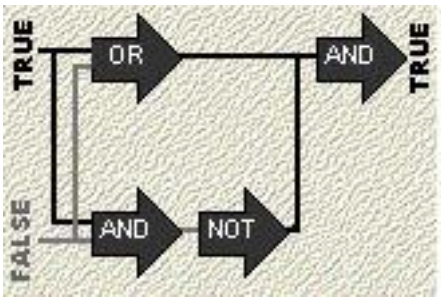

Programming Exercise 5:
Boolean Logic

Purpose: Practice with more sophisticated decision making.

Background readings from textbook: Liang, Sections 3.10-3.12, 4.4

Due date for section 001: Monday, February 15 by 10 am
Due date for section 002: Wednesday, February 17 by 10 am

Overview

As we saw in the last lab, decision making in a program can become complex. We have two choices when we need to make decisions based on multiple tests (comparisons), use nested if/if-else statements or use complex conditional statements. In lab 4, we saw that we could combine two tests that both had to be true by using AND (in Java, we did this with &&). We might also have conditions that should be true if either test is true, in which case we use OR (in Java, this is denoted using ||) or if a test is false rather than true, in which case we use NOT to negate the false value to true (in Java, not is indicated by placing ! before the test). AND, OR and NOT are *boolean* operators. Below is a table that describes the 3 boolean operators.

Boolean Operators	Meaning	Java Syntax
exp_1 **or** exp_2	True if at least one of exp_1 or exp_2 is true. False otherwise.	exp_1 || exp_2
exp_1 **and** exp_2	True if both exp_1 and exp_2 are true. False otherwise.	exp_1 && exp_2
not *exp*	True if *exp* is false. False otherwise.	!*exp*

If you have several tests that you need to combine, you can do so within the one set of parentheses if they are combined using the same boolean operator. Otherwise, you might need to nest your series of tests (called a boolean expression) in additional parentheses depending on operator precedence.

As an example of a complex conditional, imagine that we want to ensure that a user is from Clermont county and under 14 or is from Campbell county and 65 or over, or is from Warren county. We have three sets of conditions, all ORed together. That would give us `(condition1 || condition2 || condition3)`. But condition1 itself has two parts, Clermont county and under 14. This would be denoted as `county.equals("Clermont")&&age<14`. For condition2, we also have two parts, Campbell county and 65 or over, which is `county.equals("Campbell")&&age>=65`. For condition3, it is only `county.equals("Warren")`. Putting this all together, we have the following.

```
(county.equals("Clermont")&&age<14||county.equals("Campbell")&&
     age>=65||county.equals("Warren"))
```

Operator precedence says that && happens before ||, so the above condition is ok as is, but to play safe, it doesn't hurt to add parentheses. Since we want to do each && before each ||, we can rewrite it as follows.

```
((county.equals("Clermont")&&age<14)||(county.equals("Campbell")&&
     age>=65)||county.equals("Warren"))
```

Notice that the last condition does not have an && so we didn't bother to place it in its own parentheses.

Part 1: Java Examples

What follows are additional examples with explanations so that you can better understand how to express logic using boolean expressions.

1. We want to see if the input is illegal, which is the case if age is less than 0 or greater than 120.

```
if( age < 0 || age > 120 )
{
    System.out.print( age + " is not a legal value." );
}
```

2. We want to test to see if a number, n, is within a certain bounds, 1 to 100 in this case.

```
if( n >= 1 && n <= 100 )
{
    System.out.print( n + " is between within bounds." );
}
```

3. We use NOT (!) to easily negate a condition. If we want to test to see if n is not within the bounds of 1-100, we can do this.

```
if( !(n >= 1 && n <= 100) )
{
    System.out.print( n + " is not within bounds." );
}
```

We could also have tested this with if `(n<1||n>100)`. The problem is that sometimes negating a condition is tricky for programmers so its easier to use !.

Part 2: Common Pitfalls

1.

```
if( 0 <= value <= 10 )
{
    System.out.print( value +
        " is between 0 and 10." );
}
```

Syntax error!

This is actually two separate conditions that need to be combined with the "and" operator:

***0** <= **value** && **value** <= **10**.*

2.

```
if( n > 2 || n < -2 && n != 4 )
{
   System.out.print( "Absolute value of "
      + n + " is greater than 2 and "
      + n + " is not equal to 4." );
}
```

Logic error!

This returns true even if n = 4 because the "and" operator is evaluated before the "or" operator unless there are parentheses.

For n=4, n < -2 && n !=4 returns FALSE, and n>2 || FALSE returns TRUE.

Part 3: Problem

The local telephone company NkuTel offers long distance phone service to residential and commercial customers with separate rate structures for the two types of customer. Residential customers are charged a weekly rate of $5 plus 10 cents per minute for each minute over 60. Commercial customers are charged 20 cents per minute for the first 300 minutes and for each minute over 300, the rate is 15 cents per minute. NkuTel has hired you to write a program that its accounting department can use to compute the weekly long distance bill for a customer.

Ask the user to enter the type of account, **r** for a residential customer or **c** for a commercial customer. This will be stored in a variable of type **char**. If the response is neither the letter **r** nor the letter **c** then give an error message. You compare characters like you compare integers, using `==` for equality as in `(type=='r')` and != for inequality as in `(type!='r')` (assuming the variable is called type).

To input a char, we input the value as a String and then use the `charAt` method to reduce it to one character. The notation will look like this, assuming your Scanner variable is named input.

```
input.next().charAt(0)
```

You can also change the input to all upper or lower case by adding `.toUpperCase()` or `.toLowerCase()` before `.charAt(0)`. Make sure you order toUpperCase/toLowerCase and charAT in this order because `.charAt(0).toLowerCase()` would yield a syntax error since charAt gives you a single character and toLowerCase can only be used on a String (not a char).

If the user entered a correct type, then ask the user to enter the number of minutes the customer used NkuTel services for that given week and store that response in an **int** variable. Test to make sure the number of minutes is valid (greater than or equal to 0 and less than or equal to the total number of minutes in 1 week – you will have to compute this yourself). If the minutes is invalid, output an error message, otherwise continue.

If valid, continue by computing the cost. The structure for this computation will be a nested if-else structure. First, you will have to see if the type is 'r' or 'c'. If it is an 'r', then you will have to test to see if minutes is less than or equal to 60 for one computation or greater than for another. You would similarly need to break down the commercial customers between those less than or equal to 300 minutes and those greater than. Below is a skeleton for what this code will look like.

```
if( customerType=='r')
{   // residential
    if( minutes <= 60 )
    {
        ...
    }
    else  // residential minutes over 60 here
    {
        ...
    }
}
else // commercial
{
    if(minutes <= 300)
    {
        ...
    }
    else
    {
        ...
    }
}
```

After computing the cost, output the user's type, total minutes and total cost. Format the cost appropriately using DecimalFormat. For instance, the output might look like $5.45 or $12.30. Write this program. Remember to comment it well especially the if-else statements to describe the logic. Save and compile your program. Fix any syntax errors that you have before moving on to test your program.

Part 4: Test Your Program

Run your program numerous times to test it out. Use the data in the following table to see if you have it working correctly. You can make up additional tests if you like, computing what the result should be by hand.

Type	Minutes	Charge
r	20	$5.00
r	80	$7.00
c	20	$4.00
c	400	$75.00
t		Invalid client type
c	–10	Invalid number of minutes
c	11000	Invalid number of minutes

Part 5: Program Modifications and Submission

We will add two more types. Educational customer (e.g., a school) will use a type of 'e' and will be charged a flat 18 cents per minute. A preferred customer, which will be denoted as 'p', pays $10 as a base charge and 6 cents per minute. However, if the preferred customer reaches 500 minutes, then the rate is 4 cents per minute (the base stays at $10 no matter what the minutes are). Also, modify the commercial customer by giving a 30% discount for any customer who is being given a "bonus" and who has reached at least 300 minutes. You will have to add another variable called bonus (you can make this a String or a char) which, if the customer is a commercial customer of 300 minutes or more, will be input from the user ("yes"/"no" for a String, 'y'/'n' for a char). Only commercial customers with 300 or more minutes are eligible for this bonus discount. Save, compile and run your program on the data in the table below (to check your work, some of the results are given to you). Make sure your output always states the user's type, the number of minutes, if a bonus applied (only to the commercial type) and what the cost is for their bill. Collect the outputs into a text file and print and hand in or email your program and output to your instructor. Some of the outputs have the cost listed for you to check.

Type	Minutes	Bonus (for 'c' type)	Cost (for select inputs)
C	150		
e	271		
e	0		$0.00
p	-10		Illegal minutes
P	315		$28.90
R	28		$5.00
r	423		
C	301	Yes	
T			Illegal type
c	205		
C	551	No	$97.65
p	626		
E	10583		
p	45		
R	8301		
c	881	No	
a			
C	343	Yes	$46.52