Lab 3 Conditionals and Loops

Prelab Exercises

1. Rewrite each condition below in valid Java syntax (give a *boolean* expression):
   a. x > y > z
   b. *x* and *y* are both less than 0
   c. neither *x* nor *y* is less than 0
   d. *x* is equal to *y* but not equal to *z*
2. Suppose GPA is a variable containing the grade point average of a student. Suppose the goal of a program is to let a student know if he/she made the very great distinction list (the GPA must be 3.75 or above). Write an if... else... statement that prints out the appropriate message (either "Congratulations—you made the very great distinction" or "Sorry you didn't make the Very Great Distinction ").
3. Complete the program in *Salary.java* to determine the raise and new salary for an employee by adding *if ... else* statements to compute the raise. The input to the program includes the current annual salary for the employee and a number indicating the performance rating (1=excellent, 2=good, and 3=poor). An employee with a rating of 1 will receive a 6% raise, an employee with a rating of 2 will receive a 4% raise, and one with a rating of 3 will receive a 1.5% raise.

Computing a Raise

File Salary2.java contains most of a program that takes as input an employee's salary and a rating of the employee's performance and computes the raise for the employee. This is similar to question #3 in the pre-lab, except that the performance rating here is being entered as a String—the three possible ratings are "Excellent", "Good", and "Poor". As in the pre-lab, an employee who is rated excellent will receive a 6% raise, one rated good will receive a 4% raise, and one rated poor will receive a 1.5% raise. Add the *if... else...* statements to program Salary to make it run as described above.

Activities at Lake LazyDays

As activity director at Lake LazyDays Resort, it is your job to suggest appropriate activities to guests based on the weather:

```
temp >= 80: swimming

60 <= temp < 80: tennis

40 <= temp < 60: golf

temp < 40: skiing
```

1. Write a program that prompts the user for a temperature, then prints out the activity appropriate for that temperature. Use a cascading if, and be sure that your conditions are no more complex than necessary.
2. Modify your program so that if the temperature is greater than 95 or less than 20, it prints "Visit our shops!". (Hint: Use a boolean operator in your condition.) For other temperatures print the activity as before.

Rock, Paper, Scissors

Program *Rock.java* contains a skeleton for the game Rock, Paper, Scissors. Add statements to the program as indicated by the comments so that the program asks the user to enter a play, generates a random play for the computer, compares them and announces the winner (and why). For example, one run of your program might look like this:

```
$ java Rock
Enter your play: R, P, or S
r
Computer play is S
Rock crushes scissors, you win!
```

Note that the user should be able to enter either upper or lower case r, p, and s. The user's play is stored as a string to make it easy to convert whatever is entered to upper case. Use a switch statement to convert the randomly generated integer for the computer's play to a string. If you are not familiar with Rock, Paper, Scissors game search online for the details.

Date Validation

In this exercise you will write a program that checks to see if a date entered by the user is a valid date in the second millennium. A skeleton of the program is in *Dates.java*. As indicated by the comments in the program, fill in the following:

1. An assignment statement that sets monthValid to true if the month entered is between 1 and 12, inclusive.
2. An assignment statement that sets yearValid to true if the year is between 1000 and 1999, inclusive.
3. An assignment statement that sets leapYear to true if the year is a leap year. Here is the leap year rule (there's more to it than you may have thought!):

   > If the year is divisible by 4, it's a leap year UNLESS it's divisible by 100, in which case it's not a leap year UNLESS it's divisible by 400, in which case it is a leap year. If the year is not divisible by 4, it's not a leap year.

   Put another way, it's a leap year if a) it's divisible by 400, or b) it's divisible by 4 and it's not divisible by 100. So 1600 and 1512 are leap years, but 1700 and 1514 are not.

4. An if statement that determines the number of days in the month entered and stores that value in variable daysInMonth. If the month entered is not valid, daysInMonth should get 0. Note that to figure out the number of days in February you'll need to check if it's a leap year.
5. An assignment statement that sets dayValid to true if the day entered is legal for the given month and year.
6. If the month, day, and year entered are all valid, print "Date is valid" and indicate whether or not it is a leap year. If any of the items entered is not valid, just print "Date is not valid" without any comment on leap year.

Processing Grades

The file *Grades.java* contains a program that reads in a sequence of student grades and computes the average grade, the number of students who pass (a grade of at least 60) and the number who fail. The program uses a loop.

1. Compile and run the program to see how it works
2. Study the code and do the following.
    - Replace the statement that finds the sum of the grades with one that uses the += operator.
    - Replace each of three statements that increment a counting variable with statements using the increment operator
3. Run your program to make sure it works.
4. Now replace the "if" statement that updates the pass and fail counters with the conditional operator.

PreLab Exercises

In a while loop, execution of a set of statements (the body of the loop) continues until the boolean expression controlling the loop (the condition) becomes false. As for an if statement, the condition must be enclosed in parentheses. For example, the loop below prints the numbers from 1 to to LIMIT:

```
final int LIMIT = 100; // setup
int count = 1;
while (count <= LIMIT) // condition
{ // body
   System.out.println(count); // -- perform task
   count = count + 1; // -- update condition
}
```

There are three parts to a loop:

- The setup, or initialization. This comes before the actual loop, and is where variables are initialized in preparation for the first time through the loop.

- The condition, which is the boolean expression that controls the loop. This expression is evaluated each time through the loop. If it evaluates to true, the body of the loop is executed, and then the condition is evaluated again; if it evaluates to false, the loop terminates.
- The body of the loop. The body typically needs to do two things:
    - Do some work toward the task that the loop is trying to accomplish. This might involve printing, calculation, input and output, method calls—this code can be arbitrarily complex.
    - Update the condition. Something has to happen inside the loop so that the condition will eventually be false— otherwise the loop will go on forever (an infinite loop). This code can also be complex, but often it simply involves incrementing a counter or reading in a new value.

Sometimes doing the work and updating the condition are related. For example, in the loop above, the print statement is doing work, while the statement that increments count is both doing work (since the loop's task is to print the values of count) and updating the condition (since the loop stops when count hits a certain value).

The loop above is an example of a count-controlled loop, that is, a loop that contains a counter (a variable that increases or decreases by a fixed value—usually 1—each time through the loop) and that stops when the counter reaches a certain value. Not all loops with counters are count-controlled; consider the example below, which determines how many even numbers must be added together, starting at 2, to reach or exceed a given limit.

```
final int LIMIT = 16; TRACE
int count = 1; sum nextVal count
int sum = 0; --- ------- -----
int nextVal = 2;
while (sum < LIMIT) {
   sum = sum + nextVal;
   nextVal = nextVal + 2;
   count = count + 1;
}
System.out.println("Had to add together " + (count-1) + " even
numbers" + "to reach value " + LIMIT + ". Sum is " + sum);
```

Note that although this loop counts how many times the body is executed, the condition does not depend on the value of count.

Not all loops have counters. For example, if the task in the loop above were simply to add together even numbers until the sum reached a certain limit and then print the sum (as opposed to printing the number of things added together), there would be no need for the counter. Similarly, the loop below sums integers input by the user and prints the sum; it contains no counter.

```
int sum = 0; //setup
String keepGoing = "y";
int nextVal;
while (keepGoing.equals("y") || keepGoing.equals("Y")) {
   System.out.print("Enter the next integer: "); //do work
   nextVal = scan.nextInt();
   sum = sum + nextVal;
   System.out.println("Type y or Y to keep going"); //update condition
   keepGoing = scan.next();
}
System.out.println("The sum of your integers is " + sum);
```

Exercises

1. In the first loop above, the println statement comes before the value of count is incremented. What would happen if you reversed the order of these statements so that count was incremented before its value was printed? Would the loop still print the same values? Explain.
2. Consider the second loop above.
   a. Trace this loop, that is, in the table next to the code show values for variables nextVal, sum and count at each iteration. Then show what the code prints.
   b. Note that when the loop terminates, the number of even numbers added together before reaching the limit is count-1, not count. How could you modify the code so that when the loop terminates, the number of things added together is simply count?
3. Write a while loop that will print "I love computer science!!" 100 times. Is this loop count-controlled?
4. Add a counter to the third example loop above (the one that reads and sums integers input by the user). After the loop, print the number of integers read as well as the sum. Just note your changes on the example code. Is your loop now count-controlled?
5. The code below is supposed to print the integers from 10 to 1 backwards. What is wrong with it? (Hint: there are two problems!) Correct the code so it does the right thing.

```
count = 10;
while (count >= 0) {
   System.out.println(count);
   count = count + 1;
}
```

Counting and Looping

The program in *LoveCS.java* prints "I love Computer Science!!" 10 times. Compile and run it to see how it works. Then modify it as follows:

1. Instead of using constant LIMIT, ask the user how many times the message should be printed. You will need to declare a variable to store the user's response and use that variable to control the loop. (Remember that all caps is used only for constants!)
2. Number each line in the output, and add a message at the end of the loop that says how many times the message was printed. So if the user enters 3, your program should print this:
   ```
   1 I love Computer Science!!
   2 I love Computer Science!!
   3 I love Computer Science!!
   Printed this message 3 times.
   ```
3. If the message is printed N times, compute and print the sum of the numbers from 1 to N. So for the example above, the last line would now read:
   ```
   Printed this message 3 times. The sum of the numbers from 1 to 3 is 6.
   ```

   Note that you will need to add a variable to hold the sum.

Powers of 2

File *PowersOf2.java* contains a skeleton of a program to read in an integer from the user and print out that many powers of 2, starting with 20.

1. Using the comments as a guide, complete the program so that it prints out the number of powers of 2 that the user requests. Do not use Math.pow to compute the powers of 2! Instead, compute each power from the previous one (how do you get 2^n from 2^{n-1}?). For example, if the user enters 4, your program should print this:
   ```
   Here are the first 4 powers of 2:
   1
   2
   4
   8
   ```
2. Modify the program so that instead of just printing the powers, you print which power each is, e.g.:
   ```
   Here are the first 4 powers of 2:
   2^0 = 1
   2^1 = 2
   2^2 = 4
   2^3 = 8
   ```

Factorials

The *factorial* of n (written n!) is the product of the integers between 1 and n. Thus 4! = 1*2*3*4 = 24. By definition, 0! = 1. Factorial is not defined for negative numbers.

1. Write a program (*Factorial.java*) that asks the user for a non-negative integer and computes and prints the factorial of that integer. You'll need a while loop to do most of the work—

this is a lot like computing a sum, but it's a product instead. And you'll need to think about what should happen if the user enters 0.

2. Now modify your program so that it checks to see if the user entered a negative number. If so, the program should print a message saying that a nonnegative number is required and ask the user the enter another number. The program should keep doing this until the user enters a nonnegative number, after which it should compute the factorial of that number. **Hint:** you will need another while loop **before** the loop that computes the factorial. You should not need to change any of the code that computes the factorial!

A Guessing Game

File *Guess.java* contains a skeleton for a program to play a guessing game with the user. The program should randomly generate an integer between 1 and 10, then ask the user to try to guess the number. If the user guesses incorrectly, the program should ask them to try again until the guess is correct; when the guess is correct, the program should print a congratulatory message.

1. Using the comments as a guide, complete the program so that it plays the game as described above
2. Modify the program so that if the guess is wrong, the program says whether it is too high or too low. You will need an if statement (inside your loop) to do this.
3. Now add code to count how many guesses it takes the user to get the number, and print this number at the end with the congratulatory message.
4. Finally, count how many of the guesses are too high and how many are too low. Print these values, along with the total number of guesses, when the user finally guesses correctly.

More Guessing

File *Guess.java* contains the skeleton for a program that uses a while loop to play a guessing game. (This problem is described in the previous lab exercise.) Revise this program so that it uses a do ... while loop rather than a while loop. The general outline using a do... while loop is as follows:

```
// set up (initializations of the counting variables)
....
do {
   // read in a guess
   ...
   // check the guess and print appropriate messages
   ...
}
while ( condition );
```

A key difference between a while and a do ... while loop to note when making your changes is that the body of the do ... while loop is executed before the condition is ever tested. In the while loop version of the program, it was necessary to read in the user's first guess before the loop so there would be a value for comparison in the condition. In the do... while this "priming" read is no longer needed. The user's guess can be read in at the beginning of the body of the loop.

Finding Maximum and Minimum Values

A common task that must be done in a loop is to find the maximum and minimum of a sequence of values. The file *Temps.java* contains a program that reads in a sequence of hourly temperature readings over a 24-hour period. You will be adding code to this program to find the maximum and minimum temperatures. Do the following:

1. Open it and see what's there. Note that a for loop is used since we need a count-controlled loop. Your first task is to add code to find the maximum temperature read in. In general to find the maximum of a sequence of values processed in a loop you need to do two things:
   - You need a variable that will keep track of the maximum of the values processed so far. This variable must be initialized before the loop. There are two standard techniques for initialization: one is to initialize the variable to some value smaller than any possible value being processed; another is to initialize the variable to the first value processed. In either case, after the first value is processed the maximum variable should contain the first value. For the temperature program declare a variable maxTemp to hold the maximum temperature. Initialize it to -1000 (a value less than any legitimate temperature).
   - The maximum variable must be updated each time through the loop. This is done by comparing the maximum to the current value being processed. If the current value is larger, then the current value is the new maximum. So, in the temperature program, add an if statement inside the loop to compare the current temperature read in to maxTemp. If the current temperature is larger set maxTemp to that temperature. NOTE: If the current temperature is NOT larger, DO NOTHING!
2. Add code to print out the maximum after the loop. Test your program to make sure it is correct. Be sure to test it on at least three scenarios: the first number read in is the maximum, the last number read in is the maximum, and the maximum occurs somewhere in the middle of the list. For testing purposes you may want to change the HOURS_PER_DAY variable to something smaller than 24 so you don't have to type in so many numbers!
3. Often we want to keep track of more than just the maximum. For example, if we are finding the maximum of a sequence of test grades we might want to know the name of the student with the maximum grade. Suppose for the temperatures we want to keep track of the time (hour) the maximum temperature occurred. To do this we need to save the current value of the hour variable when we update the maxTemp variable. This of course requires a new variable to store the time (hour) that the maximum occurs. Declare timeOfMax (type int) to keep track of the time (hour) the maximum temperature occurred. Modify your if statment so that in addition to updating maxTemp you also save the value of hour in the

timeOfMax variable. (WARNING: you are now doing TWO things when the if condition is TRUE.)

4. Add code to print out the time the maximum temperature occurred along with the maximum.
5. Finally, add code to find the minimum temperature and the time that temperature occurs. The idea is the same as for the maximum. NOTE: Use a separate if when updating the minimum temperature variable (that is, don't add an else clause to the if that is already there).

Counting Characters

The file *Count.java* contains the skeleton of a program to read in a string (a sentence or phrase) and count the number of blank spaces in the string. The program currently has the declarations and initializations and prints the results. All it needs is a loop to go through the string character by character and count (update the countBlank variable) the characters that are the blank space. Since we know how many characters there are (the length of the string) we use a count controlled loop—for loops are especially well-suited for this.

1. Add the for loop to the program. Inside the for loop you need to access each individual character—the charAt method of the String class lets you do that. The assignment statement

   ch = phrase.charAt(i);

   assigns the variable ch (type char) the character that is in index i of the String phrase. In your for loop you can use an assignment similar to this (replace i with your loop control variable if you use something other than i). NOTE: You could also directly use phrase.charAt(i) in your if (without assigning it to a variable).

2. Test your program on several phrases to make sure it is correct.
3. Now modify the program so that it will count several different characters, not just blank spaces. To keep things relatively simple we'll count the a's, e's, s's, and t's (both upper and lower case) in the string. You need to declare and initialize four additional counting variables (e.g. countA and so on). Your current if could be modified to cascade but another solution is to use a switch statement. Replace the current if with a switch that accounts for the 9 cases we want to count (upper and lower case a, e, s, t, and blank spaces). The cases will be based on the value of the ch variable. The switch starts as follows—complete it.

```
switch (ch) {
   case 'a':
   case 'A': countA++;
   break;
   case ....
}
```

Note that this switch uses the "fall through" feature of switch statements. If ch is an 'a' the first case matches and the switch continues execution until it encounters the break hence the countA variable would be incremented.

4. Add statements to print out all of the counts.
5. It would be nice to have the program let the user keep entering phrases rather than having to restart it every time. To do this we need another loop surrounding the current code. That is, the current loop will be nested inside the new loop. Add an outer while loop that will continue to execute as long as the user does NOT enter the phrase quit. Modify the prompt to tell the user to enter a phrase or quit to quit. Note that all of the initializations for the counts should be inside the while loop (that is we want the counts to start over for each new phrase entered by the user). All you need to do is add the while statement (and think about placement of your reads so the loop works correctly). Be sure to go through the program and properly indent after adding code—with nested loops the inner loop should be indented.

Using the Coin Class

The Coin class is in the file *Coin.java*. Write a program to find the length of the longest run of heads in 100 flips of the coin. A skeleton of the program is in the file *Runs.java*. To use the Coin class you need to do the following in the program:

1. Create a coin object.
2. Inside the loop, you should use the flip method to flip the coin, the toString method (used implicitly) to print the results of the flip, and the getFace method to see if the result was HEADS. Keeping track of the current run length (the number of times in a row that the coin was HEADS) and the maximum run length is an exercise in loop techniques!
3. Print the result after the loop.

Even Odd Dialog

File *EvenOdd.java* contains the dialog box

1. Compile and run the program to see how it works.
2. Write a similar class named SquareRoots (you may modify EvenOdd) that computes and displays the square root of the integer entered.