

Department of Computer Science and Information Technology

La Trobe University

CSE1OOF/4OOF Semester 1, 2020

Assignment Part B

Due Date: Tuesday, 12th May, at 10.00 a.m.

First and Final date for SUBMISSION Tuesday 12th May at 10.00 am

Delays caused by computer downtime cannot be accepted as a valid reason for a late submission without penalty. Students must plan their work to allow for both scheduled and unscheduled downtime. There are no days late or extensions on this assignment as execution test marking will begin on the 13th

This is an **individual** Assignment. You are not permitted to work as a Pair Programming partnership or any other group when writing this assignment.

Copying, Plagiarism: Plagiarism is the submission of somebody else's work in a manner that gives the impression that the work is your own. The Department of Computer Science and Information Technology treats academic misconduct seriously. When it is detected, penalties are strictly imposed. Refer to the subject guide for further information and strategies you can use to avoid a charge of academic misconduct.

Submission Details: Full instructions on how to submit electronic copies of your source code files from your latcs8 account are given at the end. *If you have not been able to complete a program that compiles and executes containing all functionality, then you should submit a program that compiles and executes with as much functionality as you have completed. (You may comment out code that does not compile.)*

Note you must submit electronic copies of your source code files using the **submit** command on latcs8. Ensure you submit the required file. **For example**, the file **HexEditor.java** would be submitted with the command:

```
> submit OOF HexEditor.java
```

Do NOT use the LMS to submit your files, use latcs8 only

PLEASE NOTE: While you are free to develop the code for this progress check on any operating system, your solution **must** run on the latcs8 system.

Marking Scheme: This assignment is worth 10% of your final mark in this subject.

Implementation (Execution of code) 90%, explanation of code 10%

*You may be required to attend a viva voce (verbal) assessment
(to be used as a weighting factor on the final mark).*

Return of Mark sheets:

The face to face execution test marking in the lab (constitutes a return of the assignment mark, subject to the conditions on the last page of this document.

Please note carefully:

The submit server will close exactly 10:00 am on the due date.

After the submit server has closed, NO assignments can be accepted. Please make sure that you have submitted **all** your assignment files before the submit server closes.

There can be NO extensions or exceptions.

Your assignment will be marked in your normal lab just after the due date.

You **must** attend the lab you have signed up for on Allocate. Non-attendance at the lab you have signed up for on Allocate will also result in your assignment being awarded 0, except as detailed below.

If you cannot attend the lab you have signed up for on Allocate, please email p.karmakar@latrobe.edu.au to arrange another time.

Marking scheme:

90% for the code executing correctly

10%, you will be asked to explain (and / or alter) parts of your code.

Using code not taught in OOF - READ THIS

Please also note carefully that whilst we encourage innovation and exploring java beyond what has been presented in the subject to date, **above all, we encourage understanding.**

Code and techniques that are outside the material presented will not be examined, of course.

You are free to solve the Tasks below in any way, with one condition.

Any assignment that uses code that is outside what has been presented to this point **must be fully explained at the marking execution test.** Not being able to **fully** explain code outside what has been presented in the subject so far will **result in the assignment being awarded a mark of 0**, regardless of the correctness of the program.

Submitting an assignment with code outside what has been presented so far and not attending the marking execution test will result in an automatic mark of 0, regardless of the correctness of the submission.

How this assignment is marked

All assignments in OOF are marked, in the lab, in an execution test. This means that we mark running code. Your code must compile and display a result to the screen. Regrettably, we don't have the time or resources to look at code. The smallest amount of code that produces and displays a correct result will gain more marks than lots of code that doesn't compile, run or display something to the screen.

Working with files.

It is important for any good programmer to understand the contents of a file and how files work.

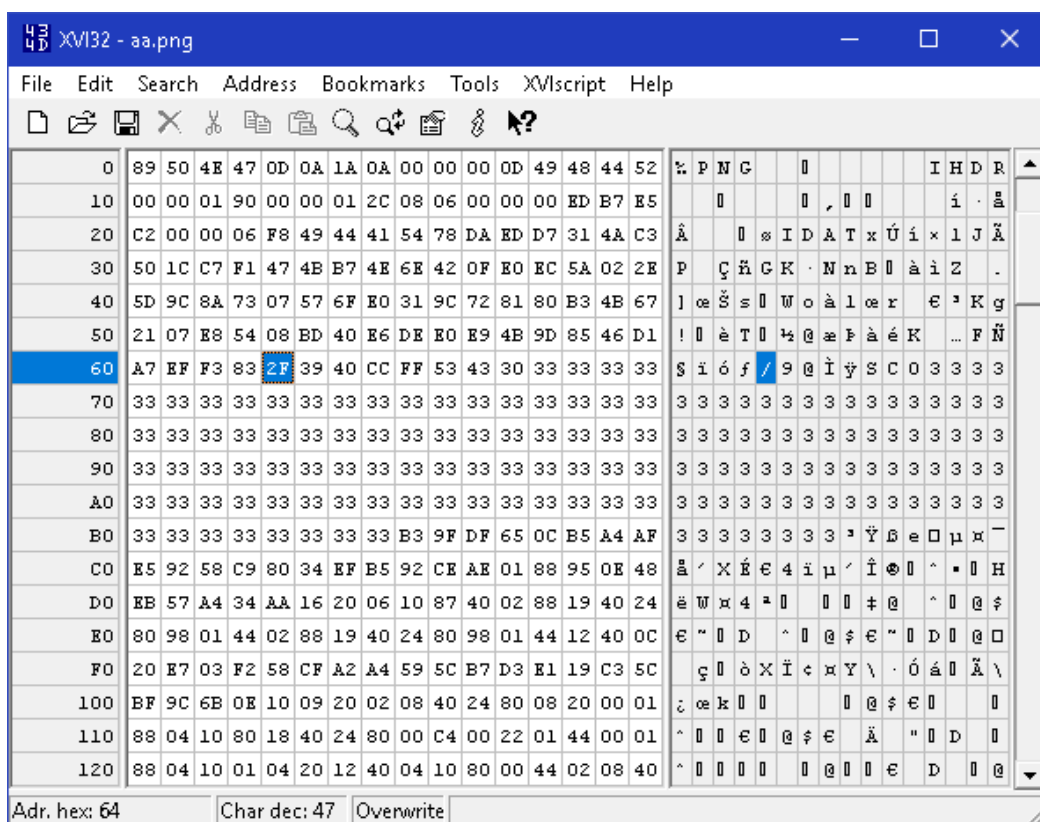
So far in OOF we have looked at text files, that is files that can be opened in Vim or Notepad.

However, most files you encounter are not text files, they are binary files. Think about things like PNG ZIP or MP3, none of these can open opened in Notepad or Vim.

The .class files that javac produces are an example of a binary file. Try opening them in Vim, are they nice to read?

Since you cannot view binary files using a text editor, we need a special tool to look at the internals of a file. This is known as a Hex Editor and is a valuable tool for any programmer.

Below is an example of a Hex Editor for windows.



Notice that it has two panels, the left side is the bytes of the file in hexadecimal. The right side is the same data but shown as ASCII characters. You may notice that many of the bytes are not valid ASCII characters and are replaced with placeholder characters or spaces.

The far-left panel is the address of the first byte of each row.

The highlighted row “60” indicates that the first byte of that row (value A7) has address 60.

Remember though that all these numbers are in hexadecimal, so “60” in hex is equal to “96” in decimal. Typically, programmers will prefix a number with “0x” to indicate that it is a hex number.

The highlighted cell (value “2F”) is at address 0x64, remember we count from zero.

Your Task

You need to implement a basic hex editor in Java that will run inside the PuTTY window.

A hex editor is a tool that you use throughout your programming career, you may find yourself using this tool long after you finish OOF. So, take the opportunity to design it well.

Your hex editor will be able to:

- 1) Open a file of any size and not crash.
- 2) Show both hex and ASCII representation of the data in a neat well formatted grid.
- 3) Be able to jump to a given address using a command.
- 4) Be able to get and set the value of a byte at a given address using a command.

When you run the program, you will provide it with the name of a file.

Then the program will show the first 256 bytes of the file, then prompt the user for a command.

The valid commands will be:

- 1) "exit" to exit the program.
- 2) "goto" to select a specific byte.
- 3) "jump" to select a byte relative to the currently selected byte.
- 4) "set" to set a byte
- 5) "hex" to switch to hex view.
- 6) "ascii" to switch to ASCII view (more on this later)
- 7) "truncate" to delete all bytes after the selected byte.
- 8) "next" to move to the next page of bytes
- 9) "previous" to move to the previous page of bytes.
- 10) "find" to find a string in the data.

Tip: only look at the first letter of the commands so that the user can use "e" for exit.

String.startsWith() may be handy for this.

The program should produce a message at the bottom of the screen saying what the last command did, for example if the user used "ascii" the program should say "Switched to ASCII Mode"

How do we even begin to do this?

First you need to have some understanding on how files work and how numbers are represented. I have provided some reading on the next few pages.

→ Please read the entire assignment before you start. ←

I have also provided some sample files in CSILIB. Use this command.

Note that there is a dot at the end of this command

```
> cp /home/student/csilib/cse100f/hexapp/*.dat .
```

Background: What is a signed & unsigned integer?

Computers work entirely in binary (1 and 0 only) so there is no room for storing the minus sign.

So how do we store negative numbers using only 1s and 0s?

The obvious way to do it would be to just use the left most bit to indicate a negative or positive number.

Which is what computers do, but with some extra steps, this is called **two's complement** encoding. Imagine if you wanted to encode the number -4 into an 8-bit binary number.

First write out 4 in binary using 8 bits like so	0000 0100
Then, flip all the bits	1111 1011
Then, plus 1	1111 1100 <- This is what a -4 byte looks like

This seems more complicated than it needed to be but there is a reason for it.

Consider what would happen if we added the number binary number 1111 1100 (negative 4) to the number 0000 1010 (positive 10)

	1111 1100	Binary of -4 (Two's complement)
+	0000 1010	Binary of 10
	<hr/>	
	10000 0110	Wait, that's 9 bits, remove the extra bit.
	0000 0110	This is the binary value for 6

With two's complement you don't ever need to do subtraction, just convert the number to negative and use addition. $-4 + 10 = 6$.

How do you know if a binary number is a two's complement encoded negative number or just a normal binary number? The answer is: you don't.

If you are given a binary value like 1111 1100 it could be -4 or it could be 252.

It depends on if you interpret the number as signed, or unsigned.

When working with signed numbers, just remember this:

If the left most bit is a 1, then the number is negative, if it's a zero, its positive.

In Java everything is signed by default (in the above example it would output -4)

Try running this code to see for yourself.

```
for(int i=0; i < 256; i++)
    System.out.printf(" Integer: %d, Byte: %d\n", i, (byte)i);
```

You will see two columns with identical numbers, until about halfway.

The divergence starts just after 127, why?

Consider what the number 128 looks like in a byte vs an int.

0000 0000 0000 0000 0000 0000 1000 0000	32 bit integer of 128
1000 0000	8 bit byte of 128

The value is identical, but ask yourself "what is the left most bit"

Remember: If the left most bit is a 1, then the number is negative.

Background: Random Access Files.

The Scanner class is useless for a hex editor application, because it only reads forwards. You need a way to jump to any random location in a file.

Read the API documentation for RandomAccessFile and File

<https://docs.oracle.com/javase/8/docs/api/java/io/RandomAccessFile.html>

<https://docs.oracle.com/javase/8/docs/api/java/io/File.html>

The methods in RandomAccessFile we are interested in are:

- int read()
- void write(int)
- int length()
- void seek(long)

When working with files it is important to know that internally there is a “file cursor” which is an integer that represents your current position within the file. Each time you call read() or write() the file cursor moves forwards.

Here is an example:

```
File myFile = new File("a.dat");
RandomAccessFile raf = new RandomAccessFile(myFile, "rw");
int byte0 = raf.read(); // read first byte and move cursor forwards.
int byte1 = raf.read(); // read second byte and move cursor forwards.
int byte2 = raf.read(); // read third byte and move cursor forwards.
raf.seek(0); // move cursor back to the start of the file.
```

The “rw” String indicates that we want to open the file for both reading and writing.

We can use seek() to move to any address within the file, even if the file is 10TiB in size.

We will only need to read small sections of the file at a time.

You may be wondering, if read() returns a byte, why is the return type an integer?

This is mainly done for convenience, using an integer allows us to avoid dealing with negative valued bytes and allows the API to return special values like -1 to indicate the end of the file has been reached. This method will always return a value between 0 and 255, or -1 indicate end of file.

Also notice that seek takes a long, consider for a moment that an int is only 32 bits long.

A 32 bit number ranges from -2147483648 to 2147483647. How many gigabytes is that?

Since you cannot have negative file size, the maximum file size we could support with 32 bits is 2147483647 bytes, or 2GiB.

You should use longs instead of ints when working with the file length to allow for large files.

Note that when you write an integer literal in Java it defaults to an Integer.

This is an issue if you are working with big numbers, so **if you need a long literal you put an L on the end** like so: `long a = 2147483647L;`

Background: What exactly does 2GiB mean?

What is a GiB?

It's a "Gibibyte" which is the technically correct way of referring to 1073741824 bytes.

The metric prefix "kilo" means 1000 since the metric system is based around powers of 10.

However, computers work in powers of 2, so a "kilobyte" was defined as 1024 bytes, not 1000.

This causes immense confusion; some software uses 1000 others use 1024 for measuring sizes.

So, to avoid confusion we will use "Kibibytes" which are defined in powers of 2 like so.

$$1\text{KiB} = 2^{10}$$

$$1\text{MiB} = 2^{20}$$

$$1\text{GiB} = 2^{30}$$

IEC Standard		
bit	bit	0 or 1
byte	B	8 bits
kibibit	Kibit	1024 bits
kilobit	kbit	1000 bits
kibibyte (binary)	KiB	1024 bytes
kilobyte (decimal)	kB	1000 bytes
megabit	Mbit	1000 kilobits
mebibyte (binary)	MiB	1024 kibibytes
megabyte (decimal)	MB	1000 kilobytes
gigabit	Gbit	1000 megabits
gibibyte (binary)	GiB	1024 mebibytes
gigabyte (decimal)	GB	1000 megabytes
terabit	Tbit	1000 gigabits
tebibyte (binary)	TiB	1024 gibibytes
terabyte (decimal)	TB	1000 gigabytes
petabit	Pbit	1000 terabits
pebibyte (binary)	PiB	1024 tebibytes
petabyte (decimal)	PB	1000 terabytes
exabit	Ebit	1000 petabits
exbibyte (binary)	EiB	1024 pebibytes
exabyte (decimal)	EB	1000 petabytes

Task 1

Create a class **HexEditor** and create a method

```
public static void repeat(String str, int count)
```

it will print out the string str count times. For example
`repeat("hello", 3)` -> "hellohellohello"
This will come in handy later.

Create a `main()` method and test your code.

Task 2

Create a static method **`String formatSize(long sizeInBytes)`**

This method needs to return the most appropriate representation for the provided file size (in bytes) to a human readable format with at least 2 decimal places (except for bytes).

Input	Output
2147483647	2.00 GiB
123	123 bytes
83647	81.69 KiB
9585631	9.14 MiB
188900977659375	171.80 TiB

Make sure you toughly test this method.

Task 3 – Create the user interface

Consider how the interface will look, for now let's assume that we will have 16 bytes per row with 16 rows on screen at a time, all in hexadecimal. Tip: instead of hard coding the value "16" everywhere, use a variable instead so that you can change it later.

Use some for loops and `System.out.printf()` to construct the following output, make sure to use the padding feature of `printf()` so that numbers like "1" get padded to " 1" the extra spaces will make it look neater later.

How do I convert an int/long to a hex string?

You could use `printf` like so: `System.out.printf("%X", 1234);`

Or you could use `String str = Long.toHexString(1234);`

Make sure to include a header row and column with a hex count.

```
207> java Hex
H | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
-----
0 |
1 |
2 |
3 |
4 |
5 |
6 |
7 |
8 |
9 |
a |
b |
c |
d |
e |
f |
```

Put this code into a method, name it something like "printHexTable".

It should take parameters, the `RandomAccessFile` object, and a "rownumber" variable.

The row number will indicate what the first row will be, so for example if I passed in a rownum of 7 the output should look like this

```
211> java Hex
H | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
-----
7 |
8 |
9 |
a |
b |
c |
d |
e |
f |
10 |
11 |
12 |
13 |
14 |
15 |
16 |
```

Task 4 – Using command line args.

Consider the following Java program

```
public class MyProgram
{
    public static void main(String[] args)
    {
        System.out.println("Number of args: "+args.length);
        System.out.println("First Arg: "+args[0]);
        System.out.println("Second Arg: "+args[1]);
    }
}
```

When you run this program from the command line you can pass it command line arguments like so

```
> java MyProgram hello world
Number of args: 2
First Arg: hello
Second Arg: world
```

As you can see you can access the command line arguments “hello” and “world” using the args variable.

Edit your program to accept a file name as a command line argument.

Do not ask the user for a file name using Scanner!

Make sure to check that the user has provided the correct number of arguments, if the user provides the wrong number of arguments you must show an error that says what the program does, and what the correct argument format is.

You must also check if the file exists before creating the `RandomAccessFile` you can do this using the `File.exists()` method. If the file does not exist, ask the user if they would like to create it, if they say yes, then use `File.createNewFile()` otherwise, exit.

Do not assume anything about the name of the file, it could be anything and contain any characters.

Do not hard code the name of the file.

Task 5 – Populating the table.

Once you have the table drawing nicely, we can think about populating it with data.

The first thing you need to do is `seek()` to the correct location. If we wanted to start from row 7, then we would need to seek to position $7 * 16 = 112$.

Once you have used `seek()` you can call `read()` and `printf()` the value inside the for loops, the bytes should be zero padded two characters ie “00” to “FF”

Note that `read()` returns -1 if you try to read past the end of the file, you should detect this and just print out some spaces.

At this point you will want to have some files to read so I have provided some use this command to copy them from CSILIB. **Note that there is a dot at the end of this command.**

```
> cp /home/student/csilib/cseloof/hexapp/*.dat .
```

Try viewing “a.dat” in your program. you should get something like this

```
232> java Hex a.dat
```

H	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	49	6E	73	74	61	6E	63	65	73	20	6F	66	20	74	68	69
10	73	20	63	6C	61	73	73	20	73	75	70	70	6F	72	74	20
20	62	6F	74	68	20	72	65	61	64	69	6E	67	20	61	6E	64
30	20	77	72	69	74	69	6E	67	20	74	6F	20	61	20	72	61
40	6E	64	6F	6D	20	61	63	63	65	73	73	20	66	69	6C	65
50	2E	20	0A	41	20	72	61	6E	64	6F	6D	20	61	63	63	65
60	73	73	20	66	69	6C	65	20	62	65	68	61	76	65	73	20
70	6C	69	6B	65	20	61	20	6C	61	72	67	65	20	61	72	72
80	61	79	20	6F	66	20	62	79	74	65	73	20	73	74	6F	72
90	65	64	20	69	6E	20	74	68	65	20	66	69	6C	65	20	73
A0	79	73	74	65	6D	2E	20	0A								
B0																
C0																
D0																
E0																
F0																

The file is less than 256 bytes long, which is why it stops part way.

You must also show the size of the file in both hex and decimal at the bottom of the screen.

Task 6 – Unicode Box Characters

In the example above I have just used basic ASCII characters to draw the borders of the table. If you look closely you can see that it's not very nice, there are gaps between the dashes and pipes. It would be nice if they were slightly longer, so it could form a continuous line.

Thankfully, there are specialised Unicode characters for this purpose.

https://en.wikipedia.org/wiki/Box-drawing_character

<https://www.unicode.org/charts/PDF/U2500.pdf>

Consider the difference between

----- vs —————

The left is just normal HYPHEN-MINUS (0x2D).

The right is BOX DRAWINGS LIGHT HORIZONTAL (0x2500).

You can get quite fancy with these, like so.

```
┌ Hello OOF ┐
└ test 123 ┘
```

Note that these only work in **monospaced** fonts, such as courier new which is what PuTTY uses.

Your task is to make the grid look nicer by using box characters of your choosing.

H	0	1	2	3	4	5	6	7	8	9	A
0											
10											
20											

You can just copy and paste the characters into the Java string literals from the wiki page.

Use right click to paste into PuTTY.

```
System.out.printf("||");
```

If you are an ASCII purist who objects to having any non-ASCII characters in their source code for fear of weird encoding problems, you may also use the hex escape sequence like so.

```
System.out.printf("\u2551");
```

Check out some websites that have Unicode information such as

<https://www.compart.com/en/unicode/block/U+2500>

<https://www.compart.com/en/unicode/block/U+25A0>

Task 7 – Interactive Design

When the program starts, display the hex data shown in Task 3 and then use `Scanner.nextLine()` to read input from the user's keyboard.

If the user enters any non-valid command, show a list of valid commands.

The program must continuously display the bytes and then ask for a command using a while loop. Only when the user enters the exit command should the program terminate.

The goto command must function as follows

"goto 1234 h" means go to byte at address hexadecimal 1234.

"goto 1234 d" means go to byte at address decimal 1234.

"goto -1234 d" means go to byte at address (file size -1234) decimal.

"goto -1234 h" means go to byte at address (file size -1234) hexadecimal.

If the user did not provide a "h" or "d" suffix, assume they meant hexadecimal.

The jump command allows moving the selection forwards or backwards relative to the currently selected byte. "jump 15" or "jump 15 h" means just forwards 0x15 bytes. "jump 15 d" means jump forwards 15 decimal bytes. Negative numbers move backwards.

You must calculate what the rownum needs to be so that you can pass it to the draw method.

The selected byte should appear near the middle of the screen, basically you should always be able to see the bytes before and after the selected byte. Make sure to adjust the row accordingly.

When using the next and previous page commands, make sure that the selected byte is always on screen. You should keep it in the same relative position on the screen.

Show the selected byte by putting brackets around it.

	0	1
50	00	00
60	44	43
70	73	6B
80	[F2]	0D
90	03	00

At the bottom of the screen show the following data:

- ➔ The address of the byte in hex, decimal, and formatted using `formatSize()`
- ➔ The value of the byte in hex (0 to FF), unsigned decimal (0 to 255) and signed decimal (-127 to 128)
- ➔ The size of the file in hex, decimal, and formatted using `formatSize()`

```
Byte address: 0x80 (128)
Byte value:   0xF2 (-14 signed) (242 unsigned)
File Size:    0x873819D1 (2268600785) 2.11GiB
```

You can omit `formatSize` if the size is less than 1024 bytes.

Make sure you support selecting bytes that are beyond the end of the file.

When the selected byte is changed, make sure to redraw the table.

Do not allow the user to select an address that is negative.

The program must say what the previous command did for example "jump 14" should say "jumped 0x14 bytes forward"

Task 8 - The set & truncate command.

The set command will accept either "set 12 d" for decimal, or "set 3A h" for hexadecimal and will write that byte to the currently selected address. The default is hex mode.

The user may also enter a single quoted ASCII character like "set 'a'"
Values that are over 255 (0xFF) are not allowed.

The set command needs to seek() to the correct location and use the write() method to set the byte. Note that write() saves the change immediately to the file. **You must also move the currently selected byte forward by 1.**

The user can select and set a byte beyond the end of the file, when you write a byte to such a location the OS will automatically backfill the intervening bytes with zeros.

The "truncate" command will just call the setLength() method in RandomAccessFile

Converting Strings to Integers

All the user input should be taken as a string since it may contain hex characters.

To convert a string to an integer use Integer.parseInt(String) like so:

```
String myStr = "123";  
int myInt = Integer.parseInt(myStr);
```

This also works with hex, but you need to remove the 0x part and add an extra argument.

```
String myStr = "1ABC";  
int myInt = Integer.parseInt(myStr, 16);
```

IMPORTANT

Your program needs to support large (>2GiB) files, which means the address may be bigger than the maximum value an integer can store. **You will need to use longs instead.**

```
String myStr = "1ABC";  
long myInt = Long.parseLong(myStr, 16);
```

Task 9 – ASCII Mode

When this option is selected the bytes of the file will be shown as characters rather than hex numbers. The program will stay in ASCII mode until the user changes back to hex mode.

However, care must be taken to not confuse PuTTY, if you just print out control characters like 0x0A (newline) or 0x09 (tab) you will cause the formatting of your grid to break.

https://en.wikipedia.org/wiki/Control_character

To deal with this, all control characters will not be printed and will instead be replaced by a symbol. The characters you use to replace them with are up to you, but they must be visually distinct from normal ASCII characters. And work in PuTTY. Things like line feeds could be shown as "LF" or "\n"

You must also show the space character (0x20) visually. So that it can be seen.

Use the Unicode 0x25CF • character since it is visually distinct from all other characters.

Most of the control characters are below 0x20, but don't forget about 0x7F.

Make sure to assign a character for 0x00.

You should move the logic for this into a separate method to keep the draw method short.

The file **b.dat** contains every possible ASCII value, use this file to check that none of the bytes cause PuTTY to misbehave.

Task 10 – Read Only Files.

You may encounter a file that you do not have write access to.

If you attempt to open such as file in "rw" mode, then your program will crash.

The File class has a method boolean **canWrite()**

Use this method to determine if the file is writable **before** you create the RandomAccessFile.

If it is, then open the file in "rw" mode, else open in "r" mode and prevent the user from using the set or truncate command.

The program should display "READ ONLY" for files that are read only.

Task 11 – Large file support

If your program is using longs instead of ints for seek() etc the you should have no problem with a file over 2GiB in size.

There is a file located at **/home/student/csilib/cse1oof/hexapp/bigfile.zip** which is more than 2GiB. Open it in your hex editor (do not copy it, it's too big)

```
> java HexEditor /home/student/csilib/cse1oof/hexapp/bigfile.zip
```

Once you have it open, try to goto position 0x80000000 (2GiB) see if your program handles it. Then try 0x86666666 (2.1GiB) and see if it still works.

Try going to a location like 0xFFFFF, you may notice that your string padding alignment on the left column didn't work so well.

This is caused by the fact you used something like printf("%4X") to do the alignment which fails once you get more than 4 digits.

Instead of hard coding a number such as "4", you should calculate how many digits you will need and use the repeat() method you crated in task 1 to output the correct number of spaces.

It should expand as required.

	0	1	2
0	[50]	4B	03
10	00	00	00
20	43	5F	56
30	82	5A	75
40	00	50	4B

	0	1	2
FFFFC0	C3	A1	A2
FFFFD0	51	57	F9
FFFFE0	80	82	1C
FFFFF0	4C	84	F9
1000000	9D	F8	90

H	0	1
FFFFC0	C3	
FFFFD0	51	
FFFFE0	80	
FFFFF0	4C	
1000000	9D	
1000010	3B	

Pro-tip: To calculate the number of spaces you need before you start printing the table. You could do some math magic with log() or you could be lazy and convert the last row header to a hex string and get the length of that string.

Task 12 – Search System

When working with large files it is helpful to be able to find interesting parts.

The “find” command will take an ASCII string and try to find it in the file.

This command will not support Unicode characters.

How to do this:

First when you take the users command, they could enter “find hello world”

You need to remove the command part, so it’s just “hello world”

Create a method for searching that takes the RandomAccess file, string to search, and the currently selected byte address. Also, the search should be case sensitive.

The search process should start from the byte after the currently selected byte, so use seek() to move here. From there use a loop to move through the file and check to see if the byte from the file matches the first character of the search (in our case “h”) if it matches, try to compare the second character “e” if that matches then try the third “l” and so on.

If you find all the characters matching, then move the selected byte to the first letter of the match. If not, leave the selected byte where it was and tell the user that the search failed.

Remember that read() returns -1 when it hits the end of the file. Show an error if you reach the end of the file and were unable to find the search string.

If the user enters the find command with no string, rerun the previous search again.

If there was no previous search, show an error.

||

Optional Bonus 1

It is annoying to need to type in long address like 0x80000000

Make your program support entering commands like “goto 1GiB”

It should support decimals, like “goto 2.4GiB” and support shorthand like “g 3.1m” for 3.1MiB

Optional Bonus 2 (hard)

Many binary files will need to store integer values which are bigger than 255.

In Java the integer 1234 would be stored in memory as 4 separate bytes (00, 00, 04, D2)

This encoding is called “Big Endian” because the most significant byte is first.

There is another style “Little Endian” which stores the bytes backwards (D2, 04, 00, 00)

Little endian is the “normal” way to do it, (homework: investigate why this is).

Your task is to add 2 commands to your program

- 1) “be X” where X is a number, switch to big endian of size x.
- 2) “le X” where X is a number, switch to little endian of size x.

The value for X can be between 1 and 8.

Once the user selects a mode the program must stay in that mode until the user changes the mode.

Byte mode is the default normal mode where you can select 1 byte at a time.

For modes with X greater than 1, you need to allow the user to select X bytes at a time.

For example, “goto 0x23” then “BE 2” should show this =>

H	0	1	2	3	4	5
0	49	6E	73	74	61	6E
10	73	20	63	6C	61	73
20	62	6F	74	[68	20]	72
30	20	77	72	69	74	69

At the bottom of the screen the big-endian value for a 2-byte integer should be shown in both hex and decimal. In this example the value of (68,20) in big endian is 0x6820 which is 26656 in decimal.

If the user were to enter the command “LE 2” then the value for (68,20) in little endian is 0x2068 or 8296 in decimal. And this would be shown at the bottom of the screen.

This must work for all values of X between 1 and 8.

Obviously selecting BE1 or LE1 is the same thing (normal single byte mode).

FO	[DD	3D	0C]	33	14	2E	5C	38	45	95	CE	4D	9B	7F
Integer Mode: 3 Byte Big Endian														
Byte address: 0xF0 (240)														
Byte value: 0xDD3DOC (-2278132 signed) (14499084 unsigned)														

The “set” command must also be able to support the new values. For example, if the program is in LE 2 mode then the maximum value that can be set is 0xFFFF or 65535.

FO	[DD	3D	0C]	33	14	2E	5C	38	45	95	CE	4D	9B
Integer Mode: 3 Byte Little Endian													
Byte address: 0xF0 (240)													
Byte value: 0xC3DDD (802269 signed) (802269 unsigned)													
File Size: 0x873819D1 (2268600785) 2.11GiB [READ ONLY]													

Sample Output

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	[49]	6E	73	74	61	6E	63	65	73	20	6F	66	20	74	68	69
10	73	20	63	6C	61	73	73	20	73	75	70	70	6F	72	74	20
20	62	6F	74	68	20	72	65	61	64	69	6E	67	20	61	6E	64
30	20	77	72	69	74	69	6E	67	20	74	6F	20	61	20	72	61
40	6E	64	6F	6D	20	61	63	63	65	73	73	20	66	69	6C	65
50	2E	20	0A	41	20	72	61	6E	64	6F	6D	20	61	63	63	65
60	73	73	20	66	69	6C	65	20	62	65	68	61	76	65	73	20
70	6C	69	6B	65	20	61	20	6C	61	72	67	65	20	61	72	72
80	61	79	20	6F	66	20	62	79	74	65	73	20	73	74	6F	72
90	65	64	20	69	6E	20	74	68	65	20	66	69	6C	65	20	73
A0	79	73	74	65	6D	2E	20	0A								
B0																
C0																
D0																
E0																
F0																

Byte address: 0x0 (0)
 Byte value: 0x49 (73 signed) (73 unsigned)
 File Size: 0xA8 (168) [READ ONLY]
 File Name: a.dat
 Command: █

Command: a

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	[I]	n	s	t	a	n	c	e	s	•	o	f	•	t	h	i
10	s	•	c	l	a	s	s	•	s	u	p	p	o	r	t	•
20	b	o	t	h	•	r	e	a	d	i	n	g	•	a	n	d
30	•	w	r	i	t	i	n	g	•	t	o	•	a	•	r	a
40	n	d	o	m	•	a	c	c	e	s	s	•	f	i	l	e
50	.	•	LF	A	•	r	a	n	d	o	m	•	a	c	c	e
60	s	s	•	f	i	l	e	•	b	e	h	a	v	e	s	•
70	l	i	k	e	•	a	•	l	a	r	g	e	•	a	r	r
80	a	y	•	o	f	•	b	y	t	e	s	•	s	t	o	r
90	e	d	•	i	n	•	t	h	e	•	f	i	l	e	•	s
A0	y	s	t	e	m	.	•	LF								
B0																
C0																
D0																
E0																
F0																

Byte address: 0x0 (0)
 Byte value: 0x49 (73 signed) (73 unsigned)
 File Size: 0xA8 (168) [READ ONLY]
 File Name: a.dat
 Swicthed to ASCII mode
 Command: █

```

Command: s ff

```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	FF	[6E]	73	74	61	6E	63	65	73	20	6F	66	20	74	68	69
10	73	20	63	6C	61	73	73	20	73	75	70	70	6F	72	74	20
20	62	6F	74	68	20	72	65	61	64	69	6E	67	20	61	6E	64
30	20	77	72	69	74	69	6E	67	20	74	6F	20	61	20	72	61
40	6E	64	6F	6D	20	61	63	63	65	73	73	20	66	69	6C	65
50	2E	20	0A	41	20	72	61	6E	64	6F	6D	20	61	63	63	65
60	73	73	20	66	69	6C	65	20	62	65	68	61	76	65	73	20
70	6C	69	6B	65	20	61	20	6C	61	72	67	65	20	61	72	72
80	61	79	20	6F	66	20	62	79	74	65	73	20	73	74	6F	72
90	65	64	20	69	6E	20	74	68	65	20	66	69	6C	65	20	73
A0	79	73	74	65	6D	2E	20	0A								
B0																
C0																
D0																
E0																
F0																

```

Byte address: 0x1 (1)
Byte value: 0x6E (110 signed) (110 unsigned)
File Size: 0xA8 (168)
File Name: a.dat

Set byte at address 0x0 to hex value 0xff
Command:

```

```

Command: g ff0a46d

```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
FF0A400	D6	91	6E	48	4C	73	37	45	B9	EE	7F	D5	FC	72	03	B0
FF0A410	9E	CO	57	5A	E4	FE	57	E5	BA	91	48	CB	35	EC	22	94
FF0A420	6B	DF	AO	74	84	2B	C1	AO	B8	17	C7	A8	D2	7D	96	C6
FF0A430	F3	BA	16	C1	6D	D6	4F	5D	F9	0B	B6	8F	49	78	88	E4
FF0A440	17	AE	48	EF	E3	19	72	89	32	5B	8E	DD	3F	03	D8	E4
FF0A450	2C	73	C2	06	24	65	6C	0F	30	6E	16	81	7D	77	51	47
FF0A460	08	15	B4	16	F3	F2	D5	3D	71	60	8A	57	B6	[F3]	FC	68
FF0A470	62	61	14	51	50	F4	5B	FC	56	57	57	A2	9F	8D	71	CE
FF0A480	21	0F	39	8E	8F	CO	7F	3E	BE	06	1D	1A	0B	D7	F6	B8
FF0A490	E2	07	8E	AC	38	04	82	C4	FA	A4	2B	2E	FA	84	76	A9
FF0A4A0	D8	84	38	BA	DB	A7	EA	C1	73	63	D6	6F	D9	FE	42	D4
FF0A4B0	C3	61	B6	66	CD	D6	01	CC	BB	9B	D3	07	C6	30	04	6C
FF0A4C0	E4	11	2A	7E	04	4C	B2	28	2E	76	8C	B6	B4	D7	5C	01
FF0A4D0	72	BA	6C	7C	D9	DD	D6	F5	C7	30	7F	AB	8F	74	7B	03
FF0A4E0	D8	99	75	A1	6B	1E	32	E2	2A	81	6A	BF	59	BA	5E	B8
FF0A4F0	5A	51	F6	8D	80	85	5B	67	74	D9	F6	47	3F	15	E2	D6

```

Byte address: 0xFF0A46D (267428973) 255.04MiB
Byte value: 0xF3 (-13 signed) (243 unsigned)
File Size: 0x873819D1 (2268600785) 2.11GiB [READ ONLY]
File Name: bigfile.zip

Moved to position: 0xff0a46d
Command:

```

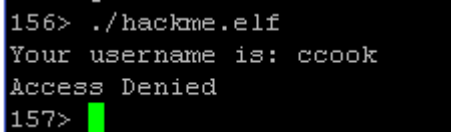
Task 13 - Using the tool

I have provided you with a compiled C program (not source code) named "hackme.elf"

```
> cp /home/student/csilib/cseloof/hexapp/hackme.elf .
```

This program checks to see if your username matches "hackerman1337" and if so, it allows you entry. If your username does not match, you get an access denied message.

Try running it yourself.



```
156> ./hackme.elf
Your username is: ccook
Access Denied
157>
```

You need to edit the program to gain access.

Since you don't have the source code you can't just recompile it.

The program must have the phrase "hackerman1337" in it somewhere.

Open the file in your hex editor and try to locate the string "hackerman1337"

If your search system is working this should be easy.

Once you have located it, change it to your username, then run the program.

Some background about C strings.

In the C programming language, strings don't have a length. Instead they mark the end of a string using a NUL byte (0x00) so when you edit the string make sure to put a 0x00 at the end.

Submit the modified elf file when you submit your assignment.

Beyond OOF

Consider some extra features that you could add to your hex editor such as:

- Search for a UTF16 & UTF8 strings.
- Open a file and append the contents of a second file to the end of the first.
- Remove bytes from the front or middle of the file.
- Move bytes from one part of the file to another.

Change the width to something other than 16

Electronic Submission of the Source Code

- Submit all the Java files that you have developed in the tasks above.
- The code must run under Unix on the latcs8 machine.
- You submit your file from your latcs8 account. Make sure you are in the same directory as the file you are submitting. Submit the file using the **submit** command.

```
submit OOF HexEditor.java
submit OOF hackme.elf
```

After submitting the files, you can run the following command that lists the files submitted from your account:

```
verify
```

You can submit the same filename as many times as you like before the assignment deadline; the previously submitted copy will be replaced by the latest one.

Please make sure that you have read page 2 about the submission close off date and time and the compulsory requirement to attend the execution test.

Failure to do both things will result in your assignment be awarded a mark of 0, regardless of the correctness of the program.

Execution test marks are provisional and subject to final plagiarism checks and checks on the compliance of your code to this assignment document.

As such final assignment marks may be lower or withdrawn completely.

Final notes

Don't let anyone look at your code and don't give anyone a copy of your code. The plagiarism checker will pick up that the assignments are the same and you will both get 0 (it doesn't matter if you can explain all your code).

And a final, final, final note, "eat the dragon in little bits". Do a little bit every night; before you know it you will be finished. The assignment is marked with running code, so you are better to have 2 or 3 parts completed that compile and run, rather than a whole lot of code that doesn't compile.

The execution test is done on latcs8 so please make sure that your code runs on latcs8 before you submit.