Java Lab Exercises

Zeller’s Congruence 1

"Zeller's Congruence" is a means of computing the day of the week of a given date (day of month, month of year, and year).

To perform the computation:

1. Obtain the day, month, and year, in numeric form.
2. Calculate values for two variables m, and y. If the month is January (1) or February (2) then m is the month number plus 12, and y is the year minus one. Otherwise, m is simply the month number and y is simply the year number.
3. Add one to the value of m and multiply the result by 13. Divide that result by five. Take the integer part of this result.
4. Add the day of the month to the result of step 3.
5. Add the value of y to the result of step 4.
6. Divide y by 4, and take the integer part of the result. Add this to the result of step 5.
7. Divide y by 100 and take the integer part of the result. ***Subtract*** this from the result of step 6.
8. Divide y by 400 and take the integer part of the result. Add this to the result of step 7.
9. Reduce the result of step 8 modulo 7. This gives the day of the week represented as 0 = Saturday through 6 = Friday.

The computation can be expressed as a *mathematical expression* (i.e not directly valid Java code) as below.

Notes:

1) This expression is written in terms of the variables y and m as calculated in step 2 above; that means these are ***modified*** from the month and year values

2) The divisions must be performed as integer, rather than floating point, divisions, or the calculation will fail.

```
day-of-week = ( day + (13 x (m + 1) / 5)
                + y + (y / 4) - (y / 100) + (y / 400)
        ) remainder 7
```

Write code that performs this calculation, take initialized variables day, month, year (initially, 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:

- Use the ternary operator (that is, <boolean> ? <if-true-value> : <if-false-value>) to perform the conditional modification of the month and year to create the m and y values.
- 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 1

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 below to convert to a Celsius temperature:
c = 5 x (f – 32) / 9.

Print out the result.

Temperature Converter 2

Define an enum type `com.mystuff.temperatures.TempNames` that names the two temperature scales `FAHRENHEIT` and `CELSIUS` (note that convention dictates that constant values are all uppercase in Java code). Add an import statement to your code for Temperature Converter 1, then concatenate the appropriate value to the output message in Temperature Converter 1.

Delete the import statement, save the file, and observe the error reported by the IDE. Search the menus of your IDE for an entry of the form `Source -> Fix Imports` (Netbeans), or `Source -> Organize Imports` (Eclipse). Execute this menu item, and observe the effect.

Again delete the import statement and modify the references of the form `TempNames.FAHRENHEIT` to `com.mystuff.temperatures.TempNames.FAHRENHEIT`

What do you notice?

String Comparisons

Declare and initialize three String variables hello, world, and helloWorld. Assign to them the literal values “Hello”, “ world”, and “Hello world” respectively (note the leading space on the second of those strings.)

Declare a fourth string, joined, and assign to it the result of concatenating the strings hello and world.

Print out the values of helloWorld and joined (both should appear identical).

Print out the value of these comparisons:

- `helloWorld == joined`
- `helloWorld.equals(joined)`

What does the output tell you?

Next, declare a variable of type `StringBuilder`. Call it `builder`, and initialize it with a `StringBuilder` created using the value of `helloWorld` as an argument. Repeat this creating a variable called builder2 referring to another `StringBuilder` initialized using the same string.

Print the following expressions:

- `builder`
- `builder.equals(helloWorld)`
- `helloWorld.equals(builder)`
- `helloWorld.equals(builder.toString())`
- `builder.equals(builder2)`

What does the output tell you?

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 the value of x and the character at that position.

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

That's Mister To You!

Prompt the user to enter text with a first and a last name, entirely in lowercase, 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 uppercase first letters.

Print the resulting text.

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 purpose of the exercise, it's not necessarily an improvement!)
- Use a `switch` statement to present the result as the name of the weekday.

Zeller's Congruence 3

Modify, or copy and modify, your solution to the Zeller's Congruence 2 exercise. Make these changes:

- In a loop, prompt the user for day, month, and year numbers, present the result as the name of the weekday.
- After printing the day of the week issue the prompt “Do you want to run again?”. If they enter yes then re-execute the loop. If they enter anything else, exit the program.

Optionally, further modify the program so that the user is required to enter either yes, or no, and if other text is entered

(such simply N) they are prompted to re-enter their answer to the “do you want to run again” prompt.

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.

Note, for this you will probably want to use the function: `ThreadLocalRandom.current().nextInt(1, 7)` which generates an integer in the range 1 to 6 (that’s not a typo!). The `ThreadLocalRandom` class is in the package `java.util.concurrent`, and will need to be imported.

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 the total of how many guesses were made.

Oracle of Delphi

Create an advice application. The program prompts the user for a question, and then delivers a vague, randomly selected, response. It then prompts for another question (notice below the prompt has changed), but if "quit" is entered, it offers another piece of generalized advice and then quits. Obviously, all the answers are random, and so vague as to be frustratingly useless! An example interaction might look like this (user input in bold italic):

```
Welcome to the Oracle of Delphi
What do you need advice about?
Should I get a new car?
Ask again later
Would you like advice on another issue?
Should I buy a used car?
The signs are positive but the future is hazy
Would you like advice on another issue?
What's the best color for a kid's bedroom?
Difficult to say, go with your instincts
Would you like advice on another issue?
No
```

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)

Students

Prepare a List containing the names of several students (represented as Strings).

Prompt the user for the name of a student and report if that name is in the list or not.

Favorite Foods

Create a Map containing some names as keys, and the name of that person's favorite food as the value associated with the key.

Prompt the user for the name of a person, and report that person's name, or "unknown" if the person is not found in the table.

Students 2

Extend the previous Students exercise so that it prints out the names of all the students in the form of a roster, alongside the word "present", like this:

```
Fred - present
Jim - present
```

Favorite Foods 2

Extend the previous Favorite Foods exercise so that it prints all the entries in the table, in the form:

```
Fred likes eggs
Jim likes pancakes
```

To do this, notice that, unlike the `List`, the `Map` cannot be used directly in the enhanced for loop. However, a `Map` can give us a "`Set`" of its keys. A `Set` is very much like a `List`, except it doesn't keep track of the order of elements, and it only permits one copy of each item it contains. Extract the set of keys and use the enhanced for loop to examine each in turn. Look that key up in the map and print the message.

Las Vegas 2

Simulate 1000 throws of a pair of dice, use a `Map<Integer, Integer>` to 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.

You will need to use the “face value” as the key in the table. If the value is not yet in the table when the value is encountered, put a value of 1. If the value is in the table, add one to it, and put the incremented value back into the table.

Las Vegas 3a

Create a `List` of (at least) six `String` objects such that the values in the `List` 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 another `List` of `Integer`, and add to it three random numbers that are suitable for use as indexes into the wheel descriptions `List`. Use these to print out the result of “pulling the handle”. For example:

```
Bar : Triple Bar : Orange
```

Las Vegas 3b

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

Create a `List<List<String>>` and populate it with four “winning” wheel combinations. Each minor `List` should have the three wheel positions (e.g. “Bar”, “Bar”, “Bar”) and a fourth item which is the textual description of the prize, e.g. “Ten Dollars!”

After the result of pulling the handle has been determined, search the outer list, looking to see if the result is a winning

position defined in one of the inner lists. 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 encryption, using a key that is 26 minus the original encryption 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 lowercase, 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

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.

Next, 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.

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.

Zeller’s Congruence 4

Modify your solution to Zeller’s Congruence 3 so that the day of week computation is extracted as a separate method. Call

that method from inside the loop that prompts the user for day, month, year, and “do you want to run again?”.

Calendar

Create a new program that prompts the user for a month and year. The program then uses loops to print out a calendar for one month.

Use the method you extracted in Zeller’s Congruence 4 to support your program (use it to determine the day of the week of the first day of the month).

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

Write a program that tests user input to see if it is a palindrome (text that has the same letter sequence read backwards as forwards)

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.

Be sure to create a method that takes the String to test as an argument and returns a boolean value indicating if that argument is a palindrome or not.

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

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

Optional: Implement the palindrome checking behavior using recursion. 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.

Divide By Zero

Write a program that reads two int values from the user, and prints the result of dividing one by the other.

Run the program and observe the effect of entering a zero for the divisor. You should see the program crashes with an error report on the lines of:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
```

Modify the program to add a `try` / `catch` construction such that if the problem arises, the program catches it and prompts the user to re-enter the two numbers.

Note: Java's exception mechanism is just another form of "conditional" logic. It does not create loops, however, the flow control of the exception mechanism is entirely interoperable with the other flow control mechanisms. So, if you declare a `boolean` variable called `success` (for example), and initialize it to `false`, then this can be used to control a `while` loop. If the division succeeds, set the variable to `true`, but if the exception arises, print the message requesting retry in the `catch` block, and leave the value of `success` alone (i.e. `false`), causing the loop to re-execute.

Say Please!

Write a program that reads a "request" from the user. Pass the request into a method called `askYourMother` which takes that request and in the normal course of events returns either "Yes" or "No". Decide whether to return "Yes" or "No" on any criteria you like (perhaps use the random number generator described in Las Vegas 1 above).

Add a loop in the main method so that as long as the response is "Yes" the user is prompted for another request. When the answer is "No", have the program exit.

Run the code and verify that it works.

Next, modify the method `askYourMother` so that if the request does not start with the word "please" the method throws:

```
new Exception("Where are your manners!?")
```

Note: Don't forget that the method must declare that it throws this exception.

Next, add a `try` / `catch` construction so that if the exception is thrown, the message in the exception is printed, and the program exits.

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.

Optional:

- 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 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.

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.

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 called `birthdays.txt` containing about a dozen birthday entries (you can either make these up, or find some from `http://www.famousbirthdays.com/`). Use a format like this:

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

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

Use this code to read the contents of the file:

```
Scanner sc = new Scanner(new
FileReader("birthdays.txt"));
sc.useDelimiter(Pattern.compile(
  "(, *)|(/ *)|( *\n *)"));
while (sc.hasNext()) {
  String last = sc.next();
  String first = sc.next();
  int day = sc.nextInt();
  int month = sc.nextInt();
  int year = sc.nextInt();
  // Create the birthday object and store it
}
```

After reading all the input and populating the `List<Birthday>`, iterate over the list and print out each `Birthday` like this:

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

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).

Modify your `Birthday` class by adding a method with this signature:

```
public String getAsText()
```

Arrange that the method returns a textual representation of this particular birthday.

Modify the code in the loop that prints out the birthdays information so that it simply says:

```
for (Birthday b : theListOfBirthdays) {
  System.out.println(b.getAsText());
}
```

Change the name of the `getAsText` method to `toString`. Modify the print statement in the loop above so it simply says:

```
  System.out.println(b);
```

What do you notice?

Create a method called getFullName. This should return the name of the person, with the first and last parts concatenated. Test the method by printing the names of every birthday.

Create another method called `getAge`. This should return the age (in whole years) of the person described in the birthday. You can use the following example to see how to obtain the day, month, and year of the current date, in numeric form:

```
LocalDate ld = LocalDate.now();
System.out.println("Day "
     + ld.getDayOfMonth());
System.out.println("Month "
     + ld.getMonth().getValue());
System.out.println("Year "
     + ld.getYear());
```

Arrange to print out each person's name and age in the form:

```
John Smith is 28 years old.
```

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. Add this to the output along with name and age.

Cars

Create a class called `Mustang`. Give it a private field called `speed` and a `public static` field called `MAX_DESIGN_SPEED`. Initialize the static field to a value of 125. Create two methods that interact with the speed field:

```
public int getSpeed() { return this.speed; }
public void setSpeed(int speed) {
  this.speed = speed;
}
```

Create a `toString` method for the `Mustang`, and arrange for this method to format a string that represents the values of `this.speed` and `MAX_DESIGN_SPEED`

Create three instances of `Mustang`, and give each a different speed, then print each object out. What do you notice about the values shown?

Now, use one of the three references to `Mustang` objects, and set the value of `MAX_DESIGN_SPEED` using it. Again print all three objects out. What do you notice about the values this time?

Birthdays 3

Next, give the class a constructor that takes the five elements as arguments and returns a fully constructed and initialized Birthday.

Change the accessibility of the five fields of the Birthday class so that they are all `private`.

Modify the calling code so that it creates the birthdays using this constructor and make any other necessary changes.

Run the code again to show that it still works as before.

Modify your constructor so that it tests to ensure that neither the first nor the last names provided are null pointers, nor are they empty strings. If they are, refuse to complete creation of the object, instead throw an IllegalArgumentException, with the message “Names must not be empty or null”.

Test this by manually calling the constructor with a null field (you cannot readily create a null entry from the text file). Arrange a try/catch block around the attempt so that the program prints a message, rather than simply crashing.

Optional: 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 (but ensure the remainder of the file is properly handled). Verify the behavior by temporarily making changes to the data file.

Things With Names

Define an interface called `Named`. Arrange that the interface declares a single method:

```
String getFullName();
```

Define a class called `Dog`, and have that class implement the `Named` interface. Give the dog fields, methods, and a constructor sufficient to support some basic dogness and satisfy the `Named` interface. (Don't get too carried away! Dogness can simply be having a method called `bark` that prints "woof" when called.)

Create a main method, and in that, create a List<Dog>, add a few (perhaps 3) dogs to it. Create a supporting method called

```
public static void printAllDogNames(
     List<Dog> dogs)
```

Arrange that the method prints out the names of everything in the list. Call the method with the list of dogs and verify that it works.

Duplicate the method but change the declaration to:

```
public static void printAllNames(
     List<? extends Named> items)
```

Call this method too, and verify that it works identically to the original.

Create a new class called `Cat`. Mimic the dog notion, creating instead an essential cat-ness. Ensure the `Cat` too implements the `Named` interface.

Create a `List<Cat>` and populate it.

What happens if you try to call `printAllDogNames` with this list?

What happens if you try to call `printAllNames` with this list?

Modify the `Birthday` class so that it implements `Named` too. What do you need to do to print all the names in the list of birthdays you already have?

Birthdays 4

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 as needed to implement 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.

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.

Birthdays 5

- Define an interface that represents a test on a birthday. The interface should be called `BirthdayCriterion` and define a single method (call it `test`) that accepts a `Birthday` as an argument, and returns a `boolean` result.
- Define a method that accepts a `List<Birthday>` and a BirthdayCriterion. Call the method `filterBirthdays`. It should return a new `List<Birthday>` such that all the birthdays that pass the test (that is, the test method returns `true` when applied to that birthday) are in the returned list.
- Define at least two classes that implement the `BirthdayCriterion` interface (for example, `SummerBirthday`, and `EvenNumberedDayBirthday`).
- Write a `main` method that builds a `List<Birthday>` with suitable data, then use the `filterBirthdays` method to extract lists of the birthdays that satisfy the given criteria. Print each list out.

Zookeeper

In this exercise, you will model “feeding time at the zoo”. Start by defining a base class `Animal`. This should model the following concrete features: weight, and noise made (e.g. “quack”, or “roar”). Define any constructor necessary to allow initialization of the private members. In particular, define two behaviors, `feed` and `favoriteFood`.

The 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). Because an unspecified kind of animal can't really feed, print an error message in this method.

The `favoriteFood` method should return a `String`, representing that animal's favorite food. Again, an unspecified animal doesn't have a favorite food, so this should return "unknown".

Define a small number of 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. Particularly, override the `feed` and `favoriteFood` methods. Give each 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.