Introduction:

In this practical, we had to write Java classes to implement supplied interfaces. We were also supplied with some basic J-Unit tests that we could run on our program. The aim was to implement two families of linear error correcting codes, the Hamming codes and the Reed-Muller codes. We chose here to implement them in 2 separate Java Classes with a common implementation of the interface IECC. The factory class helped us construct an object of the appropriate class when requested.

Design and implementation:

We will split this section into 2 parts, we will first discuss the Hamming code implementation then the Reed-Muller one.

Hamming code is a set of error-correction codes that can be used to detect and correct the errors that occur when the data is moved or stored from the sender to the receiver. The simple principle here is to generate Parity data bits in addition to the data bits. The number of parity bits is calculated like so: $2^r \geq m + r + 1$
where, r = parity bit, m = data bit

Thus, we choose the values of these parity Bits by determining if we are in the case of an even parity of odd parity. In the case of even parity, we count the number of 1's. If that count is odd, the parity bit value is set to 1, making the total count of occurrences of 1's an even number. In the opposite case, we are in odd parity and we set the value of the bit to 0.

First, we have encoding our data.
My encoding function is called doEncHamming() and this is its prototype :
BitSet doEncHamming(BitSet plaintext, int len);
The basis of our encoding method is that we are going to manipulate data inside an array and convert it to a BitSet at the end when returning.
Let's walk through the simple algorithm:

- *We create an array.*
- *We find the number of parity bits required knowing that :
2^(parity bits) must equal the current position ; Current position is (number of bits traversed + number of parity bits + 1). ; +1 is needed since array indices start from 0 whereas we need to start from 1.*
- *We know that the length of our array is length of received BitSet + number of parity bits we just found.*
- *We find a parity bit location and we set even parity bits at parity bit locations*

To find the parity case and set the correct value, We use the method getPar() :
int getPar(int b[], int power);
The process behind this is straight forward: We iterate through our array checking our set values, we use a conditional to check the correct values stored and calculate the parity value.

Finally, with our resulting array, we convert in to a BitSet that we then return.

Now concerning the Decoding, We use the doDecHamming() method :
BitSet doDecHamming(BitSet codetext, int len);
Keep in mind here that we are still manipulating arrays and using conversion to get to BitSets.

We find the correct bits to check for parity and we store the values of the parity checks in an array.
So we use a for loop to iterate and check the parities, the same amount of times as the number of parity bits added.
In this loop, We extract the bit from 2^(power). Using these values, we will now check if there is a single bit error and then correct it.
We use the following to find the error.
String s = Integer.toBinaryString(k);
parity[power] = (parity[power] + 1) % 2;
In the concerned loop, This allows us to generate the correct code that we can now use.
Finally, we extract the original data from the received (and corrected) code.

Now, the Reed-Muller code is an error correcting code, we will treat encoding and decoding separately for more detailed explanations
To encode a word, we use an encoding matrix. The encoding matrix is not stored and we can calculate it on the fly. The matrix is composed of r + 1 lines and r^2 columns. For each columns, its binary value is written in the the first r lines.The bits 0 of the values is at the first line. The last line is fill with 1 values.The others values are composed as below :

```
    * Example : r = 3
 *
 *  0 1 0 1 0 1 0 1
 *  0 0 1 1 0 0 1 1
 *  0 0 0 0 1 1 1 1
 *  1 1 1 1 1 1 1 1
```

We have to note here that I’m using the BigInteger class to do bit manipulation and to store bigger numbers as arrays were not working out here. We use a conversion method to convert types, using the same logic as the array usage. To encode our word, We get into a loop that iterates for the following length :
(int)Math.pow(2, r);
we call the following function on our word, BigInteger encode(BigInteger word, int r);
in each iteration, we go though simple steps :

- *Conversion of i in binary.*
- *Fill in with 0 if missing (0 at the left) using a fillZeroLeft method.*
- *Add "1" at the beginning.*
- *Multiplication of the word by the column of the matrix.*
- *Set the correct value for the bit.*
- *We return the biginteger with the correct bit sequence.*

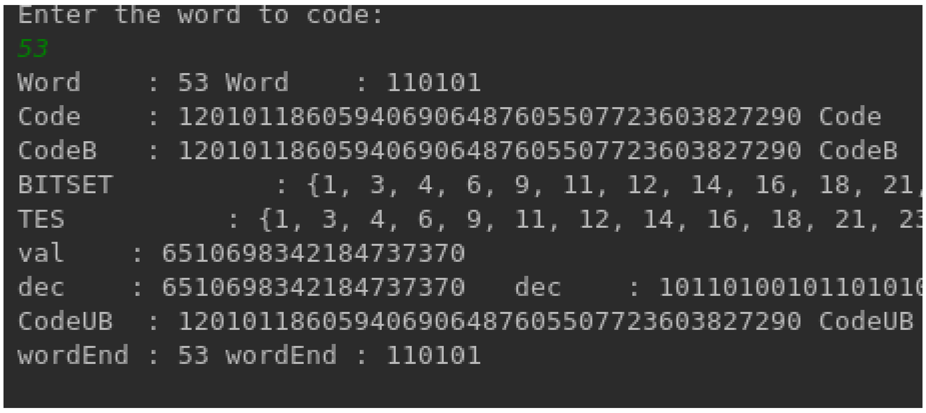
We then want to correct the error in the code word. To do so we are going to use the fourier algorithm in the BigInteger doDec(BigInteger word,int len) method.

- *We turn the word into a string and fill it with zeros.*
- *We store the word in an ArrayList with -1 if we have 1 and 1 if we have 0.*
  we do so using a for loop and the list method .add()
- *Then in another for loop, we turn I in binary, and get in function of k the nth bit of I.*
- *We test and add to store the results in an arraylist.*
- *We find the maximum in absolute value and finding the maximum in the arraylist using .max() method.*
- *We get the decoded value.*

Finally we tidy the value we just got using :
BigInteger doDone(BigInteger code, int len)
This method is straight forward and just tidies the value by inverting the bits if the first bit is equal to 1.

Testing and debugging :
For testing, I wrote my implementations in a separate Java programs and tested them for my personal input.
The Hamming code implementation is rock solid and is working well.

However, The Reed-Muller code implementation doesn’t work for specific high values. This is due to a converting function to BitSet that stops at a certain point (for no apparent reasons). I tested the logic without the conversion methods and the results are what we expect to have. The logic is good but the conversion has some issues.
Here are some screenshots showing the code working for the values that the tests don’t pass.



```
Enter the word to code:
53
Word    : 53 Word    : 110101
Code    : 120101186059406906487605507723603827290 Code
CodeB   : 120101186059406906487605507723603827290 CodeB
BITSET          : {1, 3, 4, 6, 9, 11, 12, 14, 16, 18, 21,
TES           : {1, 3, 4, 6, 9, 11, 12, 14, 16, 18, 21, 2
val    : 6510698342184737370
dec    : 6510698342184737370   dec    : 1011010010110101
CodeUB  : 120101186059406906487605507723603827290 CodeUB
wordEnd : 53 wordEnd : 110101
```

The bitset transferred here are incorrect though as we see :

BITSET : {1, 3, 4, 6, 9, 11, 12, 14, 16, 18, 21, 23, 24, 26, 29, 31, 32, 34, 37, 39, 40, 42, 45, 47, 49, 51, 52, 54, 57, 59, 60, 62}

TES : {1, 3, 4, 6, 9, 11, 12, 14, 16, 18, 21, 23, 24, 26, 29, 31, 32, 34, 37, 39, 40, 42, 45, 47, 49, 51, 52, 54, 57, 59, 60, 62}

This is wrong knowing that the length of the code is 124.

Evaluation and conclusion:

Seeking performances and clever algorithms handicapped me when It came to conversions.
However, I think that the ideas were really good. I’m not disappointed with my practical since I really tried to bring something original and performant.