Java ExecutorCompletionService: Implementing a Memoizer

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

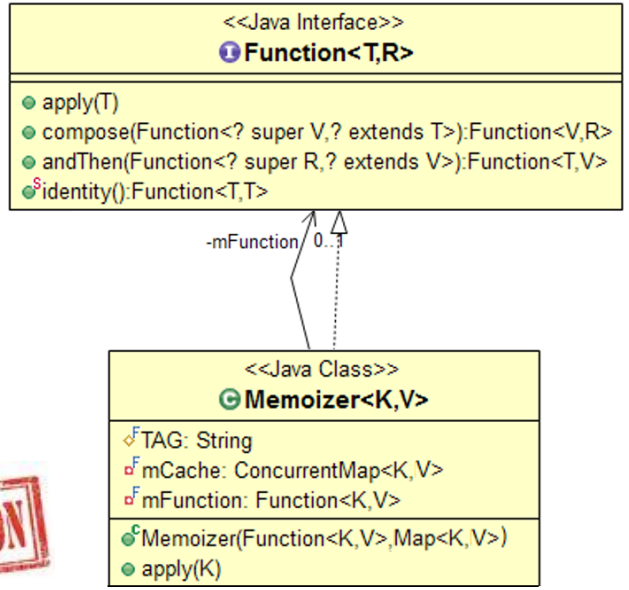
Professor of Computer Science

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 ExecutorCompletionService
- Recognize the key methods in the Java CompletionService interface
- Visualize the ExecutorCompletionService in action
- Be aware of how the Java ExecutorCompletion Service implements the CompletionService interface
- Know how to apply the Java ConcurrentHashMap class to design a "memoizer"
- Master how to implement the Memoizer class with Java ConcurrentHashMap



Memoizer caches function call results & returns cached results for same inputs

Implementing the Memoizer with ConcurrentHashMap

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final Map<K, V> mCache;


  private final Function<K, V> mFunction;

  public Memoizer(Function<K, V> func,
                  Map<K, V> map) {
    mFunction = func;
    mCache = map;
  }

  ...
```

See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final Map<K, V> mCache;

  private final Function<K, V> mFunction;

  public Memoizer(Function<K, V> func,
                  Map<K, V> map) {
    mFunction = func;
    mCache = map;
  }

  ...
```

Memoizer can be used transparently whenever a Function is expected

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final Map<K, V> mCache;
```

This map associates a key K with a value V that's produced by a function

```
  private final Function<K, V> mFunction;

  public Memoizer(Function<K, V> func,
                  Map<K, V> map) {
    mFunction = func;
    mCache = map;
  }

  ...
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final Map<K, V> mCache;


  private final Function<K, V> mFunction;

  public Memoizer(Function<K, V> func,
                  Map<K, V> map) {
    mFunction = func;
    mCache = map;
  }

  ...
```

This function produces a value based on the key

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final Map<K, V> mCache;


  private final Function<K, V> mFunction;

  public Memoizer(Function<K, V> func,
                  Map<K, V> map) {
    mFunction = func;
    mCache = map;
  }

  ...
```

Constructor initializes the fields

Both the Map & Function can be parameterized

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

Returns value associated with key in cache

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

This method is only available in Java 8+

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

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

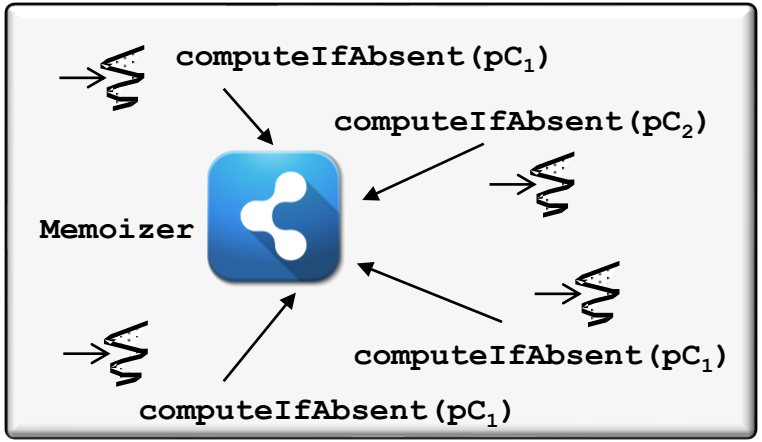
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

Multiple threads may call computeIfAbsent() concurrently for the same (non-existent) key



See www.javabrahman.com/java-8/java-8-maps-computeifabsent-computeifpresent-getordefault-methods-tutorial-with-examples

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

Try to find the key in the cache

See dig.cs.illinois.edu/papers/checkThenAct.pdf

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

If the key isn't present then atomically compute its value

See en.wikipedia.org/wiki/Lazy_initialization

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

"First thread in" for a given key won't block, but other threads trying to create this same key will block until the first computation is done

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {


    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

Return the value (either existing or newly computed)

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {



    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

Removes the key (& its value) from this memoizer

Implementing the Memoizer w/ConcurrentHashMap

- Memoizer can be configured w/ConcurrentHashMap to ensure a long-running computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {



    return mCache.computeIfAbsent(key, mFunction);
  }

  public V remove(K key) {
    return mCache.remove(key);
  }

  ...
```

Atomically remove the key (& its corresponding value) from this map

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#remove

End of Java Executor CompletionService: Implementing a Memoizer