# Futher Java
# Supervision 1

**Supervisor:** Joe Isaacs (josi2).
All work should be submitted in PDF form with page numbers 36 hours before the supervision to the email `josi2@cam.ac.uk`. If you have any questions on the course please include these at the top of the supervision work and we can talk about them in the supervision. Please include all code that you think is relevant (that you want me to look at in the PDF).

---

A *key-value store* (KVS)[1] is a typed (possible persistent) database used for storing values with a associated keys. In this supervision you will build a KVS in Java. This KVS will be a service that clients can connect to then possibly read (or write) values to (or from) this store. While you are implementing this KVS you will be carrying out simple analysis and then evaluating you implementation. You will implement a server which (one or more) client(s) can connect (simultaneously) to then query the value for one or more keys and set values for one or more keys. The state in this KVS should be serlisable (in the database sense[2]). The KVS should be typed and support `String`s, you should store large values in a *compressed* form (but not smaller ones). Compression should be used where appropriate when the client and server are communicating.

A *future* construct used to model a computation that may or may not be finished. The computation will be run asynchronously, calling `get` on a future will either return the result of the computation or cause the calling thread to wait until the computation is finished and then the calling thread will continue with the result.

**Tasks:**

1. Implement a thread safe hash map (Do not use any *thread safe* standard library classes[3]) Write a test to show that its implementation is indeed thread safe. There are frameworks for writing tests JUnit[4] if you can try using this. It is integrated into Intellij IDEA[5].

2. Discuss the protocol which clients and servers will communicate KVS operations via with you supervision partners. The operations should be [`put`, `get`, `remove`]. Describe this protocol in your answers. How do you handle faliure?

3. Implement a KVS server and client. All on the server resources are cleaned up when a client disconnects. For each part of the protocol in the previous question show where you implement this in your code. How do the client and server commutative? Meet up with you supervision parter(s) and check that both your servers and clients commutative correctly. How does well does this server scale with multiple clients (originating from you supervision parter(s) machine(s))? What is the queueing time for a response under different workloads, analyse and evaluate this.

[1] `https://en.wikipedia.org/wiki/Key-value_database`
[2] `https://en.wikipedia.org/wiki/Serializability`
[3] However you many use other standard library classes
[4] `https://junit.org/junit5/docs/current/user-guide/#writing-tests`
[5] `https://www.jetbrains.com/help/idea/configuring-testing-libraries.html`

4. Implement a future with the `Future<V>` interface[6] ignore `cancel`,

   - Naïvely (using threads)
   - A using a thread pool where work can be submitted. Implement your own thread pool. Use the `Callable<T>` interface[7]

     Write a benchmark to show relative performance of these two implementations.

5. Add a method to your two implementations of `Future<U>` called `<U> thenCombine(Function<T,Future<U>>)` which allows chaining together two futures $F_1$ and $F_2$ such that when $F_1$.`isDone` holds the result is passed to $F_2$. Add this to both implementation in the previous question. Try to only start running $F_2$ once $F_1$ has completed. Can you combine a naïve thread future with a thread pool thread future?

[6] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Future.html`

[7] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Callable.html`