In this assignment, I created a program to convert both positive and negative floating-point numbers into the IEEE754 format. I have used NetBeans to implement my project. To do so, I created a project called *Project2CSC4101*. Then, I built a class called *Convertor.java* to implement my convertor code. First, I created a static method called *IEEE754*. The inputs to this method are floating-point numbers. Therefore, the input data type should be double. Also, the output data type is a string which is called *result*.

IEEE754 has two different procedures for the floating-point numbers with the absolute value greater than one and less than one. Therefore, in the very first step, we used an if function to categorize them:

- If the input is greater than or equal 1 or less than or equal -1 (absolute value greater than or equal one), a method named *Binarize* is called. Two inputs are needed for this method, the first input is the floating-point number called *input* with double data type, and the second input is *exp_comp* which is initialized by zero with an integer data type.
- Otherwise, for the inputs ranging from -1 to 1, we created an integer type variable called *count* and initialized it with -1. After that, a float type variable is created with the absolute value of the input called *temp*. We start dividing the *temp* variable by 2^{-1}. While the *temp* value is less than one, we continue updating the *temp* variable by dividing it by 2^{-1}. At each step, we subtract one from the value of the *count*. We use the *count* variable to find out the value of exponent (*exp_comp)*. Then, we will use the *Binarize* method again. The first input is the number >1 that we reached by dividing the *input* by 2^{count}. The second input is our exponent, which is the *count* variable.

  Example:
  $0.085/2^{-1}= 0.17$ *input=0.085, temp=0.17, count=-1 and temp<1*
  $0.085/2^{-2}=0.34$ *or* $0.17/2^{-1}=0.34$ *input=0.085, temp=0.34, count=-2, and temp <1*
  $0.085/2^{-3}=0.68$ *or* $0.34/2^{-1}=0.68$ *input=0.085, temp=0.68, count=-3, and temp <1*
  $0.085/2^{-4}=1.36$ *or* $0.68/2^{-1}=1.36$ *input=0.085, temp=1.36, count=-4, and temp >1*

  Input to Binarize method: input → $input/2^{count}=0.085/2^{-4}=1.36$ *and exp_comp* → *count= -4*

From this point, the process will be the same for all floating-point numbers.

***Binarize* Method:**
As we mentioned earlier, the input to the *Binarize* method is the *input* and *exp_comp,* which are double and integer data types, respectively.
The first step in this method is to check whether the *input* is positive or negative. We use an integer type *signbit* variable initialized with 0 for it. If the *input* is less than 0, the *signbit* will change to 1, otherwise, it remains as zero. This part will build the first part of the IEEE754 format.

input>0 → signbit=0
input<0 → signbit=1

Since the inputs can be both positive and negative, we need to use the absolute value of the *input.* Therefore, we use *Math.abs* function to find it out.

Next, we need to build two parts for each number, including the whole section and decimal section for the input. We named them *whole_part* and *decimal_part*. The *whole_part* is an integer type variable, but the *decimal_part* has double data type. We calculate it by subtracting the *whole_part* from the *input* (*input - whole_part*).

Then, we use StringBuilder datatype for both the whole and decimal parts. The reason for using StringBuilder is to build mutable, or in other words, a modifiable succession of characters.

The *whole_part* will be fed into a method called *IntToBinaryString* to output a StringBuilder type variable called *wholebit*.

***IntToBinaryString* Method:**

The *whole_part* of the *input* is the input to this method with integer datatype. We have two variables here called *temp* and *remainder*, both with integer datatype. *temp* keeps the value of the input to the method at the initial step. Here, we have a StringBuilder type variable called *wholebit*. Using a for loop (for eight times), we start a division by two for the *temp*. *temp* value is updated at each step of the for loop with the division's answer. The remainder of the division will also be kept in the *remainder* variable. At each step of the remainder calculation, its value will be appended to the *wholebit* variable. However, in IEEE754 format, we need to reverse the wholebit order of data. Therefore, we did it by using the *reverse ()* function and returned it as an output of the method.

Example:
input=19.59375 → temp= 19
19/2 temp=9, remainder=1
9/2 temp=4, remainder=1
4/2 temp=2, remainder=0
2/2 temp=1, remainder=0
1/2 temp=0, remainder=1
0/2 temp=0, remainder=0

To build the fraction part of the number, we need a double temporary variable called *temp_dec* initialized with the value of the *decimal_part*. To do so, we use a for loop for 23 times. At each step, the *temp_dec* will be multiplied by 2 and the integer part of *temp_dec* will be appended to the *decimalbit* variable. Only the fraction part of the *temp_dec* will be multiplied by two, and that is the reason why we subtract the integer part of *temp_dec**2 every step from *temp_dec**2.

In the whole part of each number, only one number which should be one can be kept. Therefore, any number (0 or 1) after the first 1 should be transferred to the fraction part. We did this by using the *indexOf ("1")* function. It helps with finding the index of the first one in the whole part. Based on the number of transferred numbers to the fraction part, we build our exponent. The number of transferred numbers to the fraction part is calculated by subtracting the index of the first one from the length of the *wholebit*. Then, we add 127 to the exponent part. We need to binarize the number to find out the second part of IEEE754 format (using *IntToBinaryString* method), and it will be a StringBuilder type variable called *exponentbit*. To transfer the numbers in the whole part to the fraction part, we use a function called substring from the *firstOneInd+1* index to the end of the *wholebit*. Thus, our mantissa is both the substring of the wholebit and the *decimalbit* which was mentioned earlier. We use a toString() function to convert the *decimalbit* data type from StringBuilder to string data type.

Finally, we created a string type output including the signbit, exponent bit, and mantissa based on the required format. We also used a substring (0,23) for mantissa because the required length of the mantissa is 23 bits. In the end, we return the final output.