RIT | Golisano College of Computing and Information Sciences
School of Information

Name: _____

ISTE-120
Lab 15: Binary IO

Exercise 1 – Introduction to Steganography[1] (3 points)

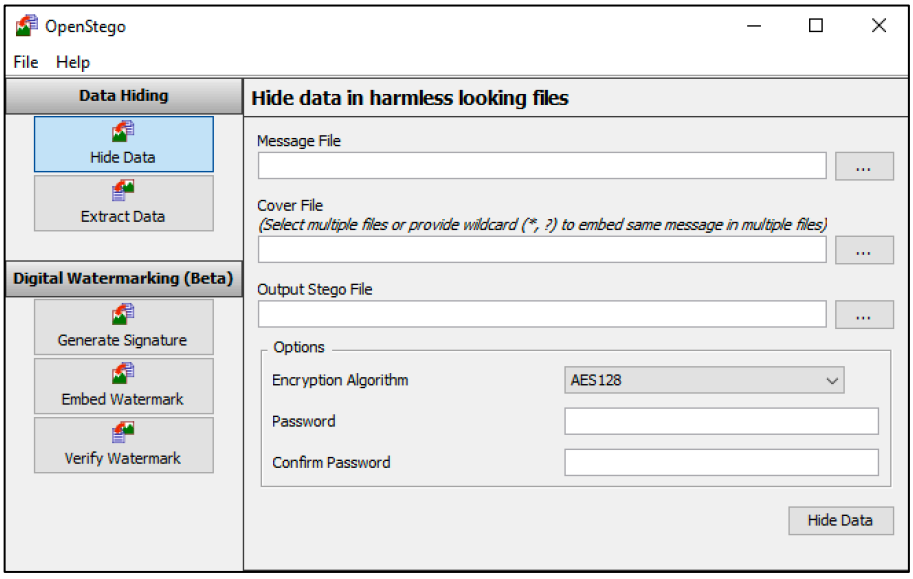
As defined in Wikipedia, steganography is the art or practice of concealing a message, image, or file within another message, image, or file. The word *steganography* combines the Ancient Greek words *steganos*, meaning "covered, concealed, or protected", and *graphein* meaning "writing." While the study of steganographic algorithms are beyond the scope of this course, an open source application, OpenStego is available that can embed one file within a second, graphics (png) image.

This lab introduces extraction of a hidden message from a graphics file as well as embedding a message within a graphics file. Graphics files, by their very nature, are binary files.

The folder, **Ex1**, in today's downloads contains three files, *campus.png*, *hidden.png* and *openstego.jar*. Open a terminal window and change to the directory **Ex1** and type the following console command. Note that double-clicking on the JAR file may also execute the file.

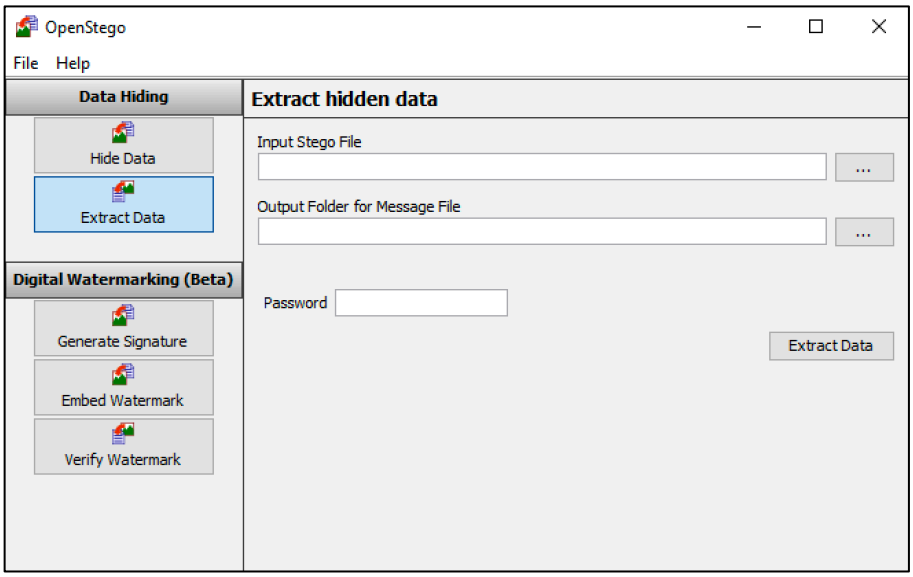
```
java -jar openstego.jar
```

A simple GUI for hiding and extracting data opens.



[1] **Differences: Stenography = writing in shorthand, whereas Steganography = concealing messages**

Selecting the “Extract Data” option gives the following:



Exercise 1 requires the use of OpenStego to extract a hidden message.

From the Windows Explorer, double-click both .png files to view the graphic images using the default graphics viewer and see if there is a visual difference. The file hidden.png contains an embedded message. The message is a familiar saying found over a doorway in the Engineering building here at RIT. To obtain it, use the following method to extract it:

Input Stego File:	hidden.png
Output Folder for Message File	**.** (“**.**” is the current working directory)
Password	RIT (case sensitive, of course)

What is the message?

Instructor / TA signoff for Ex1: _____

Extracting Embedded Data in a Bitmapped File

BMP image file layout overview

As mentioned above, steganographic algorithms are complex. That, combined with the structural differences between various graphics file formats requires knowledge beyond the scope of this course. Such is the case with Portable Network Graphics (PNG) files.

A simpler graphics format is the bitmap (BMP) file format. While complex in and of itself, it nonetheless can be simple if advanced features are ignored. The **BMP file format**, is known as a **bitmap image file** or **device independent bitmap (DIB) file format**. This is a raster graphics image file format used to store bitmap digital images, independently of the display device.

In the **Ex2** folder is the file *white.bmp*. It is an 86 KB, white rectangle that measures 200 x 144 pixels. The BMP file format is described in the diagram below. To the right is a numerical snapshot taken from a binary editor (0xED) while editing the file *white.bmp*. The left column is the byte number, expressed in decimal, while the contents are expressed as hexadecimal digit pairs, each pair representing one byte. The integer fields are formatted "little endian" and should be interpreted as the lowest byte of a particular field comes first while the highest byte is expressed last. Consider the file size field that immediately follows the ASCII "BM" signature (42 4D). That field as little endian appears as:

B6 51 01 00

When written as big endian, the hex value is 00 01 51 B6 and converts to 86454 decimal.

00000	42	4D	B6	51	01	00								BM.Q..
00006	00	00	00	00	36	00								....6.
00012	00	00	28	00	00	00								..(...
00018	C8	00	00	00	90	00								......
00024	00	00	01	00	18	00								......
00030	00	00	00	00	00	00								......
00036	00	00	00	00	00	00								......
00042	00	00	00	00	00	00								......
00048	00	00	00	00	00	00								......
00054	FF	FF	FF	FF	FF	FF								......
00060	FF	FF	FF	FF	FF	FF								......
00066	FF	FF	FF	FF	FF	FF								......

File structure of *white.bmp* in 0xED

Offset	byte 1	byte 2	byte 3	byte 4
0	42 (ASCII "B")	4D (ASCII "M")		
2	File Size	(00 01 51 B6 hex is 86,454 decimal)		
6	Reserved	(00 00 00 00 hex is 0 decimal)		
10	Offset to BOF	(00 00 00 36 hex is 54 decimal)		
14	Header Length	(00 00 00 28 hex is 40 decimal)		
18	Horizontal Width	(00 00 00 C8 hex is 200 decimal)		
22	Vertical Height	(00 00 00 90 hex is 144 decimal)		
26	# of Planes	(00 01 i.e. 1)	Bits / Pixel	(00 18 i.e. 24)
30	Compression Type	(00 00 00 00 i.e. 0 defined as no compression)		
34 38 42 46 50	Remaining 20 bytes (40 hex digits) are placeholders for Other Advanced Features			
54	Start of pixel data . .			

The file, *white.bmp*, is as simple as it gets. Every pixel is 24-bits in the RGB color space consisting of three 8-bits bytes one for each red, green and blue.

Embedded Messages

Understanding simple embedding and extraction of a message from a bmp file requires an understanding of the bmp file format (see previous section above) and the constraints that format places on the programmer to carry out the embedding process. The process of reading an existing bmp file and writing a new bmp file containing a message from a text file, involves reading sections of the input bmp file into an array, embedding a character by either:
(1) replacing two hexadecimal digits (1 byte or 1/3 of a pixel color) in the array with a character of the message and writing that array out to the output file, or (2) write all but the last byte from the read-in array, then write one of the message's character bytes to the output file.

The Java API DataInputStream and DataOutputStream classes are used for the reading and writing of binary data. Please review the `readByte()`, `readShort()`, `read(byte[] b, int start, int len)`, `writeByte()`, `writeShort()` and `write(byte[] b, int off, int len)`, methods respectively. Reading from and writing to specific locations of an existing file on disk is complex and beyond the scope of this course, but don't despair, this is covered in the next programming course.

Exercise 2 requires reading and extracting a message from a provided bmp file, and writing the message to the screen. The following describes the format of the bmp file with the hidden message:

1. Again, the first two bytes of the bmp file (using the `readByte()` method of `DataInputStream`) will be the characters 'B' and 'M'. If not, print an error message and terminate your program
2. Read the next four bytes as an int (`readInt()` method). This is the length of the image (the total number of bytes of pixel data that starts at byte 54 above)
3. Create a byte array to hold the header. This must be of size 54 – 2 – 4 = 48 (the 54-byte header, less two bytes for 'B' and 'M' (which we have already read in step 1) and less the 4 byte length which we have already read in step 2). Read this byte array in (`read(byte[] b, int off, int len) method`) just to skip to the 55th byte (the start of the pixel data, above)
4. Read the next two bytes as an int (use the `readShort()` method) - this is the number of characters in the hidden message. A short is an integer that is stored in only 2 bytes
5. Define a block factor. This is the number of bytes we will read at one time as we read the pixel factor. Our block factor will be 1000. **The idea is** that, from here on, we will read 'chunks' or 'blocks' of pixel data and only use the last byte of each block for part of our message. That is, the secret message is in the last byte of every 1000 in the pixel data
6. Check to be sure the number of blocks is greater than or equal to the number of characters in the message to be embedded. This means, checking that the size of the pixel data (see step 2 above), divided into 1000-byte blocks, has enough 1000-byte blocks to cover the length of the message (see step 4 above). If not, print an error message end terminate your program

7. Start with a String, message, that is initially the empty string ("")
8. Read 1000 characters. Take the last byte (index 999 of the byte array) as the next character in the message. Concatenate this byte to the message String.
9. Repeat the previous step until the message has been completely read from the file (that is, read one 1000 byte block for each byte in the length of the message - see step 4 above, again).
10. When the entire message has been read from the bmp file, write the message out to the screen

Exercise 2 – Extracting Embedded Data from Bitmapped Files (4 points)

Exercise 2 requires a Java program be developed to extract the hidden message from a bitmapped image file and display the message to the user. Also, write the message to an output text file.

After understanding the above file format for a file with a hidden message, **develop the algorithm** for reading the input file and extracting the embedded message. Write the program, and run it. For ease of use, get the input file name, output file name and block size values as three separate command line parameters.

Details:

- Input picture file with embedded message: *hidden1.bmp* (you can also try *hidden2.bmp*)
- Write message extracted to screen and to output file *out.txt*
- Use a block size of *1000* bytes

What was the message you extracted?

Instructor / TA signoff for Ex2: _____

Exercise 3 – Embedding Data into Bitmapped Files (3 points)

Design and develop a java program to embed messages in bmp files, as described in **Exercise 2**.

To save you time, consider making a copy of the extracting code from Exercise 2, change the class name to Embedding (Exercise 3), then Save As… for exercise 3. Modifications of the new code could save time rather than retyping a lot of similar code.

Change this code to accept four command line arguments:

- The name of the input .bmp file
- The name of a text file containing the message
- The name of the output .bmp file
- The blocksize

For this exercise, remember:

1. Check for the 'B' and 'M' at the start of the input file. Then, write them out to the output file (`writeByte()`)
2. Check that the size of the pixel data, in the 4-bytes (`readInt()`) following the 'B' and 'M' is at least *blocksize* x the number of characters in the message (one hidden character every *blocksize* bytes). Write this size out to the output file (`writeInt()`)
3. Read in the rest of the header (48 bytes) and write it out to the output file (`write(byte[] b, int off, int len)`)
4. Write the size of the message IN PLACE of the first two bytes of pixel data. That is, read in the first two bytes, ignore them, then write out the message size in their place (`writeShort()`)
5. Read in the pixel data in *blocksize*-byte blocks, REPLACE the LAST byte of each block with ONE character from the message, and then write the block out to the output file.
6. If all of the pixel data has not been read in, then you should read in and write out any remaining pixel data AFTER the message has been completely written out.

Details:

- Input picture file: *campus.bmp*
- Input message file: *in.txt*
- Output picture file: *campus_message.bmp*
- Block size: *1000*

Write the program, and run it. For ease of use, get the input file name, message file name, the output file name and block size values as four command line parameters.

Test it:

Once the code for exercise 3 successfully runs, verify the embedded message is correct, by running the code from exercise 2 code using *campus_message.bmp* as the input picture file. Also, try out different block sizes than 1000.

Instructor / TA signoff for Ex3: _____