```java
package hashbrown5;

/**
 * A class which implements the RFC 1321 definition of the Message Digest 5 Hash Algorithm.
 *
 * @author Eric Shen, Arman Siddique, Chris Wang
 * @version 1.0
 */
public class MD5 {
    //Some definitions of variables
    final protected static char[] hexArray = "0123456789ABCDEF".toCharArray();//hex char table

    private static final int[] s = { //Shift constants. Defined in RFC 1321.
            7, 12, 17, 22,
            5, 9, 14, 20,
            4, 11, 16, 23,
            6, 10, 15, 21
    };

    //Initial values for dwords A, B, C, and D. Defined as such by RFC 1321.
    private static final int INIT_A = 0x67452301;
    private static final int INIT_B = (int) 0xEFCDAB89L;
    private static final int INIT_C = (int) 0x98BADCFEL;
    private static final int INIT_D = 0x10325476;

    public static int[] K = new int[64]; //Array of constants used during the rounds of MD5.

    static {//calculate said constants
        for (int i = 0; i < 64; i++) {
            K[i] = (int) (long) ((1L << 32)/* This is 2^32*/ * Math.abs(Math.sin(i + 1))); //Also defined to be such in the RFC.
        }
    }

    /**
     * Main method. Can call MD5 method compute().
     *
```

```
37      * @param args
38      *         command line args, ignored
39      */
40     public static void main(String[] args) {
41         String input = "";//Input here
42         System.out.println(bytesToHex(compute(input.getBytes())))
   ;
43     }
44 
45     /**
46      * Converts a byte array into a String with the hex
   characters that the bytes represent.
47      *
48      * @param bytes
49      *         the byte array to turn into hex
50      * @return the resulting hex string
51      * @author maybeWeCouldStealAVan (via StackOverflow)
52      */
53     public static String bytesToHex(byte[] bytes) {
54         char[] hexChars = new char[bytes.length * 2];
55         for (int j = 0; j < bytes.length; j++) {
56             int v = bytes[j] & 0xFF;
57             hexChars[j * 2] = hexArray[v >>> 4];
58             hexChars[j * 2 + 1] = hexArray[v & 0x0F];
59         }
60         return new String(hexChars);
61     }
62 
63     /**
64      * Nonlinear function f which performs bitwise operations on
   inputs.
65      *
66      * @param b
67      *         dword b in MD5
68      * @param c
69      *         dword c in MD5
70      * @param d
71      *         dword d in MD5
72      * @return the result of applying the function
73      */
74     private static int f(int b, int c, int d) {
75         return (b & c) | ((~b) & d);
```

```
76      }
77
78      /**
79       * Nonlinear function g which performs bitwise operations on
    inputs.
80       *
81       * @param b
82       *          dword b in MD5
83       * @param c
84       *          dword c in MD5
85       * @param d
86       *          dword d in MD5
87       * @return the result of applying the function
88       */
89      private static int g(int b, int c, int d) {
90          return (b & d) | (c & (~d));
91      }
92
93      /**
94       * Nonlinear function h which performs bitwise operations on
    inputs.
95       *
96       * @param b
97       *          dword b in MD5
98       * @param c
99       *          dword c in MD5
100      * @param d
101      *          dword d in MD5
102      * @return the result of applying the function
103      */
104     private static int h(int b, int c, int d) {
105         return b ^ c ^ d;
106     }
107
108     /**
109      * Nonlinear function i which performs bitwise operations on
    inputs.
110      *
111      * @param b
112      *          dword b in MD5
113      * @param c
114      *          dword c in MD5
```

```
115      * @param d
116      *          dword d in MD5
117      * @return the result of applying the function
118      */
119     private static int i(int b, int c, int d) {
120         return c ^ (b | (~d));
121     }
122 
123     /**
124      * Computes the MD5 hash of the given message.
125      *
126      * @param message
127      *          the message to find the hash of
128      * @return a byte array containing the resulting dwords
129      */
130     private static byte[] compute(byte[] message) {
131         int lenBytes = message.length;
132         //Add 8 to account for necessary padding, divides by 64 since each block is 64 bytes, then adds 1 because the minimum possible number of blocks is 1.
133         int numBlocks = ((lenBytes + 8) >>> 6) + 1;
134         //finds the size that the message should be in bytes after padding.
135         int lenAfterPad = numBlocks << 6;
136         //Initialises an array which represents the bits to be padded. The length is the byte length of the padded message - the byte length of the original message.
137         byte[] pad = new byte[lenAfterPad - lenBytes];
138         /*
139         Padding scheme is as follows:
140         1. Append a single "1" bit.
141         2. Append zeroes until the length of the message modulo 512 is 448.
142         3. Append the original length of the message in bits, modulo 2^64.
143          */
144         pad[0] = (byte) 0x80;//10000000 in binary, accomplishes step 1 and begins step 2. The rest of step two is automatically accomplished as Java default initialises bytes as
145         // zeroes.
146         //Calculates the length of the message in bits. Uses longs since the Java long is 64 bits, and we need 64 bits per
```

```
146 the specification.
147         long lenBits = (long) lenBytes << 3;
148 
149         //Appends the length of the message(modulo 2 pow 64).
    This is accomplished by replacing the zero bytes at the end of
    the pad array with the length of the message.
150         for (int i = 0; i < 8; i++) {
151             pad[pad.length - 8 + i] = (byte) lenBits;
152             lenBits >>>= 8; //shifts eight bits over to get the
    next byte.
153         }
154 
155         //A buffer to hold the 32 bit dwords which are to be
    processed.
156         int[] buffer = new int[16];
157 
158         //Initialise internal dwords.
159         int a = INIT_A;
160         int b = INIT_B;
161         int c = INIT_C;
162         int d = INIT_D;
163 
164         //Process all blocks.
165         for (int i = 0; i < numBlocks; i++) {
166             int ind = i << 6; //Converts the current block(i) to
     a byte offset. This byte offset is how many bytes of the
    message have already been processed.
167             //Parses the 512 bit block into 16 32-bit dwords,
    which are placed into the buffer as ints.
168             for (int j = 0; j < 64; j++, ind++) {
169                 buffer[j / 4] = (int) ((ind < lenBytes) ?
    message[ind] : pad[ind - lenBytes]) << 24/*Shift to make room
    for data before pad*/ | (buffer[j >>> 2] >>> 8 /*Or with
170                 old data */);
171                 //TODO Explain this line
172             }
173 
174             //***Rounds to process dwords***
175             for (int j = 0; j < 64; j++) {
176                 int t = j / 16;
177                 int f = 0; //result of nonlinear function
178                 int g = 0; //which message dword to use
```

```
179
180                 switch (t) {
181                     case 0:
182                         f = f(b, c, d); //Applies the nonlinear
    function f
183                         g = j; //Chooses the message block to
    use
184                         break;
185                     case 1:
186                         f = g(b, c, d); //Applies the nonlinear
    function g
187                         g = (5 * j + 1) % 16; //Chooses the
    message block to use
188                         break;
189                     case 2:
190                         f = h(b, c, d); //Applies the nonlinear
    function h
191                         g = (3 * j + 5) % 16; //Chooses the
    message block to use
192                         break;
193                     case 3:
194                         f = i(b, c, d); //Applies the nonlinear
    function i
195                         g = (7 * j) % 16; //Chooses the message
    block to use
196                         break;
197                 }
198
199                 int temp = d;
200                 d = c;
201                 c = b;
202                 b = b + Integer.rotateLeft((a + f + K[j] +
    buffer[g]), s[t << 2 | (j & 3)]);
203                 /*The magical bit shifting produces this
    sequence:
204                 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3,
205                 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7,
206                 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9,
    10, 11,
207                 12, 13, 14, 15, 12, 13, 14, 15, 12, 13, 14, 15,
    12, 13, 14, 15,
208                 which is the index used for the per round shift
```

```
208 array.
209                 */
210                 a = temp;
211             }
212 
213             //Add original constants back in.
214             a += INIT_A;
215             b += INIT_B;
216             c += INIT_C;
217             d += INIT_D;
218         }
219 
220         //Process to form output.
221         byte[] result = new byte[16];
222         int ind = 0;
223 
224         for (int i = 0; i < 4; i++) {
225             int n = (i == 0) ? a : ((i == 1) ? b : (i == 2) ? c
    : d); //goes through all four dwords
226             for (int j = 0; j < 4; j++) {
227                 result[ind++] = (byte) (n); //truncates first 24
    bits of n, leaves last 8 bits
228                 n >>>= 8; //shift over 8 to access next 8 bits
229                 //TODO Draw example on board
230             }
231         }
232         return result;
233     }
234 }
```