|  |
| --- |
| **Exercise 7.1**  The busiest time of day: 18 |
| **Exercise 7.2**  Person[] people; |
| **Exercise 7.3**  boolean[] vacant; |
| **Exercise 7.5**  int[] counts;  boolean[] occupied = new boolean[5000]; |
| **Exercise 7.6**  readings = new double[60]; urls = new String[90]; machines = new TicketMachine[5]; |
| **Exercise 7.7**  None. It only creates an array to hold String objects. |
| **Exercise 7.8**  **The brackets must be square rather than round.**  double[] prices = new double[50] |
| **Exercise 7.9**  It throws an ArrayIndexOutOfBoundsException: 24 |
| **Exercise 7.10**  /\*\*  \* Print the hourly counts.  \* These should have been set with a prior  \* call to analyzeHourlyData.  \*/  public void printHourlyCounts()  {  System.out.println("Hr: Count");  int hour = 0;  while(hour < hourCounts.length) {  System.out.println(hour + ": " + hourCounts[hour]);  hour++;  }  } |
| **Exercise 7.11**  public void printGreater(double[] marks, double mean)  {  for(int index = 0; index < marks.length; index++) {  if(marks[index] > mean) {  System.out.println(marks[index]);   }  }  } |
| **Exercise 7.12**  /\*\*  \* Create an object to analyze hourly web accesses.  \* @param filename The file to be analyzed.  \*/  public LogAnalyzer(String filename)  {  // Create the array object to hold the hourly  // access counts.  hourCounts = new int[24];  // Create the reader to obtain the data.  reader = new LogfileReader(filename);  } |
| **Exercise 7.13**  /\*\*  \* Return the number of accesses recorded in the log file  \*/  public int numberOfAccesses()  {  int total = 0;  // Add the value in each element of hourCounts to total.  for(int hourCount : hourCounts) {  total = total + hourCount;   }  return total;  } |
| **Exercise 7.15**  /\*\*  \* Return the busiest hour of day  \*/   public int busiestHour()  {  int busiestHour = 0;  for(int hour = 1; hour < hourCounts.length; hour++) {  if(hourCounts[hour] > hourCounts[busiestHour]) {  busiestHour = hour;  }  }  return busiestHour;  } |
| **Exercise 7.16**  /\*\*  \* Return the quietest hour of day  \*/   public int quietestHour()  {  int quietestHour = 0;  for(int hour = 1; hour < hourCounts.length; hour++) {  if(hourCounts[hour] < hourCounts[quietestHour]) {  quietestHour = hour;  }  }  return quietestHour;  } |
| **Exercise 7.17**  In the above implementation, it is the first one that is found. |
| **Exercise 7.18**  /\*\*  \* Return the two-hour period which is busiest.  \*/   public int busiestTwoHourPeriod()  {  int busiestPeriod = 0;  int busiestPeriodCount = 0;  for(int hour = 0; hour < hourCounts.length - 1; hour++) {  int periodCount = hourCounts[hour] + hourCounts[hour+1];  if(periodCount > busiestPeriodCount) {  busiestPeriod = hour;  busiestPeriodCount = periodCount;  }  }  return busiestPeriod;  } |
| **Exercise 7.22**  Reasons for choosing a fixed size array could be:   * Performance is slightly better. * Not so good if students are added and removed from time to time.   Reasons for keeping the dynamically sized list:   * No need to keep track of the current number of students. * Good for future enhancements (for instance if we want to have a method to remove a student from the list). |
| **Exercise 7.23**  /\*\*  \* Show a list of all the files in the collection.  \*/  public void listAllFiles()  {  for(int i = 0; i < files.size(); i++) {  System.out.println(files.get(i));  }  } |
| **Exercise 7.24**   * **binarySearch: Searches its array argument for the given value. The search process relies on the contents of the array being sorted.** * **copyOfRange: Returns an array containing a copy of the data from its array parameter. The start index (inclusive) and the finish index (exclusive) define the rane of the array to be copied.** * **fill: Copies the given value into every element of the given array.** * **sort: Sorts the data in the given array. Versions of this method allow a range of the array to be sorted rather than the whole array.** |
| **Exercise 7.27**  The same patterns emerge. |
| **Exercise 7.28**  **There are two version of fill(int[]). They store a given value in either every element or a subsection of an array. The method is used in the Automaton to set the state of every cell to 0 in the reset method.** |
| **Exercise 7.29**  **The patterns are different this time. The patterns are deterministic. However, note that the question only says to alter the constructor. The reset method should also be changed to ensure this repetition on the second and subsequent runs.** |
| **Exercise 7.30**  **If a new array is not created then the original array becomes a mix of old and updated state, which affects the resulting new state.** |
| **Exercise 7.31**  **Only one cell’s value needs to be retained – that of the cell changed on the current iteration. In effect, this is what the use of the** left **variable achieves in the proposed revisions.** |
| **Exercise 7.32**  int left = i == 0 ? 0 : state[i - 1];  int center = state[i];  int right = i + 1 < state.length ? state[i + 1] : 0;  nextState[i] = (left + center + right) % 2; |
| **Exercise 7.35**  **There are 256 ways. See the information box on Wolfram codes.** |
| **Exercise 7.38**  /\*\*  \* Create a 1D automaton consisting of the given number of cells.  \* @param code The Wolfram code.  \* @param numberOfCells The number of cells in the automaton.  \*/  public Automaton(int code, int numberOfCells)  {  this.numberOfCells = numberOfCells;  // Allow an extra element to avoid 'fencepost' errors.  state = new int[numberOfCells + 1];  stateTable = new int[8];  int bits = code;  for(int i = 0; i < stateTable.length; i++) {  int previous = stateTable[i];  if((bits % 2) == 1) {  stateTable[i] = 1;  }  bits /= 2;  }  // Seed the automaton with a single 'on' cell.  state[numberOfCells / 2] = 1;  } |
| **Exercise 7.41**  int[] copy = new int[original.length];  System.arraycopy(original, 0, copy, 0, original.length); |
| **Exercise 7.42**  // Cater for non-rectangular arrays.  int[][] copy = new int[original.length][];  for(int row = 0; row < copy.length; row++) {  int length = original[row].length;  copy[row] = new int[length];  System.arraycopy(original[row], 0, copy[row], 0, length);  } |
| **Exercise 7.44**  /\*\*  \* Determine this cell's next state, based on the  \* state of its neighbors.  \* This is an implementation of the rules for  \* Conway's Game of Life.  \* @return The next state.  \*/  public int getNextState()  {  // Count the number of neighbors that are alive.  int aliveCount = 0;  for(Cell n : neighbors) {  if(n.getState() == ALIVE) {  aliveCount++;  }  }  if(state == DEAD) {  return aliveCount == 3 ? ALIVE : DEAD;  }  else {  return aliveCount < 2 || aliveCount > 3 ? DEAD : ALIVE;  }  }  **In addition, the possible states should be defined as:**  // The possible states.  public static final int ALIVE = 0, DEAD = 1;  // The number of possible states.  public static final int NUM\_STATES = 2; |