# Lab 2

Terrence Li
tzli@calpoly.edu
CPE 369-01

Collaborated with:
Timothy Chu <tchu01>
Michael Wong <mwong56>

## Logistics:

To compile:

```
- javac -cp json-20151123.jar *.java
```

To run:

```
- java -cp .:json-20151123.jar <Experiment>
```

## Implementation Overview:

- Most of the design choices for KeyValueStore and KVCollection utilized the various HashMap methods.
- For the find() method, I used an iterator to search through the Hashmap
- KeyValueStore also utilizes an internal HashMap that maps a string to a KVCollection. Limit is default set to MAX_LONG

## Conducted Experiments

Experiment 1:
- Description: To prevent Experiment 1 from running out of disk space before heap space, everything was done in-memory, meaning there were no output files. We generated an object then stored it until out of memory.
- Results: We hit OOM at 611741 objects.
- To run: java -cp .:json-20151123.jar Experiment1
- Conclusions: The program took quite a long time to throw the OOM error. There are many factors that could contribute to this such as large in-memory threshold or object generation took a while.

Experiment 2:
- Description: Same concept as Experiment 1 except with thghtShre JSON Objects
- Results: We hit OOM at 786516 objects.
- To run: java -cp .:json-20151123.jar Experiment2
- Conclusions: thghtShre has less fields in its JSON and therefore can store more objects before OOM error

Experiment 3:
- Description: We timed the amount it took to generate n objects, store the n objects into the KVCollection. Starting from 1000 objects until we hit 100000 objects.

- Results:

```
Testing 1000 items
3552535 nano seconds
Testing 10000 items
19198511 nano seconds
Testing 100000 items
37639377 nano seconds
Testing 1000000 items
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit
exceeded
```

- To run: `java -cp .:json-20151123.jar Experiment3`
- Analysis: I believe that the dramatic increase in 1000->10000 is due to the resizing of the HashMap. I also believe the increase drops because the resize method is more efficient for a larger item increase.

Experiment 4:

- Description: The goal was to determine the time of retrieval of a JSON object by the get() method and the average time for that.
- Results:

```
Average time out of 100 runs to retrieve json object by key from collection of size
100: 1431.4 nanoseconds
Standard Deviation of time out of 100 runs to retrieve json object by key from
collection of size 100: 788.813957787259 nanoseconds
Average time out of 100 runs to retrieve json object by key from collection of size
1000: 3441.54 nanoseconds
Standard Deviation of time out of 100 runs to retrieve json object by key from
collection of size 1000: 4482.185298311528 nanoseconds
Average time out of 100 runs to retrieve json object by key from collection of size
100000: 13529.92 nanoseconds
Standard Deviation of time out of 100 runs to retrieve json object by key from
collection of size 100000: 2506.1499862538158 nanoseconds
```

- To run: `java -cp .:json-20151123.jar Experiment4And5`
- Analysis: As the amount of object increases, so does the time for retrieval. It's interesting because I thought HashMaps had a constant time of retrieval. I guess the constant time of retrieval is relative to the size.

Experiment 5:

- Description: The goal was to determine the retrieval time of a JSON object by the find() method and the average time.
- Results:

```
BEFUDDLED
    Finding nonexistent key-value pairs
  Average time out of 100 runs to find all jsons with supplied key-value pair from
collection of size 100: 39031.52 nanoseconds
  Standard Deviation of time out of 100 runs to find all jsons with supplied key-
value pair from collection of size 100: 371418.4300655388 nanoseconds
  Average time out of 100 runs to find all jsons with supplied key-value pair from
collection of size 1000: 146337.24 nanoseconds
```

```
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 1439443.0879512266 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 4827812.11 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 4.799775818901476E7 nanoseconds

        Finding existing key-value pairs
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100: 97647.36 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100: 933279.3329681577 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 80965.41 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 786912.3823192781 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 2703392.28 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 2.6891907953736484E7 nanoseconds

THGHTSHRE
        Finding nonexistent key-value pairs
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100: 1621.2 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100: 9889.927625619925 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 2297.33 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 16490.782152496588 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 454667.35 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 4517453.893847231 nanoseconds

        Finding existing key-value pairs
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100: 1754.94 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100: 11380.786267934214 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 3861.86 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 1000: 32200.26425513307 nanoseconds
   Average time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 405315.16 nanoseconds
   Standard Deviation of time out of 100 runs to find all jsons with supplied key-value pair from collection of size 100000: 4026114.49909154 nanoseconds
```

To Run: java -cp .:json-20151123.jar Experiment4and5
Analysis: It seems that it’s easier to determine that an object does not exist rather than existing.

Experiment 6:

- Description: Using 100 messages and 10 KV Collection, we sorted out the messages into the KVCollections by range that the messageId falls in. For example, 1-10 would contain messages with messageId 1-10. The retrieval would then be easier because the valid key would be the messageId. Using the messageId, we can use to find the correct KVCollection. Then a modulus would take place to find the position in the KVCollection.
- Results:

  ```
  Average for range keys for 100 trials: 4128.1 nanoseconds
  Standard Deviation 1886.9900344198961
  Average for round robin for 100 trials: 136057.26 nanoseconds
  Standard Deviation 647051.1661791765

  Average for range keys for 1000 trials: 8260.985 nanoseconds
  Standard Deviation 101694.67730625231
  Average for round robin for 1000 trials: 21665.31 nanoseconds
  Standard Deviation 10996.225716758458

  Average for range keys for 10000 trials: 1941.9993 nanoseconds
  Standard Deviation 70544.84638851583
  Average for round robin for 10000 trials: 2912.66 nanoseconds
  Standard Deviation 1420.9282404118796

  Average for range keys for 100000 trials: 661.32279 nanoseconds
  Standard Deviation 58114.87827360951
  Average for round robin for 100000 trials: 3255.91 nanoseconds
  Standard Deviation 1706.953081341136
  ```
- To Run: java -cp .:json-20151123.jar Experiment6
- Analysis: Using the range sharding method, it is noticeably shorter than the round robin. This is due to the fact that the program doesn’t need to search every KVCollection until it finds the JSONObject.

---

## Reflection

I learned a lot about various ways key-value stores work. Although I used primarily internal HashMaps, there were modifications made to the HashMap methods that made the implementation special (ex. find method). This lab was also beneficial in gaining insights on the framework on distributed programing and the value of sharding. One new thing I learned from this lab was that constant retrieval time of a HashMap is dependent on it’s size. Also, it is not efficient to iterate through a HashMap.