Core / Intermediate Java

Lab Exercise Suggestions

January 2016, Simon Roberts

Zeller's Congruence 1

Find "Zeller's Congruence" on Wikipedia. This formula returns the day of the week based on three numbers (day of month, month of year, year) provided to it.

Write code that performs this calculation, take initialized variables day, month, year (hard code these values), then calculate and print out the number representing the day of the week for the day, month, and year values in the code.

Change the initialized values a few times to verify that the code works.

Notes:

- In the calculation, Zeller's Congruence modifies the month/year such that the months January and February are treated as month numbers 13 and 14 of the preceeding year. Use the ternary operator (that is, <boolean> ? <if-true-value> : <if-false-value>) to perform this modification.
- It might simplify the exercise if you calculate sub-expressions in stages, rather than the whole thing at once
- If completed, modify the program to prompt the user for day, month and year, and read these values from the keyboard.

Temperature Converter

Prompt the user to enter a number, convert this from text to a floating point value, then assuming that the entered value represents a Fahrenheit temperature apply the formula: **c = 5 x (f – 32) / 9** to convert to a Celsius temperature.

Print out the result.

String Chewing

Prompt the user to enter some text.

Generate a random number x representing the position of a character in the input text.

Print x and the character at that position.

Print the text modified by removal of the character at position x.

That's Mister To You!

Prompt the user to enter text with a first and a last name, entirely in lower case, and separated by at least one space.

Prepare text that starts with “Mr.” or “Ms.” as you choose, followed by the first

and last names modified to have upper case first letters.

Print the resulting text.

Guessing Game Setup

Consider a game in which the computer simulates picking four colored pegs, ordered from left to right. The human player would then make guesses about what colors are in what positions.

In this exercise, create an enum to represent peg colors, and an array of four such colors.

Use a loop and a random number generator to initialize the array of colors, simulating the computer setting up the game board.

Print out the game board something like this:

```
RED GREEN BLUE BLACK
```

Zeller's Congruence 2

Modify, or copy and modify, your solution to the Zeller's Congruence 1 exercise. Make these changes:

- Use `if` / `else` rather than the ternary operator to handle the adjustments to month/year values for months January and February (this is for the exercise, it's not an improvement!)
- Define an `enum` type for the days of the week
- In a loop, prompt the user for day, month, and year numbers, calculate the day of the week, and use the `enum` to present the result as the name of the weekday.

Las Vegas 1

Simulate throwing two dice, print out the values

Print special messages for each of: double six, scores totaling 7, and double one.

In Range 1

Prompt for and read a minimum number

Prompt for and read a maximum number

Repeatedly:

- Prompt for and read a number
- Indicate if the number is in range, too low, or too high

In Range 2

Notify user “I'm thinking of a number between 1 and 100”

Prompt user to enter a guess, and indicate "warm", "cold", "very cold" depending how close the guess is. Also indicate "higher!" or "lower!" depending on the direction the guess should move.

Repeat until the user guesses correctly, finally printing a congratulatory message and how many guesses were made.

Text Alignment 1

Read three short lines of text from the user, then print them out with sufficient leading spaces to make them right justified on a 60 column page (be sure your output is sent to a console using a fixed pitch font!)

Text Alignment 2

Read three short lines of text from the user, then print them out with sufficient extra spaces embedded between words to make the lines justified (that is, flushed left with both left and right margins) on a 60 column page (be sure your output is sent to a console using a fixed pitch font!)

Wake Up

Repeatedly:

- Prompt for a day of week
- If the day represents Monday→Friday, print "wake up"
- If the day represents Saturday/Sunday, print "lie in"

Options:

- Use a numeric coding (1=Sunday etc.) representing day of week
- Use `String` to represent day of week, and perform comparisons with an `if`/`else` structure
- Use `String` to represent day of week, and perform comparisons with a `switch`/`case` statements
- Use `String` to represent day of week, and perform comparisons with using `Set` membership
- Implement using `enum` / `ordinal()` and valueOf(String)

Las Vegas 2

Simulate 1000 throws of a pair of dice, count the number of times each "face value" (the sum of the two dice values) shows, and print the frequency of each face value in a table.

Las Vegas 3a

Create an array of (at least) six `String` objects such that the values in the array represent the images (use text descriptions) on the wheel of a “one-arm bandit” machine. (Examples are “Triple Bar”, “Bar”, “Cherry”, “Orange”, “Seven”, “Coin”)

Generate three random numbers that are suitable for use as indexes into the wheel array and use these numbers to print out the result of “pulling the handle”.

Example output: `Bar : Triple Bar : Orange`

Las Vegas 3b

Take your solution to Las Vegas 3a and extend it as follows.

Create a two dimensional array of (at least) four winning wheel combinations. Each minor array should have the three wheel positions (e.g. “Bar”, “Bar”, “Bar” that are the particular winning combination) and the textual description of the prize, e.g. “Ten Dollars!”

After the result of pulling the handle has been determined, search the two dimensional array, looking to see if the result is a winning position. If a win is found, print out the prize below the regular output of the three image names.

Example output:

```
Triple Bar : Triple Bar : Triple Bar
Sixty-four Thousand Dollar Jackpot!
```

Code Breaker

A “Caesar Cipher” is a simple cipher that works by “adding” an offset to the letters of the plain text. For example, given a key of 5, a letter “a” would be encoded as a letter “f”. The letter “z” would wrap round and become a letter “e”. Spaces and punctuation is left unchanged.

The goal of this lab is to create a program that uses a “brute force” approach to breaking messages that are in this code.

Decrypting a message may be handled by the same process as encyption, using a key that is 26 minus the original enryption key. This works since the processing of the message text is effectively “circular”; that is, clock arithmetic, or a

modulo calculation. Since there are 25 possible keys (key 0 has no effect), your goal is to read a message from the console, and then output the effect of all 25 keys on the message. One of them, and usually only one, will be recognizably readable, the others will not.

Suggested approach:

Read the input text into a String, convert the text to lower case, then extract an array of all the chars in the message.

Duplicate the array 25 times, keeping track of the “number” of the copy.

Then for each copy work along each character of the array in turn.
If the character, let's call it c, is in the range 'a' <= c <= 'z' then add a key equal to the copy's number to each character in the array (so, for one array, add 1 to every character, for the next add 2, and so on). Be sure that if the resulting character value is greater than 'z' you adjust it so that it effectively wrapped round from 'z' to 'a'.

Do not modify characters that are outside the range of alphabetic characters.

After creating the 25 additional arrays, print out the key number and the converted text.

Test your program on the following text, determine the message, and the key used to decrypt it:

```
qcbufohizohwcbg, mci rwr wh!
```

Guessing Game

This exercise builds on the earlier Guessing Game Setup lab. As previously described, in this game, the computer simulates picking four colored pegs, ordered from left to right. The human player makes guesses about what colors are in what positions.

At each guess, the computer reports the accuracy of the guess in the form:

```
RED X BLUE X
```

such that a named color indicates a successful guess of that color in that position, and an X indicates the guess was incorrect.

The following outline design approach might be helpful.

If you have not already done so, code the setup:

Create an enum to name five colors of your choosing and create a four-element array of these colors and initialize each element to represent a random color.

Prepare a flag variable to indicate when the user has guessed correctly, and use that flag to control a loop.

In the loop:

Prompt for and read the user's guess, as a space separated list of four colors

Convert the input text into four words, and then into an array of four color enum values.

Compare the guessed colors with the colors of the computer's guess. If a color is correct, print its name and increment a counter. If the color is incorrect, print an X.

If the count of correct guesses is four (i.e. all are correct) then set the flag indicating the user has won, and print a congratulatory message.

Calendar

Code Zeller's Congruence as a method.

Write another method that takes a month and year and with the help of the Zeller's Congruence method, uses loops to print out a calendar for one month.

From the main method, prompt the user for a month and year, convert these input values as appropriate, and use them to call the calendar printing method.

Notes:

- Start the week on Saturday, print three letter weekday names as column headings.
- Leave empty days of the month as blank space, e.g.:

```
Sat Sun Mon Tue Wed Thu Fri
                            1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30  31
```

Palindrome Checker

Implement this program using a recursive method call. As a hint, note that any String of length less than two characters is a palindrome. Further, any String that has the same first and last characters, and the rest of the String (other than the first and last) is also a palindrome, is also a palindrome.

Don't forget that you must ignore whitespace, punctuation, and capitalization.

The behavior should be as follows:

Prompt the user to enter some text.

Determine if the entered text is a palindrome or not, and print a message accordingly.

Some palindromes you can use for testing are:

- A Santa dog lived as a devil God at NASA
- Able was I ere I saw Elba
- Go deliver a dare, vile dog
- Racecar
- Some men interpret nine memos

Word List

The goal of this exercise is the create a list of all the words that occur in a text document, and print that list in alphabetical order.

A suggested approach is as follows:

For each line of the input file, read the line and split it into separate words.

Convert each word to lower case, and then add it to a `HashSet<String>`.

When the entire file has been read, make a `LinkedList<String>` from the words in the set.

Next, sort the list.

Finally iterate over the sorted list, printing each item out.

Concordance

The goal of this exercise is the create a table of all the words that occur in a text document, so that the table has the word, followed by the number of times that the word occurs in the document.

A suggested approach is as follows:

Create a `Map<String, Integer>` in which to build the table.

For each line of the input file, read the line and split it into separate words.

For each word, convert the word to lower case, and determine if the word is already in the map. If it is there, extract the number stored against it, increment that by one, and re-store the word/number pair. If the word is not in the map, add it alongside the value 1.

When all the lines of input have been processed, extract the `Set<String>` that are the keys of the map, then iterate over the elements of that set, printing out the key, and the value found in the map against that key. (See over for extra credit options)

Extra credit options for Concordance:

- Convert the key set into a list, and sort it, so the table is printed in alphabetical order of the words.
- Use the String.format, or System.out.printf formatting behaviors to print the table neatly, with the word right-justified in a 30 character wide field, and the number also right justified alongside the word, in a 5 character wide field.

File Server 1

The goal of this program is to create a (very) simple web server, allowing your browser to read read text files over the network.

Create (at least) two short plain text files in the root directory of your IDE project.

Your server program should:

Open a `ServerSocket` on a port number that is not in use (port 9000 is often available)

Then, in a loop, the program should:

Accept an inbound connection and extract the input and output streams from that connection.

Wrap the input and output in a `BufferedReader` and `PrintWriter` respectively.

Read a line from the socket input and attempt to parse it as an HTTP GET request in this form:

```
GET /<filename> HTTP/1.1
```

Extract just the filename part of the request, and then attempt to open the file, and send all the text of the file to the output of the socket.

Ensure that the socket and the file are both closed after transmission of the file.

continued...

Verify that if you direct your browser in the form:

```
http://localhost:9000/Myfile.txt
```

the program should send the contents of Myfile.txt to the browser.

Note:

- Note that the behavior of this server violates many requirements of the HTTP specification, but it's usually sufficient to work with text files in most browsers.

File Server 2

Add “Not Found” error handling to the server. If a requested file is not found, then instead of simply crashing the server, arrange to return the response String:

```
"HTTP/1.1 404 Not Found\r\n\r\n"
+ "<HTML><BODY>Not Found</BODY></HTML>"
```

Birthdays 1

The goal of this lab is to read data from a structured text file into a collection of Birthday objects, and then iterate over that collection and print the information out.

A suggested approach is as follows:

Start by defining a new `public` class called `Birthday`. Give the class five `public` fields. Two of the fields are of `String` type, called `firstName` and `lastName`, the remaining three are `int` type, called `day`, `month`, and `year`.

Create a plain-text file containing about a dozen birthday entries (you can either make these up, or find some from http://www.famousbirthdays.com/). Use a comma separated values (csv) format, for example:

```
Van Halen, Eddie, 1, 26, 1955
```

In the code, create a `List<Birthday>` to store the data read from the file.

Read the file, one line at a time.

For each line read:

Split the line on the commas, and trim any whitespace.

Create an empty `Birthday` object, and then set the first and last name fields.

Convert the three numeric fields to integers, and set the day, month, and year fields of the birthday.

Add the `Birthday` to the `List`.

After reading all the input, print out each birthday like this:

<firstname> <lastname> was born on <month> / <day> in <year>

Note:

- Some IDEs have special treatment of files with the extension `.csv`. If you find this is happening, simply rename your data file with an alternate extension such as `.csvt` (comma separates values, text) and the inconvenience should be avoided.

Birthdays 2

In this lab you will extend and improve the code you created in the previous lab, Birthdays 1. Copy your project first (so you do not damage the original). Then make the following two enhancements:

First, instead of hard-coding the name of the birthday-data file, prompt the user to enter this information. Ensure that if the file specified by the user does not exist, the program loops round and re-issues the prompt allowing the user to retry.

Second, define a new exception class `BadDateException`. when creating the `Birthday` object, perform tests to ensure that the day/month/year combination is valid. If it is not, arrange to throw a `BadDateException` to indicate this failure. Handle the exception so as to print a message reporting the problem, and skip the processing of the offending line of the data file. Verify the behavior by temporarily making changes to the data file.

Birthdays 3

In this lab you will further extend the Birthdays 2 example. Copy your project again, and then make the following changes:

Modify all five data fields to be `private`. Provide accessor (a.k.a. getter) methods to allow users of the `Birthday` object to read the values.

Modify the code that uses the `Birthday` class so that it uses the new accessor methods to get the field values from the object.

Provide the `Birthday` class with two constructors, one takes only the names, and sets the birthday to today's date. The second constructor takes all five fields and checks the validity of the date.

Modify the code that uses the `Birthday` class so that the check for bad dates is performed only in the constructor, but otherwise behaves the same.

Verify that the code works as before both with good and bad date data in the file.

Birthdays 4

Again copy your solution to Birthdays 3, then provide the `Birthday` class with a `toString` method that presents the data in the form described in Birthdays 1 above. Modify the program so it uses this behavior in presenting the output.

Add a method `getSuggestedGift()` to the `Birthday` class. The method should provide a gift suggestion based on the age of the person. For example, if someone is pre-teen, you might suggest "books and toys", or, for someone between 13 and 24 you might suggest "money". Arrange for the suggested gift to be presented alongside each birthday in the output.

Birthdays 5

Again copy your project, then arrange for the `Birthday` class to implement the `Comparable` interface. Define the order so that it is based on the `lastName` field of the object.

Next, modify the program so that after collecting the list of birthday information, the list is sorted before the data is output.

Birthdays 6

Again copy your project, and create a new class `BirthdayNameComparator` that implements `Comparator<Birthday>`. Define this class so that it orders birthdays based on the `year`, `month` and `day` fields of the birthdays. Remember that the month is only considered if the year fields are the same, and similarly for the day.

Again, print the list after sorting.

Custom Awards

In this exercise you will simulate an awards handling system. Participants in the awards system might be customers or employees or other entities. Each type of participant can earn the right to select awards of different types, and from different catalogs, based on various criteria.

To keep the exercise within a reasonable level of complexity, you will only simulate the changing catalogs, not the logic of determining which of these various privileges might apply at any given moment.

You will exercise your simulation by creating instances of the two different types of participant (customer and employee), providing them with a catalog, extracting gift options, changing the catalog and repeating the extraction of gift

options.

In a real system, awards catalogs of this type might be implemented in very different ways (e.g. one might be a webservice, while another is a database). To facilitate this, define an interface as a basic generalization of the catalog mechanism. You might call this interface `AwardsCatalog`. The key behavior of this generalization will be represented by a method somewhat like this:

```
public interface AwardsCatalog {
  Set<String> getItemList(int maxPoints);
}
```

The method `getItemList` takes a number of "points" – being award points accumulated by the participant over time – and returns textual descriptions of the awards that can be had for no more than that number of points.

Implement the `AwardsCatalog` in two unrelated classes, such that each carries a different collection of gifts (the collections may overlap if you wish).

Your system should also define two types of participant, `Customer` and `Employee`. Each may have a number of fields and behaviors that relate to the basic abstraction they model (such as name, credit limit or salary). Don't spend too long on those aspects of the model however. Both these participants should have an internal representation of the number of points they have accumulated (in a real system, this would be dynamic, but for the simulation, keep it simple; probably just set a value when creating the object).

Each participant class should also implement an interface `Awardable`, which defines two methods something like this:

```
public interface Awardable {
  void setAwardsCatalog(AwardsCatalog catalog);
  Set<String> getAwards();
}
```

The first method, `setAwardsCatalog`, is used to set or change the catalog currently offered to the participant. The second method returns the awards that this particular participant may obtain at this moment. The `getAwards` method uses the catalog provided by the setAwardsCatalog, and the number of points that are currently defined in the object.

To test the system, your main method should:

Create one of each type of participant.

Create one of each type of catalog, and apply a catalog to each participant.

Extract and print the set of gifts from each participant.

Change the award catalog of a participant, and again extract and print the set of gifts.

continued...

Optional:

- Consider how would you take this design, without modifying any existing code other than the test code in the main method, and allow two catalogs to be available to a single participant? If you have time, go ahead and implement this.

Zoo Keeper

In this exercise, you will model “feeding time at the zoo”. Start by defining an abstract base class `Animal`. This should model the following concrete features: weight, favorite food, noise made (e.g. “quack”, or “roar”). Define any constructor necessary to allow intialization of the private members. Define an abstract behavior “feed” that takes a type of food (it's probably best to model food types simply as text, to avoid over-complicating the exercise).

Define a small number of concrete subclasses, such as `Wolf`, `Lion`, and `Elephant`. Ensure that the parent `Animal` class is properly initialized when constructing instances of any of these. Give each of these animals a distinct behavior—printing out messages, so the behavior is observable—when fed (this should implement the abstract behavior declared in the `Animal` class). Give each concrete class a `toString` method that presents the animal in a readable fashion.

In the `main` method create a collection of `Animals`, then iterate the collection printing each one out.

Next, iterate the collection again, and build behavior that feeds each animal. Ensure you give each animal its favorite food.

When run, you should see a list of the animals in the zoo, and then the behavior of each as it is fed.