Lab 8: Thread-safe Collections, Tasks, and Thread Pools

> Reference: Core Java Volume I. Cay S. Horstmann. Chapter 12

Thread-safe collections (concurrent collections) in Java refers to a set of classes that allow multiple threads to access and modify a collection concurrently, without the need for explicit synchronization. (of course, the developers who implement thread-safe classes had to worry about locks, conditions, and etc. But that was their problems, not yours.)

Blocking Queues

`BlockingQueueTest.java` shows how to use a blocking queue to control a set of threads. The program searches through all files in a directory and its subdirectories, printing lines that contain a given keyword.

A producer thread enumerates all files in all subdirectories and places them in a blocking queue. This operation is fast, and the queue would quickly fill up with all files in the file system if it was not bounded.

We also start a large number of search threads (i.e., consumers). Each search thread takes a file from the queue, opens it, prints all lines containing the keyword, and then takes the next file.

We use a trick to terminate the application when no further work is required. In order to signal completion, the enumeration thread places a dummy object into the queue. (This is similar to a dummy suitcase with a label “last bag” in a baggage claim belt.) When a search thread takes the dummy, it puts it back and terminates.

Note that no explicit thread synchronization is required. In this application, **we use the queue data structure as a synchronization mechanism**.

Concurrent HashMaps

`CHMDemo.java` count all words in `.java` files of a directory tree. If counting words in a single file is a task, we can have a thread pool to execute multiple such tasks concurrently.

Typically, we will use a `HashMap` to store the result, where its key represents the word and value represents the count. Since we will update the HashMap concurrently, we should use `ConcurrentHashMap`.

When updating the map, you often need to do something special when a key is added for the first time. The `merge` method makes this particularly convenient. It has a parameter for the initial value that is used when the key is not yet present. Otherwise, the function that you supplied is called, combining the existing value and the initial value.

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue + newValue);
```

or simply

```
map.merge(word, 1L, Long::sum);
```