Java SE Security
Lab 1

1 Lab setup

Make sure you have the following items installed

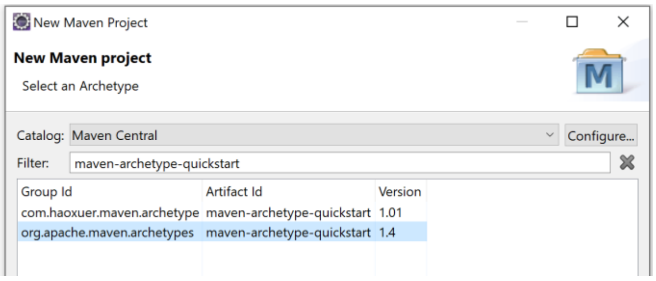
- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Eclipse Enterprise Edition or Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A suitable hex editor (HxD Hex Editor)
- A utility to extract zip files

2 Character Encoding and Byte Arrays

The source code for this lab is in the `Binary-Basics-Demo` folder.

Switch to Java EE perspective.

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-quickstart` in the filter. Eclipse may freeze temporarily while it attempts to filter through all the archetypes available at Maven Central. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Catalog: Maven Central
Filter: maven-archetype-quickstart

Group Id	Artifact Id	Version
com.haoxuer.maven.archetype	maven-archetype-quickstart	1.01
org.apache.maven.archetypes	maven-archetype-quickstart	1.4

Enter in the following details and click Finish.

Group id: `com.workshop.basics`
Artifact id: `BinaryBasicsDemo`
Version: `0.0.1-SNAPSHOT`
Package: `com.workshop.basics`

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

In the package, `com.workshop.basics` place the following new files:

```
BasicUtils
EncodingDemo
```

The `byte` data type is the one most widely used to work with pure binary data in Java. A byte value in Java is an 8-bit signed two's complement integer ranging from -128 to 127
https://www.electronics-tutorials.ws/binary/signed-binary-numbers.html
https://www.bbc.co.uk/bitesize/guides/zjfgjxs/revision/5
https://www.exploringbinary.com/twos-complement-converter/

The Java String data type can hold all the range of characters from the Unicode coding standard.

Run the `EncodingDemo` class.

We can use the `Arrays.toString` to print out the contents of a byte array.

The `displayBinaryArray` method shows the contents of the byte array in binary, signed, unsigned and hex form. Notice that for negative byte values, the MSB of 1 in the binary sequence indicates a negative number as byte values are interpreted as signed 2's complement. However, we can also interpret that binary sequence as an unsigned decimal number if we wish to.
https://www.exploringbinary.com/twos-complement-converter/

https://www.rapidtables.com/convert/number/binary-to-decimal.html

We use `getBytes` to encode a String to an array of byte using a particular character encoding scheme. If we do not specify a scheme, the platform default is used, which is typically UTF-8 on most systems.

The list of standard Charsets (character encoding) in Java is listed at:
https://docs.oracle.com/javase/8/docs/api/java/nio/charset/StandardCharsets.html

Notice that for the string `normalCharString`, the characters in there are within the ASCII range 32 - 126, and for this case, the encoding into a byte sequence within the byte array is identical regardless of whether we are using ASCII or ISO 8859-1

Different versions of the ASCII encoding can be found at:
https://www.rapidtables.com/code/text/ascii-table.html
https://www.ascii-code.com/

https://www.sciencebuddies.org/science-fair-projects/references/ascii-table

Different versions of the ISO 8859-1 encoding
https://cs.stanford.edu/people/miles/iso8859.html
https://www.htmlhelp.com/reference/charset/iso224-255.html

For the string `extendedCharString`, notice that the attempt to encode to ASCII fails as all of the characters in the string (with the exception of space) are not contained within the ASCII character set. As a result, these characters are all encoded to binary sequence with the decimal value of 63. However, these characters are available within the ISO 8859-1 encoding, and they are encoded successfully

We can see that the value of the binary sequence if it was interpreted as an unsigned positive number is the correct encoding sequence for ISO 8859-1 for that character.

When we encode using UTF-8, once we are above decimal 127 for the Unicode code point, we will encode using two bytes for a single character. For example, the character £ is encoded as two consecutive bytes with the values of 194 and 163, while the character Ø is encoded as two consecutive bytes with the values of 195 and 152, and so on.
https://www.utf8-chartable.de/unicode-utf8-table.pl?utf8=dec

For the string `arabCharString`, notice that the attempt to encode to ISO 8859-1 fails as all of the characters in the string (with the exception of space) are not contained within the ISO 8859-1 encoding. As a result, these characters are all encoded to binary sequence with the decimal value of 63. However, these characters are available within the UTF-8 encoding, and they are encoded successfully, again using two bytes for a single character. For example, the character ؆ is encoded as two consecutive bytes with the values of 216 and 134, while the character ؈ is encoded as two consecutive bytes with the values of 216 and 136
https://www.utf8-chartable.de/unicode-utf8-table.pl?start=1536&utf8=dec

The full range of Unicode characters and their coding points can be found at:
https://unicode-table.com/en/#basic-latin

The official code chart can be found at:
https://unicode.org/charts/

Finally, we call `getBytes()` without specifying any Charset, which uses the default platform encoding, which on Windows will typically be UTF-8. If we now decode back using the same encoding standard, we will get back the same sequence of characters. Obviously, if we decode back using a different encoding scheme we will get back a completely different sequence of characters since `extendedCharString` encodes to different byte sequences depending on the encoding scheme utilized.

This emphasizes the important point: We **ALWAYS** need to be mindful of the particular encoding scheme we are using when we are encoding / decoding between Strings and byte arrays. Stick to UTF-8, unless you have a good reason not to.

3 Base64 and Hex String encoding

In the package, `com.workshop.basics` place the following new files:

`Base64HexDemo`

Run the app.

When we decode a byte array with values in the printable ASCII range of 32 - 126 into a String, we can view all the characters in the String. On the other hand, if we directly decode a byte array with values in the non-printable range (0 - 31) into a String, we are unable to view some or all of the characters, regardless of the encoding scheme we use. This is because all encoding schemes (UTF-8, ASCII, ISO 8859-1) will have some values that do not map to a printable character. This makes it difficult to perform visual comparison of these strings, which is something that we will do often.
https://www.ascii-code.com/

To work around this, we can encode a byte array using into either a Base64 string or a Hex string, which can subsequently be decoded back into the original byte array. The encoded version of the original byte array allows us to visually compare the contents of two byte arrays.

Key points to note with Base64 encoding
- Uses 64 characters within the ASCII printable range: 10 digits, 26 lowercase characters, 26 uppercase characters as well as the Plus sign (+) and the Forward Slash (/)
- String length is always multiple of 4; special character = is used as padding at the string end to ensure that this is maintained
- Length is the same for group of 3 consecutive bytes

https://www.base64encode.net/

Key points to note with Hex String encoding
- Uses only characters for the Hex numbering system (0 - 9, A - F)
- String length is always twice the number of bytes (as one byte is always encoded by 2 Hex characters)

https://www.rapidtables.com/convert/number/hex-to-decimal.html

Summary:
Base64 encoding
- More compact compared to Hex string
- More difficult to manually figure out original byte sequence by examining Base64 string

Hex String
- More longer compared to Base64 string
- Easier to manually figure out original byte sequence by examining Hex string

4 File I/O with Strings and Byte Arrays

In the package, `com.workshop.basics` place the following new files:

`FileDemo`

Run the app.

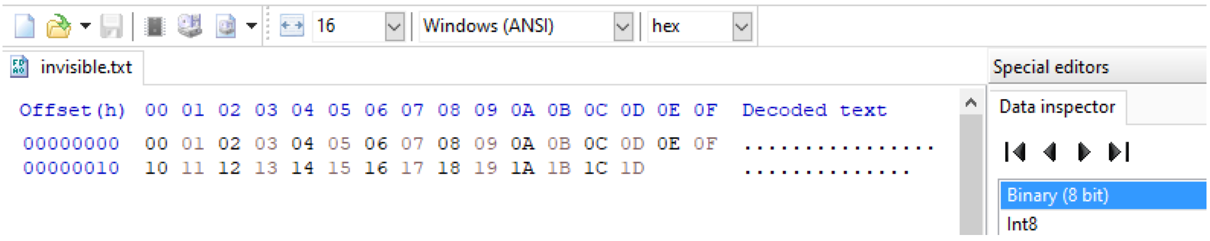
There are several ways to write a byte array to a file, the most widely used approaches are:

- Using Java NIO (fastest)
- Using FileOutputStream (older and more commonly used)

Open File Explorer in the root folder of the project. To do this, right click on the project, select Show in -> System Explorer.

Open the file `invisible.txt` in any standard editor, e.g. Notepad or Notepad++. Notice that the contents of the file are basically unreadable.

Now open the file using the HxD Hex editor. You will be able to view the contents of the file in pure binary, in byte units, shown in hex. The Data inspector on the right shows the contents of each byte in different numerical formats.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ................
00000010  10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D        ..............
```

Close the editor and press Enter to continue with program execution to read back the same file content into a byte array using the Java NIO approach.

Next, we encode the byte array to a Base64 string and save this to a file. Now we are able to view the contents of the file, which we can subsequently read back and decode successfully into a byte array again.

Finally, we encode the byte array to a Hex string and save this to a file. Again, we are able to view the contents of the file, which we can subsequently read back and decode successfully into a byte array again.

The majority of cryptographic data that we work with (private / public keys, certificates, etc) are binary data, encoding them either in Base64 or Hex strings before saving them to a file or displaying them as output provides an alternative for us to visually examine them for situations where manual study or comparison are useful.