Java ExecutorCompletionService: Designing a Memoizer

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 the Java CompletionService interface defines a framework for handling the completion of asynchronous tasks
- Know how to instantiate the Java Executor CompletionService
- Recognize the key methods in the Java CompletionService interface
- Visualize the ExecutorCompletionService in action
- Be aware of how the Java ExecutorCompletionService implements the CompletionService interface
- Know how to apply the Java ConcurrentHashMap 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.

See en.wikipedia.org/wiki/Memoization
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.

```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 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.
- 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
Designing a Memoizer with ConcurrentHashMap

- The Memoizer cache returns a value produced by applying a function to a key.
Designing a Memoizer with ConcurrentHashMap

• The Memoizer cache returns a value produced by applying a function to a key

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

See jcip.net
Designing a Memoizer with ConcurrentHashMap

• The Memoizer cache returns a value produced by applying a function to a key
• A value computed for a key is returned, rather than reapplying the function
Designing a Memoizer with ConcurrentHashMap

- The Memoizer cache returns a value produced by applying a function to a key.
- A value computed for a key is returned, rather than reapplying the function.
- Can be used when 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

- The Memoizer cache returns a value produced by applying a function to a key.
- A value computed for a key is returned, rather than reapplying the function.
- Can be used when a Function is expected.

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

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

Use memoizer
Designing a Memoizer with ConcurrentHashMap

- The Memoizer cache returns a value produced by applying a function to a key
- A value computed for a key is returned, rather than reapplying the function
- Can be used when a Function is expected

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

... Don't use memoizer

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

- The Memoizer cache returns a value produced by applying a function to a key
- A value computed for a key is returned, rather than reapplying the function
- Can be used when 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*

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

See upcoming part of this lesson on "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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html)
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

```
SynchronizedMap
```

In contrast, a SynchronizedMap uses a single lock

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

• Memoizer’s apply() hook method uses computeIfAbsent() to ensure a function only runs when a key is added to cache.
Designing a Memoizer with ConcurrentHashMap

- Memoizer’s apply() hook method uses computeIfAbsent() to ensure a function only runs when a key is added to cache, e.g.
- This method implements “atomic check-then-act” semantics
  
  ```java
  return map.computeIfAbsent(key,
  k -> mappingFunc(k));
  ```

See [dig.cs.illinois.edu/papers/checkThenAct.pdf](dig.cs.illinois.edu/papers/checkThenAct.pdf)
Memoizer’s apply() hook method uses computeIfAbsent() to ensure a function only runs when a key is added to cache, e.g.

- This method implements “atomic check-then-act” semantics

Here’s the equivalent sequence of Java (non-atomic/-optimized) 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’s apply() hook method uses computeIfAbsent() to ensure a function only runs when a key is added to cache, e.g.
- This method implements “atomic check-then-act” semantics
- Here’s the equivalent sequence of Java (non-atomic/-optimized) code
- Only one computation per key is performed even if multiple threads simultaneously call computeIfAbsent() using the same key
End of Java ExecutorCompletion Service: Designing a Memoizer