COMP 282 - Project 3

Fall 2018

Instructions for Submission

You must provide all source files in a single `.zip` file. This file must contain no directories whatsoever. That is, if I were to unzip it, the contents would be only the `.java` files required for the project. For example, in a *nix environment, I would create a submission by navigating to my source directory issuing a `zip AdamClark *.java`. **Under no circumstances will a submission be accepted if it is the product of you zipping your IDE's project directory!**

There are to be absolutely no packages used in your submissions. Any code correction necessary to make the grader run will result in a failure. The use of the default package is, generally, not advised for enterprise-level work; this, however, is not that.

Additionally, you must not include any `main` functions in your submission, only those files required for each project part are needed.

The following files **must** be present in your submission:

- `DBLite.java`

A series of "checker" functions have been provided for each part of the project. Ensure your project passes these basic checks prior to submission. **If any files are excluded, or fail to pass the checker, the grader *will* fail, resulting in a 0 on the assignment.**

Part 1 - Storing and Retrieving Data

You will undoubtedly be required to store data when building anything but the most trivial software. To this end, we will focus on smart ways to store such data so that you may quickly retrieve it when necessary. But first, we might want to cover the basics.

Implement the following interface in a class named `DBLite`:

```
import java.io.*;

public interface IReadWritable {
  public byte[] read() throws IOException;
  public void write(byte[] value) throws IOException;
  public void write(String value) throws IOException;
}
```

The `DBLite` class will open a file given by a constructor argument – `name` – appended with a suffix of `.dbl`. For example, consider the following code:

```
try {
  IFileBased db = new DBLite("test");
  db.write("My File Content");
} catch (IOException e) {
  System.err.println("Uh oh.");
}
```

This code would result in there being a file – `test.dbl` – created, containing the text `My File Content`. We may read it with something like the following:

```
try {
  IFileBased db = new DBLite("test");
  System.out.println(new String(db.read()));
} catch (IOException e) {
  System.err.println("Uh oh.");
}
```

The checker for this project part is as follows:

```
import java.io.*;

public class ReadWriteCheck {
  public static void run() throws IOException {
    IReadWritable db1 = new DBLite("read_write_check");
    db1.write("This is a test.");

    IReadWritable db2 = new DBLite("test");
    assert(new String(db2.read()).equals("This is a test."));
  }
}
```

Note: you generally do *not* want to construct a class interface in the way we are throughout this project. The fact that our database class is storing information in files should be a (poorly kept) "secret" from the rest of the code base. You would always want to keep these sorts of implementation details a secret, unless your grade depended on it...

Part 2 - Our First Format

In general, determining the exact right format for data in files is a tricky prospect. For example, we have spent an entire semester studying how to properly store data into an abstract type in order to facilitate quick retrieval, insertions, and deletions. In terms of actual computer code, we will start with something a bit less sophisticated than the things we have been studying thus far.

In order to make use of our toy database, we must have key/value pairs that will allow us to retrieve specific pieces of data, without having to look at the entire database file. This shall be done with the following interface:

```
import java.io.*;

public interface IRegistrar {
  public String load(String key) throws IOException;
  public void store(String key, String value) throws IOException;
}
```

For example, we may want to store a string into our database:

```
DBLite db = new DBLite("my_db");
db.store("first", "My simple example.");
db.store("second", "My next example.");
```

Then, in order to retrieve it again:

```
DBLite db = new DBLite("my_db");
String loaded = db.load("first");
System.out.println(loaded); // "My simple example."
```

In order to get started quickly, we shall adopt a linear-search-based approach to our file store. That is, we will store everything as it is added in the class, and search for the key later. Effectively, we want to keep *everything* in memory, and write to our file whenever we get a new key/value pair:

```
public void store(String key, String value) throws IOException {
  switch (this.strategy) {
  case LINEAR: this.linearStore(key, value); break;
  }
}
```

Therefore, we need to have two constructors:

```
public DBLite(String name) throws IOException {
  this(name, Strategy.LINEAR);
}
```

```
public DBLite(String name, Strategy s) throws IOException {
  this.strategy = s;
  // Some file or path construction here.
}
```

In an actual file, the you should store both the key and the string. Each key and value should be separated by a null terminator (`\0`). Each entry in the file should, likewise, be separated by a null terminator (`\0`). This ensures that we will be able to store any arbitrary string values we need.

A checker for this section follows:

```
import java.io.*;

public class RegistrarCheck {
  public static void run() throws IOException {
    IRegistrar db1 = new DBLite("registrar_check");
    db1.store("first", "My example string.");
    db1.store("second", new Integer(1234).toString());
    assert(db1.load("first").equals("My example string."));
    assert(db1.load("second").equals("1234"));

    IRegistrar db2 = new DBLite("registrar_check");
    int i = Integer.parseInt(db2.load("second"));
    assert(db2.load("first").equals("My example string."));
    assert(i == 1234);
  }
}
```

Note: I would *strongly* advise you not to use `Map` for anything in this project. While their use for this part might make a lot of sense, they will require you to engage in a lot of hacks in future sections.

Part 3 - A Hash File

Typically, when files are used as a storage medium to aggregate smaller pieces of data, they include a header structure that will tell you where a particular piece of data is located further into the file. For example, your file might look something like this:

```
first,17second,35My example string.1234
```

Here, we can see that the first key "points" to character 17 in the file. One should expect, therefore, to be able to treat the file as one big array, and find the information we care about.

How we go about finding this "header reference" is the main optimization of this section. Whereas we previously attempted to do a linear search across the

entire file, we now want to use a more sophisticated data structure to speed up data retrieval. It turns out hash tables are a perfect candidate for this type of optimization.

First, we must modify the above example to allow us to store keys of arbitrary size. That means we need a special character to denote the boundary between a pair of keys and offsets. We may use the `\0` character for this:

```
first,0\0second,18\0\0My example string.\01234\0
```

You will notice that we not only utilize this character in the header, but the rest of the file as well. This is so we do not have to keep track of two numbers in the header: the offset of the data, and its length. We may simply find the offset, and read from there until we reach the next instance of `\0`. Also notice the `\0\0`; this marks the end of the file header, and the beginning of data. Adopting this convention allows us to use offsets *relative* to the start of the data. Saving us from having to recompute file offsets whenever a new key is inserted.

Implementation of this new strategy will require you to modify your code to include a separate constructor:

```
public class DBLite implements IReadWritable, IRegistrar {
  public static enum Strategy { LINEAR, HASHING };
  public DBLite(String name) throws IOException {
    this(name, Strategy.LINEAR);
  }
  public DBLite(String name, Strategy s) throws IOException {
    // ...
  }
}
```

You must also add new methods to accomplish a `load` and `store`:

```
public String hashingLoad(String key) throws IOException {
  // ...
}
public void hashingStore(String key, String value) throws IOException {
  // ...
}
```

The `hashingStore` method may simply append new values to the end of the file, the data itself need not be in hashed order.

You will be expected to use the following as your hashing function for keys:

```
private int hash(String key) {
  int h = 0;
  for (int i = key.length() - 1; i >= 0; --i)
    h += (int) key.charAt(i);
  return h;
}
```

Your are also expected to use open addressing in order to resolve collisions. In the event of such a collision, you will simply search for the value in the next (positive direction, only) key/value pair.

When constructing the table, you must maintain only the number of slots required to store your current set of keys, and no more. This means you will have to recompute the table on each insertion, as your table will have grown by one.

Perhaps unsurprisingly, the checker for this section is very similar to the last:

```
import java.io.*;

public class HashingCheck {
  public static void run() throws IOException {
    IRegistrar db1 = new DBLite(
      "hashing_check",
      DBLite.Strategy.HASHING
    );
    db1.store("first", "My example string.");
    db1.store("second", Integer.toString(1234));
    assert(db1.load("first").equals("My example string."));
    assert(db1.load("second").equals("1234"));

    IRegistrar db2 = new DBLite(
      "hashing_check",
      DBLite.Strategy.HASHING
    );
    String s = db2.load("first");
    int i = Integer.parseInt(db2.load("second"));
    assert(db2.load("first").equals("My example string."));
    assert(i == 1234);
  }
}
```