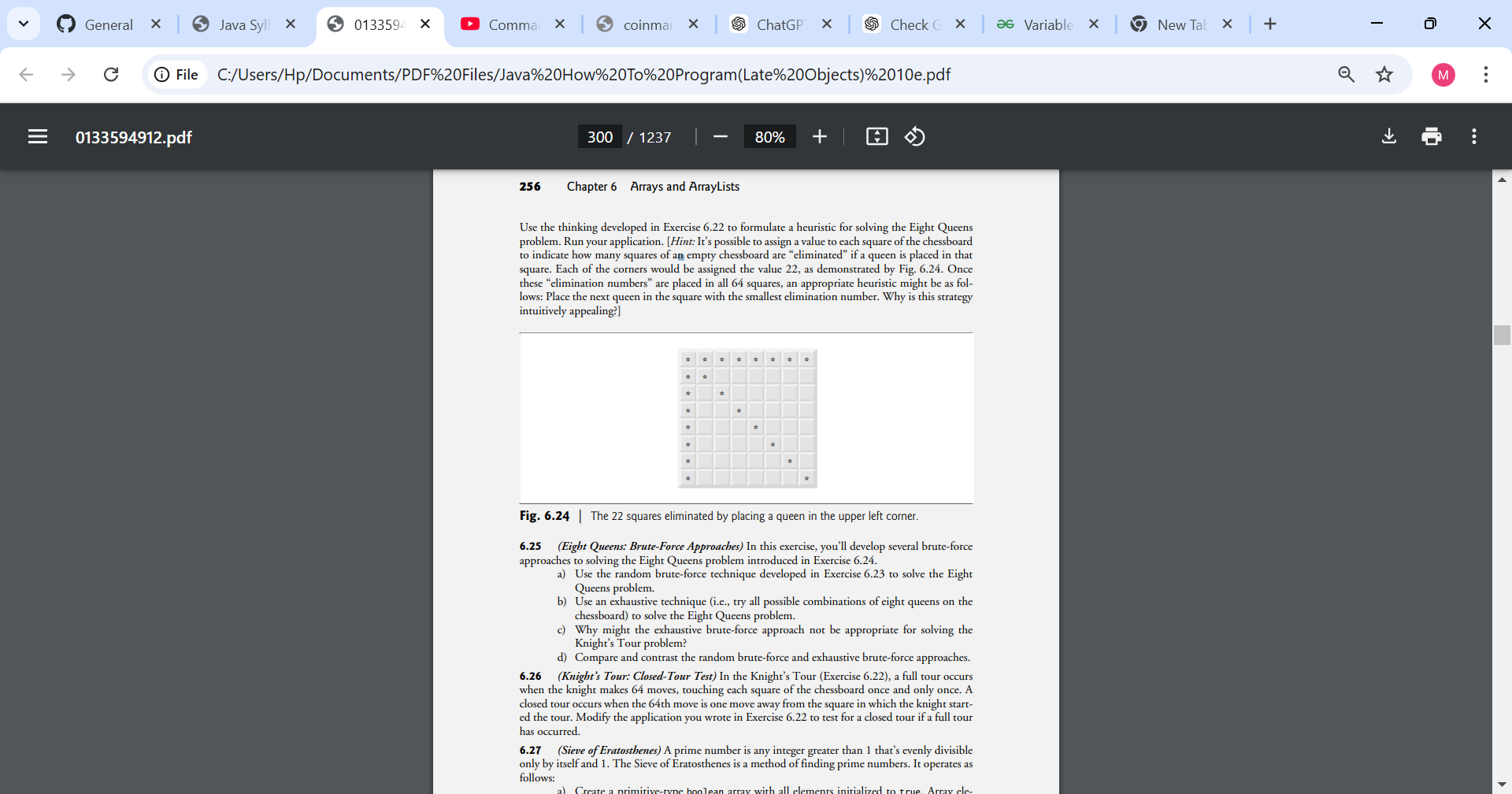
**6.22 (Knight’s Tour: Brute-Force Approaches)** In part (c) of Exercise 6.22, we developed a solution to the Knight’s Tour problem. The approach used, called the “accessibility heuristic,” generates many solutions and executes efficiently.

As computers continue to increase in power, we’ll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let’s call this approach “brute-force” problem solving.

1. Use random-number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves) at random. Your application should run one tour and display the final chessboard. How far did the knight get?
2. Most likely, the application in part (a) produced a relatively short tour. Now modify your application to attempt 1,000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your application finishes attempting the 1,000 tours, it should display this information in neat tabular format. What was the best result?
3. Most likely, the application in part (b) gave you some “respectable” tours, but no full tours. Now let your application run until it produces a full tour. [Caution: This version of the application could run for hours on a powerful computer.] Once again, keep a table of the number of tours of each length, and display this table when the first full tour is found. How many tours did your application attempt before producing a full tour? How much time did it take?
4. Compare the brute-force version of the Knight’s Tour with the accessibility-heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute-force approach? Argue the pros and cons of brute-force problem solving in general.

**6.24 (Eight Queens)** Another puzzler for chess buffs is the Eight Queens problem, which asks the following: Is it possible to place eight queens on an empty chessboard so that no queen is “attacking” any other (i.e., no two queens are in the same row, in the same column or along the same diagonal)?

Use the thinking developed in Exercise 6.22 to formulate a heuristic for solving the Eight Queens problem. Run your application. [Hint: It’s possible to assign a value to each square of the chessboard to indicate how many squares of an empty chessboard are “eliminated” if a queen is placed in that square. Each of the corners would be assigned the value 22, as demonstrated by Fig. 6.24. Once these “elimination numbers” are placed in all 64 squares, an appropriate heuristic might be as follows: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?]



6.25 (Eight Queens: Brute-Force Approaches) In this exercise, you’ll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 6.24.

a) Use the random brute-force technique developed to solve the Eight Queens problem.

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard) to solve the Eight Queens problem.

c) Why might the exhaustive brute-force approach not be appropriate for solving the Knight’s Tour problem?

d) Compare and contrast the random brute-force and exhaustive brute-force approaches.

**6.26 (Knight’s Tour: Closed-Tour Test)** In the Knight’s Tour (Exercise 6.22), a full tour occurs when the knight makes 64 moves, touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the square in which the knight started the tour. Modify the application you wrote in Exercise 6.22 to test for a closed tour if a full tour has occurred.

**6.27 (Sieve of Eratosthenes)** A prime number is any integer greater than 1 that’s evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) Create a primitive-type boolean array with all elements initialized to true. Array elements with prime indices will remain true. All other array elements will eventually be set to false.

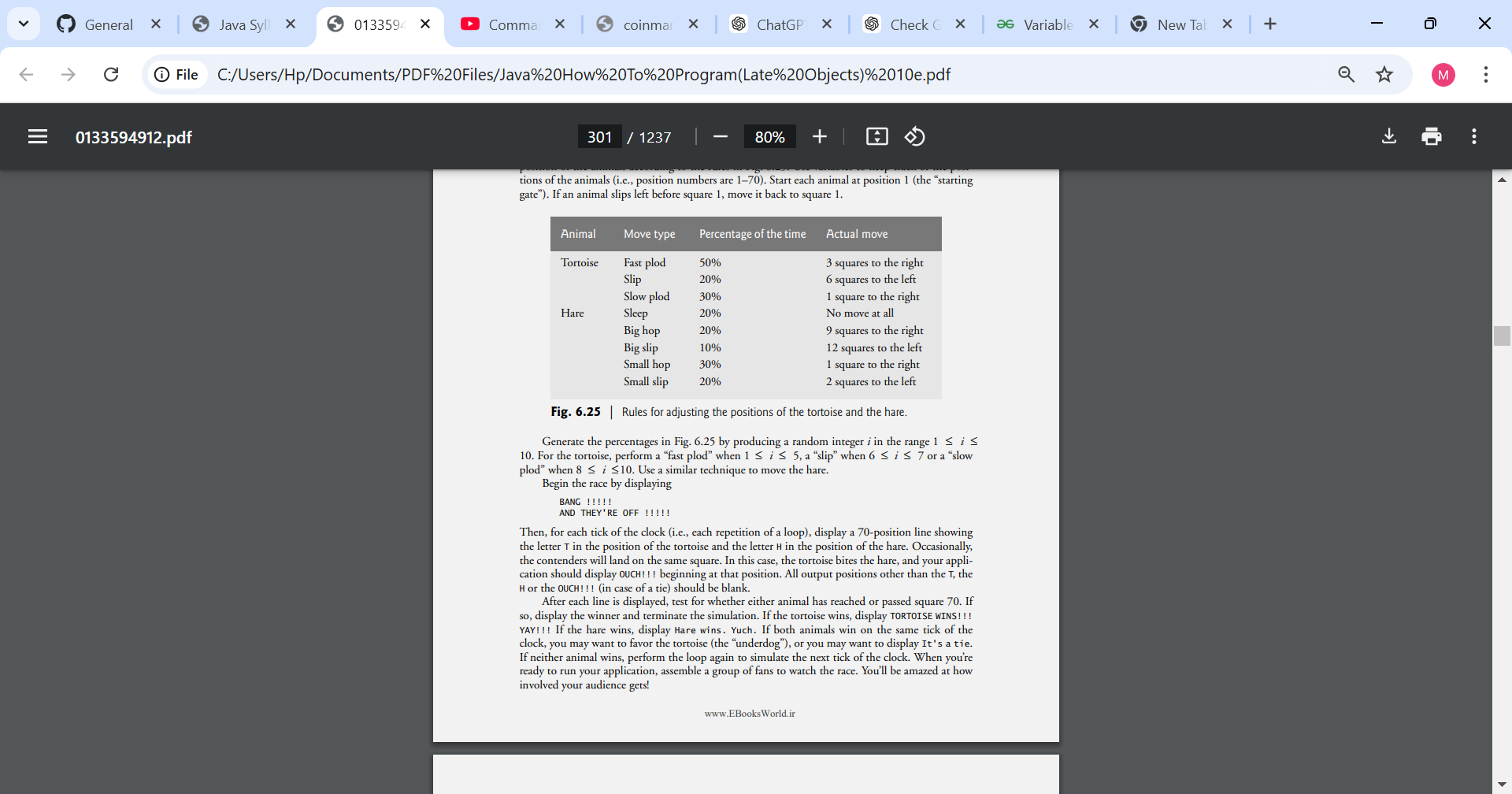
b) Starting with array index 2, determine whether a given element is true. If so, loop through the remainder of the array and set to false every element whose index is a multiple of the index for the element with value true. Then continue the process with the next element with value true. For array index 2, all elements beyond element 2 in the array that have indices which are multiples of 2 (indices 4, 6, 8, 10, etc.) will be set to false; for array index 3, all elements beyond element 3 in the array that have indices which are multiples of 3 (indices 6, 9, 12, 15, etc.) will be set to false; and so on.

When this process completes, the array elements that are still true indicate that the index is a prime number. These indices can be displayed. Write an application that uses an array of 1,000 elements to determine and display the prime numbers between 2 and 999. Ignore array elements 0 and 1.

**6.28 (Simulation: The Tortoise and the Hare)** In this problem, you’ll re-create the classic race of the tortoise and the hare. You’ll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick of the clock, your application should adjust the position of the animals according to the rules in Fig. 6.25. Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (the “starting gate”). If an animal slips left before square 1, move it back to square 1.



Generate the percentages in Fig. 6.25 by producing a random integer i in the range 1 ≤ i ≤ 10. For the tortoise, perform a “fast plod” when 1 ≤ i ≤ 5, a “slip” when 6 ≤ i ≤ 7 or a “slow plod” when 8 ≤ i ≤10.

Use a similar technique to move the hare. Begin the race by displaying

BANG !!!!!

AND THEY'RE OFF !!!!!

Then, for each tick of the clock (i.e., each repetition of a loop), display a 70-position line showing the letter T in the position of the tortoise and the letter H in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your application should display OUCH!!! beginning at that position. All output positions other than the T, the H or the OUCH!!! (in case of a tie) should be blank.

After each line is displayed, test for whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation. If the tortoise wins, display TORTOISE WINS!!! YAY!!! If the hare wins, display Hare wins. Yuch. If both animals win on the same tick of the clock, you may want to favor the tortoise (the “underdog”), or you may want to display It's a tie. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you’re ready to run your application, assemble a group of fans to watch the race. You’ll be amazed at how involved your audience gets!

Later in the book, we introduce a number of Java capabilities, such as graphics, images, animation, sound and multithreading. As you study those features, you might enjoy enhancing your tortoise-and-hare contest simulation.

**6.29 (Fibonacci Series)** The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms.

a) Write a method fibonacci(n) that calculates the nth Fibonacci number. Incorporate this method into an application that enables the user to enter the value of n.

b) Determine the largest Fibonacci number that can be displayed on your system.

c) Modify the application you wrote in part (a) to use double instead of int to calculate and return Fibonacci numbers, and use this modified application to repeat part (b).

**Making a Difference**

**6.33 (Polling)** The Internet and the web are enabling more people to network, join a cause, voice opinions, and so on. Recent presidential candidates have used the Internet intensively to get out their messages and raise money for their campaigns. In this exercise, you’ll write a simple polling program that allows users to rate five social-consciousness issues from 1 (least important) to 10 (most important). Pick five causes that are important to you (e.g., political issues, global environmental issues). Use a one-dimensional array topics (of type String) to store the five causes. To summarize the survey responses, use a 5-row, 10-column two-dimensional array responses (of type int), each row corresponding to an element in the topics array. When the program runs, it should ask the user to rate each issue. Have your friends and family respond to the survey. Then have the program display a summary of the results, including:

a) A tabular report with the five topics down the left side and the 10 ratings across the top, listing in each column the number of ratings received for each topic.

b) To the right of each row, show the average of the ratings for that issue.

c) Which issue received the highest point total? Display both the issue and the point total.

d) Which issue received the lowest point total? Display both the issue and the point total.