**Development Philosophy and Explanation**

The application, GreekBibleStudent, has been developed using a significant number of classes. The basic reasons for the choice of these classes has been:

1. To avoid overloading the main class, and
2. To attempt to group methods (procedures) on a logical basis.

(I’m not entirely convinced that I’ve always succeeded in both of these aims – especially where features have been developed late in the project lifetime.)

1. **Where do we start?**

The main form has been renamed **frmMain**. This means that processing starts in **frmMain.cs**.

You will note that **frmMain** begins with a number of class definitions that are global to the file. We will deal with these throughout this document.

1. **classGlobal, classGlobal.cs and globalVars**

All truly global variables have been put in **classGlobal**, instantiated as **globalVars**. This is essentially the main communication mechanism between classes. Most of these variables are defined as part of the **frmMain** creation or the initialisation of various other classes.

Note that quite a lot of key variables are defined here. In particular, the Registry key and the base location of project files

1. **classRegistry, classRegistry.cs and appRegistry**

The Registry is used to store:

1. file locations;
2. the size and position of the main form so that, when the user restarts the application, it will be the same size and position as the previous session;
3. the position of the main “splitter” – the selectable divider that separates left and right sides of the main form;
4. colours, fonts and font sizes of all the editable fields;
5. a few details relating to notes and reference history.
6. **GreekProcessing, GreekProcessing.cs and greekUtilities**

You may note that this is actually the first area of processing.

This class provides methods supporting the processing of the Greek text. All text is implemented as Unicode (two-byte) characters. Note that the base Greek alphabet (and a few additional characters) have hex values between 0x0386 and 0x03ce. (You can find a table of these at <https://www.unicode.org/charts/PDF/U0370.pdf>.) More complex (combined) Greek characters (such as vowels with rough or smooth breathings, vowels with various accents, combinations of the two and so on) have hex values between 0x1f00 and 0x1ffc (see the Unicode consortium table at <https://www.unicode.org/charts/PDF/U1F00.pdf>.)

A large part of this class is the storage of these tables with codes that allow transition from one to another. So, for example, if the user has previously typed an *alpha* and now types a *rough breathing*, the simple *alpha* character will be replaced by an *alpha* with a *rough breathing*. It also allows for transitions in the other direction: if a specific character has an accent, for example, the program can identify its non-accented form.

1. **classText, classText.cs and mainText**
   1. *storeAllText*

This is important enough to warrant more detailed explanation. Once the Registry details have been initialised and the Greek lists have been constructed, this method will be called. It loads all text-related data into memory.

This method will perform the following steps:

1. Get a list of New Testament (NT) book names (in presentation sequence) and the name of each file in which text data is stored;
2. Steps through each file in sequence to retrieve and store the data;
3. Repeats step i) for the Septuagint (LXX)
4. Repeats step ii) for LXX
5. Populates the comboBoxes on the main form.
6. ***Memory Structure***

The information from step i) is stored in the Sorted Dictionary, **listOfNTBooks**. This means that there is a NT book code (or bookId) by which details can be retrieved. It is effectively a zero-base array but stored in a dictionary.

The data in steps ii) and iv) are stored in a hierarchy of classes:

|  |  |
| --- | --- |
| **Class** | **Content** |
| classBookContent | chapterInBook  chapterLookup  chapterSequence |
| classChapterContent | versesInChapter  verseLookup  verseSequence |
| classVerseContent | wordIndex |
| classWordContent |  |

In each case:

* xxxInyyy = a list of instances of classChapterContent in the given book, classVerseContent in the given chapter, and so on – all keyed by a simple sequence;
* xxxLookup = a list of actual chapter or verse numbers in the same sequence as the xxInyyy list;
* xxxSequence = a means of finding the sequence given the actual chapter or verse number.

Since words don’t have a recognised reference, these are only referred to by sequence. Each classWordContent instance contains specific information about the word. Since it is the bottom of the hierarchy, there is no list to be kept.

1. ***Chapter Sequence and Chapter Number***

The chapter sequence is simply an integer value (starting at zero) that allows us to uniquely identify an instance of the classChapterContent. It is generated in the sequence in which the chapter details are encountered in the source data. The chapter number, on the other hand, is a string that is picked up from the data.

The purpose of this distinction is that it caters from non-contiguous chapters. If a book jumps from chapter 9 to chapter 11, the program will cope with it. (This is unlikely in chapters but, given that the verse numbering is treated in the same way, some verses *are* omitted.) This approach was actually forced on us by the text of LXX which sometimes has non-numeric information in chapters and verses and where chapters are occasionally presented in a chaotic order.

1. ***Data retained about words***

Grammatical information about each word is provided in the source data in two, encoded fields. These are stored in the variables **catString** and **ParseString** and are decoded elsewhere.

The following information is stored in relation to the actual word is stored in three formats:

|  |  |
| --- | --- |
| **Variable Name** | **Significance/use** |
| textWord | The word as it appears in the text |
| accentlessTextWord | textWord with accents removed (but not breathings) |
| bareTextWord | textWord with all decorations removed – i.e. only alphabetic characters |
| rootWord | The defined root of the textWord (which may be very different) |

Additional variables are:

|  |  |
| --- | --- |
| **Variable Name** | **Significance/use** |
| uniqueValue | A calculated value that uniquely identifies the grammatical function of the word |
| punctuation | Any punctuation that follows the TextWord |
| preWordChars | Any text that precedes textWord |
| postWordChars | Any text that follows textWord |

These variations on the source text make the various functions of the application less time consuming: the work is done once, at initialisation.

* 1. *Mouse clicks on the NT and LXX text*

This is not, strictly speaking, part of classText but is an important element of processing text.

Whenever a mouse click occurs in either text area (**rtxtMainText** or **rtxtLxxMainText**), the event, **textAreaMouseDown** is called. This calculates the text position at which the click occurred (a sort of character count) and stores it in the global variable, **latestMousePosition**. (Note that this is used for *both* NT and LXX and only stores what is literally the last position clicked on.) It also works out hat verse has been clicked on and updates the combo box, **cbVerse** or **cbLxxVerse**, as suitable. (The role of the verse combo boxes will be discussed later.)

* 1. *Selecting a new Book*

If a new book is selected from the pull-down list, **cbBook**, the event, **cbBook\_SelectedIndexChanged**, is called. This, in turn, calls the method, **handleChangeOfNTBook**. This will reset the chapter combo box to the first chapter in the new book and then call **displayNTChapter**, which contains the core code in displaying the text for that book and chapter. It will, in turn, reset the verse combo box to the first verse of the chapter.

If the current text is the LXX, (i.e. it is **cbLxxBook** that is changed), then the same logic occurs but for LXX equivalents.

* 1. *Selecting a new Chapter*

If the chapter changes (by selecting **cbChapter** or **cbLxxChapter**), similar processing occurs but the book is not changed.

Note that, for §§5.3 and 5.4, something similar occurs if the chapters are incrementally increased or decreased or if an item of history is selected.

For all other processes, I suggest you wourk progressively through the controls of the main form and review the events from all actions on them. I have tried to explain the logic within code comments. Feel free to contact me, if there are inconsistencies or the logic isn’t clear.