```
/*
 * Copyright 2011 David Simmons
 * http://cafbit.com/entry/implementing_des
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import java.util.Arrays;

/**
 * Super-slow DES implementation for the overly patient.
 *
 * The following resources proved valuable in developing and testing
 * this code:
 *
 * "Data Encryption Standard" from Wikipedia, the free encyclopedia
 * http://en.wikipedia.org/wiki/Data_Encryption_Standard
 *
 * "The DES Algorithm Illustrated" by J. Orlin Grabbe
 * http://orlingrabbe.com/des.htm
 *
 * "DES Calculator" by Lawrie Brown
 * http://www.unsw.adfa.edu.au/~lpb/src/DEScalc/DEScalc.html
 *
 * April 6, 2011
 *
 * @author David Simmons - http://cafbit.com/
 *
 */
public class DES {

    ////////////////////////////////////////////////////////////////
////
    //
    // Various static data tables used by the DES algorithm.
    //
    // Many of these tables are based on bit permutations, where the
    // index of the array corresponds to the output bit, and the value
    // indicates which bit of the input should be used.
```

```
    //
    // The bit position values, provided by Wikipedia, start counting
    // with the left-most bit as "1".
    //
    ////////////////////////////////////////////////////////////////
////

    // High-level permutations

    /**
     * Input Permutation.  The message block is permuted by this
     * permutation at the beginning of the algorithm.
     */
    private static final byte[] IP = {
        58, 50, 42, 34, 26, 18, 10, 2,
        60, 52, 44, 36, 28, 20, 12, 4,
        62, 54, 46, 38, 30, 22, 14, 6,
        64, 56, 48, 40, 32, 24, 16, 8,
        57, 49, 41, 33, 25, 17, 9,  1,
        59, 51, 43, 35, 27, 19, 11, 3,
        61, 53, 45, 37, 29, 21, 13, 5,
        63, 55, 47, 39, 31, 23, 15, 7
    };

    /**
     * Final Permutation.  The final result is permuted by this
     * permutation to generate the final ciphertext block.
     */
    private static final byte[] FP = {
        40, 8, 48, 16, 56, 24, 64, 32,
        39, 7, 47, 15, 55, 23, 63, 31,
        38, 6, 46, 14, 54, 22, 62, 30,
        37, 5, 45, 13, 53, 21, 61, 29,
        36, 4, 44, 12, 52, 20, 60, 28,
        35, 3, 43, 11, 51, 19, 59, 27,
        34, 2, 42, 10, 50, 18, 58, 26,
        33, 1, 41, 9, 49, 17, 57, 25
    };

    // Permutations relating to the Feistel function.

    /**
     * Expansion Permutation.  The Feistel function begins by applying
     * this permutation to its 32-bit input half-block to create an
     * "expanded" 48-bit value.
     */
    private static final byte[] E = {
        32, 1,  2,  3,  4,  5,
        4,  5,  6,  7,  8,  9,
        8,  9,  10, 11, 12, 13,
```

```
        12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21,
        20, 21, 22, 23, 24, 25,
        24, 25, 26, 27, 28, 29,
        28, 29, 30, 31, 32, 1
    };

    /**
     * Substitution Boxes.  A crucial step in the Feistel function is
     * to perform bit substitutions according to this table.  A 48-bit
     * value is split into 6-bit sections, and each section is
permuted
     * into a different 6-bit value according to these eight tables.
     * (One table for each section.)
     *
     *  According to Wikipedia:
     *  "The S-boxes provide the core of the security of DES - without
     *  them, the cipher would be linear, and trivially breakable."
     */
    private static final byte[][] S = { {
        14, 4,  13, 1,  2,  15, 11, 8,  3,  10, 6,  12, 5,  9,  0,  7,
        0,  15, 7,  4,  14, 2,  13, 1,  10, 6,  12, 11, 9,  5,  3,  8,
        4,  1,  14, 8,  13, 6,  2,  11, 15, 12, 9,  7,  3,  10, 5,  0,
        15, 12, 8,  2,  4,  9,  1,  7,  5,  11, 3,  14, 10, 0,  6,  13
    }, {
        15, 1,  8,  14, 6,  11, 3,  4,  9,  7,  2,  13, 12, 0,  5,
10,
        3,  13, 4,  7,  15, 2,  8,  14, 12, 0,  1,  10, 6,  9,  11, 5,
        0,  14, 7,  11, 10, 4,  13, 1,  5,  8,  12, 6,  9,  3,  2,
15,
        13, 8,  10, 1,  3,  15, 4,  2,  11, 6,  7,  12, 0,  5,  14, 9
    }, {
        10, 0,  9,  14, 6,  3,  15, 5,  1,  13, 12, 7,  11, 4,  2,  8,
        13, 7,  0,  9,  3,  4,  6,  10, 2,  8,  5,  14, 12, 11, 15, 1,
        13, 6,  4,  9,  8,  15, 3,  0,  11, 1,  2,  12, 5,  10, 14, 7,
        1,  10, 13, 0,  6,  9,  8,  7,  4,  15, 14, 3,  11, 5,  2,  12
    }, {
        7,  13, 14, 3,  0,  6,  9,  10, 1,  2,  8,  5,  11, 12, 4,
15,
        13, 8,  11, 5,  6,  15, 0,  3,  4,  7,  2,  12, 1,  10, 14, 9,
        10, 6,  9,  0,  12, 11, 7,  13, 15, 1,  3,  14, 5,  2,  8,  4,
        3,  15, 0,  6,  10, 1,  13, 8,  9,  4,  5,  11, 12, 7,  2,  14
    }, {
        2,  12, 4,  1,  7,  10, 11, 6,  8,  5,  3,  15, 13, 0,  14, 9,
        14, 11, 2,  12, 4,  7,  13, 1,  5,  0,  15, 10, 3,  9,  8,  6,
        4,  2,  1,  11, 10, 13, 7,  8,  15, 9,  12, 5,  6,  3,  0,
14,
        11, 8,  12, 7,  1,  14, 2,  13, 6,  15, 0,  9,  10, 4,  5,  3
    }, {
        12, 1,  10, 15, 9,  2,  6,  8,  0,  13, 3,  4,  14, 7,  5,
```

```
11,
        10, 15, 4,  2,  7,  12, 9,  5,  6,  1,  13, 14, 0,  11, 3,  8,
        9,  14, 15, 5,  2,  8,  12, 3,  7,  0,  4,  10, 1,  13, 11, 6,
        4,  3,  2,  12, 9,  5,  15, 10, 11, 14, 1,  7,  6,  0,  8,  13
    }, {
        4,  11, 2,  14, 15, 0,  8,  13, 3,  12, 9,  7,  5,  10, 6,  1,
        13, 0,  11, 7,  4,  9,  1,  10, 14, 3,  5,  12, 2,  15, 8,  6,
        1,  4,  11, 13, 12, 3,  7,  14, 10, 15, 6,  8,  0,  5,  9,  2,
        6,  11, 13, 8,  1,  4,  10, 7,  9,  5,  0,  15, 14, 2,  3,  12
    }, {
        13, 2,  8,  4,  6,  15, 11, 1,  10, 9,  3,  14, 5,  0,  12, 7,
        1,  15, 13, 8,  10, 3,  7,  4,  12, 5,  6,  11, 0,  14, 9,  2,
        7,  11, 4,  1,  9,  12, 14, 2,  0,  6,  10, 13, 15, 3,  5,  8,
        2,  1,  14, 7,  4,  10, 8,  13, 15, 12, 9,  0,  3,  5,  6,  11
    } };

    /**
     * "P" Permutation.  The Feistel function concludes by applying
this
     * 32-bit permutation to the result of the S-box substitution, in
     * order to spread the output bits across 6 different S-boxes in
     * the next round.
     */
    private static final byte[] P = {
        16, 7,  20, 21,
        29, 12, 28, 17,
        1,  15, 23, 26,
        5,  18, 31, 10,
        2,  8,  24, 14,
        32, 27, 3,  9,
        19, 13, 30, 6,
        22, 11, 4,  25
    };

    // Permutations relating to subkey generation

    /**
     * PC1 Permutation.  The supplied 64-bit key is permuted according
     * to this table into a 56-bit key.  (This is why DES is only a
     * 56-bit algorithm, even though you provide 64 bits of key
     * material.)
     */
    private static final byte[] PC1 = {
        57, 49, 41, 33, 25, 17, 9,
        1,  58, 50, 42, 34, 26, 18,
        10, 2,  59, 51, 43, 35, 27,
        19, 11, 3,  60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7,  62, 54, 46, 38, 30, 22,
        14, 6,  61, 53, 45, 37, 29,
```

```
        21, 13, 5,  28, 20, 12, 4
    };

    /**
     * PC2 Permutation.  The subkey generation process applies this
     * permutation to transform its running 56-bit keystuff value into
     * the final set of 16 48-bit subkeys.
     */
    private static final byte[] PC2 = {
        14, 17, 11, 24, 1,  5,
        3,  28, 15, 6,  21, 10,
        23, 19, 12, 4,  26, 8,
        16, 7,  27, 20, 13, 2,
        41, 52, 31, 37, 47, 55,
        30, 40, 51, 45, 33, 48,
        44, 49, 39, 56, 34, 53,
        46, 42, 50, 36, 29, 32
    };

    /**
     * Subkey Rotations.  Part of the subkey generation process
     * involves rotating certain bit-sections of the keystuff by
either
     * one or two bits to the left.  This table specifies how many
bits
     * to rotate left for each of the 16 steps.
     */
    private static final byte[] rotations = {
        1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
    };

    ////////////////////////////////////////////////////////////////
////
    //
    // Numerical utility methods
    //
    ////////////////////////////////////////////////////////////////
////

    // convenience methods for performing the basic permutations.

    private static long IP(long src)  { return permute(IP, 64,
src);                } // 64-bit output
    private static long FP(long src)  { return permute(FP, 64,
src);                } // 64-bit output
    private static long E(int src)    { return permute(E, 32,
src&0xFFFFFFFFL);      } // 48-bit output
    private static int  P(int src)    { return (int)permute(P, 32,
src&0xFFFFFFFFL); } // 32-bit output
    private static long PC1(long src) { return permute(PC1, 64,
```

```
src);                } // 56-bit output
    private static long PC2(long src) { return permute(PC2, 56,
src);                } // 48-bit output

    /**
     * Permute an input value "src" of srcWidth bits according to the
     * supplied permutation table.  (Note that our permutation tables,
     * supplied by Wikipedia, start counting with the left-most bit as
     * "1".)
     */
    private static long permute(byte[] table, int srcWidth, long src)
{
        long dst = 0;
        for (int i=0; i<table.length; i++) {
            int srcPos = srcWidth - table[i];
            dst = (dst<<1) | (src>>srcPos & 0x01);
        }
        return dst;
    }

    /**
     * Permute the supplied 6-bit value based on the S-Box at the
     * specified box number.  (Box numbers start at 1, to be
consistent
     * with the literature.)
     */
    private static byte S(int boxNumber, byte src) {
        // The first and last bits determine which 16-value row to
        // reference, so we transform the 6-bit input into an
        // absolute index based on the following bit shuffle:
        // abcdef => afbcde
        src = (byte) (src&0x20 | ((src&0x01)<<4) | ((src&0x1E)>>1));
        return S[boxNumber-1][src];
    }

    /**
     * Utility method to convert 8 bytes (starting at the specified
     * offset to the supplied byte array) into a single 64-bit long
     * value.  If the supplied byte array does not contain 8 elements
     * starting at offset, the missing bytes are regarded as zero
     * padding.
     */
    private static long getLongFromBytes(byte[] ba, int offset) {
        long l = 0;
        for (int i=0; i<8; i++) {
            byte value;
            if ((offset+i) < ba.length) {
                value = ba[offset+i];
            } else {
                value = 0;
```

```
            }
            l = l<<8 | (value & 0xFFL);
        }
        return l;
    }

    /**
     * Utility method to convert a 64-bit long value into eight bytes,
     * which are written into the supplied byte array at the specified
     * offset.  If the destination byte array does not have eight
bytes
     * starting at offset, the remaining bytes are silently discarded.
     */
    private static void getBytesFromLong(byte[] ba, int offset, long
l) {
        for (int i=7; i>=0; i--) {
            if ((offset+i) < ba.length) {
                ba[offset+i] = (byte) (l & 0xFF);
                l = l >> 8;
            } else {
                break;
            }
        }
    }

    ////////////////////////////////////////////////////////////////
////
    //
    // Primary DES algorithm methods
    //
    ////////////////////////////////////////////////////////////////
////

    /**
     * The Feistel function is the heart of DES.
     */
    private static int feistel(int r, /* 48 bits */ long subkey) {
        // 1. expansion
        long e = E(r);
        // 2. key mixing
        long x = e ^ subkey;
        // 3. substitution
        int dst = 0;
        for (int i=0; i<8; i++) {
            dst>>>=4;
            int s = S(8-i, (byte)(x&0x3F));
            dst |= s << 28;
            x>>=6;
        }
        // 4. permutation
```

```
    return P(dst);
}

/**
 * Generate 16 48-bit subkeys based on the provided 64-bit key
 * value.
 */
private static long[] createSubkeys(/* 64 bits */ long key) {
    long subkeys[] = new long[16];

    // perform the PC1 permutation
    key = PC1(key);

    // split into 28-bit left and right (c and d) pairs.
    int c = (int) (key>>28);
    int d = (int) (key&0x0FFFFFFF);

    // for each of the 16 needed subkeys, perform a bit
    // rotation on each 28-bit keystuff half, then join
    // the halves together and permute to generate the
    // subkey.
    for (int i=0; i<16; i++) {
        // rotate the 28-bit values
        if (rotations[i] == 1) {
            // rotate by 1 bit
            c = ((c<<1) & 0x0FFFFFFF) | (c>>27);
            d = ((d<<1) & 0x0FFFFFFF) | (d>>27);
        } else {
            // rotate by 2 bits
            c = ((c<<2) & 0x0FFFFFFF) | (c>>26);
            d = ((d<<2) & 0x0FFFFFFF) | (d>>26);
        }

        // join the two keystuff halves together.
        long cd = (c&0xFFFFFFFFL)<<28 | (d&0xFFFFFFFFL);

        // perform the PC2 permutation
        subkeys[i] = PC2(cd);
    }

    return subkeys; /* 48-bit values */
}

/**
 * Encrypt a 64-bit block of plaintext message into a 64-bit
 * ciphertext.
 */
public static long encryptBlock(long m, /* 64 bits */ long key) {
    // generate the 16 subkeys
    long subkeys[] = createSubkeys(key);
```

```
        // perform the initial permutation
        long ip = IP(m);

        // split the 32-bit value into 16-bit left and right halves.
        int l = (int) (ip>>32);
        int r = (int) (ip&0xFFFFFFFFL);

        // perform 16 rounds
        for (int i=0; i<16; i++) {
            int previous_l = l;
            // the right half becomes the new left half.
            l = r;
            // the Feistel function is applied to the old left half
            // and the resulting value is stored in the right half.
            r = previous_l ^ feistel(r, subkeys[i]);
        }

        // reverse the two 32-bit segments (left to right; right to
left)
        long rl = (r&0xFFFFFFFFL)<<32 | (l&0xFFFFFFFFL);

        // apply the final permutation
        long fp = FP(rl);

        // return the ciphertext
        return fp;
    }

    /**
     * Wrapper around encryptBlock() that allows arguments to be byte
     * arrays instead of longs.
     */
    public static void encryptBlock(
        byte[] message,
        int messageOffset,
        byte[] ciphertext,
        int ciphertextOffset,
        byte[] key
    ) {
        long m = getLongFromBytes(message, messageOffset);
        long k = getLongFromBytes(key, 0);
        long c = encryptBlock(m, k);
        getBytesFromLong(ciphertext, ciphertextOffset, c);
    }

    ////////////////////////////////////////////////////////////////
////
    //
    // High-level interface to the DES algorithm
```

```
    //
    ////////////////////////////////////////////////////////////////
////

    /**
     * Encrypt the supplied message with the provided key, and return
     * the ciphertext.  If the message is not a multiple of 64 bits
     * (8 bytes), then it is padded with zeros.
     *
     * This method uses the Electronic Code Book (ECB) mode of
     * operation -- each 64-bit block is encrypted individually with
     * the same key.
     */
    public static byte[] encrypt(byte[] message, byte[] key) {
        byte[] ciphertext = new byte[message.length];

        // encrypt each 8-byte (64-bit) block of the message.
        for (int i=0; i<message.length; i+=8) {
            encryptBlock(message, i, ciphertext, i, key);
        }

        return ciphertext;
    }

    /**
     * Encrypt the supplied message with the provided key, and return
     * the ciphertext.  If the message is not a multiple of 64 bits
     * (8 bytes), then it is padded with zeros.
     *
     * This method uses the Electronic Code Book (ECB) mode of
     * operation -- each 64-bit block is encrypted individually with
     * the same key.
     *
     * The provided password is converted into a key with the bits
     * of each byte reversed, to generate a stronger key.
     * See passwordToKey() for more details.
     */
    public static byte[] encrypt(byte[] challenge, String password) {
        return encrypt(challenge, passwordToKey(password));
    }

    /**
     * Convert a password string into a byte array, reversing the bits
     * of each byte to place more useful key bits into non-discarded
     * bit-positions of the 64-bit DES key input.  The ever-popular
     * 7-bit ASCII characters have useful information in the least
     * significant bit which is discarded by DES, and always have zero
     * in the most significant bit, so reversing the bit order of the
     * password bytes results in a stronger key.
     *
```

```
     * This is consistent with the "VNC Authentication" scheme used in
     * the RFB protocol:
     *
     * "The RFB specification says that VNC authentication is done by
     * receiving a 16 byte challenge, encrypting it with DES using the
     * user specified password, and sending back the resulting 16
bytes.
     * The actual software encrypts the challenge with all the bit
fields
     * in each byte of the password mirrored."
     *     - http://www.vidarholen.net/contents/junk/vnc.html
     */
    private static byte[] passwordToKey(String password) {
        byte[] pwbytes = password.getBytes();
        byte[] key = new byte[8];
        for (int i=0; i<8; i++) {
            if (i < pwbytes.length) {
                byte b = pwbytes[i];
                // flip the byte
                byte b2 = 0;
                for (int j=0; j<8; j++) {
                    b2<<=1;
                    b2 |= (b&0x01);
                    b>>>=1;
                }
                key[i] = b2;
            } else {
                key[i] = 0;
            }
        }
        return key;
    }

    /* Decrypting is left as an exercise for the reader. ;) */

    ////////////////////////////////////////////////////////////////
////
    //
    // Test methods
    //
    // The rest of the file is devoted to some simple test
infrastructure
    // for providing confidence in this DES implementation.
    //
    ////////////////////////////////////////////////////////////////
////

    private static int charToNibble(char c) {
        if (c>='0' && c<='9') {
            return (c-'0');
```

```
        } else if (c>='a' && c<='f') {
            return (10+c-'a');
        } else if (c>='A' && c<='F') {
            return (10+c-'A');
        } else {
            return 0;
        }
    }
    private static byte[] parseBytes(String s) {
        s = s.replace(" ", "");
        byte[] ba = new byte[s.length()/2];
        if (s.length()%2 > 0) { s = s+'0'; }
        for (int i=0; i<s.length(); i+=2) {
            ba[i/2] = (byte) (charToNibble(s.charAt(i))<<4 |
charToNibble(s.charAt(i+1)));
        }
        return ba;
    }
    private static String hex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for (int i=0; i<bytes.length; i++) {
            sb.append(String.format("%02X ",bytes[i]));
        }
        return sb.toString();
    }

    public static boolean test(byte[] message, byte[] expected, String
password) {
        return test(message, expected, passwordToKey(password));
    }

    private static int testCount = 0;
    public static boolean test(byte[] message, byte[] expected, byte[]
key) {
        System.out.println("Test #"+(++testCount)+":");
        System.out.println("\tmessage:  "+hex(message));
        System.out.println("\tkey:      "+hex(key));
        System.out.println("\texpected: "+hex(expected));
        byte[] received = encrypt(message, key);
        System.out.println("\treceived: "+hex(received));
        boolean result = Arrays.equals(expected, received);
        System.out.println("\tverdict: "+(result?"PASS":"FAIL"));
        return result;
    }

    public static void getCipherText(byte[] message, byte[] key) {
        System.out.println("Get Cipher Text #"+(++testCount)+":");
        System.out.println("\tmessage:  "+hex(message));
        System.out.println("\tkey:      "+hex(key));
        byte[] received = encrypt(message, key);
```

```
        System.out.println("\tciphertext "+hex(received));
    }

    public static void main(String[] args) {

                if (args.length < 2) {
                        System.out.println("Usage:");
                        System.out.println("java DES key plaintext");
                        return;
                }

                String key = args[0];
                String plaintext = args[1];

                getCipherText(parseBytes(plaintext),
parseBytes(key));
    }
}
```