**CSIS210 - Data Structures**

Intro to Programming

### Laboratory 6

**Lab 1**

# Names \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# frankernest971218.jpg

# General Lab Procedures

* You should have a directory (folder) in your home account called csis120. At the beginning of each lab, create a new sub-directory called labX, where X is the lab number.
* Files used in the lab can be found on the course blackboard webpages.
* Turn in this lab stapled to print outs of the code you produce in each assigned section from the laboratory manual. These sheets must be in order.
* When reading in input from the Keyboard, always prompt for that input first.
* You can find documentation for the Java class libraries at the URL

<http://java.sun.com/javase/6/docs/api/>

## Lab 6

#### Objectives:

* To utilize for each, while and do-while loops.
* To manipulate ArrayLists.
* To practice reading input from the keyboard.
* To learn how random numbers work.

#### Part 1: Sentinel based loops

In this part of the lab, you will write a while loop and/or a do-while loop implementation of a classic looping problem: reading in values from the keyboard until you read in a zero.

* Create a **lab6** folder on your Z: drive.
* Obtain the **lab6.zip** and **RandomNumbers.java** files from Blackboard.
* Extract the files from the lab6.zip file into your lab 6 folder.
* In BlueJ, create a new project in your lab6 folder called **Lottery**.
* Create a new class called **Pick6**. You may remove all default fields, constructors, and methods.
* Create a **public static void main()** method.
* Within main, use a Scanner variable to read in a number between 1 and 49 from the keyboard. Test your code to be sure that it works.
* Modify your code, using a while loop, so whenever a user inputs a number that is not between 1 and 49, output an error message to the screen and ask them again for another number.
	+ Repeatedly ask them for another number until a valid one is entered.
	+ You may assume that only integers are entered by the user.
* Once completed, demonstrate your output for your instructor and have him or her initial here. If you do not finish during the exercise period, then demonstrate your application at the beginning of the next exercise period.

Instructor’s Initials \_\_\_\_\_\_\_\_

* Now, add to your code the functionality that would have the user enter in a series of numbers between 1 and 49 until the user enters in a zero.
* As the user is entering numbers, if the number is invalid, output an error message and ignore the input. If the number is valid, output a message verifying the input value. E.g., *7 is a valid input*
* When finished, test your work until you are sure that it functions properly. When ready, **add your name(s) as a comment at the top of the class, print out a copy of the class and attach it to your lab submission.**

#### Part 2: Introduction to ArrayLists

In this part of the lab, you will utilize a built-in data structure in the Java Programming Language called an **ArrayList**. The ArrayList holds a collection of values, unlike a primitive variable which holds exactly one value. You will be answering questions about the ArrayList method.

* Open the RandomNumbers class in an editor and look at the code.
* What import statements are needed to run this code?
* How many methods are defined for this class and what are their names?
* What is the name of the variable of type ArrayList<Integer>?
* What does it mean that the word Integer is placed between the < > symbols in the method signature and in the local variable declaration statement?
* What does rnd.nextInt(49) return? Do not just answer “an integer” or “a number.” Be specific about what it means to call nextInt with a parameter value of 49.
	+ HINT: You are allowed to look up the nextInt method of the Random class in the Java 1.6 API.
* Why is the number returned from nextInt incremented by one before being assigned to the variable choice?
* How are numbers added to the ArrayList variable?
* What does the *contains* method for the arrList variable do?
* The for-loop line states that the loop is supposed to run exactly six times. In what circumstance would it run a different number of times? In this case, would it run more or less than six times?

#### Part 3: Updating our Lottery Class

In this part of the lab, you will modify your original Pick6 class to perform like an actual lottery machine. Six distinct numbers will be entered from the keyboard, and then compared to a randomly selected set of numbers. Your program will then determine how many numbers inputted by the user match the computer-generated set.

* In your Pick6 class, modify your input code, using a while loop, to ensure that six distinct numbers are entered by the user.
	+ You may now assume that the user will only enter in numbers between 1 and 49.
	+ If a user enters a number already chosen, print out the error message indicating that particular number was already chosen. For example, if 6 was already used, then an appropriate error message would be

Number 6 has already been chosen. Choose again.

* + Be sure that the loop continues to prompt for numbers until 6 are read in.
	+ HINT: Use an ArrayList to store values from input. The ArrayList should be created to hold objects of type **Integer**. Note that it is **Integer**, not **int**. ArrayLists cannot store primitive data types, only object data types. If a particular value from the user is not in the current ArrayList, then you can be sure that the user has just entered in a distinct value. The ArrayList can then be updated appropriately.
* Using a RandomNumbers variable, obtain an ArrayList of randomly selected integers. Store this in an ArrayList<Integer> reference variable called **winningNumbers**. Using a for each loop, print out the contents of winningNumbers to the screen.
* Finally, using a for each loop, check the numbers between your ArrayList and the ones contained in winningNumbers.
* Your code should produce output similar to Figure 1’s output.



Figure : Sample Input/Output

* When finished, test your work until you are sure that it functions properly. When ready, **add your name(s) as a comment at the top of the class, print out a copy of the class and attach it to your lab submission.**

#### Part 4: Club Membership

In this part of the lab, you will model club membership records. Member objects will be added to and manipulated by an ArrayList.

## In BlueJ, open the club project. Your task is to complete the Club class, an outline of which has been provided in the project. The Club class is intended to store Member objects in a collection.

* Within the **Club** class, define a field named **members** for an ArrayList of Member objects. Use an appropriate import statement for this field. In the constructor, create the collection object and assign it to the field. Make sure that all the files in the project compile before moving on to the next section.
* Complete the **numberOfMembers** method to return the current size of the collection. Until you have a method to add objects to the collection this will always return zero, of course, but it will be ready for further testing later.
* Membership in a club is represented by an instance of the **Member** class. A complete version of **Member** is already provided for you in the **club** project, and it should not need any modification. An instance contains details of a person’s name, and the month and year in which they joined the club. All member details are filled out when an instance is created. A new **Member** object is added to a **Club** object’s collection via the **Club** object’s **join** method, which has the following description:

 **/\*\***

 **\* Add a new member to the club’s collection of members.**

 **\* @param member The member object to be added**

 **\*/**

 **public void join(Member member)**

 Complete the **join** method.

* When you wish to add a new **Member** object to the **Club** object from the object bench, you must first create a **Club** object. Then, you can add a member in one of two ways:
	1. Create a new **Member** object on the object bench, call the **join** method on the **Club** object, and click on the **Member** object to supply the parameter;
	2. or call the **join** method on the **Club** object and type into the constructor’s parameter dialogue box (where month and year are numbers):

 **new Member(“name”, month, year)**

Each time you add one, use the **numberOfMembers** method to check both that the **join** method is adding to the collection, and that the **numberOfMembers** method is giving the correct result.

* The **joinedInMonth** method of the **Club** class has the following description:

 **/\*\***

**\* Determine the number of members who joined in the**

**\* given month.**

**\* @param month The month we are interested in.**

**\* @return The number of members**

**\*/**

**public int joinedInMonth(int month)**

Modify this method so that if the **month** parameter is outside the valid range of 1-12 an error message is printed and zero is returned. Otherwise, a for each loop is used to determine (and then return) the number of members who joined that month is returned.

* The **purge** method of the **Club** class has the following description:

 **/\*\***

**\* Remove from the club’s collection all members who joined in**

**\* the given month, and return them stored in a separate**

**\* collection object.**

**\* @param month The month of the membership.**

**\* @param year The year of the membership**

**\*/**

**public ArrayList<Member> purge(int month)**

Modify this method so that if the **month** parameter is outside the valid range of 1-12 and error message is printed and a collection object with no objects stored in it is returned. Otherwise, locate and remove all members who joined in that month. To do this:

1. Use a for each loop to populate an ArrayList with all members who should be purged for the **month**.
2. Use the ArrayList of purged members and the **removeAll** method for ArrayLists to remove all of the members for a given month at once.
3. Return the ArrayList of purged members.

Note: The **purge** method is significantly harder to write than any of the others in this class, but you can do it!

* When finished, test your work until you are sure that it functions properly. When ready, **add your name(s) as a comment at the top of the class, print out a copy of the class and attach it to your lab submission.**

