**DB assignment**

**Task 1: Introduction**

This workbook will lead you through the first assessed exercise, the aim of which is to build a database server (from the ground up !). This will not only give you more practice writing complex Java programmes, but will also provide you with hands-on experience of using query languages. Note that this assignment WILL contribute to your unit mark (with a weighting of 40%).

This assignment will be marked on the lab machines, so it is essential that you check that you can compile and run your code using Maven on these machines before submission.

In order to identify the use of "derived" code (i.e. material "found" online or generated by AI), we will make use of code similarity checking tools. Any code that has been determined to have been derived will be discounted during the marking process (resulting in a reduced final mark). Remember that you will only receive credit for code that YOU yourself have written.

You are encouraged to talk about the assignments and discuss possible high-level implementation approaches with other students. HOWEVER it is **essential** that you protect your own work (and don't share code with other students). With this in mind, never exchange code with other students (via IM/email, GIT, forums, printouts, photos or any other means). If you ask a question on the unit discussion forum, try to keep discussion at a high level (i.e. not pasting in chunks of your code). If it is unavoidable to include code, only share small fragments of the essential sections you need assistance with. In addition, avoid the use of pair programming on this unit - you must produce all your own work ! Students submitting very similar work will be treated as collusion and will be dealt with using the university academic integrity processes.

**Task 2: Assignment Overview**

In this assignment, you will build a relational database server from scratch. This server must receive incoming requests (conforming to a standard query language) and then interrogate and manipulate a set of stored records. Your server will maintain persistent data as a number of files on your filesystem. You will not be required to implement a client application - this will be provided for you (to allow you to connect to your server and check that is it working correctly).

As usual, you have been provided with a [Maven project](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/cw-db) to help get you started with the assignment. As before, there is a [template test script](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/cw-db/src/test/java/edu/uob/ExampleDBTests.java) for you to use - make sure you add suitable test cases to this script to ensure that your application is fully and systematically tested.

It is **essential** that your server is *robust* - you should detect and trap errors effectively and ensure that the server continues running at all times. Just imagine a world in which servers had to be manually restarted every time something unexpected was encountered. It's going to be very difficult for your server to pass the marking tests if it has crashed !

Note that your main class MUST be called DBServer and MUST include the following constructor method and input handling method:

* public DBServer()
* public String handleCommand(String)

If you change the name of the class or the signature of either of the above methods, we won't be able to run your code ! We will be using automated marking scripts and if you server does not conform to the above, we won't be able to test it ! Your submission will be assessed on the success with which it implements the described query language, as well as the flexibility, error handling and robustness with which your server operates. Remember that we also care about "Code Quality" - so be sure to adhere to the coding standards and conventions covered in the lectures.

**Task 3: Persistent Storage**

[[一張含有 螢幕擷取畫面, 圓形, 標誌, 圖形 的圖片

AI 產生的內容可能不正確。](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/03%20Persistent%20Storage/slides/segment-1.pdf)](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/03%20Persistent%20Storage/slides/segment-1.pdf)[一張含有 圓形, 螢幕擷取畫面, 鮮豔, 行 的圖片

AI 產生的內容可能不正確。](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/03%20Persistent%20Storage/video/segment-1.mp4)

Any database system must be able to persistently store data (otherwise it will lose everything each time it is restarted). In this assignment, you must use the file system for this purpose. Your database will consist of a number of tables (aka 'entities'), each containing a collection of rows that store 'records' - see the table shown later in this section for an example.

Each table should be stored in a separate file using tab separated text. A [sample data file](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/people.tab) has been provided to illustrate this file format. Note that the constructor method of the DBServer class in the template project initialises a storageFolderPath variable to indicate which folder should be used to store the data files. This variable has been set to a folder called databases - it is essential that you store ALL of your data inside this directory (and nowhere else outside of it).

The first (0th) column in each table must contain a unique numerical identifier or 'primary key' (which should always be called id). The id value of each row will NOT be provided by the user, but rather they should be automatically generated by the server. It is up to you how you do this, however you should ensure that each id is unique (within the table where is resides).

Both Database names and Table names should be case insensitive (since some file systems have problems differentiating between upper and lower case filenames). Any database/table names provided by the user should be converted into lowercase before saving out to the filesystem. You should treat column names as case insensitive for querying, but you should *preserve* the case when storing them (i.e. do NOT convert them to lower case). This is so that the user can define attribute names using CamelCase if they wish (which is a useful aid to readability).

[一張含有 文字, 字型, 螢幕擷取畫面, 數字 的圖片

AI 產生的內容可能不正確。](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/03%20Persistent%20Storage/images/table.jpg)

**Hints & Tips:**  
As a useful starting point, your first task is to write a method that reads in the data from the [sample data file](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/people.tab) using the Java [File IO API](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/package-summary.html). View the slides and video at the start of this section for an overview of these packages. At this stage you need only print out the content of these records to the terminal (in a later task you will store this data in a suitable in-memory data structure).

When working with file paths, it is **essential** that you do not use any platform-specific file separators. Some platforms use \ and some platforms use / for separating folder names in a file path. Java code should work on ALL platforms - for this reason, you should make use of the [File.separator](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/File.html#separator) constant (which will contain the relevant character for the platform that the code is currently running on).

Feel free to extend the file storage mechanism described above if you need to implement additional features. Your server will only ever need to read in files that your server has created. During testing, all databases and tables will be created by sending queries to your server - we will NOT be copying any existing data files of our own into the databases folder. Note that if you encounter a tab file with invalid formatting when reading in data from the filesystem, your file parsing method should throw an IOException. You should however ensure that this exception is subsequently caught by another part of your server - remember: don't let your server crash !

**Task 4: Maintaining Relationships**

Relationships between records in different tables should be recorded using 'foreign keys'. For example, the PurchaserID in the table illustrated below is a reference to the id of a person from the example table in the previous task. This additional data file is [provided for you](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/sheds.tab) in order to aid you in developing a solution. Note that only single element keys are in use (i.e. you do not need to cope with 'composite' keys).

It is important that the id of a record must NOT change at any time during the operation of the system (once a record has been assigned an id this must stay fixed for as long as that record is kept in the database). Additionally, you should not "recycle" the IDs in any table (i.e. reusing the IDs for new rows after old rows have been deleted). This is because the IDs might be used as foreign keys elsewhere in the database and there is the risk that unintentional relationships are created.

For simplicity, no primary or foreign key marker keywords are provided in the query language - the server relies upon the user remembering which attributes are keys. This might seem like a problematic limitation in the query language, however we have to draw the line somewhere (otherwise the assignment would become too complex to implement in the given timeframe).

Note that it is *not* your responsibility to normalise the database - this is a job for the developers who have designed the database schema and who make use of your database server. If you don't know what normalisation is, then don't worry - you don't need to know for this assignment anyway !

[一張含有 文字, 字型, 螢幕擷取畫面, 數字 的圖片

AI 產生的內容可能不正確。](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/04%20Maintaining%20Relationships/images/sheds.jpg)

**Task 5: Java Data Structures**

Once the data has been read in from the filesystem, you will need to store it in memory. You will need to devise a suitable set of classes to represent this data inside your Java program. Think carefully about the tabular nature of relational databases and then write a set of classes that match this structure. You will need to consider a wide range of different elements of the database, including: tables, rows, columns, keys, table names, column names, data values, ids and relationships between entities.

Remember that this teaching block focuses on "development" (not just "coding") and as such we are attempting to develop your analysis and design skills. This exercise is more than just implementing a pre-defined specification - it requires you to understand the domain, be able to deconstruct the problem and make informed design decisions to achieve a successful solution. As such, it is not easy - you are likely to make mistakes and will need to refactor your code at different stages over the next few weeks.

Once you have defined a collection of classes that you feel are appropriate to the problem, use the file reading methods that you wrote in the previous section to populate instances of your classes with data read in from the sample data files. In order to fully exercise your classes as you develop your server further, you will need to write additional data files to augment those given to you. We advise creating these data files using queries in your testing scripts, rather than manually writing them. This is because you are likely to need to recreate them on a regular basis during the course of your development work. Additionally, you should not write any test scripts that assume the presence of pre-existing tab files, since there is no guarantee that these will be present in the databases folder !

Once you have successfully stored the imported data in your classes, the next step is to write a method to save these structures *back out* to the filesystem again (using appropriate features of the Java [File IO API](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/package-summary.html)).

**Hints & Tips:**  
In order to check that your code is successfully reading and writing data from the files, you should alter the data *whilst it is in memory* (i.e. *after* you have read it in but *before* you write it back out again). You could for example increment the age of all people in the table by one year each time the data is loaded. By changing the data in this way, you can check to make sure that the stored files are actually being re-written with the updated information !

**Task 6: Communication**

It is not the aim of this assignment to address the topic of network or socket programming. For this reason, the networking aspects of the server have been provided for you in [the template server class](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/cw-db/src/main/java/edu/uob/DBServer.java). This server can be run from the command line using mvnw exec:java@server. The database server listens on port 8888 in order to receive incoming messages. These incoming commands are then passed to the handleCommand method for processing. Your task is to add to the handleCommand method to respond to the commands.

It is up to you how you format the printing of the table content. You should however attempt to make the response as human-reader-friendly as possible. You will see from the sample test script provided in the template project that the tests have been written using simple word matching. This has been done because output formatting will vary from implementation to implementation and the test cases need to work with all students' code. You should use the same approach when writing your own tests.

It is essential that your response is returned by the handleCommand method and NOT just printed out in the local terminal/console. When we test your server during the marking process, we will be monitoring what is returned via the network. You won't get any marks for just doing println messages in the terminal !

To help you ensure that your server conforms to the correct protocol, a [command-line client](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/cw-db/src/main/java/edu/uob/DBClient.java) has been provided for you. This client can be run from the command line using mvnw exec:java@client. You should not change any of the code in the client - any features that you implement in this class will not be executed during the marking process. The client has been provided purely to allow you to manually check that your server is operating correctly. During the marking process, your server will be marked entirely by automated test scripts.

For the sake of simplicity, you may assume only a single client connects to the server at any one time (i.e. there is no need to handle parallel queries or deal with issues of contention).

**Task 7: Query Language**

Although we have described the communication mechanism that should be used, we still need something to actually transmit ! Clients will communicate with the server using a simplified version of the SQL database query language. Your task is to write a handler for the incoming messages which will: parse the incoming command, perform the specified query, update the data stored in the database and return an appropriate result to the client. Be sure to save any changed data back to the file system - you don't want to risk losing any in-memory updates (in case your server crashes, or is kill off).

The query language we will use for this assignment supports the following queries:

* USE: switches the database against which the following queries will be run
* CREATE: constructs a new database or table (depending on the provided parameters)
* INSERT: adds a new record (row) to an existing table
* SELECT: searches for records that match the given condition
* UPDATE: changes the existing data contained within the rows of a table
* ALTER: changes the structure (columns) of an existing table
* DELETE: removes records that match the given condition from an existing table
* DROP: removes a specified table from a database, or removes the entire database
* JOIN: performs an **inner** join on two tables (returning all permutations of all matching records)

A grammar that fully defines the simplified query language is provided in [this BNF document](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/BNF.txt). You will note that BNF grammar contains two distinct types of rule:

* Symbols with angle brackets <name> denote rules which MAY contain arbitrary additional whitespace
* Symbols with square brackets [name] indicate rules that can NOT contain additional whitespace

As a consequence of these rules, your server should be able to correctly parse incoming commands irrespective of the number of *additional* whitespace characters between certain tokens. So for example:

SELECT \* FROM people WHERE Name == 'Steve' ;

is valid and acceptable, being equivalent to:

SELECT \* FROM people WHERE Name=='Steve';

Dealing the such additional whitespace might seem like a daunting and challenging task. Don't panic, we will provide you some help and advice for dealing with such queries.

**Hints & Tips:**  
Note that you should NOT use any existing parsers or parser generators (Yacc, Lex, Antlr etc.) The aim of this assignment is to implement the command parsing using your own code.

To help further illustrate the use of the simplified query language, we have provided [a sample transcript](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/example-transcript.docx) which shows a series of example queries and the response we expect to see the the server. In this assignment, we will not be providing you with an extensive set of tests cases for you to check the effectiveness of your code. Learning to write your own comprehensive set of test cases is a key part of becoming a 'developer'. The aim of the query transcript and the BNF grammar is to provide you with as much help as possible in achieving this objective.

**Task 8: Query Specifics**

When you start to dig deeper into the complexities of the assignment, you will no doubt have a variety of questions of specific aspects of system functionality. This section attempts to provide more detail regarding the intended operation of the database server.

***SQL keywords***

Convention has it that query examples are typically shown with uppercase keywords (to differentiate them from identifiers and literals). However, SQL keywords are in fact case insensitive, so select \* from people; is equivalent to SELECT \* FROM people;. This is true for all keywords in the BNF (including TRUE/FALSE, AND/OR, LIKE etc.). In addition to this, all SQL keywords are reserved words, therefore you should not allow them to be used as database/table/attribute names. The server should return an error if the user attempts to misuse a reserved keyword (see the following task for details of the error handling mechanism to be used).

***Comparisons***

It is not necessary to implement a datatype system within the database - you can just store everything as simple text strings. You should perform >, >=, <, <= comparisons wherever it is possible to do something sensible (e.g. with floats, ints, strings etc). In situations where no appropriate comparison is possible (e.g. firstName > 12) don't try to trap the query and return an error, just return no data in the results table. The LIKE comparator is intended just for use with strings and provides a simple case sensitive substring matcher (NOT the full SQL LIKE operator with % wildcarding). If the user attempts to perform a LIKE query on non-string data (e.g. ints, floats, booleans) your server should not return an error, but just perform the query as though the data were a string (even if this causes some strange results).

***JOIN Attribute Ordering***

Table attributes of our simplified query language are 'unqualified' (i.e. they do not include the name of the table within which an attribute resides). When performing a JOIN query therefore, we must ensure that the ordering of the specified tables is the same as the ordering of the specified attributes. For example, a query should be of the form: JOIN tableOne AND tableTwo ON attributeFromTableOne AND attributeFromTableTwo; Enforcing this ordering makes it possible for your server to easily identify the specific attributes referred to by a query.

***Response tables***

The order of values returned by a SELECT should be the same as that specified in the query (e.g. SELECT name, id FROM marks would return the name column first, followed by the id column). Note that SELECT \* should return the values in the order that they are stored in the table. The table returned by a JOIN should contain attribute names in the form OriginalTableName.AttributeName (see the [query transcript](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/example-transcript.docx) for specific examples). This is so the user can determine which attributes came from which tables (as well as coping with the situation where two tables have attributes of the same name). Note that these composite names are purely for the display of the tables - attribute names containing . characters are not permitted by the BNF and are therefore not valid for use in queries. The joined table should NOT contain the ID columns from the original tables, but rather should include a new ID column containing freshly generated IDs.

**Task 9: Error Handling**

Your query interpreter should identify any errors in the structure and content of incoming queries. The response returned from your server back to the client must begin with one of the following two "status" tags:

* [OK] for valid and successful queries, followed by the results of the query.
* [ERROR] if the query is invalid, followed by a suitable human-readable message that provides information about the nature of the error.

Errors should be returned to the client if the SQL is malformed (i.e. doesn't conform to the BNF) or when the user attempts to perform prohibited actions, including (but not limited to):

* trying to insert too many (or too few) values into a table entry
* attempting to create a database or table using a name that already exists
* creating a table with duplicate column names (or trying to add a column with an existing name)
* attempting to remove the ID column from a table
* changing (updating) the ID of a record
* queries on non-existent databases, tables and columns
* queries which use invalid element names (e.g. reserved SQL keywords)

Errors should NOT be returned in situations where the user performs:

* a valid query that has no matches: just return the column names and no data rows
* a query to delete columns/tables/rows/database that contain data: the user should be free to perform destructive actions
* a comparison of two different data types: attempt a sensible comparison if possible, return blank results if no comparison is possible

Note that because you are not required to maintain type information for the attributes in a table, it will not be possible for you to validate the type of inserted data. It is therefore the responsibility of the user to ensure that only numerical data is stored in numerical attributes and string data is stored in string attributes etc.

You may wish to make use of exceptions to handle errors internally within your server, however these should NOT be returned to the client. Java exceptions are for the benefit of Java programmers - they are not intended for use as user error messages. It is essential that your response back to the client begins with the correct status tag (either [OK] or [ERROR]). These will be used by the automated testing scripts during the marking process. The "human-readable" error message text will not be assessed during the marking process - we trust you to make the effort of returning something appropriate !

**Task 10: Final Submission**

Note that this is an individual assignment, not a group activity ! Automated checkers will be used to flag partly (or fully) duplicate submissions from students in this year's cohort. If markers feel that collusion between students has taken place, the incident will be referred to an academic malpractice panel. The outcome of this panel may be a mark of zero for assignment or even the entire unit (if it is a repeat offence).

You must create a zip archive of your entire Maven project (i.e. the cw-db folder) and upload this into the relevant submission folder of the Assessment, submission and feedback Blackboard page. It is essential that you ensure your code compiles and runs on the lab machines using mvnw before you submit it.

Your submission will be assessed on the extent and success with which it implements the described query language, as well as the flexibility and robustness with which it operates. You should make sure your code can respond to *at the very least* the "standard" query spacing (as illustrated in the [example transcript](https://github.com/drslock/JAVA2024/blob/main/Weekly%20Workbooks/07%20DB%20assignment/resources/example-transcript.docx)). As with any real implementation of SQL, you should also however support variability of whitespace. Remember that different users may use different spacing standards and styles - it is desirable to support them, not constrain them !

Make sure your code does not contain anything specific to your computer or platform (e.g. absolute file paths, operating system specific code etc). Before submitting your code, we advise you to test your project on a computer *other than the one it was developed on*. If you developed your code on your own computer, make sure it operates correctly on the lab machines. If you developed your code on the lab machines, make sure it operates correctly on another platform (e.g. ideally a Windows computer - if you have access to one). Clear out all files from the databases folder and then ensure the code compiles and runs correctly with Maven (using: mvnw clean compile test). We will apply a penalty mark if we cannot run your code "out of the box" - we can't spend time "fixing" everyones' projects before we mark them !

It is VERY important that you do NOT change the name or parameters of any of the classes and methods that you have been given. Scripts will be used to automatically run your code to make sure it operates correctly - if you rename somethings you shouldn't, we won't be able to mark your code ! Your main class MUST be called DBServer and should not change the signature of the constructor or the handleCommand method. It is **ESSENTIAL** that you check your code passes the original skeleton test script - if it does not pass these basic tests then your code will not pass any of the marking test scripts.

Remember that this unit is not just about assessing the operation of executable code - we are also interested in wider "development" issues. With this in mind, your submission will be assessed on "code quality" as outlined in the lectures. Finally, remember that we will use code similarity checking tools to determine "derived" code (material "found" online or generated by AI) you will only receive credit for code that YOU yourself have written.