

```
1: /**
2:  * This program emulates the California SuperLotto PLUS(R) Quick Pick(R)
3:  * functionality. Each ticket contains a maximum of 10 lines. Remaining lines
4:  * will overflow into as many additional tickets as needed. Each line of numbers
5:  * contains 5 unique numbers between 1 and 43 (printed in ascending order with
6:  * leading zeroes) and a sixth "Meganumber," a number between 1 and 27 which
7:  * does not need to be unique from the other numbers. Each line is alpha
8:  * enumerated with uppercase letters from A to J (correlating to 1 to 10).
9:  * The mega column is headed with the word "MEGA" and there should be a header
10:  * at the top of each ticket.
11:  *
12:  * Here is a sample call to the program:
13:  *
14:  *     > java SuperLotto 14
15:  *
16:  *     #####
17:  *     ### CALIFORNIA SUPERLOTTO ###
18:  *     #####
19:  *
20:  *                                     MEGA
21:  *     A    06   14   24   28   43   12
22:  *     B    08   11   18   31   36   15
23:  *     C    14   18   33   41   42   21
24:  *     D    07   19   30   38   39   26
25:  *     E    07   08   19   32   41   26
26:  *     F    09   14   16   19   41   24
27:  *     G    03   09   24   28   37   05
28:  *     H    08   09   14   24   30   10
29:  *     I    03   17   37   39   42   17
30:  *     J    08   12   14   23   41   13
31:  *
32:  *
33:  *     #####
34:  *     ### CALIFORNIA SUPERLOTTO ###
35:  *     #####
36:  *
37:  *                                     MEGA
38:  *     A    17   18   23   32   37   02
39:  *     B    02   12   19   24   42   07
40:  *     C    01   07   09   11   31   02
41:  *     D    01   06   15   26   32   15
42:  *
43:  * @name      SuperLotto
44:  * @author    Ravi S. Ramphal
45:  * @class     CCSF CS111B
46:  * @date      2017.07.10
47:  * @version   1.0
48:  */
49:
50: import java.util.*;
51:
52: public class SuperLotto
53: {
54:     /**
55:      * This is a die method that prints a message to the stderr stream and
56:      * exits the program with an error code.
57:      *
58:      * @param errorMessage The error message to be displayed to users
59:      */
60:     private static void die(String errorMessage)
61:     {
62:         System.err.println(errorMessage);
63:         System.exit(1);
64:     }
65:
66:     /**
67:      * This method repeats a character at the beginning of a String, padding the
68:      * String until it reaches a desired width.
```

```
69:      *
70:      * @param input The String to be padded
71:      * @param width An integer for the desired final width of the String
72:      * @param padder The character to be repeated to go get to final width
73:      * @return String The final padded String
74:      */
75: private static String leftPad (String input, int width, char padder)
76: {
77:     // cache number of characters in input
78:     int inputWidth = input.length();
79:
80:     // if user desires a width shorter than input width, return unchanged
81:     if (inputWidth >= width) return input;
82:
83:     // initialize a StringBuilder with capacity set to desired width
84:     StringBuilder output = new StringBuilder(width);
85:
86:     // repeat characters to fill missing width
87:     for (int i = 0; i < (width - inputWidth); i++)
88:     {
89:         output.append(padder);
90:     }
91:
92:     // append original input to end of repeated characters and return String
93:     return output.append(input).toString();
94: }
95:
96: /**
97:  * This method converts an integer into a String and then repeats a
98:  * character at the beginning of the String until it reaches a desired
99:  * width.
100:  *
101:  * @param input The integer to be padded
102:  * @param width An integer for the desired final width of the String
103:  * @param padder The character to be repeated to go get to final width
104:  * @return String The final padded string
105:  */
106: private static String leftPad (int input, int width, char padder)
107: {
108:     return leftPad("" + input, width, padder);
109: }
110:
111: /**
112:  * This method returns a random integer between the provided lower limit
113:  * and upper limit.
114:  *
115:  * @param a An int representing the lower limit (inclusive)
116:  * @param b An int representing the upper limit (inclusive)
117:  * @return int A random number between the two limits
118:  */
119: private static int getRandomNumber(int a, int b)
120: {
121:     return ((int)((b - a + 1) * Math.random() + a));
122: }
123:
124: /**
125:  * This method iterates through an array searching for a target number and
126:  * returns a boolean if it is/is not found. It performs sequential search,
127:  * but it only searches over elements that have already been populated
128:  * (represented by the 'limit' param).
129:  *
130:  * @param set An array of integers to check against
131:  * @param target The integer that is being searched for
132:  * @param limit The last index that has already been populated
133:  * @return boolean A boolean for if the number already exists in the set
134:  */
135: private static boolean isRepeated(int[] set, int target, int limit)
136: {
```

```
137:         for (int i = 0; i < limit + 1; i++)
138:         {
139:             if (target == set[i]) return true;
140:         }
141:
142:         return false;
143:     }
144:
145:     /**
146:      * This method generates an array of given number of random integers between
147:      * the lower and upper limits provided and returns the array.
148:      *
149:      * @param count The number of numbers that should be generated
150:      * @param lower An int representing the lower limit
151:      * @param upper An int representing the upper limit
152:      * @return int[] An array of integers that were generated
153:      */
154:     private static int[] getUniqueNumbers(int count, int lower, int upper)
155:     {
156:         int[] numbers = new int[count];
157:
158:         for (int i = 0; i < numbers.length; i++)
159:         {
160:             int randomNumber = getRandomNumber(lower, upper);
161:             while (isRepeated(numbers, randomNumber, i))
162:             {
163:                 randomNumber = getRandomNumber(lower, upper);
164:             }
165:             numbers[i] = randomNumber;
166:         }
167:
168:         return numbers;
169:     }
170:
171:     /**
172:      * This method accepts an array of lines where each element is itself
173:      * an array of integers and formats the output to resemble a
174:      * California SuperLotto PLUS(R) Quick Pick(R) tickets.
175:      *
176:      * NOTE: This method does NOT validate input format or sanitize the data
177:      *       in any way. The user must do this before invoking this function.
178:      *
179:      * @param lines An array of lines (a line being an array of integers)
180:      */
181:     private static void printTicket (int[][] lines)
182:     {
183:         System.out.println();
184:         System.out.println("#####");
185:         System.out.println("### CALIFORNIA SUPERLOTTO ###");
186:         System.out.println("#####");
187:         System.out.println();
188:         System.out.println("                                MEGA ");
189:
190:         // iterate over lines
191:         for (int i = 0; i < lines.length; i++)
192:         {
193:             System.out.print(" " + String.valueOf((char)(i + 65)) + " ");
194:
195:             // iterate over numbers
196:             for (int j = 0; j < lines[i].length; j++)
197:             {
198:                 System.out.print(leftPad(lines[i][j], 2, '0') + " ");
199:             }
200:             System.out.println();
201:         }
202:
203:         System.out.println();
204:     }
```

```
205:
206: /**
207:  * This method generates a California SuperLotto PLUS(R) Quick Pick(R) line.
208:  * Each line contains 5 unique numbers between 1 and 43 and a sixth
209:  * "Meganumber," a number between 1 and 27 which does not need to be unique
210:  * from the other numbers.
211:  *
212:  * @return int[] An array of integers representing a Quick Pick(R) line
213:  */
214: private static int[] generateLine ()
215: {
216:     int[] line = new int[6];
217:     int[] numbers = getUniqueNumbers(5, 1, 43);
218:     int meganumber = getRandomNumber(1, 27);
219:
220:     Arrays.sort(numbers);
221:
222:     for (int i = 0; i < numbers.length; i++) line[i] = numbers[i];
223:     line[5] = meganumber;
224:
225:     return line;
226: }
227:
228: /**
229:  * This method takes a desired number of lines and generates a
230:  * California SuperLotto PLUS(R) Quick Pick(R) ticket (where each ticket
231:  * can only have a maximum of 10 lines).
232:  *
233:  * NOTE: This method does NOT validate input format or sanitize the data
234:  *       in any way. The user must do this before invoking this function.
235:  *
236:  * @param numOfLines The number of lines to generate a ticket for
237:  * @return int[][][] An array of lines (which are arrays of integers)
238:  */
239: private static int[][] generateTicket (int numOfLines)
240: {
241:     int[][] ticket = new int[numOfLines][6];
242:
243:     for (int i = 0; i < numOfLines; i++)
244:     {
245:         ticket[i] = generateLine();
246:     }
247:
248:     return ticket;
249: }
250:
251: /**
252:  * This method takes a number to chunk and the chunk size and returns an
253:  * array of integers representing the chunks.
254:  *
255:  * For example, 'generateChunks(41, 10)' returns '{ 10, 10, 10, 10, 1 }'
256:  *
257:  * @param totalNum The total number that is to be chunked
258:  * @param chunkSize The integer size of each chunk
259:  * @return int[] An array of integers representing the chunks
260:  */
261: private static int[] generateChunks (int totalNum, int chunkSize)
262: {
263:     if (totalNum < 1 || chunkSize < 1) return new int[] {};
264:     if (totalNum <= chunkSize) return new int[] { totalNum };
265:
266:     int numOfChunks = totalNum / chunkSize;
267:     int partialChunkSize = totalNum % chunkSize;
268:
269:     boolean hasPartialChunk = partialChunkSize != 0;
270:
271:     if (hasPartialChunk) numOfChunks++;
272: }
```

```
273:         int[] chunks = new int[numOfChunks];
274:
275:         Arrays.fill(chunks, chunkSize);
276:
277:         if (hasPartialChunk) chunks[numOfChunks - 1] = partialChunkSize;
278:
279:         return chunks;
280:     }
281:
282:     /**
283:      * This method accepts a desired number of lines and then generates tickets
284:      * (an array of lines where each element is itself an array of integers)
285:      * and prints each ticket out to the user.
286:      *
287:      * @param numOfLines The number of lines to generate and print tickets for
288:      */
289:     private static void printTickets (int numOfLines)
290:     {
291:         int[] ticketChunkSizes = generateChunks(numOfLines, 10);
292:
293:         for (int size : ticketChunkSizes)
294:         {
295:             int[][] ticket = generateTicket(size);
296:             printTicket(ticket);
297:         }
298:     }
299:
300:     /**
301:      * This is the 'main' method of this class. It validates arguments passed
302:      * in and then prints (a) ticket(s) containing the desired number of lines.
303:      *
304:      * @param array An array of arguments provided to the program
305:      */
306:     public static void main (String ... args)
307:     {
308:         if (args.length == 0) die("Missing argument!");
309:         if (args.length > 1) die("Too many arguments!");
310:
311:         try
312:         {
313:             int numOfLines = Integer.parseInt(args[0]);
314:
315:             if (numOfLines < 1) die("Invalid number of lines!");
316:
317:             printTickets(numOfLines);
318:         }
319:         catch (NumberFormatException e)
320:         {
321:             die("Invalid argument!");
322:         }
323:     }
324: }
```