1. **import** java.util.\*;
2. **import** java.io.IOException;
3. **import** java.io.UnsupportedEncodingException;
4. **import** java.math.\*;
5. **import** java.nio.charset.Charset;
6. **import** java.nio.charset.StandardCharsets;
7. **import** java.util.BitSet;
9. *//The program is an implementation of SHA3-256.*
10. **public** **class** Keccak
11. {
12. *//Secure coding rule: MSC55-J followed*
13. *//Converts a given string of characters to a string of 0's and 1's*
14. **static** String ConvertStringToBinary(String s)
15. {
16. String binary="";
17. **try**
18. {
19. **for**(**int** i=0;i<s.length();i++)
20. {
21. binary += Integer.toBinaryString(s.charAt(i));
22. }
23. **return** binary;
24. }
25. **catch**(Exception e)
26. {
27. System.out.println("error");
28. }
29. **return** binary;
30. }
32. *//Converts a given string of 0's and 1's to a string of characters*
33. **static** String ConvertBinaryToString(String s)
34. {
35. String textString= "";
37. **int** stringLimit=s.length()-7;
38. **for**(**int** i=0;i<=stringLimit;i+=7)
39. {
40. **int** charCode = Integer.parseInt(s.substring(i, (i+7)), 2);
41. textString += (**char**)charCode;
42. }
43. **return** textString;
44. }
46. *//Converts a given string of 0's and 1's to a new BitSet*
47. **static** BitSet ConvertBinaryStringToBitSet(String s)
48. {
49. BitSet B= **new** BitSet(s.length());
50. *//Secure coding rule: DCL53-J, NUM09-J followed*
51. **for**(**int** i=0;i<s.length();i++)
52. {
53. *//Secure coding rule: EXP52-J followed*
54. **if**(s.charAt(i)=='1')
55. {
56. B.set(i,**true**);
57. }
58. **else**
59. {
60. B.set(i,**false**);
61. }
62. }
63. **return** B;
64. }
66. *//Converts a given BitSet to a string of 0's and 1's*
67. **static** String ConvertBitSetToBinaryString(BitSet b)
68. {
69. String s="";
70. *//Secure coding rule: DCL53-J, NUM09-J followed*
71. **for**(**int** i=0;i<b.size();i++)
72. {
73. *//Secure coding rule: EXP52-J followed*
74. **if**(b.get(i)==**true**){
75. s+="1";
76. }
77. **else**{
78. s+="0";
79. }
80. }
81. **return** s;
82. }
84. *//Creates a new 3D BitSet and returns the BitSet object*
85. **static** BitSet[][] Create3DBitSet()
86. {
87. BitSet[][] X= **new** BitSet[5][5];
88. *//Secure coding rule: DCL53-J, NUM09-J followed*
89. **for**(**int** x=0;x<5;x++)
90. {
91. **for**(**int** y=0;y<5;y++)
92. {
93. X[x][y]= **new** BitSet(64);
94. }
95. }
96. **return** X;
97. }
99. *//Copies given 3D BitSet into another a new BitSet object*
100. **static** BitSet[][] Copy3DBitSet(BitSet A[][])
101. {
102. BitSet NewBitSet[][]=Create3DBitSet();
103. *//Secure coding rule: DCL53-J, NUM09-J followed*
104. **for**(**int** x=0;x<5;x++)
105. {
106. **for**(**int** y=0;y<5;y++)
107. {
108. **for**(**int** z=0;z<64;z++)
109. {
110. *//Secure coding rule: EXP52-J followed*
111. **if**(A[x][y].get(z)==**true**){
112. NewBitSet[x][y].set(z,**true**);
113. }
114. **else**{
115. NewBitSet[x][y].set(z,**false**);
116. }
117. }
118. }
119. }
120. **return** NewBitSet;
121. }
123. *//Function to perform single bit xoring within a BitSet*
124. **static** **boolean** SingleBitXOR(BitSet B,**int** i,**int** j)
125. {
126. *//Secure coding rule: EXP52-J followed*
127. **if**(B.get(i)==B.get(j)){
128. **return** **false**;
129. }
130. **else** {
131. **return** **true**;
132. }
134. }
136. *//Step 1 of the Keccak round function*
137. **static** BitSet[][] ThetaStep(BitSet[][] A)
138. {
139. BitSet OrigA[][]=Copy3DBitSet(A);
140. *//Secure coding rule: DCL52-J followed*
141. BitSet[]C= **new** BitSet[5];
142. BitSet[]D= **new** BitSet[5];
143. **for**(**int** a=0;a<5;a++)
144. {
145. C[a]= **new** BitSet(64);
146. D[a]= **new** BitSet(64);
147. }
148. *//Step 1 of theta*
149. *//Secure coding rule: DCL53-J, NUM09-J followed*
150. **for**(**int** x=0;x<5;x++)
151. {
152. A[x][0].xor(A[x][1]);
153. A[x][0].xor(A[x][2]);
154. A[x][0].xor(A[x][3]);
155. A[x][0].xor(A[x][4]);
156. C[x]=A[x][0];
157. }
158. Boolean a=**false**;
159. *//copying bitset C into D*
160. *//Secure coding rule: DCL53-J, NUM09-J followed*
161. **for**(**int** x=0;x<5;x++)
162. {
163. **for**(**int** z=0;z<64;z++)
164. {
165. **if**(C[x].get(z)==**true**){
166. D[x].set(z,**true**);
167. }
168. **else**{
169. D[x].set(z,**false**);
170. }
171. }
172. }
174. *//rotate each lane in C by 1*
175. **for**(**int** x=0;x<5;x++)
176. {
177. *//Secure coding rule: DCL53-J, NUM09-J followed*
178. **for** (**int** z=0;z<64;z++)
179. {
180. *//Secure coding rule: EXP53-J, NUM02-J, NUM51-J followed*
181. **int** i = (((z-1)%64)+64)%64;
182. a= D[x].get(i);
183. **if**(a==**true**){
184. C[x].set(z,**true**);
185. }
186. **else**{
187. C[x].set(z,**false**);
188. }
189. }
190. }
192. **for**(**int** x=0;x<5;x++)
193. {
194. *//Secure coding rule: EXP53-J, NUM02-J, NUM51-J followed*
195. D[(((x-1)%5)+5)%5].xor(C[(x+1)%5]);
196. }
197. *//Secure coding rule: DCL53-J, NUM09-J followed*
198. **for**(**int** x=0;x<5;x++)
199. {
200. **for**(**int** y=0;y<5;y++)
201. {
202. *//Secure coding rule: EXP53-J, NUM02-J, NUM51-J followed*
203. OrigA[x][y].xor(D[(((x-1)%5)+5)%5]);
204. }
205. }
206. **return** OrigA;
207. }
209. *//Step 2 of the Keccak round function*
210. **static** BitSet[][] RhoStep(BitSet[][] A)
211. {
212. BitSet OrigRho[][]=Copy3DBitSet(A);
213. *//Secure coding rule: DCL52-J followed*
214. **int** x=1;
215. **int** y=0;
216. *//Secure coding rule: DCL53-J, NUM09-J followed*
217. **for**(**int** t=0;t<24;t++)
218. {
219. **for**(**int** z=0;z<64;z++)
220. {
221. *//Secure coding rule: EXP53-J, NUM02-J, NUM51-J followed*
222. **int** i= (((z-((t+1)\*(t + 2))/2)%64)+64)%64;
223. **if**(A[x][y].get(i)==**true**){
224. OrigRho[x][y].set(z,**true**);
225. }
226. **else**{
227. OrigRho[x][y].set(z,**false**);
228. }
229. **int** temp=x;
230. x=y;
231. *//Secure coding rule: EXP53-J, NUM02-J followed*
232. y=(((2\*temp)+(3\*y))%5);
233. }
234. }
236. **return** OrigRho;
237. }
239. *//Step 3 of the Keccak round function*
240. **static** BitSet[][] PiStep(BitSet[][] A)
241. {
242. BitSet OrigPi[][]=Copy3DBitSet(A);
243. *//Secure coding rule: DCL53-J, NUM09-J followed*
244. **for**(**int** x=0;x<5;x++)
245. {
246. **for**(**int** y=0;y<5;y++)
247. {
248. *//Secure coding rule: NUM02-J followed*
249. OrigPi[x][y]=A[(x+(3\*y))%5][x];
250. }
251. }
252. **return** OrigPi;
253. }
255. *//Step 4 of the Keccak round function*
256. **static** BitSet[][] ChiStep(BitSet[][] A)
257. {
258. BitSet OrigChi[][]=Copy3DBitSet(A);
259. BitSet AllOnes[][]= Create3DBitSet();
261. *//Initializing all values of 3D BitSet to true*
262. *//Secure coding rule: DCL53-J, NUM09-J followed*
263. **for**(**int** x=0;x<5;x++)
264. {
265. **for**(**int** y=0;y<5;y++)
266. {
267. **for**(**int** z=0;z<64;z++)
268. {
269. AllOnes[x][y].set(z,**true**);
270. }
271. }
272. }
274. *//NOT operation*
275. *//Secure coding rule: DCL53-J followed*
276. **for**(**int** x=0;x<5;x++)
277. {
278. **for**(**int** y=0;y<5;y++)
279. {
280. A[x][y].xor(AllOnes[x][y]);
281. }
282. }
283. *//AND operation*
284. *//Secure coding rule: DCL53-J, NUM09-J followed*
285. **for**(**int** x=0;x<5;x++)
286. {
287. **for**(**int** y=0;y<5;y++)
288. {
289. *//Secure coding rule: NUM02-J followed*
290. A[(x+1)%5][y].and(OrigChi[(x+2)%5][y]);
291. }
292. }
293. *//XOR operation*
294. *//Secure coding rule: DCL53-J, NUM09-J followed*
295. **for**(**int** x=0;x<5;x++)
296. {
297. **for**(**int** y=0;y<5;y++)
298. {
299. *//Secure coding rule: NUM02-J followed*
300. OrigChi[x][y].xor(A[(x+1)%5][y]);
301. }
302. }
303. **return** OrigChi;
304. }
306. *//Round constant function for Step 5: Iota*
307. **static** **boolean** rc(**int** t)
308. {
309. **if**((t%255)==0)
310. **return** **true**;
311. **else**
312. {
313. String R="10000000";
314. BitSet R1;
315. *//Secure coding rule: DCL53-J, NUM09-J followed*
316. **for**(**int** i=1;i<(t%255);i++)
317. {
318. R="0"+R;
319. R1=ConvertBinaryStringToBitSet(R);
320. R1.set(0,SingleBitXOR(R1,0,8));
321. R1.set(4,SingleBitXOR(R1,4,8));
322. R1.set(5,SingleBitXOR(R1,5,8));
323. R1.set(6,SingleBitXOR(R1,6,8));
324. R=ConvertBitSetToBinaryString(R1);
325. R=R.substring(0, 8);
326. }
327. **char** retValue;
328. retValue=R.charAt(0);
329. **if**(retValue=='1'){
330. **return** **true**;
331. }
332. **else**{
333. **return** **false**;
334. }
335. }
336. }
338. *//Step 5 of the Keccak round function*
339. **static** BitSet[][] IotaStep(BitSet[][] A, **int** iR)
340. {
341. BitSet[][] OrigIota= Copy3DBitSet(A);
342. BitSet RC= **new** BitSet(64);
343. **int** l=0;
344. *//Secure coding rule: NUM02-J followed*
345. l=(**int**)Math.floor(Math.log(64)/Math.log(2));
347. **for**(**int** j=0;j<l;j++)
348. {
349. *//Secure coding rule: EXP53-J followed*
350. **int** temp=(**int**)((Math.pow(2, j))-1);
351. RC.set(temp,(rc(j+(7\*iR))));
352. }
353. OrigIota[0][0].xor(RC);
354. **return** OrigIota;
355. }
357. **static** BitSet KeccakFunction(BitSet X)
358. {
359. BitSet S[][]= **new** BitSet[5][5];
360. S=ConvertBitSetTo3DState(X);
361. *//24 rounds of all 5 steps of Keccak permutation!*
362. *//Secure coding rule: DCL53-J, NUM09-J followed*
363. **for**(**int** i=0;i<25;i++)
364. {
365. S=ThetaStep(S);
366. S=RhoStep(S);
367. S=PiStep(S);
368. S=ChiStep(S);
369. S=IotaStep(S,i);
370. }
371. *//convert 3D bitset back to 1D*
372. **return** (Convert3DStateToBitSet(S));
373. }
375. **static** BitSet Convert3DStateToBitSet(BitSet[][] X)
376. {
377. BitSet S= **new** BitSet(1600);
378. String interimS="";
379. *//Secure coding rule: DCL53-J, NUM09-J followed*
380. **for**(**int** x=0;x<5;x++)
381. {
382. **for**(**int** y=0;y<5;y++)
383. {
384. interimS+=ConvertBitSetToBinaryString(X[x][y]);
385. }
386. }
387. S=ConvertBinaryStringToBitSet(interimS);
388. **return** S;
389. }

392. **static** BitSet[][] ConvertBitSetTo3DState(BitSet X)
393. {
394. BitSet S[][]= Create3DBitSet();
395. *//Secure coding rule: DCL53-J, NUM09-J followed*
396. **for**(**int** x=0;x<5;x++)
397. {
398. **for**(**int** y=0;y<5;y++)
399. {
400. **for**(**int** z=0;z<64;z++)
401. {
402. *//Secure coding rule: EXP53-J followed*
403. **int** i=(64\*((5\*y)+x))+z;
404. **boolean** a= X.get(i);
405. **if**(a==**true**){
406. S[x][y].set(z,**true**);
407. }
408. **else**{
409. S[x][y].set(z,**false**);
410. }
411. }
412. }
413. }
414. **return** S;
415. }
417. **static** String KeccakPad(**int** r, **int** m)
418. {
419. *//Secure coding rule: EXP53-J, NUM02-J, NUM51-J followed*
420. **int** j=((((-m-2)%r)+r)%r);
421. String zeroPad="";
422. **for**(**int** i=0;i<j;i++)
423. {
424. zeroPad+="0";
425. }
426. String P="1"+zeroPad+"1";
427. *//System.out.println("Pad: "+P);*
428. **return** P;
429. }
431. **static** String SPONGE(String N,**int** d)
432. {
433. **int** r=576, b=1600;
434. String P=N+KeccakPad(r,N.length());    *//value of r is hardcoded here*
435. *//int n= (P.length()/r); n= number of input message blocks,*
436. *//the above statement is commented because it was part of the NIST algorithm but had no visible use*
437. **int** c=b-r;
438. BitSet S= **new** BitSet(1600);             *//state array of 'b' bits, here b=1600*
439. BitSet C= **new** BitSet(c);
441. **for**(**int** x=0;x<P.length();x+=r)
442. {
443. *//BitSet P0=new BitSet(r);*
444. String P0=(P.substring(x,r));
445. BitSet B0= **new** BitSet(b);
446. B0=ConvertBinaryStringToBitSet(P0+ConvertBitSetToBinaryString(C));
447. S.xor(B0);
448. S=KeccakFunction(S);
449. }
450. *//Secure coding rule: DCL52-J followed*
451. String z="";
452. String S1="";
453. **while**(z.length()<d)
454. {
455. S1= ConvertBitSetToBinaryString(S);
456. z = z + S1.substring(0, r);
457. **if**(z.length()>d)
458. {
459. **break**;
460. }
461. S=KeccakFunction(S);
462. }
463. **return** (z.substring(0, d));
464. }
466. *//main function*
467. **public** **static** **void** main (String args[]) **throws** IOException
468. {
469. Scanner kbd= **new** Scanner(System.in);
470. System.out.println("**\n**Enter the text message: ");
471. String input=kbd.nextLine();
472. *//Secure coding rule: EXP03-J followed*
473. **if**(input.equals(""))
474. {
475. System.out.println("Invalid input!");
476. }
477. **else**
478. {
479. String binary=ConvertStringToBinary(input);
480. String output="";
481. System.out.println("Entered text in binary: "+"**\n**"+binary);
482. output=SPONGE(binary,256);
483. String messageDigest1="";
484. messageDigest1=ConvertBinaryToString(output);
485. System.out.println("Final hash value: "+messageDigest1);
486. }
487. System.out.println("Bye!");
488. kbd.close();
489. }*//end of main*
490. }*//end of class*