Here is the Java simulation program in three files. The program uses the blocksize (variable \( N \) in file `Shannon.java`, accessible as a command line argument). The program calculates \( 2^{*N} \) as the size of the code table (variable `expN` in file `Shannon.java`). The length of each codeword in bytes is also a variable (CWS in file `Shannon.java`) accessible as a command line argument. Thus the number of bits in each codeword is \( 8*CWS \). The main data structure is the coding table: `expN` entries each of size CWS bytes. Each entry is the class `Word`, and the table itself is of class `Table`. This coding table is allocated inside `Table.java`, and each entry is allocated inside `Word.java` and filled with random bits. The simulation is repeated `simSize` many times (another command line argument inside `Shannon.java`). At each iteration, a random index in the coding table is chosen (length \( N \) bits), and the corresponding codeword (length CWS bytes) is fetched from the table. The codeword is ”perturbed” by reversing each bit with probability \( 1 - \ p = 0.25 \), where \( \ p \) is a variable inside `Shannon.java`. The table is then checked for the closest match to this new perturbed word. Here ”closest” means to check each entry to see the number of bit positions in which it differs from the perturbed word. The program focuses on the word or words in the table that differ from the perturbed word in the smallest number of bit positions. If there is more than one ”closest match”, this is regarded as an error, as is the case in which the closest match is a word different from the original unperturbed word. (In case of more than one closest match, one could choose a word at random, but this program does not do that.) The error rate is simply the percent of errors compared with all trials. The program uses a reasonably clever and efficient method for comparing codewords (as bit strings). They are compared byte-by-byte. To compare two bytes, say \( b_1 \) and \( b_1 \), in function `countDiffs` inside file `Table.java`, the function first calculates \( b = b_1 \overset{\text{ xor}}{\land} b_2 \) (the bit-wise exclusive-or). A 1 bit in \( b \) represents a difference in the two byte values, so one needs only to count the number of 1s in the byte \( b \). This is done with a table lookup in the array \( c \), declared in `Word.java`, but used in `Table.java`. The variable \( b \) ranges from -128 to 127 inclusive, so it is necessary to access \( c[b+128] \) and to create \( c \) to give the correct answers when used in this way. The array of Strings \( s \) (inside `Word.java`) gives the bit representation of each value of \( b \), but this was only used for debugging.

---

**Java class: Word**

```java
// Word.java: an array of CWS (codeword size) bytes
import java.util.Random;
public class Word {
   public static int[] c = {
      // number of 1 bits in 2s complement value (use value+128)
      // used in class Table
      1,2,2,3,2,3,4, 2,3,3,4,3,4,4,5, 2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6,
      2,3,3,4,3,4,5, 3,4,4,5,4,5,5,6, 3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7,
      2,3,3,4,3,4,5, 3,4,4,5,4,5,5,6, 3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7,
      3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7, 4,5,5,6,5,6,6,7, 5,6,6,7,6,7,7,8,
```
0, 1, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7);

```java
public byte[] w; // the only data field in this class

// Word: construct and fill bytes with random values
public Word(Random ranNumGen) {
    w = new byte[Shannon.CWS]; // allocate CWS bytes
    for (int j = 0; j < Shannon.CWS; j++)
        w[j] = (byte)(256*ranNumGen.nextDouble() - 128);
}

// Word: construct and copy input Word u into new class
public Word(Random ranNumGen, Word u) {
    w = new byte[Shannon.CWS];
    for (int j = 0; j < Shannon.CWS; j++)
        w[j] = u.w[j];
}
```

Java class: Table

```java
// Table.java: the code table for Shannon’s random code
import java.util.Random;
public class Table {
    public Word[] t; // the only data field in this class

    // Table: constructor. Allocate expN = 2**N random words
    public Table(Random ranNumGen) {
        t = new Word[Shannon.expN];
        for (int i = 0; i < Shannon.expN; i++)
            t[i] = new Word(ranNumGen);
    }

    // search: search Table t for an input word w
    public int search (Word w) {
        int comp;
        int minComp = Shannon.CWS*8 + 1;
        int minCompCount = -100000000;
        int index = -200000000;
        for (int i = 0; i < Shannon.expN; i++) {
            comp = compare(t[i], w); // count bits that differ
            if (comp == minComp) // an old minimum
                minCompCount++;
            if (comp < minComp) { // a new minimum
                index = i;
                minComp = comp;
                minCompCount = 1;
            }
        }
        if (minCompCount == 1) return index; // unique minimum
        else return -minCompCount; // several different minimums
    }
}```
// compare: return count of differences of bits of input words
private int compare(Word u, Word v) {
    int diffs = 0;
    for (int i = 0; i < Shannon.CWS; i++)
        diffs += countDiffs(u.w[i], v.w[i]);
    return diffs;
}

// countDiffs: return count of differences of bits of input bytes
private int countDiffs(byte b1, byte b2) {
    byte b = (byte)(b1^b2); // xor gives 1 where bytes differ
    return Word.c[b+128]; // table lookup gives # of 1 bits
}

// getWord: fetch a word at a given index: part of simulation
public Word getWord(int index) {
    return t[index];
}

// printTable: print the whole table, debug only
public void printTable() {
    for (int i = 0; i < Shannon.expN; i++) {
        System.out.print("Entry " + i + ": ");
        t[i].printWord();
    }
}

Java class: Shannon

// Shannon.java: a simulation of Shannon’s random coding
import java.util.Random; // use fancy rng for reproducability
public class Shannon {
    public static final double P = 0.75; // prob of no error
    public static int N; // blocksize, from command line
    public static int expN; // = 2**N, table size, calculated from N
    public static final double C = capacity(P); // channel capacity
    public static int CWS; // the codeword size, bytes, from cmd line
    private static Random ranNumGen = new Random(); // diff each time

    public static double log2(double d) { // for log2 in Java
        return Math.log(d)/Math.log(2.0);
    }

    public static double capacity(double p) { // channel capacity
        if (p == 0 || p == 1) return 1;
        return 1 + p*log2(p) + (1 - p)*log2(1 - p);
    }

    public static int randInt(int i) { // rand int, between 0 and i-1
        return (int)(ranNumGen.nextDouble()*i);
    }
}
// perturb: alter bits of input word, each time with prob 1-P
public static Word perturb(Word v) {
    Word u = new Word(ranNumGen, v);
    int[] mask = {1, 2, 4, 8, 16, 32, 64, -128};
    for (int i = 0; i < Shannon.CWS; i++)
        for (int j = 0; j < 8; j++)
            if (ranNumGen.nextDouble() > Shannon.P) {
                u.w[i] = (byte)(mask[j]^u.w[i]);
            }
    return u;
}

public static void main(String[] args) {
    int simSize = Integer.parseInt(args[0]); // # of trials
    N = Integer.parseInt(args[1]); // block size
    CWS = Integer.parseInt(args[2]); // codeword size
    expN = 1;
    for (int i = 0; i < N; i++)
        expN = expN*2; // expN = 2^**N, table size in Table.java
    System.out.println("simSize: " + simSize +
    "; Blocksize: " + Shannon.N +
    "; Codeword size (bytes): " + Shannon.CWS +
    ", expN: " + Shannon.expN);
    // count matches and two kinds of mismatches
    int numMatch = 0, numNonMatch = 0, numMultiMatch = 0;
    Table tab = new Table(ranNumGen); // the coding table
    for (int k = 0; k < simSize; k++) {
        int ind = randInt(Shannon.expN); // index of rand code word
        Word w = tab.getWord(ind); // w is the random code word
        Word u = perturb(w); // u is w with random noise added
        int ind2 = tab.search(u); // closest match, perturbed code word
        if (ind2 == ind) numMatch++;
        else if (ind2 >= 0) { // matched wrong code word, not one sent
            numNonMatch++;
        }
        else if (ind2 < 0) numMultiMatch++; // multiple matches
        if (k%500 == 499) {
            System.out.print("Error Rate: " +
            (k+1 - numMatch)/(double)(k+1));
        System.out.println("; Match: " + numMatch +
            ", Non-Match: " + numNonMatch +
            ", Multiples: " + numMultiMatch);
    }
}
    System.out.print("Error Rate: " +
    (simSize - numMatch)/(double)simSize);
    System.out.println("; Match: " + numMatch +
    ", Non-Match: " + numNonMatch +
    ", Multiples: " + numMultiMatch);