The Twilight Zone
Text Encodings, FileIO, and Exceptions

Melvyn Ian Drag

October 7, 2021

Abstract

Tonight we will learn how the ASCII, UTF-8 and UTF-16 text encodings work. Then we will see how they apply to Java File I/O Operations. Finally we will take a quick look at Exceptions in Java, as basic knowledge is required to work with File IO classes.

1 Welcome

Welcome to the Twilight Zone. For those who aren't aware, the *Twilight Zone* is a strange place, filled with shadows, where things aren't quite what they seem. There are monsters here that you didn't know existed that are waiting to cause you immeasurable discomfort and anxiety. You will doubt yourself here. You will doubt that you are able, you will at times forget that you can. We have to keep our heads up high and our minds clear in this place because there are simple rules governing everything - we just can't lose sight of them as we get tired and distressed.

2 ASCII

Many years ago, a short while after the beginning, there was an encoding called "ASCII". You see, the computers and the programmning languages in the beginning were designed in the USA and the UK mainly, and so all of the technology was dealt with in English. This is why the programming languages say "character" and "integer" and not whatever their equivalents are in other languages. At the same time, math and science publications were moving away from German, French and Russian and everything was being written in English, for ease of international exchange of ideas.

And so, as this was happening, it is not surprising that computers were designed to represent English language characters, punctuation, numbers, and a few other things needed by a computer.

hand out ASCII table

What I am handing to you is a table showing the binary encoding for text in the ASCII system. If your programming language supported ASCII, you would therefore write the character '0' as 0x30. You would represent 'a' as 0x97. The ASCII encoding could do other things too - if you wanted a blank line you could write '\n', stored in memory as 0x0A. If you wanted a tab you could write '\t', which was stored in memory as a 0x09.

The encoding even featured a way to make your computer make a beeping sound! You can try to print an '\a' on your computer ('\a' is stored in memory as 0x07) and depending on how your laptop/pc is configured it might still beep. If there is time after class we can try together.

2.1 ASCII Art

If you have heard the term ASCII before, there is a good chance that you have heard about it in the context of 'ASCII-art'. Pictures like this are still somewhat popular in certain text circles:

Note: In class just go to the url to show the class

```
       (\-"""-/)
        |     |
        \ ^ ^ /  .-.
         \_o_/  / /
        /`   `\/  |
       /       \  |
       \ (   ) /  |
      / \_) (_/ \ /
     |   (\-/)   |
     \  --^o^--  /
      \ '.___.' /
jgs  .'  \-=-/  '.
    /   /`   `\   \
   (//./       \.\\)
    `"`         `"`
```

`https://www.asciiart.eu/animals/marsupials`

2.2 ASCII is a Seven Bit Encoding

Show that the table goes up to 127. On the board, multiply 2x2x2x2x2x2x2 = 128 = $2\hat{7}$, with seven bits we get numbers 0-127.

3 Exercise

How many bytes are in a Character in Java? Instruct the class to write a code to get the number of bytes in a character. Use the *Character.SIZE* member to find out.

Answer: 2 bytes / 16 bits.

4 UTF16

I told you we were in the twilight zone - things are just starting to get weird. Java uses two bytes for a character, and yet I told you that the ASCII encoding uses only seven bits, which fits in one byte!

When ASCII came out, this was many years ago as I told you. But Java came out in the early 1990s *(Check me out on this, I might be off bya couple of years)*, and at this time personal computers were starting to spread around the world. The computers back then were primitive compared to what we have now **Show class a picture of the Commodore 64**, but the computers were being used by speakers of many languages. At this time it was becoming very important for people to write in their own native languages. Some people used Cyrillic letters, others used Sanskrit letters, Japanese and Chinese had many many characters, and so on. The ASCII encoding had one bit left over, so by turning on the remaining bit in the encoding there were now 128 new characters that could be represented by ASCII.

Write on board 8 bits, show how putting a 1 in the leading position allowed for 128 new values. For all the old 128 values, there was a new value corresponding to it if you turned the first bit on.

Do you think that all the alphabets in the world fit in the allocated 128 extra characters? *Wait for a resounding no.*

Different vendors tried different things. You could install different character sets on your computer, each characterset representing different letters, but this was cumbersome. Many different internationalization schemes appeared in the technological world. It's very interesting stuff and if you want to know more you can poke around the internet.

As I mentioned before, Java came out in the early 1990s, and it was about this time that people were clamoring for a standard internationalization scheme so the mess of competing character sets didn't continue to spiral out of control. What was decided was that all characters could be represented - instead of in 8 bits - in 16 bits! The new standard was dubbed unicode, and this implementation was called UTF-16.

Remember now, we are in the twilight zone, where there are gotchas and strange pits for you to fall into. Shortly after the release of Java, after the language designers adopted this 16 bit character for their new, modern language, the standards committee changed it's mind. The Unicode consortium continued on with UTF-16, but they realized that all the characters in the world could not be represented in 16 bits. So the amended the UTF-16 standard such that it would *usually* use 16 bits for characters, but othertimes it would use 32 bits as was necessary.

As such, UTF-16 is known as a variable-width encoding, whereas ASCII was a fixed, 7 bit wide encoding. Do not forget! UTF-16 can hold 2 byte characters, or 4 byte characters.

And so Java, already designed and released to the world, had a 16 bit character type, that was *usually* good enough for text, but sometimes things went haywire.

5 UTF-16 and Java

Do not forget! All characters in Java are 16 bits wide. Other programming languages will behave differently - forget everything you know! Java characters are 16 bits wide.

Now we will dip our toes into the mysterious waters of Text Encodings and Java. Run the 'StringLength.java' code, found in the class repository.

Have a look at the output. Note that there are two strings, each of length 5. One is written in Russian, the other in English. Nevertheless, Java says the length of each is 5.

Both strings have a strange "FE FF" in the beginning - ignore that for now. Java reports 12 bytes each, but it's actually 10 bytes plus these extra two weird bytes - we'll just ignore them for a minute.

As a class verify that the utf-16 bytes are correct for both Fyodor and hello.
Verify the Fyodor bytes using this table: `https://en.wikipedia.org/wiki/Cyrillic_script_in_Unicode#Basic_Cyrillic_alphabet`

5.1 UTF-16 is the bomb. But the BOM is not.

The BOM is the "FE FF" we see at the beginning of the byte arrays gotten from the strings. Luckily, Java characters are stored in Big Endian order. That means, that the first byte goes first, then the second byte. As we discussed last week, that is not universally the case. Some UTF-16 methods will come with a "FE FF" or "FF FE" pair of bytes at the beginning. **This is the twilight zone!!** Sometimes the Byte order mark / BOM is there, sometimes it is not. Tread carefully when you write code that manipulates utf16 text. Sometimes if you get data from a different source, the internet for example, the data will be in Little Endian order. This text will have the BOM/Byte order mark "0xFF, 0xFE". You may need to check this, depending on your use case.

5.2 Look at Unicode points U+XXXX syntax

Have a look here. `https://www.fileformat.info/info/unicode/char/0424/index.htm` Across the internet there are various places where you can find unicode data. Note that the cyrrillic f is U+0424.

5.3 Writing UTF16 values with the \uXXXX syntax

We've seen as we looked on the internet that Unicode code points are often refered to as U+XXXX. The XXXX are hex values. For UTF-16, when the code point fits in two bytes, we can write the code point as \uXXXX. (In Java, not in all programming languages).

See the code in 'UXXXX.java'.

Run the code. Note that you should see the "0" in the output. Depending on your machine, you may or may not see the \u0080 symbol.

Ask around and see if anyone got the symbol to print

5.4 Exercise

You have 5 minutes to complete this task.

If you tap out you can look at IAmACheater.java

One of my favorite Portuguese words is:

recordao

Note I am missing the font to write this. Must write it on the board and sort out how to include the font in this docunment.

Write a Java string for this word using only unicode escape sequences. The unicode escape sequences for the english letters can be found using your ascii table. e.g. r = \u0052. Print the string to the console.

Your code should look like:

```
String s = "\u0052\u0065...";
System.out.println(s);
```

Hints for the Portuguese letters:

1. https://www.fileformat.info/info/unicode/char/00e7/index.htm
2. https://www.compart.com/en/unicode/U+00E3

Show the answer before moving on...

5.5 xxd

In this lecture we will make use of the “xxd” command line tool. This tool should be present on Linux and Mac computers. It is also present in the **git bash** shell if you are on windows.

xxd gives us a hex dump of a text file. If you quickly run “xxd filename” you will see a bunch of hex bytes dumped out in your terminal. You may, in your leisure, look for another tool that you like better. I love xxd, and so do many other people. It is a classic Linux tool and I have never needed a substitute.

Back to lecture. . .

5.6 Something Weird Before the Break

Now, are you ready to see something weird?

show ModifiedXXXX.java

Run the ModifiedUXXX code like this:

```
java ModifiedUXXXX | xxd
```

Look at the output. On my machine I see C280 and not 0080 as I’ve told you to expect. Does anyone have anything other than C280?

Remember - xxd is showing us the hexadecimal output of our program. We asked our program to print out 0x0080 - but it printed out 0xc280. Remember, I told you text encodings are dangerous things. Startups have failed, programmers have falied to deliver results, all because of what I’ve just shown you. Many people do not understand what I’ve just shown you.

6 Final Jeopardy & Break Time

- What are all the primitive "whole number types" in Java?
- Are Java integers signer or unsigned?
- Are Java "whole number" primitives stored in Big Endian or Little Endian Format?
- Which bit is always the sign bit in "whole number types" in Java?
- What is the BOM in UTF-16 text?
- Write "hello" in UTF-16LE, and prepend an appropriate BOM.

7 One More Thing About UTF-16 in Java

Remember when I told you that UTF-16 could be stored in 2 bytes or 4 bytes? Poor Java. . . the developers tried so hard to make a language that was modern and supported the latest internationalization standards . . . then the internationalization committee screwed them over and changed utf-16 from a fixed 16 bit encoding to a variable 16 or maybe 32 bit encoding.

No matter, the Java folks solved that problem. The two byte UTF-16 characters come in what is called a Surrogate pair. You take a two-byte prefix and a two byte suffix in line with the UTF-16 specification, and this forms a character. Neither the prefix nor the suffix forms a valid code point on its own - neither one represents a symbol on its own - but together, they represent a symbol.

Java UTF-16 strings can be created in several ways when the value requires 4 bytes.

See 'PolishedExamples/Surrogate.java' to see how to create a string using Surrogate pairs in UXXXX format or using an integer value from the U+XXXXXX specification. Notice that the treble clef in the example has 5 character in the U+XXXXX, whereas the others we've seen have only 4. That's because it needs more than 2 bytes. Notice how you can create it with an integer from the U+XXXXX number however.

After that example

Look at 'PrintEmoji.java' This is either in PolishedExamples or Scratchpad, can't rememebr which.

8 Checkpoint

Remember that Java internally stores text data in chars, Characters and Strings in a UTF-16 encoding. Not ASCII. And not any of the other million encodings that exist. Always UTF-16. And always big-endian.

9 UTF-8

9.1 Preamble

Unfortunately again for Java (and for Windows as well), UTF-16 is not a widely used or loved text encoding. If you write a program that ingests data from the internet or desktop applications, the text is probably encoded in what is called "UTF-8". As I mentioned before, when Java (and Windows) came about in the early 1990s, the Unicode consortium was working dilligently to choose the best encoding for international text. And silly things like emojis, and mathematical symbols, etc..

Shortly after they settled on UTF16 being a 16 bit encoding, they decided it should be variable 16/32. Then they decicded it should actually be 8/16/24/32 bits. This new encoding, that ranged from 1 - 4 bytes, was called "UTF-8".

Before I tell you how UTF-8 works, I'll just tell you that my laptop (and yours probably) has a UTF-8 encoded terminal. Whatever text you see in my terminal is internally encoded as UTF-8. Just as Java used UTF-16. So, returning to that strange thing I showed you before we wnet on break, where I tried to print 0x0080 to the console, but my terminal said I had printing 0xC280 - Well lets look at the Unicode page: `https://www.fileformat.info/info/unicode/char/0080/index.htm`

you will see that in UTF-16 it is encoded as 0x00 0x80, but in UTF-8 it is encoded as 0xC2 0x80.

9.2 How UTF-8 works

What I am going to give you now is a cool exerpt from a little hacker journal that's quite popular. I've given you three articles, one which is relevant to today's lecture, one is relevant to your life, and one is relevant to next week's lecture.

Handout POC —— GTFO pamphlet

Take a minute to skim through this now and see if anything in there makes sense to you (allow 2 minutes). Now, soon you'll be able to read article one. Maybe not tonigh, but definitely after doing the homework you'll be able to read article one and make some sense of what they're talking about. Article 2 is a great read for people who like doing crafts and projects. Article 3 is likely to go over your head, but you can skim it to set the stage for next week's lecture.

Anyway let's return to lecture. UTF-8 is one of the coolest encoding schemes in the world. It is a variable width encoding, but it has some nice properties that make it easy to parse as a computer.

UTF-8 is great because it is compatible with ASCII! All one-byte UTF-8 characters are valid ASCII characters. So now you understand about half of how UTF-8 works. If you look on Page XXXX of the pamphlet I just gave you you'll find a table explaining how UTF-8 works. Note that the one byte charactrs are just like ASCII. Remember that ASCII is a seven bit encoding with a zero in the first position. I've given you an ASCII table - if you have time later you can choose a few of those hex values and convert them to binary and you'll see they all start with a leading zero.

write on board the one byte UTF-8 boiler plate *Note that we have 7 bits (the xs) to represent our code point. Any code point that needs to convey only 7 bits of info will fit there*

Now, how does UTF8 work for larger UTF code points?

Note the positions of the xs, ys and zs in the table I gave you for UTF8, the table in the pamphlet.

UTF-8 is a prefix encoding.

The rules:

1. If the code point is one byte long, start the byte with a zero.
2. If the code point is two bytes long, start the first byte with 2 ones, then a zero, the your data. The second byte must start with '10'
3. If the code point is three bytes long, start the first byte wit 3 ones, then a zero, then your data. The second and third bytes start with '10'
4. If the code point is four bytes long, start the first byte with 4 ones, then a zero, then your data. The second, third and fourth bytes start with '10'

The rules (part 2):

1. Get the hex value for the Unicode code point you want to encode in UTF-8 (That's the U+XXXX(y?) value.) For the treble clef, the code point was **U+1D11E**. Note we are using the value associated with the big U.
2. Convert the hex after the U+ to binary
3. See how many binary digits it requires
4. See if you will need 1, 2, 3, or 4 UTF-8 code points to accomodate that many digits.
5. Starting from the left, fill the blank spaces in the UTF-8 boiler plat with the binary digits from the code point you just wrote in binary.
6. If there are any left over places, fill with zeros.
7. Done. Convert to hex if you want.

Do the treble clef on the board

```
0x1D11E = 0b11101000100011110 (17 bits)
One byte UTF-8 gives 7 free bits for data
Two bytes give 11 free bits
Three bytes give 16 free bits
So we must use 4 bytes, which gives 21 free bits.
Now we just fill in:
1111XXXX 10XXXXXX 10XXXXXX 10XXXXXX
```

```
11110000 100XXXXX 10XXXXXX 10XXXXXX // pad with zeros, we only
    need 17 bits
11110000 10011101 10000100 10011110 // Fill in the 17 bits
    from the U+1D11E

11110000:10011101:10000100:10011110 // VERIFY AGAINST
    https://www.fileformat.info/info/unicode/char/1d11e/index.htm
```

9.3 Exercise

There is a language spoken in Southeast Asia calle *Tamil*. One of the letters of the alphabet is the one given here: `https://www.fileformat.info/info/unicode/char/0b8a/index.htm`

The unicode code point is U+0B8A. Convert this code point to UTF-8 and verify your answer with the answer online.

Now you more or less understand UTF-8. You might need to look at it a bit longer. That article I've given you shows some problems with UTF-8 in how it's implemeted in various languages. If you know a couple of programming languages and are bored you could read the article, find flaws in your chosen languages and write to the author of the article. You'd get some cool points.

The authors said they'll buy a beer for anyone who finds an error that they didn't document in the article.

10 Break and Pop Quiz

1. How far apart (in decimal) are a and A?
2. How far apart (in decimal) are b and B?
3. How far apart (in decimal) are z and Z?
4. How is the number 0 stored in binary in Java on your computer?
5. How is the Character 0 stored in binary in the ASCII encoding?
6. How is the number 1 stored in binary in Java on your computer?
7. How is the Character 1 stored in binary in the ASCII encoding?

11 FileIO

Now welcome back to the twilight zone.

So now we know a few ways that text can be encoded by a computer. When I first came to Java I felt really worried about how to read and write files because there were so many options and I wasn't sure about the way Java encoded text and what means Java had for writing/reading files in various encodings.

In general I felt like I was drowning in all of the different options for reading and writing files in Java. I looked on google ' how to read a file in Jiava' and the answers were terrifying. The same of writing. And the most frustrating part is that every tutorial seemed to use a different means for writing a file. Recently I came across this stackoverflow post that sort of echoed my feelings:

`https://stackoverflow.com/questions/13959766/best-file-i-o-option-in-java`

There are many ways to handle file i/o in Java. Indeed for the homework I assigned before where you had to write a file I said to use the FileWriter class, and yet it seemed that everyone in class who wrote a file used a different class that wasn't *FileWriter*. This made me think either

1. you knew way more Java than me
2. you only knew one way to write a file that you learned in another class, and you used that.

A rule of thumb I found in that post for handling file I/O on that page:

- Class contains 'Stream' = Handles bytes
- Class contains 'Reader/Writer' = Handles text. Usually using a default encoding.

In general, we prefer streams that allow us to specify the encoding we are reading/writing. FileReader/FileWriter uses the machine default encoding. My machin is UTF-8 encoded. But what if the data file I am reading is UTF-16 encoded? Then the FileReader/FileWriter won't work.

11.1 Beware the FileReader/FileWriter

These two classes read and write based on the machine's default encoding. These two classes read a text file, character by character.

Run FileReaderExample with hello-utf8.txt Run FileReaderExample with hello-utf16.txt Run FileReaderExample with dostoyevsky-utf8.txt Run FileReaderExample with dostoyevsky-utf16.txt

Note in the above examples that the file reader correctly reads 5 characters from the utf files, but reads 10 characters from the utf16 files.

There is an additional example in the code about FileWriter which is maybe subtly wrong. I don't have time ot go into it. If you have alot of time to waste you can run it and see what it does.

11.2 InputStreamReader and BufferedReader

Note as we run the following examples we are using the same code and the same text files. Yet, depending on how we set things up we either get junk or the correct output.

java InputStreamReaderExample dostoyevsky-utf8.txt UTF-8 java InputStreamReaderExample dostoyevsky-utf16.txt UTF-8 java InputStreamReaderExample dostoyevsky-utf16.txt UTF-16 java InputStreamReaderExample dostoyevsky-utf8.txt UTF-16

If you wrote the code wrong and worked as a statistician, you might in one case think the person who wrote the text you are analyzing was Japanese, or Russian. One of these conclusions is correct, one is wrong. It's up to you as a programmer to write the code correctly.

11.3 OutputStreamWriter

Here is one way to control the output format of your bytes

java OutputStreamWriterExample UTF-8 then **cat outputUTF-8.txt — xxd** You will see utf8 bytes for "Hello"

java OutputStreamWriterExample UTF-16 then **cat outputUTF-16.txt — xxd** You will see utf16 bytes for "Hello", along with the BOM.

There are many other ways to do this. I've spent hours experimenting with different file IO classes in Java. If I've given you useful example, then great! If you are still confused after these few examples I've given, then you'll have to run the code I gave you, experiment with it, and think until it becomes clear to you. Remember - I told you in the beginning that playing with text encodings will make you feel powerless and afraid. It will make you doubt yourself. There is a good chance you'll feel scared and like you aren't good enough. I've explained the rules of this game to you - now you need to go on and apply them so you don't get lost in this mysterious place.

12 Exceptions

Sorry I haven't had time to work on the Exceptions part of the lecture too much. I worked on it before but I'm not sure if we'll have time in this lecture to cover it in great depth. I'll give a whirlwind tour of exceptions here just so you know what all these Exceptions are in the code I've been showing you today.

13 Conclusions

We worked hard, and achieved very little.