توضیحات کد

هسته اصلی پروژه از پنج کلاس اصلی تشکیل شده است. کلاس ها عبارتند از

- **HuffmanEncode** : کلاس مربوط به انکود کردن (تبدیل کاراکتر به کد باینری حاصل از درخت هافمن)
- **HuffmanDecode** : کلاس مربوط به دیکود یا تبدیل باینری به کاراکتر
- **Controller** : کلاس کنترلر مربوط به JAVAFX
- **Crypto** : کلاس مربوط به رمزگذاری به همراه کلاس اکسپشن CryptoException
- **Main** : کلاس اصلی مربوط به اجرا و تست اپلیکیشن

:HuffmanEncode.java

کتابخانه های استفاده شده در این کلاس :

```
import java.io.*;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;
```

کتابخانه io برای کار با فایل

کلاس Comparator که شیء مربوط به این کلاس هنگام ایجاد کردن یک شیء از کلاس PriorityQueue به عنوان یک تابع مقایسه کننده استفاده می شود.

دو کلاس Map و HashMap که برای نگه داری فرکانس و کد باینری هر کاراکتر استفاده می شود.

کلاس PriorityQueue که ساختمان داده از پیش تعریف شده صف اولویت را در خود داراست

HashMap :

ساختمان داده هش مپ یا هش تیبل یک ساختار برای نگه داری داده بر اساس نگاشت است که برای ذخیره داده های جفت بصورت key & value استفاده می شود.

توابعی که از این کلاس در این پروژه استفاده شده است :

- put(Key k, Value v)
  برای قرار دادن یک جفت کلید و مقدار مثل جفت کاراکتر و فرکانس آن یا جفت کاراکتر و کد هافمن آن یا بر عکس
- size()
  تعداد عناصر موجود در ساختار داده را برمی گرداند
- containsKey(Object Key)
  بررسی می کند که آیا کلید ورودی در ساختار داده وجود دارد یا خیر
- get(Key k)
  مقدار متاظر با کلید ورودی را برمی گرداند

این کلاس بصورت جنریک تعریف شده است یعنی جفتی که در این ساختمان داده ذخیره می شوند می توانند از هر نوعی باشند مثلا می توانیم داشته باشیم

```
HashMap< Character, String> hmapCode = new HashMap< Character, String>();
```

که در این قسمت یک شیء از کلاس HashMap ساخته می شود که کلید آن از نوع کاراکتر و مقدار آن از نوع رشته می باشد.

دقت کنید که کلاس های که بصورت مقابل می باشند : `ClassName< Type1, Type2, ..., TypeN>` بصورت جنریک (Generic) می باشند یعنی به جای هر کدام از type ها میتوان تایپ مورد نظر را قرار داد. به نوتیشن "<>" دقت کنید.

برای مطالعه بیشتر راجع به HashMap میتوانید به لینک مقابل مراجعه کنید : https://beginnersbook.com/2013/12/hashmap-in-java-with-example/

PriorityQueue :

صف اولویت دار بر خلاف صف معمولی که از ساختار FIFO استفاده می کند ترتیب خروج عناصرش را بر اساس اولویتی که برای آن تعیین شده است مقرر می کند. عناصر صف اولویت بر اساس یک ترتیب طبیعی یا بر اساس یک تابع مقایسه کننده دیگر که در هنگام صدا زدن constructor به آن داده میشود مرتب می شوند.

توابع پر استفاده از این کلاس:

- isEmpty()
  بررسی خالی بودن صف
- poll()
  این متد سر صف را حذف می کند و برمیگرداند یا در صورت خالی بودن صف null را برمیگرداند
- peek()
  این متد سر صف را فقط برمیگرداند ولی آنرا حذف نمی کند
- offer(E e)
  این متد عنصر ورودی را در صف insert میکند.

این کلاس همانند HashMap بصورت Generic تعریف شده است.

constructor این کلاس بصورت زیر است :

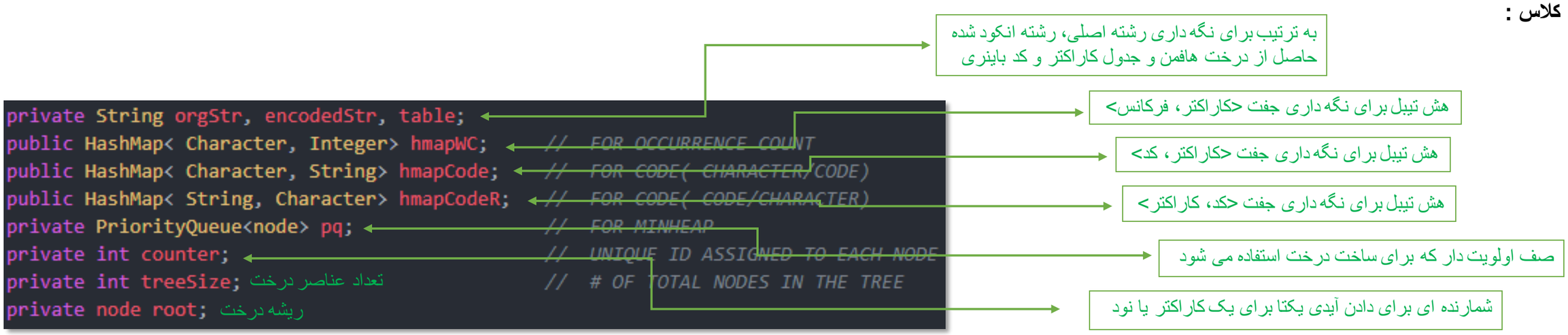
```
PriorityQueue(int initialCapacity, Comparator<E> comparator)
```

که در آن initialCapacity ظرفیت اولیه صف می باشد و comparator کلاس مقایسه کننده است در این کلاس تابعی به نام int compare باید مطابق با خواسته ما override شود

اطلاعات بیشتر در مورد صف اولویت : https://en.wikipedia.org/wiki/Priority_queue

https://www.geeksforgeeks.org/priority-queue-class-in-java-2/

متغیر های کلاس :



زیرکلاس node :

```
//  INNER CLASS
private class node {
    int uid, weight;
    char ch;
    node left, right;

    //  CONSTRUCTOR FOR CLASS NODE
    private node( Character ch, Integer weight, node left, node right) {
        uid = ++counter;
        this.ch = ch;
        this.weight = weight;
        this.left = left;
        this.right = right;

    }
}
```

حاوی متغیر های uid برای نگه داری آیدی یکتا برای هر نود. weight برای نگه داری وزن نود که در این پروژه وزن این نود فرکانس کاراکتر می باشد. ch برای نگه داری کاراکتر و نود چپ و راست برای نگه داری فرزندان چپ و راست.

در تابع کانستراکتور هم هر متغیر از یک شیء از این کلاس مقدار دهی میشود.

تابع سازنده کلاس HuffmanEncode :

```
public HuffmanEncode( String fname) {
    this.orgStr = readFile( fname);     //  STEP 0: READ INPUT FILE
    this.counter = 0;
    this.treeSize = 0;
    hmapWC = new HashMap< Character, Integer>();
    hmapCode = new HashMap< Character, String>();
    hmapCodeR = new HashMap< String, Character>();
    pq = new PriorityQueue< node>( 1, new Comparator<node>() {
        @Override
        public int compare(node n1, node n2) {
            if( n1.weight < n2.weight)
                return -1;
            else if( n1.weight > n2.weight)
                return 1;
            return 0;
        }
    });
    countWord();        //  STEP 1: COUNT FREQUENCY OF WORD
    buildTree();        //  STEP 2: BUILD HUFFMAN TREE
    buildCodeTable();   //  STEP 3: BUILD HUFFMAN CODE TABLE
    getCodeTable();     //  STEP 4: GET CODE TABLE
    encode();           //  STEP 5: GENERATE ENCODED DATA
    writeFile( fname);  //  STEP 6: WRITE THE COMPRESSED DATA INTO A FILE

}
```

ورودی تابع نام فایلی که میخواهیم انکود کنیم.

در این تابع ابتدا فایل ورودی توسط تابع readFile خوانده می شود و در رشته orgStr ریخته می شود. متغیر های counter و treeSize مقداردهی اولیه می شوند و توابع سازنده HashMap ها صدا زده می شود همچنین تابع سازنده PriorityQueue هم صدا زده می شود که میبینیم ورودی اول آن 1 است یعنی ظرفیت اولیه آن یک است و ورودی دوم آن هم یک شیء از کلاس Comparator است که هنگام ساختن آن تابع compare این کلاس را هم override کردیم به این صورت که تابعی که تعریف شده است باعث می شود که صف اولویت بر اساس نود با وزن کمتر به بیشتر مرتب شود یعنی عنصر مینیمم در سر صف قرار می گیرد.

سپس توابع دیگری به ترتیب صدا زده می شوند که جلوتر به بررسی همگی آنها می پردازم.

تابع readFile :

```java
public static String readFile( String fname) {
    StringBuilder sb = new StringBuilder();
    File filename = new File( fname);
    try {
        BufferedReader in = new BufferedReader( new FileReader( filename));
        String line = in.readLine();
        while( line != null) {
            sb.append( line + "\n");
            line = in.readLine();
        }
        in.close();
    } catch ( IOException e) {
        System.out.println( e);
    }
    return sb.toString();
}
```

نام فایل به عنوان ورودی این تابع می باشد سپس فایل توسط کلاس BufferedReader خوانده می شود که این کلاس قابلیت خواندن خط به خط فایل را با تابع readLine میدهد. در تابع readFile تا جایی که دیگر خطی از فایل نمانده باشد از آن فایل خوانده می شود و در متغیر sb که از نوع StringBuilder است ریخته می شود. برای اضافه کردن هر خطی که خوانده می شود به sb از تابع append استفاده شده است. در آخر مقدار رشته ای sb از تابع برگردانده می شود.

تابع countWord :

```java
private void countWord() {
    Character ch;
    Integer weight;
    for( int i = 0; i < orgStr.length(); ++i) {
        ch = new Character( orgStr.charAt( i));
        if( hmapWC.containsKey( ch) == false)
            weight = new Integer( 1);
        else
            weight = hmapWC.get( ch) + 1;
        hmapWC.put( ch, weight);
    }
}
```

در این تابع در هر تکرار حلقه یک کاراکتر از orgStr (که در تابع قبلی مقدار میگیرد) خوانده می شود سپس بررسی می شود که آیا این کاراکتر در هش تیبل hmapWC وجود دارد یا نه که اگر وجود نداشت آن کاراکتر با مقدار وزن (فرکانس) ۱ در آن هش تیبل قرار داده می شود و اگر هم وجود داشت به مقدار وزن آن یکی اضافه می کند.

کاراکتر موجود در اندیس i را میخواند

چک می کند آیا کاراکتر در هش مپ وجود ندارد

اگر وجود نداشت وزن را ۱ کن

اگر وجود داشت وزنش را یکی اضافه کن

جفت <کاراکتر، وزن> را در هش مپ قرار بده اگر کاراکتر از قبل در هش مپ وجود داشته باشد تنها مقدار وزن آن overwrite می شود

تابع buildTree :

```
private void buildTree() {
    buildMinHeap(); //  SET ALL LEAF NODES INTO MINHEAP
    node left, right;
    while( ! pq.isEmpty()) {
        left = pq.poll(); عنصر سر صف (نود مینیمم) برداشته می شود و در left ریخته میشود
        treeSize++;
        if( pq.peek() != null) {
            right = pq.poll();
            treeSize++;
            root = new node( '\0', left.weight + right.weight, left, right);
        } else {    //  ONLY LEFT CHILD. RIGHT = null
            root = new node( '\0', left.weight, left, null);
        }

        if( pq.peek() != null) {
            pq.offer( root);
        } else {    //  = TOP ROOT. FINISHED BUILDING THE TREE
            treeSize++;
            break;
        }
    }
}
```

تابع buildMinHeap مینیمم هیپ اولیه را می سازد و متغیر pq را مقداردهی میکند.

حلقه تا زمانی که صف اولویت خالی نباشد ادامه می یابد

بررسی می شود که آیا عنصر دیگری در صف وجود دارد اگر وجود داشت آن را بردار و در right بریز. نود جدیدی با کاراکتر نال و وزن جدید "مجموع وزن right و left" بساز و right و left را به ترتیب فرزندان راست و چپ آن قرار بده

اگر عنصر دیگری وجود نداشت نود جدیدی بساز با کاراکتر نال و وزن left و فرزند چپ آنرا left قرار بده و بجای فرزند راست null قرار بده

در این قسمت بررسی می شود که صف پس از ساختن internal node خالی نشده باشد اگر خالی نشده باشد internal node را که در قسمت های قبل ساختیم در صف insert کن در غیر اینصورت از حلقه خارج شو

تابع buildMinHeap :

در این تابع یک حلقه وجود دارد که همانند حلقه foreach بین عناصر متغیر hmapWC که شامل جفت <کاراکتر، فرکانس> بود را پیمایش می کند و به ازای هر جفت یک نود می سازد و کاراکتر و فرکانس آن کاراکتر را در نود قرار میدهد سپس آن نود را در متغیر pq (همان صف اولویت) قرار میدهد.

```
private void buildMinHeap() {
    for( Map.Entry< Character, Integer> entry : hmapWC.entrySet()) {
        Character ch = entry.getKey();
        Integer weight = entry.getValue();
        node n = new node( ch, weight, null, null);
        pq.offer( n);
    }
}
```

تابع buildCodeTable :

تابع ساخت جدول کد ها. که یک جدول هش تیبل با جفت <کاراکتر، کد> و جدول دیگر هش تیبل با جفت <کد، کاراکتر> می باشد.
در خود تابع buildCodeTable تابع دیگری صدا زده می شود که این تابع بصورت بازگشتی جدول را میسازد.

```
private void buildCodeTable() {
    String code = "";
    node n = root;
    buildCodeRecursion( n, code);   //  RECURSION
}

private void buildCodeRecursion( node n, String code) {
    if( n != null) {
        if( ! isLeaf( n)) {     //  n = INTERNAL NODE
            buildCodeRecursion( n.left, code + '0');
            buildCodeRecursion( n.right, code + '1');
        } else {    //  n = LEAF NODE
            hmapCode.put( n.ch, code);
            hmapCodeR.put( code, n.ch);
        }
    }
}
```

تابع buildCodeRecursion :

این تابع از ریشه شروع به پیمایش درخت می کند و تا جایی که به یک نود برگ نرسد ادامه میدهد در پیمایش هنگامی که به سمت فرزند چپ حرکت می کند به کد 0 و هنگامی که به سمت راست حرکت می کند به کد 1 را اضافه می کند. در آخر یعنی هنگامی که به یک نود برگ رسید در متغیر hmapCode جفت <کاراکتر، کد> را قرار میدهد و در متغیر hmapCodeR جفت <کد، کاراکتر> را قرار میدهد.

برای بررسی برگ بودن یک نود از تابع isLeaf استفاده شده است :

```
private boolean isLeaf( node n) {
    return ( n.left == null) && ( n.right == null);
}
```

در این تابع بررسی میشود که آیا نود فرزندی دارد یا نه اگر فرزندی نداشت پس این نود برگ است در غیر این صورت برگ نیست.

تابع getCodeTable :

در این تابع جدول کد-کاراکتر را بصورت یک رشته از متغیر hmapCodeR (که هش تیبل مربوط به ذخیره جفت <کد، کاراکتر> بود) استخراج می کند و آن ها را با فرمت code~char در هر خط قرار میدهد. برای ساخت این رشته از StringBuilder و تابع append استفاده شده است. در خط اول تعداد کاراکتر های یکتا و در خطوط بعدی جدول کد-کاراکتر قرار می گیرد. سپس این مقدار در متغیر table ریخته می شود.

```
private void getCodeTable() {
    StringBuilder sb = new StringBuilder();
    sb.append( hmapCodeR.size() + "\n");
    for( Map.Entry< String, Character> entry : hmapCodeR.entrySet()) {
        String code = entry.getKey();
        int ch = entry.getValue();
        sb.append(code + "~" + ch + "\n");
    }
    table = sb.toString();
}
```

تابع encode :

در تابع انکود با استفاده از جدول کاراکتر-کد که در هش مپ hmapCode قرار دارد هر کاراکتر از رشته ورودی به ترتیب خوانده می شود و کد متناظر با آن در sb قرار داده می شود و در آخر مقدار رشته sb در متغیر encodedStr قرار می گیرد. بنابراین تبدیل رشته ورودی به کد با استفاده از درخت کد هافمن انجام می شود. در مرحله بعد رشته انکود شده را در فایل میریزیم.

```
private void encode() {
    StringBuilder sb = new StringBuilder();
    Character ch;
    for( int i = 0; i < orgStr.length(); ++i) {
        ch = orgStr.charAt( i);
        sb.append( hmapCode.get( ch));
    }
    encodedStr = sb.toString();
}
```

تابع writeFile :

در این تابع مقادیر متغیر های table (که شامل جدول کد-کاراکتر بود) و encodedStr (که در مرحله قبل بدست آمد) در فایل ریخته می شود.

توجه داشته باشید که پسوند فایل های انکود شده 8z می باشد.

```
private void writeFile( String fname) {
    StringBuilder sb = new StringBuilder();
    File fileIn = new File( fname);
    String Iname = fileIn.getName();
    String Oname = fileIn.getPath().replace( "txt", "8z");
    File fileOut = new File( Oname);

    try {
        BufferedWriter bw = new BufferedWriter( new FileWriter( fileOut));
        bw.write( table + encodedStr);
        bw.close();
    } catch ( IOException e) {
        System.out.println( e);
    }
}
```

در آخر هم توابعی برای محاسبه اندازه ورودی و خروجی و نرخ فشردگی :

```
public double rate() {
    double os = orgSize();
    double es = encSize();
    return ( os / es);
}

public long orgSize() {
    return  orgStr.length() * 8;
}

public long encSize() {
    return encodedStr.length();
}
```

:HuffmanDecode.java

در این کلاس از کتابخانه های io برای کار با فایل و کتابخانه HashMap برای ساخت دوباره جدول کد-کاراکتر برای decode کردن استفاده شده است.

متغیر های این کلاس :

متغیر های encodedStr و decodedStr به ترتیب برای نگه داری رشته انکود شده و دیکود شده.

متغیر counter برای نگه داری تعداد کاراکتر های یکتا

متغیر hmapCodeR از نوع HashMap برای نگه داری جدول کد-کاراکتر

```
private String encodedStr, decodedStr;
private int counter;
public HashMap< String, Character> hmapCodeR;
```

تابع سازنده کلاس HuffmanDecode :

تابع سازنده هش تیبل متغیر hmanCodeR صدا زده می شود.

مراحل خواندن از فایل انکود شده، دیکود کردن رشته ورودی، و نوشتن داده دیکود شده در فایل تکست به ترتیب توسط توابع انجام می شود که به ترتیب آن ها را توضیح خواهم داد.

```
public HuffmanDecode( String fname) {

    hmapCodeR = new HashMap< String, Character>();

    readFile( fname);   //  STEP 0: READ CODE TABLE AND ENCODED DATA
    decode();           //  STEP 1: DECODE ENCODED DATA
    writeFile( fname);  //  STEP 2: WRITE DECODED DATA

}
```

تابع readFile :

```
public void readFile( String fname) {
    StringBuilder sb = new StringBuilder();
    File fileName = new File( fname);
    try {
        BufferedReader br = new BufferedReader( new FileReader( fileName));
        counter = Integer.parseInt( br.readLine());
        for( int i = 0; i < counter; ++i) {
            String[] s = br.readLine().split("~");
            String code = s[ 0];
            Character ch = ( char)Integer.parseInt( s[1]);
            hmapCodeR.put( code, ch);
        }
        sb.append( br.readLine());
        br.close();
    } catch (IOException e) {
        System.out.println( e);
    }

    encodedStr = sb.toString();
}
```

خط اول فایل که شامل تعداد کاراکتر های یکتا می باشد خوانده می شود و در متغیر counter ریخته می شود

این حلقه به تعداد کاراکتر های یکتا تکرار می شود و به همان اندازه خط های بعدی فایل را می خواند و هر خط را از فرمت code~char خارج می کند و code و char را از آن استخراج می کند سپس این جفت را در هش تیبل hmapCodeR قرار می دهد.

پس از اتمام کار حلقه، خط بعدی که همان خط آخر فایل است و شامل رشته انکود شده می باشد را خوانده و در متغیر sb از نوع StringBuilder قرار میدهیم

در آخر مقدار رشته موجود در sb در متغیر encodedStr ریخته می شود.

تابع decode :

در این تابع در ابتدا یک رشته خالی داریم. در حلقه رشته انکود شده ورودی بیت به بیت خوانده می شود و هر بیت آن به آن رشته اضافه می شود و سپس چک می شود که آیا رشته در جدول کد-کاراکتر (hmapCodeR) وجود دارد یا نه اگر وجود نداشت بیت بعدی خوانده می شود و به رشته اضافه می شود و سپس بررسی می شود و این روند تا زمانی ادامه می یابد که رشته در جدول پیدا شود. هنگامی که رشته را در جدول پیدا کردیم کاراکتر متناظر با رشته را از جدول استخراج کرده و در sb می نویسیم و رشته را خالی می کنیم و به این روند ادامه می دهیم تا جایی که رشته انکود شده ورودی تمام شود.

در آخر مقدار رشته sb را در متغیر decodedStr میریزیم.

```
public void decode() {
    StringBuilder sb = new StringBuilder();
    String t = "";

    for( int i = 0; i < encodedStr.length(); ++i) {
        t += encodedStr.charAt( i);
        if( hmapCodeR.containsKey( t)) {
            sb.append( hmapCodeR.get( t));
            t = "";
        }
    }
    decodedStr = sb.toString();
}
```

تابع writeFile :

در تابع writeFile هم متغیر decodeStr که شامل رشته دیکود شده است و در مرحله قبل مقدار گرفت در فایل نوشته می شود.

```
public void writeFile( String fname) {
    StringBuilder sb = new StringBuilder();
    File fileIn = new File( fname);
    String Oname = fileIn.getPath().replace( ".8ze", ".txt");

    File fileOut = new File( Oname);

    try {
        BufferedWriter bw = new BufferedWriter( new FileWriter( fileOut));
        bw.write( decodedStr);
        bw.close();
    } catch ( IOException e) {
        System.out.println( e);
    }

}
```

در رابطه با کلاس controller :

این کلاس یکی از کلاس های اصلی برای کنترل اپلیکیشن های javafx است. در این کلاس المنت هایی که در جریان کار اپلیکیشن دخیل هستند کنترل می شوند مثلا برای یک کلید در برنامه تابعی در این کلاس قرار می دهیم تا در هنگام فشردن آن کلید آن تابع اجرا شود. توضیح بیشتر راجع به این کلاس از حوصله بحث خارج است.

در رابطه با کلاس crypto :

این کلاس شامل توابعی برای رمزنگاری و رمزگشایی فایل ها می باشد در این کلاس یک تابع اصلی با نام doCrypto وجود دارد.

ورودی های این تابع به ترتیب مد (رمزنگاری یا رمز گشایی)، رشته کلید (رمز)، فایل ورودی و فایل خروجی می باشند.

در این الگوریتم بسته به مد ورودی عمل رمز نگاری/گشایی انجام می پذیرد این اعمال با توابع init و doFinal انجام می شود. الگوریتم مورد استفاده در عمل رمزنگاری AES می باشد.

در این کلاس دو تابع encrypt و decrypt وجود دارند که هر کدام متناسب با عمل مربوط به عنوانش تابع doCrypto را با مد مورد نظر صدا می زند.

```
private static void doCrypto( int cipherMode, String key, File input, File output)
    throws CryptoException {
    try {
        Key secretKey = new SecretKeySpec( key.getBytes(), ALGORITHM);
        Cipher cipher = Cipher.getInstance( TRANSFORMATION);
        cipher.init( cipherMode, secretKey);

        FileInputStream inputStream = new FileInputStream( input);
        byte[] inputBytes = new byte[ ( int) input.length()];
        inputStream.read( inputBytes);

        byte[] outputBytes = cipher.doFinal( inputBytes);

        FileOutputStream outputStream = new FileOutputStream( output);
        outputStream.write( outputBytes);

        inputStream.close();
        outputStream.close();
    } catch ( NoSuchPaddingException | NoSuchAlgorithmException
            | InvalidKeyException | BadPaddingException
            | IllegalBlockSizeException | IOException ex) {
        throw new CryptoException("Error encrypting/decrypting file", ex);
    }
}
```