Dmitri Stanchevici
Unit 9
Module 1

Ex. 1.1

Part 2

Java files can be found in the following directories:

- apps
- tests

Part 3

Following are the subdirectories inside org, with Java files in them and with the number of lines in each Java file:

- org/gateway
    - Main.java -- 28
- org/gateway/browser
    - ElementTypes.java -- 7
    - GSBrowser.java -- 618
    - GSDoc.java -- 308
    - GSDocElement.java -- 41
    - GSWord.java -- 29
- org/gateway/dbase
    - DB.java -- 66
    - DBServer.java -- 344
    - Table.java -- 153
- org/gateway/server
    - GSServer.java -- 246
    - ServerApp.java -- 20
- org/gateway/util
    - FirstClassLoader.java -- 111
    - Log.java -- 49
    - MyRecord.java -- 169
    - NameValue.java -- 24
    - ParseEngine.java -- 265
    - ParseResult.java -- 16

To find out the subdirectories and the Java file, I explored the org directory in Finder. The find out the numbers of lines in the Java files, I opened each of them in Emacs, which I had configured to display line numbers.

Ex. 1.2

Except for the presence of README and LICENSE text files, I don't see any similarities between the directories of Jetty and our GS application. Jetty's main directories are

- bin
- etc
- lib
- modules

while the GS main directories are

- apps
- databases
- gsmlpages
- logs
- org
- properties
- tests

Inside Jetty's main directories, I did not find any directories or files that would allow me to indicate a similarities between Jetty and GS.

Ex. 1.3

In test3.gsml, I see the following tag that has not been described in the instructions:

<vspace height=10>

I assume this tag inserts vertical space.

Ex. 1.5

When the "Go" button is clicked, goURL() is called:

METHOD	ACTION
`void goURL ()`	`jumpToLink (urlField.getText().trim())` is called.

`void jumpToLink (String url)`	Because we are working with a file, the condition
	
	`if (url.startsWith("file://")) {`
	`  // It's a local file.`
	`  currentHost = null;`
	`  doc = readLocalFile`

	`(url.substring(7,url.length()));`
	`}`
	
	is fulfilled.
	
	The url's substring starting with index 7 (that is after `"file://")` is sent to readLocalFile.

`static GSDoc readLocalFile (String filename)`	Having checked for the existence of the file, this method uses a LineNumberReader to read the lines of this file into an ArrayList of Strings. This ArrayList is sent to the constructor of the GSDoc class in which these lines are assigned to the instance variable containing these raw lines in an ArrayList.
Back in `void jumpToLink (String url)`	The GSDoc is returned from readLocalFile and assigned to **doc,** a GSDoc instance variable of GSBrowser. After a check to ensure that doc != null, doc.parse() is called inside the GSDoc class.
Inside class GSDoc `public boolean parse()`	Methods inside the GSDoc class are called to parse the raw lines and store the result as various GSDocElements.
Back in `void jumpToLink (String url)`	The urlField.setText (fullURL) ensures that the field will show the full URL. Then this.repaint () is called.
`public void paintComponent (Graphics g)`	The elements from **doc** are rendered to the panel with a Graphics tool. First, the **doc.titleString** is drawn, and then a for-each loop goes over the remaining GSDocElements inside **doc** to render them depending on their types.

Ex. 1.6

The variables from GSWord are used in the following methods.

In `drawText()`, the coordinates and dimensions of those gwords are saved that may serve as a hyperlink if clicked on. To determine if a word or words form a hyperlink the gword.linkURL variable (from GSWord) is checked.

Correspondingly, in `mouseClicked()`, clicking on these words (or rather **clicking on the area specified by their GSWord variables**) will follow the link that is saved in the linkURL variable in GSWord.

Ex. 1.8

Objective: Drawing from two files, *actors.table* and *movies.table*, print all movies in which a given actor has played.

1. Create an array list **actorRecords** of all records in *actors.table,* each element of the array being a hashmap for (1) movieID and (2) actor information.
2. Create an array list **movieIDsTitles** of all records in *movies.table,* each element of the array being a hashmap for (1) movieID and (2) title.
3. Save the given actor's name in String **actorName**.
4. Create an array list **actorMovieIDs.**
5. Traverse **actorRecords,** to find all **movieIDs** associated with **actorName.** Store these **IDs** in **actorMovieIDs**.
6. Traverse **actorMovieIDs,** retrieving from **movieIDsTitles** the title associated with each ID. Print this title.

Ex. 1.11

Running DBJoinTiming with a JOIN command takes **1930ms**.

When the query is changed to "FETCH movies:movies," the running time is **54ms**.

The JOIN command obviously requires more time than FETCH does because JOIN involves two tables rather than one and requires more calculation, checking, and manipulation (including the creation of a third--joined--table) than FETCH.

Ex. 1.12

It seems that the naive version is quadratic $O(n^2)$ because of the nested for-each loops at the end of join() in Table:

```
for (MyRecord r: rows) {
  for (MyRecord r2: t.rows) {
     MyRecord r3 = r.join (r2);
     if (r3 != null) {
        result.rows.add (r3);
     }
  }
}
```

The revised version also has a pair of nested loops, but the internal loop traverses a hashmap that contains only the rows from one table with the value based on which the two tables are joined. So, overall we are dealing with a liner algorithm here: O(n).

Ex. 1.14

“GS” stands for “Greatly Simplified.”