Applying Java FutureTask to Design a Memoizer Cache

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand how Java FutureTask conveys a result from a computation running in a thread to thread(s) retrieving the result.
- Recognize key methods in Java FutureTask.
- Know what a Memoizer is & motivates how FutureTask can optimize its performance in concurrent programs.

Memoizer caches function call results & returns cached results for same inputs.
Motivating FutureTask with a Memoizer
Motivating FutureTask with a Memoizer

- Memoization is optimization technique used to speed up programs

See en.wikipedia.org/wiki/Memoization
Memoization is an optimization technique used to speed up programs.

It caches the results of expensive function calls.

V `computeIfAbsent(K key, Function func)` {
  1. If key doesn’t exist in cache perform a long-running function associated w/key & store the resulting value via the key
  2. Return value associated with key
}
Memoization is an optimization technique used to speed up programs.

It caches the results of expensive function calls.

```java
V computeIfAbsent(K key, Function<? super K, ? extends V> func) {
    1. If key doesn’t exist in cache perform a long-running function associated w/key & store the resulting value via the key
    2. Return value associated with key
}
```
Memoization is an optimization technique used to speed up programs.

It caches the results of expensive function calls.

```java
V computeIfAbsent(K key, Function func) {
    1. If key doesn’t exist in cache perform a
       long-running function associated w/key
       & store the resulting value via the key
    2. Return value associated with key
}
```
Memoization is an optimization technique used to speed up programs.

- It caches the results of expensive function calls.
- When the same inputs occur again, the cached results are simply returned.

```java
V computeIfAbsent(K key, Function func) {
    1. If key already exists in cache
        return cached value associated w/key
}
```
Memoization is optimization technique used to speed up programs

- It caches the results of expensive function calls
- When the same inputs occur again the cached results are simply returned

```java
V computeIfAbsent(K key, Function func) {
    // 1. If key already exists in cache
    return cached value associated w/key
}
```
Motivating FutureTask with a Memoizer

- Memoizer defines a cache that returns a value produced by applying a (long-running) function to a key

See PrimeExecutorServiceFutureTask/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java
Memoizer defines a cache that returns a value produced by applying a (long-running) function to a key.

This class is based heavily on the book “Java Concurrency in Practice” by Brian Goetz et al.

See jcip.net
Motivating FutureTask with a Memoizer

- Memoizer defines a cache that returns a value produced by applying a (long-running) function to a key
- A value that’s already been computed for a key is just returned, rather than applying the function to recompute it
Motivating FutureTask with a Memoizer

- Memoizer defines a cache that returns a value produced by applying a (long-running) function to a key
  - A value that’s already been computed for a key is just returned, rather than applying the function to recompute it
  - By implementing Function a memoizer can be used whenever a Function is expected

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html
Motivating FutureTask with a Memoizer

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead.

See [docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html](docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html)
Motivating FutureTask with a Memoizer

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A different lock guards each hash bin

Contention is low due to use of multiple locks

Motivating FutureTask with a Memoizer

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A different lock guards each hash bin
- A SynchronizedMap just uses one lock

**SynchronizedMap**

```
0 1 2 3 4  ...  n
```

- `mCache`: ConcurrentHashMap<K,Future<V>>
- `mFunction`: Function<K,V>
- Memoizer(Function<K,V>)
- apply(K)
- computeValue(K): Future<V>
- getFutureValue(K,Future<V>)

Contention is higher due to use of one lock

See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap
Motivating FutureTask with a Memoizer

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A different lock guards each hash bin
- `computeValue()` uses `FutureTask` to ensure a function runs only when key is first added to cache

See docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html
Motivating FutureTask with a Memoizer

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A different lock guards each hash bin
- `computeValue()` uses FutureTask to ensure a function runs only when key is first added to cache

Only one computation occurs if multiple threads simultaneously call `computeValue()` for same key
End of Applying Java FutureTask to Design a Memoizer Cache