```java
package com.chaoticcognitions.aenigma.models.rotors;

import android.util.Log;

/**
 * Class to handle the functionality of the Enigma machine rotors.
 * @author Dan Cassidy
 */
public class Rotor {
    public enum Direction {RIGHT_TO_LEFT, LEFT_TO_RIGHT}

    private static final char CHAR_OFFSET = 'A';
    private final int CHAR_SET_SIZE;

    // Once set, these fields will not be changed. Comprise basic rotor settings. They are stored
    // inside the Rotor class itself to avoid having an excessive amount of calls to the various
    // methods of the RotorType enum.
    private final RotorType rotorType;
    private final String wiring;
    private final String reverseWiring;
    private final String turnoverChars;
    private final boolean isSteppingRotor;
    private final boolean isMarkedWithNumbers;

    // User-changeable fields.
    private char visiblePosition = 'A'; //TODO can change name to something more representative?
    private char ringSetting = 'A';

    // Automatic field to deal with turnover and stepping.
    private boolean isAtTurnoverPosition = false;
    private boolean justStepped = false; //TODO needed?
    private boolean steppingBuffer = false; //TODO needed?

    /**
     * 1-parameter constructor.
     * @param rotorType The type of rotor to construct.
     */
    public Rotor(RotorType rotorType) {
        this.rotorType = rotorType;

        wiring = this.rotorType.wiring();
        reverseWiring = this.rotorType.reverseWiring();
        turnoverChars = this.rotorType.turnoverChars();
        isSteppingRotor = this.rotorType.isSteppingRotor();
        isMarkedWithNumbers = this.rotorType.isMarkedWithNumbers();

        CHAR_SET_SIZE = wiring.length();

        checkTurnover();
    }

    // BEGIN GETTERS AND SETTERS -->
    public RotorType getRotorType() {
        return rotorType;
    }

    public String getVisiblePosition() {
        if (isMarkedWithNumbers) {
            int visibleNumber = visiblePosition - 'A' + 1;
            return (visibleNumber < 10 ? "0" + visibleNumber : "" + visibleNumber);
        } else
            return Character.toString(visiblePosition);
    }

    public void setVisiblePosition(char visiblePosition) throws IllegalArgumentException {
        if (!isValidChar(visiblePosition))
            throw new IllegalArgumentException("Invalid position setting.");

        this.visiblePosition = visiblePosition;
        checkTurnover();
    }

    public char getRingSetting() {
        return ringSetting;
    }

    public void setRingSetting(char ringSetting) throws IllegalArgumentException {
        if (!isValidChar(ringSetting))
            throw new IllegalArgumentException("Invalid ring setting.");
        this.ringSetting = ringSetting;
    }

    public boolean justStepped() {
        return justStepped;
    }

    public boolean isAtTurnoverPosition() {
        return isAtTurnoverPosition;
    }
    // <-- END GETTERS AND SETTERS

    /**
     * Encodes a character where the input is on the main (right) side of the rotor.
     * @param inputChar The character to encode.
     * @return The encoded character.
     */
    public char encode(char inputChar, Direction direction) throws IllegalArgumentException {
        if (!isValidChar(inputChar))
            throw new IllegalArgumentException("Invalid");

        // Helper code for stepping. //TODO determine if actually needed
        if (steppingBuffer)
            steppingBuffer = false;
        else
            justStepped = false;

        // Determine the current offset.
        int offset = ringSetting - visiblePosition;
        Log.i("Rotor", "Rotor " + rotorType + " [encode(" + direction.toString().charAt(0) + ")]
        current offset: " + offset + ".");

        // Remove the offset to get the true input character.
        inputChar -= offset;
        Log.i("Rotor", "Rotor " + rotorType + " [encode(" + direction.toString().charAt(0) + ")]
        true input character (pre-normalization): " + inputChar + ".");

        // Normalize the true input character to handle any rollover. (E.g. - A character beyond 'Z'
        // will get rolled over from the end of the rotor back to the beginning.)
        inputChar = normalize(inputChar);
        Log.i("Rotor", "Rotor " + rotorType + " [encode(" + direction.toString().charAt(0) + ")]
            true input character (post-normalization): " + inputChar + ".");

        // Get the corresponding character that the true input character is wired to on the rotor.
        char outputChar;
        if (direction == Direction.RIGHT_TO_LEFT)
            outputChar = wiring.charAt(inputChar - CHAR_OFFSET);
        else
            outputChar = reverseWiring.charAt(inputChar - CHAR_OFFSET);
        Log.i("Rotor", "Rotor " + rotorType + " [encode(" + direction.toString().charAt(0) + ")]
            true output character: " + outputChar + ".");

        // Add the offset back to the character.
        outputChar += offset;
        Log.i("Rotor", "Rotor " + rotorType + " [encode(" + direction.toString().charAt(0) + ")]
            final output character (pre-normalization): " + outputChar + ".");

        // Normalize the offset output character to handle any rollover and get the final output
        // character.
        outputChar = normalize(outputChar);
        Log.i("Rotor", "Rotor " + rotorType + " [encode(" + direction.toString().charAt(0) + ")]
            final output character (post-normalization): " + outputChar + ".");

        // Return the final output character.
        return outputChar;
    }

    /**
     * Steps the rotor.
     */
    public void doStep() {
        // If this rotor does not step, just return, otherwise proceed with stepping.
        if (!isSteppingRotor)
            return;

        // TODO: Verify rotor stepping method. <-- Tentatively good.
        Log.i("Rotor", "Rotor " + rotorType + " stepping from '" + visiblePosition + "' to '" +
                (visiblePosition == 'Z' ? 'A' : (char)(visiblePosition + 1)) + "'.");

        // Step the rotor and normalize (handle rollover) if needed.
        visiblePosition = normalize(++visiblePosition);

        // Set some flags to advertise the fact this rotor just stepped. TODO needed?
        justStepped = true;
        steppingBuffer = true;

        // Check to see if this rotor is at a turnover position.
        checkTurnover();
    }

    /**
     * Determines whether the rotor is at a turnover position or not.
     */
    private void checkTurnover() {
        isAtTurnoverPosition = (turnoverChars.indexOf(visiblePosition) != -1);
    }

    /**
     * Determines whether the argument is a valid character.
     * @param charToValidate The character to validate.
     * @return boolean, representing whether the argument is a valid character (true) or not (false).
     */
    private boolean isValidChar(char charToValidate) {
        return (wiring.indexOf(charToValidate) != -1);
    }

    /**
     * Normalize the given character to within the rotor's character set. In other words, handle
     * the rollover from the end of the character set to the beginning, or from the beginning of the
     * character set to the end.
     * @param charToNormalize The character to normalize.
     * @return char, containing the normalized character.
     */
    private char normalize(char charToNormalize) {
        if (charToNormalize < CHAR_OFFSET)
            charToNormalize += CHAR_SET_SIZE;
        else if (charToNormalize >= CHAR_OFFSET + CHAR_SET_SIZE)
            charToNormalize -= CHAR_SET_SIZE;
        return charToNormalize;
    }
}

```

```java
/*-----------------------------------------------------------------------------------------------------
 * Author:      Dan Cassidy
 * Date:        2015-08-04
 * Assignment:  Project
 * Source File: RotorType.java
 * Language:    Java
 * Course:      CSCI-C 490, Android Programming, MoWe 08:00
-----------------------------------------------------------------------------------------------------*/
package com.chaoticcognitions.aenigma.models.rotors;

/**
 * Enum to store the relevant information about the different types of rotors in a single place.
 * @author Dan Cassidy
 */
public enum RotorType {
    // Enigma I - German Army and Air Force (Wehrmacht, Luftwaffe)
    // Stator
    I_ETW,
    // Rotors
    I_I, I_II, I_III, I_IV, I_V,
    // Reflectors
    I_UKW_A, I_UKW_B, I_UKW_C,

    // Norway Enigma - Enigma I used postwar by Norway
    // Stator
    N_ETW,
    // Rotors
    N_I, N_II, N_III, N_IV, N_V,
    // Reflectors
    N_UKW,

    // Enigma M3 - German Navy (Kriegsmarine)
    // Stator
    M3_ETW,
    // Rotors
    M3_I, M3_II, M3_III, M3_IV, M3_V, M3_VI, M3_VII, M3_VIII,
    // Reflectors
    M3_UKW_B, M3_UKW_C,

    // Enigma M4 - U-Boot Enigma
    // Stator
    M4_ETW,
    // Rotors
    M4_I, M4_II, M4_III, M4_IV, M4_V, M4_VI, M4_VII, M4_VIII, M4_BETA, M4_GAMMA,
    // Reflectors
    M4_UKW_B, M4_UKW_C,

    // Enigma G - Zählwerk Enigma A28 and G31
    // Stator
    G_ETW,
    // Rotors
    G_I, G_II, G_III,
    // Reflectors
    G_UKW,

    // Enigma D - Commercial Enigma A26
    // Stator
    D_ETW,
    // Rotors
    D_I, D_II, D_III,
    // Reflectors
    D_UKW,

    // Enigma K - Commercial Enigma A27
    // Stator
    K_ETW,
    // Rotors
    K_I, K_II, K_III,
    // Reflectors
    K_UKW,

    // Swiss K - Swiss Enigma K Variant (Swiss Air Force)
    // Stator
    KS_ETW,
    // Rotors
    KS_I, KS_II, KS_III,
    // Reflectors
    KS_UKW,

    // Enigma KD - Enigma K with UWK-D *** (Rewirable) *** //TODO figure out what to do with the KD enigma
    // Stator
    KD_ETW,
    // Rotors
    KD_I, KD_II, KD_III,
    // Reflectors
    KD_UKW,

    // Railway Enigma - Modified Enigma K
    // Stator
    R_ETW,
    // Rotors
    R_I, R_II, R_III,
    // Reflectors
    R_UKW,

    // Enigma T - Japanese Enigma (Tirpitz)
    // Stator
    T_ETW,
    // Rotors
    T_I, T_II, T_III, T_IV, T_V, T_VI, T_VII, T_VIII,
    // Reflectors
    T_UKW;

    /**
     * Get the wiring for the rotor based on its type. For instance, 'A' is wired to the first
     * letter of this string, 'B' is wired to the second letter, and so on.
     * @return The wiring for the rotor.
     */
    public String wiring() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
                return "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
            case G_ETW:
            case D_ETW:
            case K_ETW:
```

```
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
                return "QWERTZUIOASDFGHJKPYXCVBNML";
            case T_ETW:
                return "KZROUQHYAIGBLWVSTDXFPNMCJE";

            // Rotors
            case I_I:
            case M3_I:
            case M4_I:
                return "EKMFLGDQVZNTOWYHXUSPAIBRCJ";
            case I_II:
            case M3_II:
            case M4_II:
                return "AJDKSIRUXBLHWTMCQGZNPYFVOE";
            case I_III:
            case M3_III:
            case M4_III:
                return "BDFHJLCPRTXVZNYEIWGAKMUSQO";
            case I_IV:
            case N_IV:
            case M3_IV:
            case M4_IV:
                return "ESOVPZJAYQUIRHXLNFTGKDCMWB";
            case I_V:
            case M3_V:
            case M4_V:
                return "VZBRGITYUPSDNHLXAWMJQOFECK";
            case N_I:
                return "WTOKASUYVRBXJHQCPZEFMDINLG";
            case N_II:
                return "GJLPUBSWEMCTQVHXAOFZDRKYNI";
            case N_III:
                return "JWFMHNBPUSDYTIXVZGRQLAOEKC";
            case N_V:
                return "HEJXQOTZBVFDASCILWPGYNMURK";
            case M3_VI:
            case M4_VI:
                return "JPGVOUMFYQBENHZRDKASXLICTW";
            case M3_VII:
            case M4_VII:
                return "NZJHGRCXMYSWBOUFAIVLPEKQDT";
            case M3_VIII:
            case M4_VIII:
                return "FKQHTLXOCBJSPDZRAMEWNIUYGV";
            case M4_BETA:
                return "LEYJVCNIXWPBQMDRTAKZGFUHOS";
            case M4_GAMMA:
                return "FSOKANUERHMBTIYCWLQPZXVGJD";
            case G_I:
            case D_I:
            case K_I:
                return "LPGSZMHAEOQKVXRFYBUTNICJDW";
            case G_II:
            case D_II:
            case K_II:
                return "SLVGBTFXJQOHEWIRZYAMKPCNDU";
            case G_III:
            case D_III:
```

```
            case K_III:
                return "CJGDPSHKTURAWZXFMYNQOBVLIE";
            case KS_I:
                return "PEZUOHXSCVFMTBGLRINQJWAYDK";
            case KS_II:
                return "ZOUESYDKFWPCIQXHMVBLGNJRAT";
            case KS_III:
                return "EHRVXGAOBQUSIMZFLYNWKTPDJC";
            case KD_I:
                return "VEZIOJCXKYDUNTWAPLQGBHSFMR";
            case KD_II:
                return "HGRBSJZETDLVPMQYCXAOKINFUW";
            case KD_III:
                return "NWLHXGRBYOJSAZDVTPKFQMEUIC";
            case R_I:
                return "JGDQOXUSCAMIFRVTPNEWKBLZYH";
            case R_II:
                return "NTZPSFBOKMWRCJDIVLAEYUXHGQ";
            case R_III:
                return "JVIUBHTCDYAKEQZPOSGXNRMWFL";
            case T_I:
                return "KPTYUELOCVGRFQDANJMBSWHZXI";
            case T_II:
                return "UPHZLWEQMTDJXCAKSOIGVBYFNR";
            case T_III:
                return "QUDLYRFEKONVZAXWHMGPJBSICT";
            case T_IV:
                return "CIWTBKXNRESPFLYDAGVHQUOJZM";
            case T_V:
                return "UAXGISNJBVERDYLFZWTPCKOHMQ";
            case T_VI:
                return "XFUZGALVHCNYSEWQTDMRBKPIOJ";
            case T_VII:
                return "BJVFTXPLNAYOZIKWGDQERUCHSM";
            case T_VIII:
                return "YMTPNZHWKODAJXELUQVGCBISFR";

            // Reflectors
            case I_UKW_A:
                return "EJMZALYXVBWFCRQUONTSPIKHGD";
            case I_UKW_B:
            case M3_UKW_B:
                return "YRUHQSLDPXNGOKMIEBFZCWVJAT";
            case I_UKW_C:
            case M3_UKW_C:
                return "FVPJIAOYEDRZXWGCTKUQSBNMHL";
            case N_UKW:
                return "MOWJYPUXNDSRAIBFVLKZGQCHET";
            case M4_UKW_B:
                return "ENKQAUYWJICOPBLMDXZVFTHRGS";
            case M4_UKW_C:
                return "RDOBJNTKVEHMLFCWZAXGYIPSUQ";
            case G_UKW:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
                return "IMETCGFRAYSQBZXWLHKDVUPOJN";
            case KD_UKW:
                return "NSUOMKLIHZFGEADVXWBYCPRQTJ"; // Rewireable!
            case R_UKW:
```

```java
                return "QYHOGNECVPUZTFDJAXWMKISRBL";
            case T_UKW:
                return "GEKPBTAUMOCNILJDXZYFHWVQSR";

            default:
                return "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        }
    }

    /**
     * Get the reverse wiring for the different types of rotors.
     * @return
     */
    public String reverseWiring() {
        String wiring = this.wiring();
        char[] reverseWiring = new char[wiring.length()];

        final char CHAR_OFFSET = 'A';
        for (int index = 0; index < wiring.length(); index++)
            reverseWiring[wiring.charAt(index) - CHAR_OFFSET] = (char)(index + CHAR_OFFSET);

        return new String(reverseWiring);
    }

    /**
     * Get the turnover characters for the different types of rotors.
     */
    public String turnoverChars() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
            case G_ETW:
            case D_ETW:
            case K_ETW:
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
            case T_ETW:
                return "";

            // Rotors
            case I_I:
            case N_I:
            case M3_I:
            case M4_I:
                return "Q";
            case I_II:
            case N_II:
            case M3_II:
            case M4_II:
                return "E";
            case I_III:
            case N_III:
            case M3_III:
            case M4_III:
                return "V";
            case I_IV:
```

```
            case N_IV:
            case M3_IV:
            case M4_IV:
                return "J";
            case I_V:
            case N_V:
            case M3_V:
            case M4_V:
                return "Z";
            case M3_VI:
            case M4_VI:
            case M3_VII:
            case M4_VII:
            case M3_VIII:
            case M4_VIII:
                return "ZM";
            case M4_BETA:
            case M4_GAMMA:
                return "";
            case G_I:
                return "SUVWZABCEFGIKLOPQ";
            case G_II:
                return "STVYZACDFGHKMNQ";
            case G_III:
                return "UWXAEFHKMNR";
            case D_I:
            case K_I:
            case KS_I:
            case R_III:
                return "Y";
            case KS_II:
            case D_II:
            case K_II:
            case R_II:
                return "E";
            case D_III:
            case K_III:
            case KS_III:
            case R_I:
                return "N";
            case KD_I:
            case KD_II:
            case KD_III:
                return "SUYAEHLNQ";
            case T_I:
            case T_III:
                return "WZEKQ";
            case T_II:
            case T_IV:
                return "WZFLR";
            case T_V:
            case T_VII:
                return "YCFKR";
            case T_VI:
            case T_VIII:
                return "XEIMQ";

            // Reflectors
            case I_UKW_A:
            case I_UKW_B:
```

```
            case I_UKW_C:
            case N_UKW:
            case M3_UKW_B:
            case M3_UKW_C:
            case M4_UKW_B:
            case M4_UKW_C:
            case G_UKW:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
            case KD_UKW:
            case R_UKW:
            case T_UKW:
                return "";

            default:
                return "";
        }
    }

    /**
     * Get whether the rotor steps or not based on the type.
     */
    public boolean isSteppingRotor() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
            case G_ETW:
            case D_ETW:
            case K_ETW:
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
            case T_ETW:
                return false;

            // Rotors
            case I_I:
            case I_II:
            case I_III:
            case I_IV:
            case I_V:
            case N_I:
            case N_II:
            case N_III:
            case N_IV:
            case N_V:
            case M3_I:
            case M3_II:
            case M3_III:
            case M3_IV:
            case M3_V:
            case M3_VI:
            case M3_VII:
            case M3_VIII:
            case M4_I:
            case M4_II:
```

```
            case M4_III:
            case M4_IV:
            case M4_V:
            case M4_VI:
            case M4_VII:
            case M4_VIII:
            case G_I:
            case G_II:
            case G_III:
            case D_I:
            case D_II:
            case D_III:
            case K_I:
            case K_II:
            case K_III:
            case KS_I:
            case KS_II:
            case KS_III:
            case KD_I:
            case KD_II:
            case KD_III:
            case R_I:
            case R_II:
            case R_III:
            case T_I:
            case T_II:
            case T_III:
            case T_IV:
            case T_V:
            case T_VI:
            case T_VII:
            case T_VIII:
                return true;
            case M4_BETA:
            case M4_GAMMA:
                return false;

            // Reflectors
            case I_UKW_A:
            case I_UKW_B:
            case I_UKW_C:
            case N_UKW:
            case M3_UKW_B:
            case M3_UKW_C:
            case M4_UKW_B:
            case M4_UKW_C:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
            case KD_UKW:
            case R_UKW:
            case T_UKW:
                return false;
            case G_UKW:
                return true;

            default:
                return false;
        }
    }
```

```java

    /**
     * Get whether the rotor is marked with numbers or not based on the type.
     */
    public boolean isMarkedWithNumbers() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
            case G_ETW:
            case D_ETW:
            case K_ETW:
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
            case T_ETW:
                return false;

            // Rotors
            case I_I:
            case I_II:
            case I_III:
            case I_IV:
            case I_V:
            case N_I:
            case N_II:
            case N_III:
            case N_IV:
            case N_V:
                return true;
            case M3_I:
            case M3_II:
            case M3_III:
            case M3_IV:
            case M3_V:
            case M3_VI:
            case M3_VII:
            case M3_VIII:
            case M4_I:
            case M4_II:
            case M4_III:
            case M4_IV:
            case M4_V:
            case M4_VI:
            case M4_VII:
            case M4_VIII:
            case M4_BETA:
            case M4_GAMMA:
            case G_I:
            case G_II:
            case G_III:
            case D_I:
            case D_II:
            case D_III:
            case K_I:
            case K_II:
            case K_III:
            case KS_I:
```

```
            case KS_II:
            case KS_III:
            case KD_I:
            case KD_II:
            case KD_III:
            case R_I:
            case R_II:
            case R_III:
            case T_I:
            case T_II:
            case T_III:
            case T_IV:
            case T_V:
            case T_VI:
            case T_VII:
            case T_VIII:
                return false;

            // Reflectors
            case I_UKW_A:
            case I_UKW_B:
            case I_UKW_C:
            case N_UKW:
            case M3_UKW_B:
            case M3_UKW_C:
            case M4_UKW_B:
            case M4_UKW_C:
            case G_UKW:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
            case KD_UKW:
            case R_UKW:
            case T_UKW:
                return false;

            default:
                return false;
        }
    }

    /**
     * Get whether the rotor is a stator or not.
     */
    public boolean isStator() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
            case G_ETW:
            case D_ETW:
            case K_ETW:
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
            case T_ETW:
                return true;

            // Rotors
            case I_I:
            case I_II:
            case I_III:
            case I_IV:
            case I_V:
            case N_I:
            case N_II:
            case N_III:
            case N_IV:
            case N_V:
            case M3_I:
            case M3_II:
            case M3_III:
            case M3_IV:
            case M3_V:
            case M3_VI:
            case M3_VII:
            case M3_VIII:
            case M4_I:
            case M4_II:
            case M4_III:
            case M4_IV:
            case M4_V:
            case M4_VI:
            case M4_VII:
            case M4_VIII:
            case M4_BETA:
            case M4_GAMMA:
            case G_I:
            case G_II:
            case G_III:
            case D_I:
            case D_II:
            case D_III:
            case K_I:
            case K_II:
            case K_III:
            case KS_I:
            case KS_II:
            case KS_III:
            case KD_I:
            case KD_II:
            case KD_III:
            case R_I:
            case R_II:
            case R_III:
            case T_I:
            case T_II:
            case T_III:
            case T_IV:
            case T_V:
            case T_VI:
            case T_VII:
            case T_VIII:
                return false;

            // Reflectors
            case I_UKW_A:
            case I_UKW_B:
```

```java
            case I_UKW_C:
            case N_UKW:
            case M3_UKW_B:
            case M3_UKW_C:
            case M4_UKW_B:
            case M4_UKW_C:
            case G_UKW:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
            case KD_UKW:
            case R_UKW:
            case T_UKW:
                return false;

            default:
                return false;
        }
    }

    /**
     * Get whether the rotor is an actual rotor or not.
     */
    public boolean isRotor() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
            case G_ETW:
            case D_ETW:
            case K_ETW:
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
            case T_ETW:
                return false;

            // Rotors
            case I_I:
            case I_II:
            case I_III:
            case I_IV:
            case I_V:
            case N_I:
            case N_II:
            case N_III:
            case N_IV:
            case N_V:
            case M3_I:
            case M3_II:
            case M3_III:
            case M3_IV:
            case M3_V:
            case M3_VI:
            case M3_VII:
            case M3_VIII:
            case M4_I:
            case M4_II:
```

```
            case M4_III:
            case M4_IV:
            case M4_V:
            case M4_VI:
            case M4_VII:
            case M4_VIII:
            case M4_BETA:
            case M4_GAMMA:
            case G_I:
            case G_II:
            case G_III:
            case D_I:
            case D_II:
            case D_III:
            case K_I:
            case K_II:
            case K_III:
            case KS_I:
            case KS_II:
            case KS_III:
            case KD_I:
            case KD_II:
            case KD_III:
            case R_I:
            case R_II:
            case R_III:
            case T_I:
            case T_II:
            case T_III:
            case T_IV:
            case T_V:
            case T_VI:
            case T_VII:
            case T_VIII:
                return true;

            // Reflectors
            case I_UKW_A:
            case I_UKW_B:
            case I_UKW_C:
            case N_UKW:
            case M3_UKW_B:
            case M3_UKW_C:
            case M4_UKW_B:
            case M4_UKW_C:
            case G_UKW:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
            case KD_UKW:
            case R_UKW:
            case T_UKW:
                return false;

            default:
                return false;
        }
    }

    /**
```

```
     * Get whether the rotor is a reflector or not.
     */
    public boolean isReflector() {
        switch (this) {
            // Stators
            case I_ETW:
            case N_ETW:
            case M3_ETW:
            case M4_ETW:
            case G_ETW:
            case D_ETW:
            case K_ETW:
            case KS_ETW:
            case KD_ETW:
            case R_ETW:
            case T_ETW:
                return false;

            // Rotors
            case I_I:
            case I_II:
            case I_III:
            case I_IV:
            case I_V:
            case N_I:
            case N_II:
            case N_III:
            case N_IV:
            case N_V:
            case M3_I:
            case M3_II:
            case M3_III:
            case M3_IV:
            case M3_V:
            case M3_VI:
            case M3_VII:
            case M3_VIII:
            case M4_I:
            case M4_II:
            case M4_III:
            case M4_IV:
            case M4_V:
            case M4_VI:
            case M4_VII:
            case M4_VIII:
            case M4_BETA:
            case M4_GAMMA:
            case G_I:
            case G_II:
            case G_III:
            case D_I:
            case D_II:
            case D_III:
            case K_I:
            case K_II:
            case K_III:
            case KS_I:
            case KS_II:
            case KS_III:
            case KD_I:
```

```java
            case KD_II:
            case KD_III:
            case R_I:
            case R_II:
            case R_III:
            case T_I:
            case T_II:
            case T_III:
            case T_IV:
            case T_V:
            case T_VI:
            case T_VII:
            case T_VIII:
                return false;

            // Reflectors
            case I_UKW_A:
            case I_UKW_B:
            case I_UKW_C:
            case N_UKW:
            case M3_UKW_B:
            case M3_UKW_C:
            case M4_UKW_B:
            case M4_UKW_C:
            case G_UKW:
            case D_UKW:
            case K_UKW:
            case KS_UKW:
            case KD_UKW:
            case R_UKW:
            case T_UKW:
                return true;

            default:
                return false;
        }
    }

    /**
     * Returns the string representation of the rotor type.
     * @return The string representation of the rotor type.
     */
    @Override public String toString() {
        return super.toString().substring(super.toString().indexOf('_') + 1).replace('_', '-');
    }
}

```

```
//TODO create file comment
package com.chaoticcognitions.aenigma.models.plugboards;

import android.util.Log;

/**
 * Created by Dan on 2015-07-28.
 */
public class Plugboard {
    private static final char NULLCHAR = '\0';
    private int numPairs = 0;
    private char[] plugSettings = new char[26];

    public void addPlugSettings(char char1, char char2) {
        if (char1 < 'A' || char1 > 'Z' || char2 < 'A' || char2 > 'Z' || char1 == char2)
            throw new IllegalArgumentException("Invalid characters for plugboard pair.");
        else if (plugSettings[char1 - 'A'] != NULLCHAR || plugSettings[char2 - 'A'] != NULLCHAR)
            // At least one of the two characters are paired already.
            return;

        plugSettings[char1 - 'A'] = char2;
        plugSettings[char2 - 'A'] = char1;
        numPairs++;
    }

    //TODO create method comment
    public char encode(char inputChar) {
        Log.i("Plugboard", "Plugboard encoding '" + inputChar + "' to '" + (plugSettings[inputChar
        - 'A'] == NULLCHAR ? inputChar : plugSettings[inputChar - 'A']) + "'");
        return (plugSettings[inputChar - 'A'] == NULLCHAR ? inputChar :
                plugSettings[inputChar - 'A']);
    }
}

```

```java
//TODO create file comment
package com.chaoticcognitions.aenigma.models.machines;

import com.chaoticcognitions.aenigma.models.plugboards.Plugboard;
import com.chaoticcognitions.aenigma.models.rotors.Rotor;
import com.chaoticcognitions.aenigma.models.rotors.RotorType;

import static com.chaoticcognitions.aenigma.models.rotors.Rotor.Direction;

/**
 * TODO finish class comment
 * @author Dan Cassidy
 */
public class Machine {
    public enum RotorPosition {RIGHT, MIDDLE, LEFT, GREEK, REFLECTOR}

    //TODO comment field groupings
    private final MachineType machineType;
    private final boolean isEnigmaStepped;
    private final int numberOfRotors;
    private final boolean hasVisibleReflector;
    private final boolean hasPlugboard;

    private final RotorType[] possibleStators;
    private final RotorType[] possibleRotors;
    private final RotorType[] possibleReflectors;

    private Plugboard plugboard;

    private Rotor stator;
    private Rotor[] rotors;
    private Rotor reflector;

    //TODO create method comment
    public Machine(MachineType machineType) {
        this.machineType = machineType;

        isEnigmaStepped = this.machineType.isEnigmaStepped();
        numberOfRotors = this.machineType.numberOfRotors();
        hasVisibleReflector = this.machineType.hasVisibleReflector();
        hasPlugboard = this.machineType.hasPlugboard();
        possibleStators = this.machineType.possibleStators();
        possibleRotors = this.machineType.possibleRotors();
        possibleReflectors = this.machineType.possibleReflectors();

        plugboard = new Plugboard();

        rotors = new Rotor[numberOfRotors];
    }

    // BEGIN GETTERS AND SETTERS -->
    public MachineType getMachineType() {
        return machineType;
    }

    public void setStator(RotorType statorType) {
        if (!isValidStator(statorType))
            throw new IllegalArgumentException("Invalid stator type.");
        this.stator = new Rotor(statorType);
    }

    public void setReflector(RotorType reflectorType) {
        if (!isValidReflector(reflectorType))
            throw new IllegalArgumentException("Invalid reflector type.");
        reflector = new Rotor(reflectorType);
    }

    public void setRotor(RotorType rotorType, RotorPosition position) {
        if (!isValidRotor(rotorType) || !isValidPosition(position))
            throw new IllegalArgumentException("Invalid rotor type or position.");
        rotors[position.ordinal()] = new Rotor(rotorType);
    }

    public void setPlugboardPairs(String plugPairs) {
        for (int index = 0; index < plugPairs.length() && index + 1 < plugPairs.length(); index += 2)
            plugboard.addPlugSettings(plugPairs.charAt(index), plugPairs.charAt(index + 1));
    }

    public void setRingSetting(char ringSetting, RotorPosition position) {
        if (!isValidPosition(position))
            throw new IllegalArgumentException("Invalid position.");
        if (position == RotorPosition.REFLECTOR)
            reflector.setRingSetting(ringSetting);
        else
            rotors[position.ordinal()].setRingSetting(ringSetting);
    }

    public void setVisiblePosition(char visiblePosition, RotorPosition position) {
        if (!isValidPosition(position))
            throw new IllegalArgumentException("Invalid position.");
        if (position == RotorPosition.REFLECTOR)
            reflector.setVisiblePosition(visiblePosition);
        else
            rotors[position.ordinal()].setVisiblePosition(visiblePosition);
    }
    // <-- END GETTERS AND SETTERS

    //TODO create method comment
    public char encode(char inputChar) {
        // step
        doStep();
        // encode plugboard
        if (hasPlugboard)
            inputChar = plugboard.encode(inputChar);
        // encode stator
        inputChar = stator.encode(inputChar, Direction.RIGHT_TO_LEFT);
        // encode rotor array
        for (Rotor rotor : rotors)
            inputChar = rotor.encode(inputChar, Direction.RIGHT_TO_LEFT);
        // encode reflector
        inputChar = reflector.encode(inputChar, Direction.RIGHT_TO_LEFT);
        // encode rotor array (reverse)
        for (int index = numberOfRotors - 1; index >= 0; index--)
            inputChar = rotors[index].encode(inputChar, Direction.LEFT_TO_RIGHT);
        // encode stator (reverse)
        inputChar = stator.encode(inputChar, Direction.LEFT_TO_RIGHT);
        // encode plugboard (reverse)
        if (hasPlugboard)
            inputChar = plugboard.encode(inputChar);
        return inputChar;
    }

    //TODO create method comment
    public String encode(String inputString) {
        String toReturn = "";
        for (char inputChar : inputString.toCharArray())
            toReturn += encode(inputChar);
        return toReturn;
    }

    //TODO create method comment
    private void doStep() {
        //TODO see if this method can be optimized at all
        if (isEnigmaStepped) {
            if (rotors[RotorPosition.RIGHT.ordinal()].isAtTurnoverPosition()) {
                // normal stepping
                if (rotors[RotorPosition.MIDDLE.ordinal()].isAtTurnoverPosition()) {
                    if (rotors[RotorPosition.LEFT.ordinal()].isAtTurnoverPosition()) {
                        reflector.doStep();
                    }
                    rotors[RotorPosition.LEFT.ordinal()].doStep();
                }
                rotors[RotorPosition.MIDDLE.ordinal()].doStep();
            } else {
                // double stepping (?)
                if (rotors[RotorPosition.MIDDLE.ordinal()].isAtTurnoverPosition() &&
                        rotors[RotorPosition.MIDDLE.ordinal()].justStepped()) {
                    rotors[RotorPosition.LEFT.ordinal()].doStep();
                    rotors[RotorPosition.MIDDLE.ordinal()].doStep();
                }
            }
            rotors[RotorPosition.RIGHT.ordinal()].doStep();
        } else {
            if (rotors[RotorPosition.RIGHT.ordinal()].isAtTurnoverPosition()) {
                if (rotors[RotorPosition.MIDDLE.ordinal()].isAtTurnoverPosition()) {
                    if (rotors[RotorPosition.LEFT.ordinal()].isAtTurnoverPosition()) {
                        reflector.doStep();
                    }
                    rotors[RotorPosition.LEFT.ordinal()].doStep();
                }
                rotors[RotorPosition.MIDDLE.ordinal()].doStep();
            }
            rotors[RotorPosition.RIGHT.ordinal()].doStep();
        }
    }

    //TODO create method comment
    @Override public String toString() {
        String toReturn = "";
        for (Rotor rotor : rotors)
            toReturn = rotor.getVisiblePosition() + (toReturn.isEmpty() ? "" : " ") + toReturn;
        if (hasVisibleReflector)
            toReturn = reflector.getVisiblePosition() + " " + toReturn;
        return toReturn;
    }

    //TODO create method comment
    private boolean isValidStator(RotorType statorToValidate) {
        for (RotorType stator : possibleStators)
            if (statorToValidate == stator)
```

```
                return true;

        return false;
    }

    //TODO create method comment
    private boolean isValidRotor(RotorType rotorToValidate) {
        for (RotorType rotor : possibleRotors)
            if (rotorToValidate == rotor)
                return true;

        return false;
    }

    //TODO create method comment
    private boolean isValidReflector(RotorType reflectorToValidate) {
        for (RotorType reflector : possibleReflectors)
            if (reflectorToValidate == reflector)
                return true;

        return false;
    }

    //TODO create method comment
    private boolean isValidPosition(RotorPosition positionToValidate) {
        return !(positionToValidate == RotorPosition.GREEK && numberOfRotors != 4);
    }

    //TODO create method comment
    private boolean isReady() {
        for (Rotor rotor : rotors)
            if (rotor == null)
                return false;
        return (stator != null && reflector != null && plugboard != null);
    }
}
```

```java
/*-----------------------------------------------------------------------------------------------------
 * Author:      Dan Cassidy
 * Date:        2015-08-04
 * Assignment:  Project
 * Source File: MachineType.java
 * Language:    Java
 * Course:      CSCI-C 490, Android Programming, MoWe 08:00
-----------------------------------------------------------------------------------------------------*/
package com.chaoticcognitions.aenigma.models.machines;

import com.chaoticcognitions.aenigma.models.rotors.RotorType;

/**
 * Enum to store the relevant information about the different types of Enigma machines in a single
 * place.
 * @author Dan Cassidy
 */
public enum MachineType {
    ENIGMA_I,
    NORWAY_ENIGMA,
    ENIGMA_M3,
    ENIGMA_M4,
    ENIGMA_G,
    ENIGMA_D,
    ENIGMA_K,
    SWISS_K,
    ENIGMA_KD,
    RAILWAY_ENIGMA,
    ENIGMA_T;

    /**
     * Gets a list of possible stators for the given machine type.
     * @return The possible stators for the given machine type.
     */
    public RotorType[] possibleStators() {
        switch (this) {
            case ENIGMA_I:
                return new RotorType[]{RotorType.I_ETW};
            case NORWAY_ENIGMA:
                return new RotorType[]{RotorType.N_ETW};
            case ENIGMA_M3:
                return new RotorType[]{RotorType.M3_ETW};
            case ENIGMA_M4:
                return new RotorType[]{RotorType.M4_ETW};
            case ENIGMA_G:
                return new RotorType[]{RotorType.G_ETW};
            case ENIGMA_D:
                return new RotorType[]{RotorType.D_ETW};
            case ENIGMA_K:
                return new RotorType[]{RotorType.K_ETW};
            case SWISS_K:
                return new RotorType[]{RotorType.KS_ETW};
            case ENIGMA_KD:
                return new RotorType[]{RotorType.KD_ETW};
            case RAILWAY_ENIGMA:
                return new RotorType[]{RotorType.R_ETW};
            case ENIGMA_T:
                return new RotorType[]{RotorType.T_ETW};

            default:
                return new RotorType[]{};
        }
    }

    /**
     * Gets a list of possible rotors for the given machine type.
     * @return The possible rotors for the given machine type.
     */
    public RotorType[] possibleRotors() {
        switch (this) {
            case ENIGMA_I:
                return new RotorType[]{RotorType.I_I, RotorType.I_II, RotorType.I_III,
                RotorType.I_IV, RotorType.I_V};
            case NORWAY_ENIGMA:
                return new RotorType[]{RotorType.N_I, RotorType.N_II, RotorType.N_III,
                RotorType.N_IV, RotorType.N_V};
            case ENIGMA_M3:
                return new RotorType[]{RotorType.M3_I, RotorType.M3_II, RotorType.M3_III,
                RotorType.M3_IV, RotorType.M3_V, RotorType.M3_VI, RotorType.M3_VII,
                RotorType.M3_VIII};
            case ENIGMA_M4:
                return new RotorType[]{RotorType.M4_I, RotorType.M4_II, RotorType.M4_III,
                RotorType.M4_IV, RotorType.M4_V, RotorType.M4_VI, RotorType.M4_VII,
                RotorType.M4_VIII, RotorType.M4_BETA, RotorType.M4_GAMMA};
            case ENIGMA_G:
                return new RotorType[]{RotorType.G_I, RotorType.G_II, RotorType.G_III};
            case ENIGMA_D:
                return new RotorType[]{RotorType.D_I, RotorType.D_II, RotorType.D_III};
            case ENIGMA_K:
                return new RotorType[]{RotorType.K_I, RotorType.K_II, RotorType.K_III};
            case SWISS_K:
                return new RotorType[]{RotorType.KS_I, RotorType.KS_II, RotorType.KS_III};
            case ENIGMA_KD:
                return new RotorType[]{RotorType.KD_I, RotorType.KD_II, RotorType.KD_III};
            case RAILWAY_ENIGMA:
                return new RotorType[]{RotorType.R_I, RotorType.R_II, RotorType.R_III};
            case ENIGMA_T:
                return new RotorType[]{RotorType.T_I, RotorType.T_II, RotorType.T_III,
                RotorType.T_IV, RotorType.T_V, RotorType.T_VI, RotorType.T_VII, RotorType.T_VIII};

            default:
                return new RotorType[]{};
        }
    }

    /**
     * Gets a list of possible reflectors for the given machine type.
     * @return The possible reflectors for the given machine type.
     */
    public RotorType[] possibleReflectors(){
        switch (this) {
            case ENIGMA_I:
                return new RotorType[]{RotorType.I_UKW_A, RotorType.I_UKW_B, RotorType.I_UKW_C};
            case NORWAY_ENIGMA:
                return new RotorType[]{RotorType.N_UKW};
            case ENIGMA_M3:
                return new RotorType[]{RotorType.M3_UKW_B, RotorType.M3_UKW_C};
            case ENIGMA_M4:
                return new RotorType[]{RotorType.M4_UKW_B, RotorType.M4_UKW_C};
            case ENIGMA_G:
```

```java
                return new RotorType[]{RotorType.G_UKW};
            case ENIGMA_D:
                return new RotorType[]{RotorType.D_UKW};
            case ENIGMA_K:
                return new RotorType[]{RotorType.K_UKW};
            case SWISS_K:
                return new RotorType[]{RotorType.KS_UKW};
            case ENIGMA_KD:
                return new RotorType[]{RotorType.KD_UKW};
            case RAILWAY_ENIGMA:
                return new RotorType[]{RotorType.R_UKW};
            case ENIGMA_T:
                return new RotorType[]{RotorType.T_UKW};

            default:
                return new RotorType[]{};
        }
    }

    /**
     * Gets whether the machine is Enigma stepped or not based on the type.
     * @return The possible stators for the given machine type.
     */
    public boolean isEnigmaStepped() {
        switch (this) {
            case ENIGMA_I:
            case NORWAY_ENIGMA:
            case ENIGMA_M3:
            case ENIGMA_M4:
            case ENIGMA_D:
            case ENIGMA_K:
            case SWISS_K:
            case ENIGMA_KD:
            case RAILWAY_ENIGMA:
            case ENIGMA_T:
                return true;
            case ENIGMA_G:
                return false;

            default:
                return true;
        }
    }

    /**
     * Gets whether the machine's reflector is visible or not based on the type.
     * @return Whether the machine's reflector is visible.
     */
    public boolean hasVisibleReflector() {
        switch (this) {
            case ENIGMA_I:
            case NORWAY_ENIGMA:
            case ENIGMA_M3:
            case ENIGMA_M4:
                return false;
            case ENIGMA_G:
            case ENIGMA_D:
            case ENIGMA_K:
            case SWISS_K:
            case ENIGMA_KD:
```

```
            case RAILWAY_ENIGMA:
            case ENIGMA_T:
                return true;

            default:
                return false;
        }
    }

    //TODO create method comment
    public boolean hasPlugboard() {
        switch (this) {
            case ENIGMA_I:
            case NORWAY_ENIGMA:
            case ENIGMA_M3:
            case ENIGMA_M4:
                return true;
            case ENIGMA_G:
            case ENIGMA_D:
            case ENIGMA_K:
            case SWISS_K:
            case ENIGMA_KD:
            case RAILWAY_ENIGMA:
            case ENIGMA_T:
                return false;

            default:
                return false;
        }
    }

    /**
     * Gets the number of rotors a machine has based on its type. Note that this is only describing
     * the number of actual rotors and does not include the stator or reflector.
     * @return The number of rotors the machine has.
     */
    public int numberOfRotors() {
        switch (this) {
            case ENIGMA_I:
            case NORWAY_ENIGMA:
            case ENIGMA_M3:
            case ENIGMA_G:
            case ENIGMA_D:
            case ENIGMA_K:
            case SWISS_K:
            case ENIGMA_KD:
            case RAILWAY_ENIGMA:
            case ENIGMA_T:
                return 3;
            case ENIGMA_M4:
                return 4;

            default:
                return 3;
        }
    }

    /**
     * Returns the string representation of the machine type.
     * @return The string representation of the machine type.
```

```
         */
        @Override public String toString() {
            switch (this) {
                case ENIGMA_I:
                    return "Enigma I";
                case NORWAY_ENIGMA:
                    return "Norway Enigma";
                case ENIGMA_M3:
                    return "Enigma M3";
                case ENIGMA_M4:
                    return "Enigma M4";
                case ENIGMA_G:
                    return "Enigma G";
                case ENIGMA_D:
                    return "Enigma D";
                case ENIGMA_K:
                    return "Enigma K";
                case SWISS_K:
                    return "Swiss-K";
                case ENIGMA_KD:
                    return "Enigma KD";
                case RAILWAY_ENIGMA:
                    return "Railway Enigma";
                case ENIGMA_T:
                    return "Enigma T";

                default:
                    return "Unknown";
            }
        }
    }

```