BibleReader 1.0

Joel Dever

Final Project – CSCI 297

12/14/09

Introduction

In this document I will try to cover the relevant topics to understand the composition of this program. I will not go into great technical detail. If that is desired consult the commented source code. The overall function of this program is a Bible reader for one's mobile phone. The text of the Bible is of the King James Version.

Start-Up

The MIDlet is launched in Main.java. One of the first thing that occurs is there is a check on the total heap memory that exists. If this value is not 8MB or more it will alert the user of this fact and exit. Otherwise the program moves towards checking for the existence of a Bible file. This occurs in the FileAccess class. The program checks to see if this file exists on the local file system if it does then the program continues to parse the file. If it does not exist the program will prompt the user to navigate through the local file system and choose a directory to download the file to. Once that is done the file will be downloaded off the internet.

Once the file is certainly on the phone the program will conduct some basic parsing. This is done by creating hash table with 66 keys, one for each book of the bible. The value will be an instance of my class Book. Book will store its own hash table where the key is a chapter number and the value a byte offset for that chapter. At the completion of parsing the file the hash table will be full and 66 instances of Book will be in this hash table.

After initial indexing is complete the program will launch an instance of ReaderCanvas, which will control the display of the text for the user. Initially a welcome message is displayed.

At this point the program has finished loading. The whole process takes only a few seconds even if the file is not already on the system. The file is only 1.2 MB, which allows for a quick download. I also added a splash screen that alerts the user of what is going on while it's loading. It also gives the user instructions on selecting a directory to download to.

Persistence

Persistence is a nice feature that I thought would improve usability. I implemented one feature and that is storing the file name. The program will store this value and will know where to load the file from each time the user starts the program. If the feature didn't exist the user would have to enter somehow where this file existed or the file would have a static location. I think my implementation is most user friendly of the options. The filename is used many times throughout the program and I felt it would be inefficient to constantly be loading this string to and from the record store. As a solution I load the file name once from the record store then store it as a class variable. This variable is used to load the file for the future uses.

The Persistence class is used to handle the record store access. By this I mean I have provided constructors and methods that will get and set byte arrays which the record store uses. This class provides extensibility for future work on the application.

The Download

I was able to find a parseable bible file that made this whole program possible. Initially when I began work this file was available as a web accessible text file. In November or thereabouts I noticed by downloads were failing through my application. I found this was because the web site hosting my application no longer offered the txt file in that format. It was still available but now in compressed form using gzip compression. This dropped the file size to roughly 30%, which was great for my program. This minimizes the most time intensive and unpredictable part of the program which is the download over the internet.

The file however now had to be decompressed before I could use it. I found a library that offered a GZIP implementation that was suitable for J2ME. This was found at www.java4ever.com and quickly integrated to my program. The download and decompression take only a few seconds.

Another that this brought about however was the need for about 6MB of free heap space. The file needed to be downloaded into a single byte array which needed to be approximately 1.3 MB. After compression it was to be about 4.3 MB. At the point directly after decompression I would need 6MB of heap to operate. There is no way to download and decompress only part of the file. The gzip algorithm requires the entire file to be present in order to decompress. Immediately after decompression the file is written to disk at which the application can free the heap. From this point on plenty of heap will be available for the user.

Important Data Structures

I mentioned briefly during the Start-Up section that the entire Bible file is parsed and a hash table is set up that holds 66 entries where the key is the book name and the value an instance of a class Book. This hash table is efficient because it stores strings as its key.

The value of the Bible hash table is an instance of the class Book. The Book class has its own hash table called index. This hash table has an integer as a key and a value. The key will hold the chapter number. If a particular book has 30 chapters the hash table will have 30 keys. The value will by the byte offset in the file where that chapter is located. When the user wants to jump to a random chapter in the Bible this hash table will be used to find the exact byte that the chapter begins on, which is very efficient.

There also exists a class called Chapter which is only used when the user selects a chapter to jump to. It has two data structures worth mentioning. It will have a vector called splitVerses, which stores the text of the verses in a readable manner. For example if the screen only can fit 25 characters and the verse is 115 characters the verse will need to be split across multiple lines. My program will parse all the verses in a given chapter and put up to a 25 character string in each element of the vector. It will do this in a way that no word in the verse is ever cut off and no character is omitted. It will store the location of the last space and display up to that space. If a word is cutoff it will be displayed in its entirety on the next line. This vector however leaves a gap in program function. If a 150 element vector contains 35 verses how do I know on which index verse 14 starts? This is solved by using yet another hash table. This hash table, which is also a member variable of Chapter, will have a key of verse and a value of the index in the vector where the verse starts.

I used a hash table yet again for mapping the abbreviated versions of the book names, which are found in the text file, to the full names e.g. "2Th" →"2 Thessalonians". Again a hash table is an efficient solution because we are using random access of strings.

I ran into another issue where I would need to know what the next book would be. Consider an example where a user is on the last chapter of the book of Job and wants to move to the next chapter. I had no mechanism to understanding what that book would be. I solved this by making a vector of books called orderedBooks which is 66 elements in size and contains a string for each book name. This is used to set the next and previous books for each instance of Book.

Text Display

One of the biggest design issues I faced was how to display the text. I examined the option of using a form, but this would bring about issues of how to store the text that wasn't on the screen among others. I considered a canvas as well. I knew it would be more complicated but would give me more control. The paint feature allowed me to only display what I needed. It gave me full control over scrolling and I always knew what was on the screen.

One of the nice features of this phone is that it will work with phones of varying screen size. The screen size properties are queried at start-up and the text is parsed accordingly.

When the user selects a chapter to display it will be parsed and then immediately painted to the screen. After the paint the program will then parse the previous and next chapters in a separate thread. This makes the program more efficient. When the user asks to view the next or previous chapters they are immediately ready and the user didn't have to wait to view the current chapter before parsing the next and previous ones. Every time the next or previous chapters are fetched it is done in a separate thread so that the user won't have to wait.

Problems/Issues

The most challenging issue I ran into involved calls failing due to thread blocks. I spent the better part of a weekend dealing with this. The calls that were failing were IO calls using FileConnection. I was totally in the dark to what was causing the problem. The internet had little help for me. I followed the only piece of advice I found online and made a function call, which was embedded in a command action method, run in a separate thread. This solved that problem. I later ran into the same problem again later. This time I was making method calls from within the keyPressed function. These were also made from a separate thread. This however threads introduce a whole new set of challenging problems with regard to shared resources, which was solved through some deep thinking.

One minor issue that caused me some headache was dealing with capitalization. The bible file and my abbreviations hash table have the books with a starting capital letter e.g. Genesis. Later into development I thought it would be a nice feature to allow the user to enter the book names to be non-case sensitive. I solved this problem by forcing most of my uses to the book name to lower case using String.toLowerCase(). However when it is displayed I wanted the

book to appear at the top of the screen capitalized. I kept the member variable bookName in the Book class the capitalized version, otherwise used the lower case version.

Another very annoying problem I ran into was during parsing of the file. I was finding my program not parsing past about a dozen chapters into Genesis. I was perplexed for a couple hours. I later looked at the file I was reading from and realized it was corrupt around that point. It is possible the download was unstable or the decompression was unstable. I also found that in one area I was not closing my stream handles. One of these issues led to the problem and I did not run into this issue again.

A common issue that was prevalent was bounds checking. An example of this would be trying to access the previous book while the current book is Genesis. This applied for verses, chapters, and books. A lot of checks needed to be made so I didn't break out of bounds on arrays. This applied to user input as well as using the directional keys on the screen. I checked the edge cases to ensure that this would not be an outstanding issue.

Conclusion

I am pleased with the results of this project. My main initial concerns were with speed and size. The size issue was solved through compression and intelligent use of the file system. I store in memory what I need but no more. The speed issue also was overcome. I download and process the entire Bible in a few short seconds. During run time the program also responds very quickly. The user never feels like he is waiting while the program processes data.

The program is easy to use and works very well. It enabled me to build on and fine tune the J2Me development skills I learned in this class.