Java Concurrent Collections:
Designing a Memoizer with ConcurrentHashMap

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

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Lesson

- Understand the capabilities of Java’s concurrent collections
- Recognize the capabilities of Java’s ConcurrentHashMap & BlockingQueue
- Know how to apply the Java Concurrent HashMap class to design a “memoizer”

Memoizer caches function call results & returns cached results for same inputs
Overview of Memoizer
Overview of Memoization

- Memoization is an optimization technique used to speed up programs.
Overview of Memoization

• Memoization is 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
}
```
Overview of Memoization

- 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
}
```
Overview of 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
}

Memoizer

CACHE
Overview of Memoization

- 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
}
```
Overview of Memoization

- 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
}
```
Designing a Memoizer with ConcurrentHashMap
Memoizer defines a cache that returns a value produced by applying a (long-running) function to a key.
Designing a Memoizer with ConcurrentHashMap

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

This class is based on “Java Concurrency in Practice” by Brian Goetz et al.
Designing a Memoizer with ConcurrentHashMap

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

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

```java
Function<Long, Long> func =
    doMemoization
    ? new Memoizer<>
        (PrimeCheckers::isPrime,
         new ConcurrentHashMap());
    : PrimeCheckers::isPrime;

... new PrimeCallable(randomNumber, func);```

See [docs.oracle.com/javase/8/docs/api/java/util/function/Function.html](docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)
Designing a Memoizer with ConcurrentHashMap

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

```java
Function<Long, Long> func =
    doMemoization
    ? new Memoizer<>
        (PrimeCheckers::isPrime,
         new ConcurrentHashMap())
    : PrimeCheckers::isPrime;

... new PrimeCallable(randomNumber, func)); ...
```
Designing a Memoizer with ConcurrentHashMap

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

```java
Function<Long, Long> func = doMemoization
    ? new Memoizer<>(
        PrimeCheckers::isPrime,
        new ConcurrentHashMap<>());
    : PrimeCheckers::isPrime;

... new PrimeCallable(randomNumber, func)); ...

Don’t use memoizer
```
Designing a Memoizer with ConcurrentHashMap

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

```java
Function<Long, Long> func =
    doMemoization
        ? new Memoizer<>
            (PrimeCheckers::isPrime,
                 new ConcurrentHashMap());
    : PrimeCheckers::isPrime;
```

... func is identical, regardless of which branch is chosen

See upcoming lesson on “Java ExecutorCompletionService: Application to PrimeChecker App"
Designing a Memoizer with ConcurrentHashMap

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html
Designing a Memoizer with ConcurrentHashMap

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
- A group of locks guard different subsets of the hash buckets

Contention is low due to use of multiple locks

Designing a Memoizer with ConcurrentHashMap

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
- A group of locks guard different subsets of the hash buckets

In contrast, a SynchronizedMap uses a single lock

See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap
Designing a Memoizer with ConcurrentHashMap

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A group of locks guard different subsets of the hash buckets
- apply() uses computeIfAbsent() to ensure a function only runs when key/value pair is added to cache

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent
Designing a Memoizer with ConcurrentHashMap

• Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  • A group of locks guard different subsets of the hash buckets
• apply() uses computeIfAbsent() to ensure a function only runs when key/value pair is added to cache, e.g.
  • This method implements “atomic check-then-act” semantics
    return map.computeIfAbsent (key,
      k -> mappingFunc(k));

See dig.cs.illinois.edu/papers/checkThenAct.pdf
Designing a Memoizer with ConcurrentHashMap

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A group of locks guard different subsets of the hash buckets
- apply() uses computeIfAbsent() to ensure a function only runs when key/value pair is added to cache, e.g.
  - This method implements “atomic check-then-act” semantics
- Here’s the equivalent sequence of non-atomic/-optimized Java code

```java
V value = map.get(key);
if (value == null) {
    value = mappingFunc.apply(key);
    if (value != null) map.put(key, value);
}
return value;
```

See dig.cs.illinois.edu/papers/checkThenAct.pdf
Designing a Memoizer with ConcurrentHashMap

- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead
  - A group of locks guard different subsets of the hash buckets
- apply() uses computeIfAbsent() to ensure a function only runs when key/value pair is added to cache

Only one computation per key is performed even if multiple threads simultaneously call computeIfAbsent() using the same key
End of Java Concurrent Collections: Designing a Memoizer with ConcurrentHashMap