Lab Questions
Big Java, Late Objects / Java for Everyone, 2e
Chapter Number: Arrays and ArrayLists

1.1) Let's work on some basic skills by creating a program that contains a few experiments with arrays. If you are asked to print out a value, be sure to print a label that indicates what the value represents, unless otherwise indicated. For example, if you are asked to print a variable `z`, the output might look like this:

```
z :   8.0
```

Perform each of the following tasks:

a) Create an array `x` of doubles with an initializer list that contains the following values: 8, 4, 5, 21, 7, 9, 18, 2, and 100.
b) Print the number of items in the array by printing the expression `x.length`.
c) Print the first array item, `x[0]`.
d) Print the last array item. Be careful to choose the right index.
e) Print the expression `x[x.length - 1]`. Why is this value the same as in part d?
f) Use a standard `for` loop to print all the values in the array without labels.
g) Use a standard `(` loop to print all the values in the array with labels to indicate what each element is.
h) Use a standard `for` loop to print all the values in the array in reverse order with labels to indicate what each element is.
i) Use an enhanced `for` loop to print all the values in the array without labels.

1.2) Let's continue to work on our fundamental array skills with a few more experiments. As in Lab 6.1.1, if you are asked to print out a value, be sure to print a label that indicates what the value represents, unless otherwise indicated. For example, if you are asked to print a variable `z`, the output might look like this:

```
z :   8.0
```

Perform each of the following tasks:

a) Create an array `x` of doubles with an initializer list that contains the following values: 8, 4, 5, 21, 7, 9, 18, 2, and 100.
b) Use an enhanced `for` loop to compute to print the total of all the elements in array `x`.
c) Use a standard `for` loop to compute and print the total of all the elements in array `x`.
d) Compute and print the minimum value in `x`. Your program should find this value by examining each item in the array.
e) Repeat part d, but print the minimum value and its index in the array.

1.3) Write a method that is passed an array, `x`, of doubles and an integer rotation amount, `n`. The method creates a new array with the items of `x` moved forward by `n` positions. Elements that are rotated off the array will appear at the end. For example, suppose `x` contains the following items in sequence:

1 2 3 4 5 6 7

After rotating by 3, the elements in the new array will appear in this sequence:

4 5 6 7 1 2 3

Array `x` should be left unchanged by this method. Use the following code to help you get started. Be sure to test your program with different rotation amounts.

```
public class Arrays
{
   public static void main(String[] args)
   {
      double[] x = {8, 4, 5, 21, 7, 9, 18, 2, 100};
      System.out.println("Before rotation:  ==============================");
      for (int i = 0; i < x.length; i++)
      {
         System.out.println("x[" + i + "]:  " + x[i]);
      }
      x = rotate(x, 3);
      System.out.println("After rotation:  ==============================");
      for (int i = 0; i < x.length; i++)
      {
         System.out.println("x[" + i + "]:  " + x[i]);
      }
   }

   // Your method goes here.
}
```

2.1) Create a class called `CustomerLister` with a `main` method that instantiates an array of `String` objects called `customerName`. The array should have room for five `String` objects. Use an initializer list to put the following names into the array:

Cathy
Ben
Jorge
Wanda
Freddie

Print the array of names.

2.2) Use `new` to create a second `double` array called `customerBalance` in the `main` method. Allow room for five customer balances, each stored as a `double`. In the loop that prints each customer name, add some code to prompt the user to enter a balance for that customer. Read the keyboard input with a `Scanner` object. Use the following balances for the input:

100.00
234.56
2.49
32.32
400.00

After all the balances have been entered, print out each customer and his/her balance.

3) British puzzle maker H. E. Dudeney concocted an interesting puzzle about a bored postman called the "Peevish Postman Problem". According to Dudeney, the postman worked in a small post office with consecutive letter boxes numbered 1 to 100. Each box was equipped with a door that could be opened and closed. Late one evening the postman made a "pass" through the boxes and opened every door. Still bored, he walked back to the beginning and made a second pass, this time visiting boxes 2, 4, 6, …, 100. Since those doors were now open, he closed them. On the third pass he visited boxes 3, 6, 9, 12, …, 99 and if a door was open he closed it, and if the door was closed he opened it. He continued to make passes through the boxes and always followed the same rule: On each pass *i* from 1 to 100, he visited only boxes that were multiples of *i*, … and changed the state of each door he visited. After making 100 passes at the doors, he surveyed the results and was surprised by the pattern of doors that he saw.

The code below uses a Boolean array to represent the doors. A true value in the array represents an open door, and a false value represents a closed one. You will have to write two nested loops in order to manipulate the array as described above. The inner loop will control the door number visited on a single pass, and the outer loop will control the number of passes. Print the state of each door after the 100th pass.

The puzzle was conceived as a paper and pencil entertainment. Can you explain the pattern of doors?

Hint: Because the doors are numbered starting at one, we will waste the first position in the array. In this case, the default value will be set to false. By ignoring the first position, the door numbers match their index positions in the array.

```
public class Peevish
{
   public static void main(String[] args)
   {
      boolean[] door;
      final int NODOORS = 101;    // We will not use door[0]
      final boolean OPEN = true;
      final boolean CLOSED = false;
      // Initialize the doors
      door = new boolean[NODOORS];
      for (int i = 0;  i < NODOORS; i++)
      {
         door[i] = CLOSED;
      }
      // Print the state of each door (only doors 1-100)
      for (int i = 1; i < NODOORS; i++)
      {
         if (door[i] == true)
         {
```



```
            System.out.println("Door " + i + " is open.");
         }
         else
         {
            System.out.println("Door " + i + " is closed.");
         }
      }

      // Your code here
   }
}
```

4) We can use the `Math.random` method to generate random integers. For example, `Math.random()` generates a random integer greater than or equal to 0 and less than 1. The expression `Math.random() * 6` generates random numbers between 0 and 5, simulating the throw of a die. In this lab assignment, you will use an array to test whether the random generator is fair; that is, whether each possible value is generated approximately the same number of times.

Your program should ask the user:

> How many random numbers should be generated?
> What is the number of values for each random draw? (e.g., 6)

Make an array with one element for each possible outcome. Set each element to 0. Then keep calling the random number generator. If it returns a value `v`, then increment the counter belonging to `v`.

After all numbers have been generated, print the counters. Here is a typical program run:

```
How many random numbers do you want to generate? 1000
What is the number of values for each random draw? 10.

0    78
1   101
2   118
3    97
4   103
5   102
6   112
7    91
8    94
9   104
```

What is the code for your program?

5) The Universal Sudoku Puzzle in the newspaper provides directions for completing a Sudoku puzzle: "Complete the grid so that every row, column, and 3 x 3 box contains every digit from 1 to 9 inclusively." Here is an example of a completed Sudoku:

```
258 || 137 || 649
```



```
146 || 985 || 327
793 || 246 || 851
=================
472 || 863 || 195
581 || 492 || 736
639 || 571 || 482
=================
315 || 728 || 964
824 || 619 || 573
967 || 354 || 218
```

Notice that each row and column contains the integers 1 to 9. This is also true of the nine 3 x 3 subsquares.

Mathematicians would say that a Sudoku puzzle is a special example of another mathematical object called a Latin square. Latin squares of size *n* x *n* satisfy the property that every row and every column contain the integers 1, 2, …, *n*. In fact we say that a row or column has the "Latin property" if it contains exactly the integers 1, 2, …, *n*. For Sudoku we can extend the notion to subsquares as well, so if a 3 x 3 subsquare contains 1, 2, …, 9, we say the subsquare satisfies the Latin property. With these definitions, a Sudoku square has the property that every row, column, and 3 x 3 subsquare is Latin.

Here is an invalid Sudoku arrangement:

```
258 || 137 || 649
146 || 985 || 227
793 || 246 || 851
=================
472 || 863 || 195
581 || 492 || 736
639 || 571 || 482
=================
315 || 728 || 964
824 || 619 || 573
967 || 354 || 218
```

Notice that in the square above, the second row, the seventh column, and the top-right subsquare do not satisfy the Latin property, while all other rows, columns and subsquares do satisfy the Latin property.

The Java code listed below has a `main` method that creates and prints three Sudoku puzzles. Each puzzle is created by calling the static method `makeSudoku(String s)`, passing it an 81 character `String` that contains the contents of a single puzzle. The rows of the puzzle are listed in sequence in the string. The method converts the passed `String` to a two-dimension array of strings (each `String` containing a single digit) which is returned to the calling program.

The code below also contains a method called `getPrintableSudoku(String[][] x)`. This method is passed the two-dimensional array representation of the puzzle, and returns a formatted `String` which can be printed to produce a two-dimensional representation on a page.

You will need to provide the body for the method `isValidSudoku(String[][] x)`. Complete this method so it returns true or false based on whether the puzzle arrangement is a complete and valid Sudoku puzzle.

There are many approaches you can take in tackling this problem. Because the solution involves testing rows, columns, and subsquares, you will probably want to introduce a collection of helper methods to break the problem into simpler subtasks.

At the top level, we need to develop a method `isValidSudoku` so that it checks that 1) each row is Latin, 2) each column is Latin, and 3) each subsquare contains 1, 2, …, 9.

Let's decompose `isValidSudoku` into three Boolean methods called `rowsAreLatin`, `colsAreLatin`, and `goodSubsquares`, which correspond to the tasks listed above. Each of these three methods can be further decomposed as well. For example, `rowsAreLatin` can be decomposed into a subordinate method `rowIsLatin(int i)` which examines a single row. (The other methods can be decomposed in a similar way.)

In designing `rowIsLatin(String[][] x, int i)`, it is evident that we need a method of insuring that each symbol 1, 2, …, 9 appears in row `i`. One technique is to create a `boolean` array called `found` with the property that `found[k]` is true if symbol `k` occurs in row[`i`]. We can start by initializing the `found` array to all `false` values to indicate that none of the symbols 1, 2, …, 9 have yet be found in the current row.

For each `String` item in a Sudoku row, you change it to an `int` by invoking `Integer.parseint(String)` and use the resulting number as a subscript into the `found` array, setting the corresponding element `true`. As an example, if the first item in a row is 6, we need to set `found[6] = true`. (*Hint:* You will need to code something like `found[k] = true` where `k` takes on each integer value represented by the `String` values in the row.) When you have completed examination of a row, the entire `found` array should contain only true values if each symbol has occurred exactly once in the row.

To help you get started, we have provided a `main` method that creates and prints three Sudoku objects. It also contains the needed method headers that we discussed in our approach to decomposing the problem.

```
public class Sudoku
{
   public static void main(String[] args)
   {
      // Row and column Latin but with invalid subsquares
      String config1 = "123456789234567891345678912456789123567891234 6"
         + "789123457891234568912345679123456 78";
      String[][] puzzle1 = makeSudoku(config1);
      if (isValidSudoku(puzzle1))
      {
         System.out.println("This puzzle is valid.");
      }
      else
      {
         System.out.println("This puzzle is invalid.");
      }
      System.out.println(getPrintableSudoku(puzzle1));
      System.out.println("-----------------------------------------------");

      // Row Latin but column not Latin and with invalid subsquares
      String config2 = "123456789123456789123456789123456789123456 78"
         + "912345678912345678912345678912345 6789";
      String[][] puzzle2 = makeSudoku(config2);
      if (isValidSudoku(puzzle2))
```



```
      {
         System.out.println("This puzzle is valid.");
      }
      else
      {
         System.out.println("This puzzle is invalid.");
      }

      System.out.println(getPrintableSudoku(puzzle2));
      System.out.println("-----------------------------------------------------");

      // A valid sudoku
      String config3 = "258137649146985327793246851472863195581492736 63"
         + "957148231572896482461957396735421 8";
      String[][] puzzle3 = makeSudoku(config3);
      if (isValidSudoku(puzzle3))
      {
         System.out.println("This puzzle is valid.");
      }
      else
      {
         System.out.println("This puzzle is invalid.");
      }
      System.out.println(getPrintableSudoku(puzzle3));
      System.out.println("-----------------------------------------------------");

   }

   public static String[][] makeSudoku(String s)
   {
      int SIZE = 9;
      int k = 0;
      String[][] x = new String[SIZE][SIZE];
      for (int i = 0; i < SIZE; i++)
      {
         for (int j = 0; j < SIZE; j++)
         {
            x[i][j] = s.substring(k, k + 1);
            k++;
         }
      }
      return x;
   }

   public static String getPrintableSudoku(String[][] x)
   {
      int SIZE = 9;
      String temp = "";
      for (int i = 0; i < SIZE; i++)
      {
         if ((i == 3) || (i == 6))
         {
            temp = temp + "=================\n";
         }
         for (int j = 0; j < SIZE; j++)
         {
            if ((j == 3) || (j == 6))
```



```
            {
               temp = temp + " || ";
            }
            temp = temp + x[i][j];
         }
         temp = temp + "\n";
      }
      return temp;
   }

   public static boolean isValidSudoku(String[][] x)
   {
      return rowsAreLatin(x) && colsAreLatin(x) && goodSubsquares(x);
   }

   public static boolean rowsAreLatin(String[][] x)
   {
      // fill in your code here
   }

   public static boolean rowIsLatin(String[][] x, int i)
   {
      // fill in your code here
   }

   public static boolean colsAreLatin(String[][] x)
   {
      // fill in your code here
   }

   public static boolean colIsLatin(String[][] x, int j)
   {
      // fill in your code here
   }

   public static boolean goodSubsquares(String[][] x)
   {
      // fill in your code here
   }

   public static boolean goodSubsquare(String[][] x, int i, int j)
   {
      // fill in your code here
   }
}
```

6) Array lists are objects that, like arrays, provide you the ability to store items sequentially and recall items by index. Working with array lists involves invoking `ArrayList` methods, so we will need to develop some basic skills. Let's start with the code below:

```
import java.util.ArrayList;

public class ArrayListRunner
{
   public static void main(String[] args)
```



```
    {
        ArrayList<String> names = new ArrayList<String>();
        System.out.println(names);
    }
}
```

The `main` method imports `java.util.ArrayList` and creates an `ArrayList` that can hold strings. It also prints out the `ArrayList` and, when it does, we see that the list is empty: [ ].

Complete the following tasks by adding code to this skeleton program. If you are asked to print a value, provide a suitable label to identify it when it is printed.

a) Invoke `add()` to enter the following names in sequence: Alice, Bob, Connie, David, Edward, Fran, Gomez, Harry. Print the `ArrayList` again.
b) Use `get()` to retrieve and print the first and last names.
c) Print the `size()` of the `ArrayList`.
d) Use `size()` to help you print the last name in the list.
e) Use `set()` to change "Alice" to "Alice B. Toklas". Print the `ArrayList` to verify the change.
f) Use the alternate form of `add()` to insert "Doug" after "David". Print the `ArrayList` again.
g) Use an enhanced `for` loop to print each name in the `ArrayList`.
h) Create a second `ArrayList` called `names2` that is built by calling the `ArrayList` constructor that accepts another `ArrayList` as an argument. Pass `names` to the constructor to build `names2`. Then print the `ArrayList`.
i) Call `names.remove(0)` to remove the first element. Print `names` and `names2`. Verify that Alice B. Toklas was removed from `names`, but not from `names2`.

