7. homework assignment; JAVA, Part 1, Academic year 2017/2018; FER

Napravite prazan Maven projekt, kao u 1. zadaći: u Eclipsovom workspace direktoriju napravite direktorij `hw07-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.jmbag0000000000:hw07-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit:junit:4.12`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Pročitajte posljednju stranicu upute. Jeste? OK; ova zadaća sastoji se od dva dijela.

Problem 1.

You will write a program `Crypto` that will allow the user to encrypt/decrypt given file using the AES crypto-algorithm and the 128-bit encryption key or calculate and check the SHA-256 file digest. Since this kind of cryptography works with binary data, use octet-stream Java based API for reading and writing of files. What needs to be programmed is illustrated by the following use cases (I have skipped *classpath* parameter; set it to appropriate value). You will find in repository additional file which is used in this example: `hw07test.bin`. Download this file and place it in your projects current directory. Program outputs are shown red, user input is shown in blue.

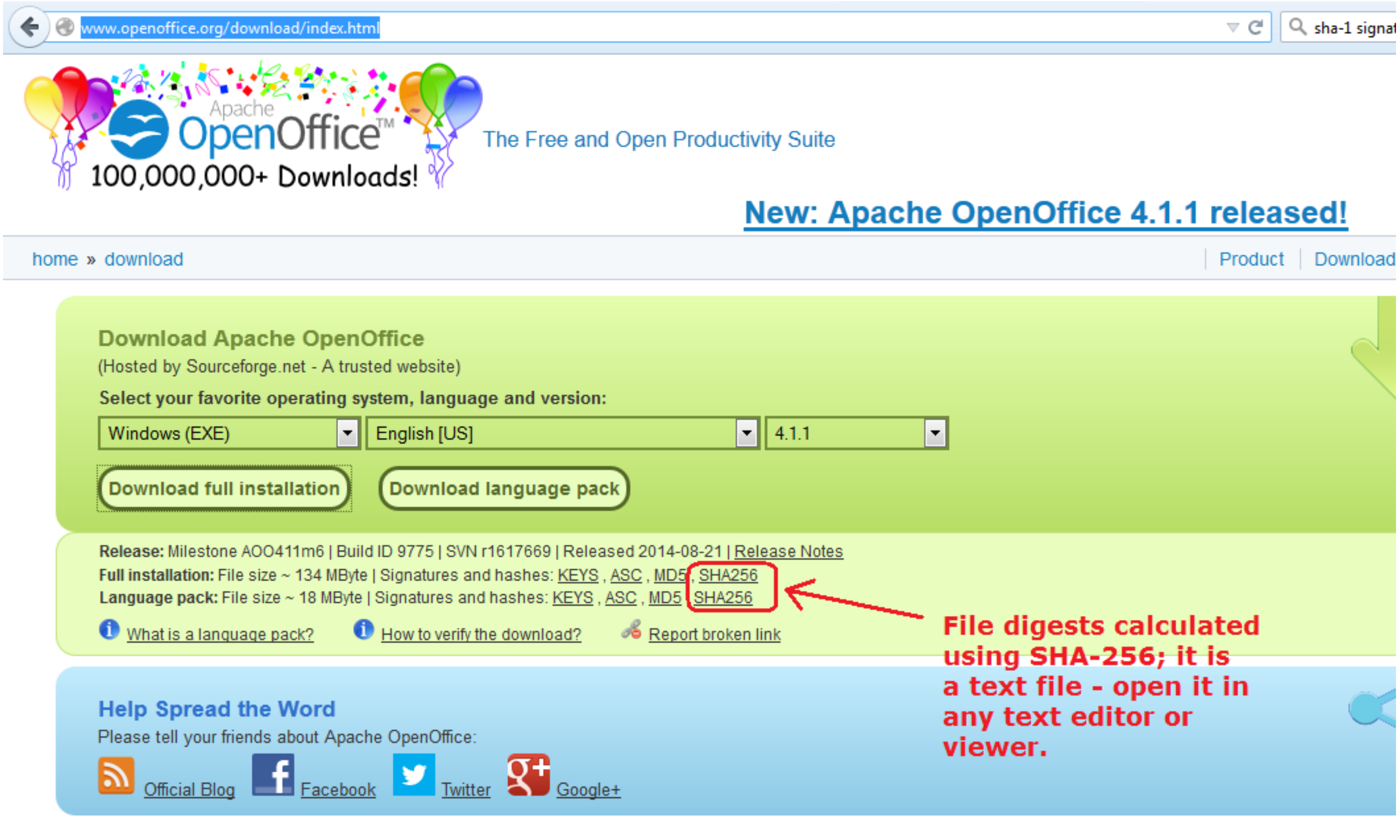
```
java hr.fer.zemris.java.hw07.crypto.Crypto checksha hw07test.bin
Please provide expected sha-256 digest for hw07test.bin:
> 2e7b3a91235ad72cb7e7f6a721f077faacfeafdea8f3785627a5245bea112598
Digesting completed. Digest of hw07test.bin matches expected digest.

java hr.fer.zemris.java.hw07.crypto.Crypto checksha hw07test.bin
Please provide expected sha-256 digest for hw07test.bin:
> d03d4424461e22a458c6c716395f07dd9cea2180a996e78349985eda78e8b800
Digesting completed. Digest of hw07test.bin does not match the expected digest. Digest
was: 2e7b3a91235ad72cb7e7f6a721f077faacfeafdea8f3785627a5245bea112598

java hr.fer.zemris.java.hw07.crypto.Crypto encrypt hw07.pdf hw07.crypted.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> e52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Encryption completed. Generated file hw07.crypted.pdf based on file hw07.pdf.

java hr.fer.zemris.java.hw07.crypto.Crypto decrypt hw07.crypted.pdf hw07orig.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> e52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Decryption completed. Generated file hw07orig.pdf based on file hw07.crypted.pdf.

java hr.fer.zemris.java.hw07.crypto.Crypto decrypt hw07test.bin hw07test.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> e52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Decryption completed. Generated file hw07test.pdf based on file hw07test.bin.
```

First two examples test your implementation of digest calculation. Third and fourth example test is your implementation of file encryption and decryption compatible with itself. The fifth example tests is your decryption procedure compatible with the encryption procedure which was done by me. If this last step works, you will be able to open `hw07test.pdf` in PDF viewer and read its content.

Lets just briefly explain some of the concepts from this problem.

Message digest is a fixed-size binary digest which is calculated from arbitrary long data. The idea is simple. You have some original data (lets denote it D); this can be a file on disk. Then you calculate a digest for this data (lets denote it S); for example, if S is calculated with SHA-256 algorithm, the digest will always be 256-bits long, no matter how long is the original file you digested. Generally speaking, the original data can not be reconstructed from the digest and this is not what the digests are used for. Digests are used to verify if the data you have received (for example, when downloading the data from the Internet) arrived unchanged. You will verify this by calculating the digest on the file you have downloaded and then you will compare the calculated digest with the digest which is published on the web site from which you have started the download. If something has changed during the download, there is extremely high probability that the calculated digest will be different from the one published on the web site. You can see this on many of web-pages which offer file download. Visit, for example, Open Office download page:

http://www.openoffice.org/download/index.html


www.openoffice.org/download/index.html
sha-1 signat
Apache OpenOffice™
100,000,000+ Downloads!
The Free and Open Productivity Suite
New: Apache OpenOffice 4.1.1 released!
home » download
Product
Download
Download Apache OpenOffice
(Hosted by Sourceforge.net - A trusted website)
Select your favorite operating system, language and version:
Windows (EXE)
English [US]
4.1.1
Download full installation
Download language pack
What is a language pack?
How to verify the download?
Report broken link
Help Spread the Word
Please tell your friends about Apache OpenOffice:
Official Blog
Facebook
Twitter
Google+

Release: Milestone AOO411m6 | Build ID 9775 | SVN r1617669 | Released 2014-08-21 | Release Notes
Full installation: File size ~ 134 MByte | Signatures and hashes: KEYS , ASC , MD5 , SHA256
Language pack: File size ~ 18 MByte | Signatures and hashes: KEYS , ASC , MD5 , SHA256

File digests calculated using SHA-256; it is a text file - open it in any text editor or viewer.

Note: Digests will be integral part of *digital signature* – a mechanism which is today broadly used online as a replacement for persons physical signature. At FER you will learn more on this if you enroll the course Advanced operating systems (*Computing Master programme*, profile *Computer Science*).

Encryption is the conversion of data into a form, called a *ciphertext*, that can not be easily understood by unauthorized people. *Decryption* is the reverse process: it is a transformation of ciphertext back into its original form. There are two families of cryptography: *symmetric* in which both encryption and decryption use the same key (i.e. “password”), and *asymmetric* in which a pair of keys is used (public key and private key which are mutually inverse: what is encrypted with one can only be decrypted with other). Since the decryption of encrypted data must be possible, there can be no loss of data (as is the case with digests). Encrypted data will always be as big (or even bigger) as were the original data. In this homework we will use a symmetric crypto-algorithm AES which can work with three different key sizes: 128 bit, 192 bit and 256 bit. Since AES is *block cipher*, it always consumes 128 bit of data at a time (or adds padding if no more data is available) and produces 128 bits of encrypted text. Therefore, the length (in bytes) of encrypted file file will always be divisible by 16.

In this homework you are not expected to implement these algorithms. You only have to learn how to use them in your programs. Java already offers appropriate implementations. Please consult the following references:

http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#MessageDigest
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Cipher
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#MDEx
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#SimpleEncrEx

Encryption keys and initialization vectors are byte-arrays each having 16 bytes. In the above example it is expected from the user to provide these as hex-encoded texts.

Implement these methods. To obtain properly initialized `Cipher` object, use following code snippet:

```
String keyText = … what user provided for password …
String ivText = … what user provided for initialization vector …
SecretKeySpec keySpec = new SecretKeySpec(Util.hextobyte(keyText), "AES");
AlgorithmParameterSpec paramSpec = new IvParameterSpec(hextobyte(ivText));
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(encrypt ? Cipher.ENCRYPT_MODE : Cipher.DECRYPT_MODE, keySpec, paramSpec);
```

Create a class `Util` with two public static methods: `hextobyte(keyText)` and `bytetohex(bytearray)`. Method `hextobyte(keyText)` should take hex-encoded String and return appropriate `byte[]`. If string is not valid (odd-sized, has invalid characters, …) throw an `IllegalArgumentException`. For zero-length string, method must return zero-length byte array. Method must support both uppercase letters and lowercase letters.

For example: `hextobyte("01aE22")` should return `byte[] {1, -82, 34}`.

Method `bytetohex(bytearray)` takes a byte array and creates its hex-encoding: for each byte of given array, two characters are returned in string, in big-endian notation. For zero-length array an empty string must be returned. Method should use lowercase letters for creating encoding.

For example: `bytetohex(new byte[] {1, -82, 34})` should return `"01ae22"`.

You are expected to write your own implementation of these two methods (using other classes for this is not allowed). Write unit tests for this methods to ensure they work correctly.

Please note, **you are not allowed** to use CipherInputStream or CipherOutputStream (or any of its subclasses); you are required to implement encryption/decryption directly using Cipher object and a series of update/update/update/... completed by doFinal(). Also, you are not allowed to read a complete file into memory, then encrypt/decrypt it and then write the result back to disk since the input file can be huge. You are only allowed to read a reasonable amount of file into memory at each single time (for example, 4k) – use byte streams for this. The same goes for constructing the resulting file. For reading file you must use an instance of FileInputStream and for writing an instance of FileOutputStream. Ensure that they are both buffered – use appropriate method for constructing buffered file streams directly from Files class.

Be aware that algorithms we use here for encryption and decryption are block-based. They must get a block of bytes in order to create new encrypted/decrypted block of bytes. This is the reason for existence of methods update and doFinal. When you pass some bytes by calling update, algorithm processes as many blocks as possible and return processed blocks while retaining still-to-process bytes in internal buffer. When you call update again and deliver new bunch of bytes, processing continues and you get new processed blocks as result. You repeat this in loop, reading data from input stream, and writing processed data to output stream. Once the input stream is drained, you must complete processing by calling doFinal. Using this call, you signal to the encryption/decryption procedure that there will be no more data, and that in order to complete processing, it should itself add padding in needed in order to create last block of data and then process it, returning to you the processed data.

Taking this into account, do not be surprised if you find out that the encrypted file is a bit larger then the original file: its size will be multiple of algorithm block size. When performing decryption, the algorithm will be aware that the padding was added, and you will obtain only the data you sent in when encrypting: reconstructed file will have the same size as the original file.

Problem 2.

Download the file `hw07part2.bin` from Ferko repository and save it in you current directory. Now run your program:

```
java hr.fer.zemris.java.hw07.crypto.Crypto checksha hw07part2.bin
Please provide expected sha-256 digest for hw07part2.bin:
> 603ce08075a10ea3f781301bfafc01e3e9c9487ba33790d4afa7fd15dffd2b94
Digesting completed. Digest of hw07part2.bin matches expected digest.
```

If you obtain different result, there is something wrong; either the file `hw07part2.bin` is corrupted (redownload it again) or you have bug in your program (fix it). When you do obtain result as expected, run following command:

```
java hr.fer.zemris.java.hw07.crypto.Crypto decrypt hw07part2.bin hw07part2.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> e52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Decryption completed. Generated file hw07part2.pdf based on file hw07part2.bin.
```

Open the file you just generated, read it and proceed as instructed by the text in that file.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

If you need any help, I have reserved usual timeslots for consultations: Monday Noon, Tuesday at 10 AM, and Wednesday at 9 AM; also Thursday and Friday at Noon. Feel free to drop by my office.

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are expected to write tests for Util class methods.
You are encouraged to write tests for other problems.

Once you complete the problems 1 and 2, you will discover what is left, in order to complete the entire homework.

When your **complete** homework is done, pack it in zip archive with name `hw07-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 28th 2018. at 23:59.