```
package enigma;

import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

import java.util.ArrayList;
import java.util.NoSuchElementException;
import java.util.Objects;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import static enigma.EnigmaException.*;
/**
 * (Full repo unavailable for UC Berkeley privacy concerns)
 * This is the main class of Enigma, a virtualized replica of the Enigma Machine that helped shorten World War II by 5 years.
 * Contains all rotor operations.
 * @author Ansh Vashisth
 */
final class Main {

    /** Process a sequence of encryptions and decryptions, as
     *  specified by ARGS, where 1 <= ARGS.length <= 3.
     *  ARGS[0] is the name of a configuration file.
     *  ARGS[1] is optional; when present, it names an input file
     *  containing messages.  Otherwise, input comes from the standard
     *  input.  ARGS[2] is optional; when present, it names an output
     *  file for processed messages.  Otherwise, output goes to the
     *  standard output. Exits normally if there are no errors in the input;
     *  otherwise with code 1. */
    public static void main(String... args) {
        try {
            new Main(args).process();
            return;
        } catch (EnigmaException excp) {
            System.err.printf("Error: %s%n", excp.getMessage());
        }
        System.exit(1);
    }

    /** Check ARGS and open the necessary files (see comment on main). */
    Main(String[] args) {
```

```
    if (args.length < 1 || args.length > 3) {
        throw error("Only 1, 2, or 3 command-line arguments allowed");
    }

    _config = getInput(args[0]);
    _configTwo = getInput(args[0]);

    if (args.length > 1) {
        _input = getInput(args[1]);
    } else {
        _input = new Scanner(System.in);
    }

    if (args.length > 2) {
        _output = getOutput(args[2]);
    } else {
        _output = System.out;
    }
}

/** Return a Scanner reading from the file named NAME. */
private Scanner getInput(String name) {
    try {
        return new Scanner(new File(name));
    } catch (IOException excp) {
        throw error("could not open %s", name);
    }
}

/** Return a PrintStream writing to the file named NAME. */
private PrintStream getOutput(String name) {
    try {
        return new PrintStream(new File(name));
    } catch (IOException excp) {
        throw error("could not open %s", name);
    }
}

/** Configure an Enigma machine from the contents of configuration
 *  file _config and apply it to the messages in _input, sending the
 *  results to _output. */
private void process() {
    String fullRegex = "[^\\s]{1,}";
    Pattern pattern = Pattern.compile(fullRegex);
```

```
Scanner firstLine = new Scanner(_input.nextLine());
if (!configMade) {
    if (firstLine.hasNext(pattern)) {
        if (!(Objects.equals(firstLine.next(pattern), "*"))) {
            throw error("No asterisk given at beginning of input");
        }
    } else {
        throw error("No asterisk provided.");
    }
    run = readConfig();
    configMade = true;
}
run.clearRotors();
String[] rotorsUsed = new String[run.numRotors()];
for (int i = 0; i < run.numRotors(); i++) {
    if (firstLine.hasNext(pattern)) {
        String temp = firstLine.next(pattern);
        boolean contains = false;
        for (int j = 0; j < run.getAvail().size(); j++) {
            if (Objects.equals(temp, run.getAvail().get(j).name())) {
                for (int k = 0; k < run.getUse().size(); k++) {
                    if (!Objects.equals(temp,
                            run.getUse().get(k).name())) {
                        continue;
                    }
                    throw error("Duplicate rotors not allowed.");
                }
                rotorsUsed[i] = temp;
                contains = true;
                break;
            }
        }
    } else {
        throw error("Invalid rotor input.");
    }
}
run.insertRotors(rotorsUsed);
if (firstLine.hasNext(pattern)) {
    setUp(run, firstLine.next(pattern));
}
StringBuilder plugBuild = new StringBuilder();
while (firstLine.hasNext(pattern)) {
    plugBuild.append(firstLine.next(pattern));
}
```

```
    run.setPlugboard(new Permutation(plugBuild.toString(),
            run.getAlphabet()));
    StringBuilder finBuild = new StringBuilder();
    while (_input.hasNextLine()) {
        finBuild = new StringBuilder();
        String thisLine = _input.nextLine();
        Scanner thisLinesc = new Scanner(thisLine);
        int thisLineSize = 0;
        while (thisLinesc.hasNext(pattern)) {
            String temp = thisLinesc.next(pattern);
            thisLineSize += temp.length();
            if (Objects.equals(temp, "*")) {
                _output.print(printMessageLine(finBuild.toString(),
                        thisLineSize));
                process();
                return;
            } else {
                String printOut = run.convert(temp);
                finBuild.append(printOut);
            }
        }
        _output.println(printMessageLine(finBuild.toString(),
                thisLineSize));
    }
}

/** Return an Enigma machine configured from the contents of configuration
 *  file _config. */
private Machine readConfig() {
    try {
        String rotorNameRegex = " *[^\\s]{1,} *";
        String rotorTypeAndNotch = " [RNM][.*] *";
        String perm = "\\([^ ]{1,}\\) ";
        Pattern pattern1 = Pattern.compile(rotorNameRegex);
        Pattern pattern3 = Pattern.compile(perm);
        int rotorSlots, numPawls;
        ArrayList<Rotor> allRotors = new ArrayList<Rotor>();
        String alphabetLine = "";
        if (_config.hasNext(pattern1)) {
            alphabetLine = _config.next(pattern1);
        } else {
            throw error("No alphabet given.");
        }
        if (_config.hasNext(pattern1)) {
```

```
    rotorSlots = Integer.parseInt(_config.next(pattern1));
} else {
    throw error("No number of rotor slots given.");
}
if (_config.hasNext(pattern1)) {
    numPawls = Integer.parseInt(_config.next(pattern1));
} else {
    throw error("No number of pawls given.");
}
while (_config.hasNext(pattern1)) {
    String rotorName = "";
    char rotorType = 0;
    String notches = "";
    String permute = "";
    if (_config.hasNext()) {
        rotorName = _config.next(pattern1);
    } else {
        throw error("No rotor name given.");
    }
    if (_config.hasNext(pattern1)) {
        String twoThings = _config.next(pattern1);
        rotorType = twoThings.charAt(0);
        notches = twoThings.substring(1, twoThings.length());
    } else {
        throw error("No rotor type/notches given.");
    }
    int sizeHold = 0;
    StringBuilder strb = new StringBuilder(permute);
    while ((_config.hasNext(pattern1))
            & (sizeHold < alphabetLine.length())) {
        String temp = _config.next(pattern1);
        strb.append(temp);
        sizeHold += (temp.length() - 2);
    }
    if (rotorType == 'R') {
        allRotors.add(new Reflector(rotorName,
                new Permutation(strb.toString(),
                        new Alphabet(alphabetLine))));
    } else if (rotorType == 'N') {
        allRotors.add(new Rotor(rotorName,
                new Permutation(strb.toString(),
                        new Alphabet(alphabetLine))));
    } else if (rotorType == 'M') {
        allRotors.add(new MovingRotor(rotorName,
```

```
                    new Permutation(strb.toString(),
                            new Alphabet(alphabetLine)),
                    notches));
            }
        }
        return new Machine(new Alphabet(alphabetLine),
                rotorSlots, numPawls, allRotors);
    } catch (NoSuchElementException excp) {
        throw error("configuration file truncated");
    }
}

/** Return a rotor, reading its description from _config. */
private Rotor readRotor() {
    try {
        return null;
    } catch (NoSuchElementException excp) {
        throw error("bad rotor description");
    }
}

/** Set M according to the specification given on SETTINGS,
 *  which must have the format specified in the assignment. */
private void setUp(Machine M, String settings) {
    int j = 0;
    for (int i = settings.length() - 1; i >= 0; i--) {
        M.getUse().get(i + 1).set(settings.charAt(i));
        j++;
    }
}

/** Print MSG in groups of five (except that the last group may
 *  have fewer letters).
 * @param len Length of the string passed in.
 * @return Outputs the final strings.
 **/
private String printMessageLine(String msg, int len) {
    String fiveChars = "([^\\s]{0,5})";
    Pattern pattern = Pattern.compile(fiveChars);
    Scanner sc = new Scanner(msg);
    Matcher matcher = pattern.matcher(msg);

    String fin = "";
    int track = 0;
```

```
        StringBuilder finOut = new StringBuilder(fin);
        if (len <= 2) {
            if (matcher.find()) {
                return finOut.append(matcher.group()).toString();
            }
        }
        while (matcher.find() & (len - 10 > 0)) {
            track += 5;
            String t = matcher.group();
            finOut.append(matcher.group() + " ");
        }
        if (matcher.find()) {
            finOut.append(matcher.group());
        }
        return finOut.toString();
    }

    /** Alphabet used in this machine. */
    private Alphabet _alphabet;

    /** Source of input messages. */
    private Scanner _input;

    /** Source of machine configuration. */
    private Scanner _config;

    /** Backup config. **/
    private Scanner _configTwo;

    /** File for encoded/decoded messages. */
    private PrintStream _output;

    /** Machine given by _config. */
    private Machine run;

    /** Status of machine. */
    private boolean configMade = false;
}
```