Designing a Memoizer for Use With the Java ExecutorCompletionService

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

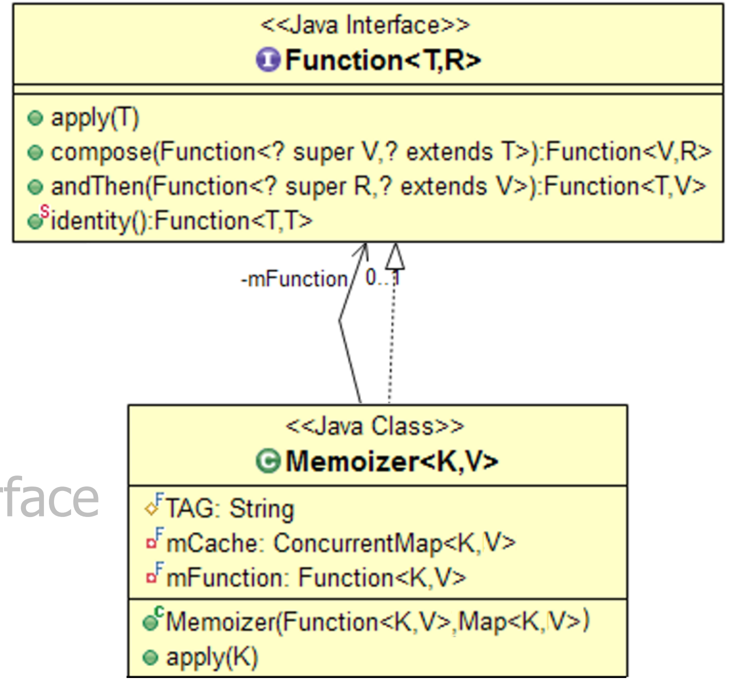
Institute for Software Integrated Systems

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 ExecutorCompletion Service implements the CompletionService interface
- 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 optimization technique used to speed up programs

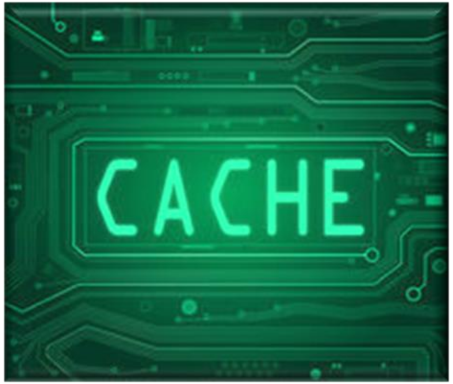
See en.wikipedia.org/wiki/Memoization

Overview of Memoization

- Memoization is 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

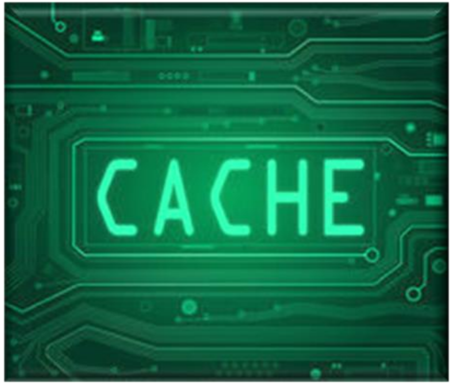


Overview of Memoization

- Memoization is 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

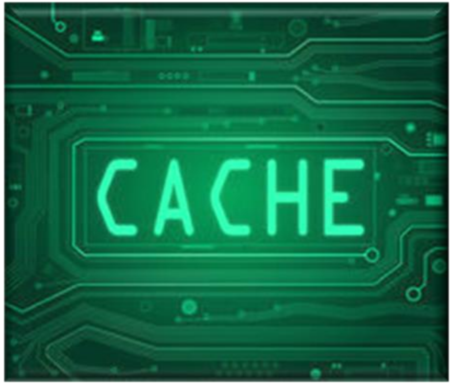


Overview of Memoization

- Memoization is 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

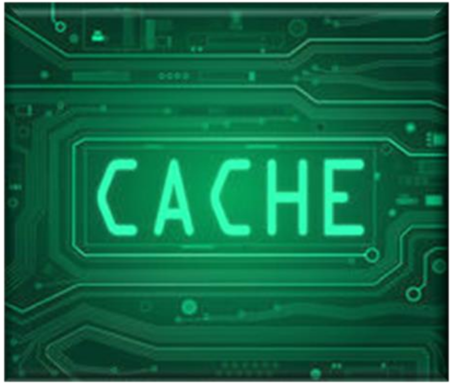


Overview of Memoization

- 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

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

Memoizer

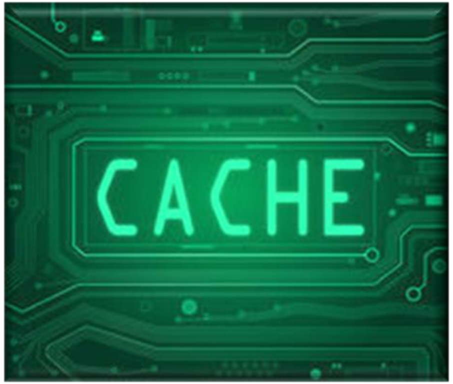


Overview of Memoization

- 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

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

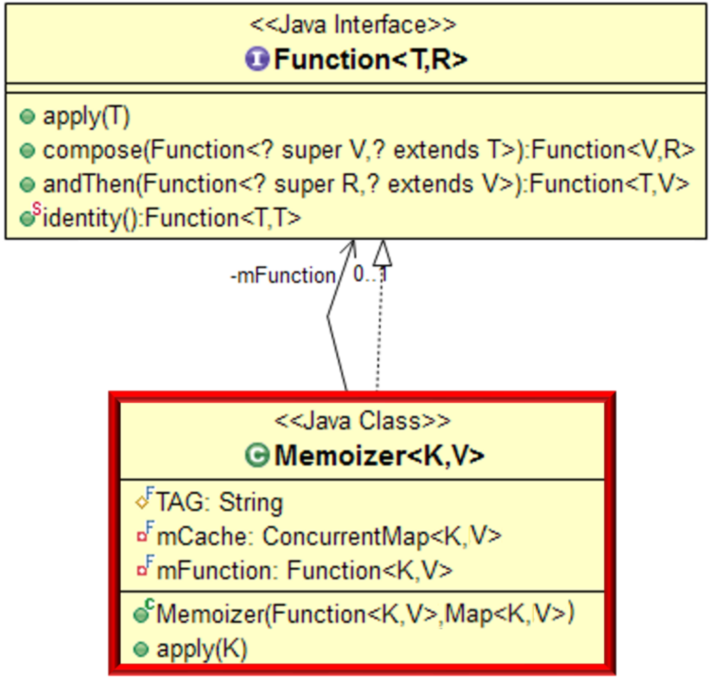
Memoizer



Designing a Memoizer with ConcurrentHashMap

Designing a Memoizer with ConcurrentHashMap

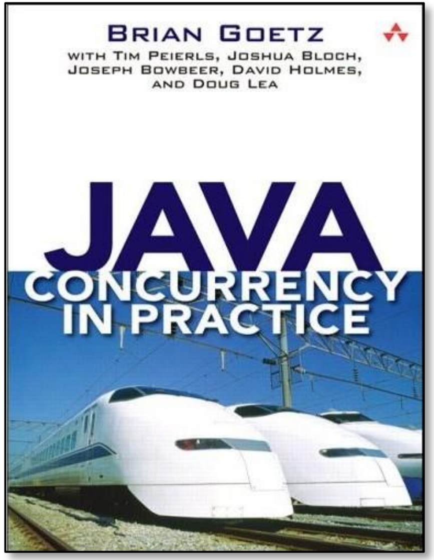
- The Memoizer cache returns a value produced by applying a function to a key

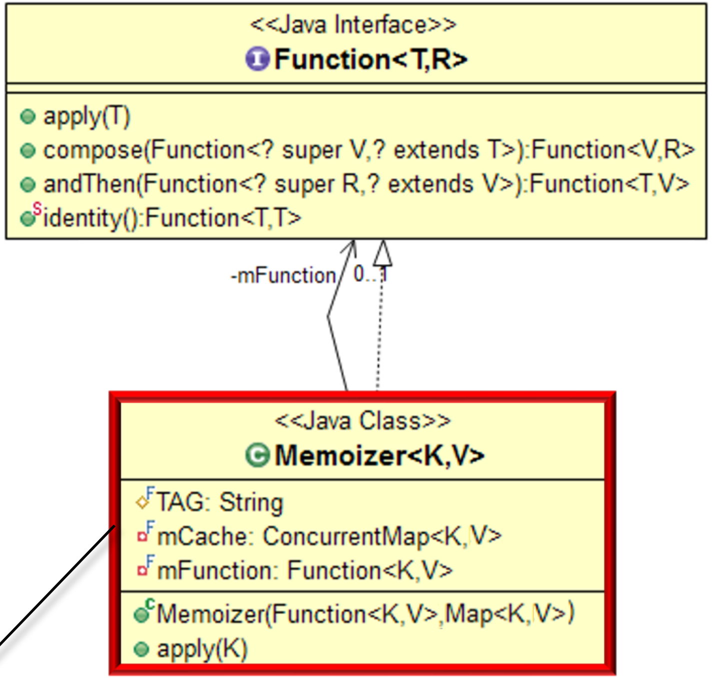


See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java

Designing a Memoizer with ConcurrentHashMap

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



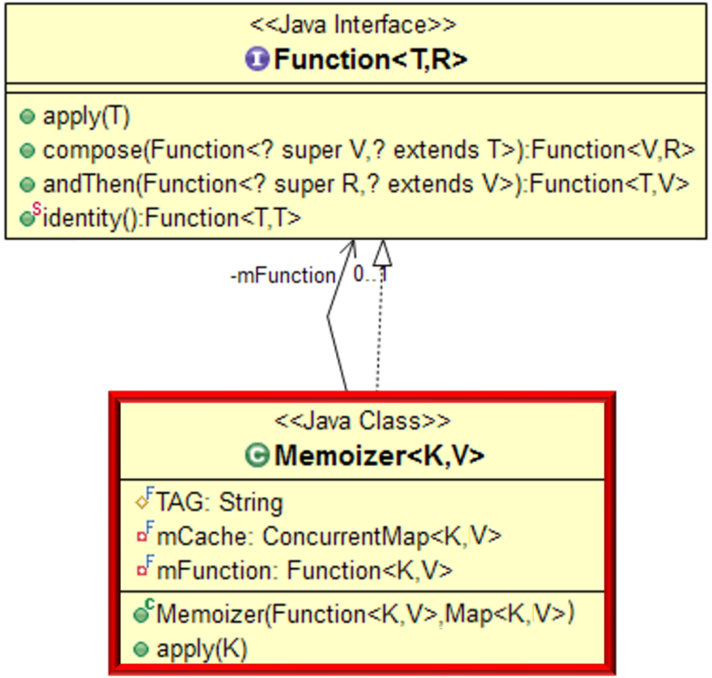


This class is based on "Java Concurrency in Practice" by Brian Goetz et al.

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

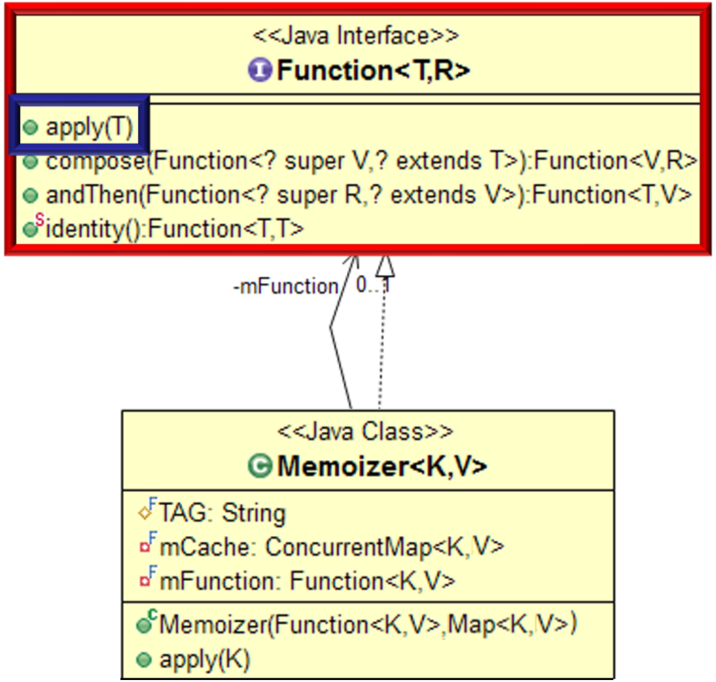


Designing a Memoizer with ConcurrentHashMap

- 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

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


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



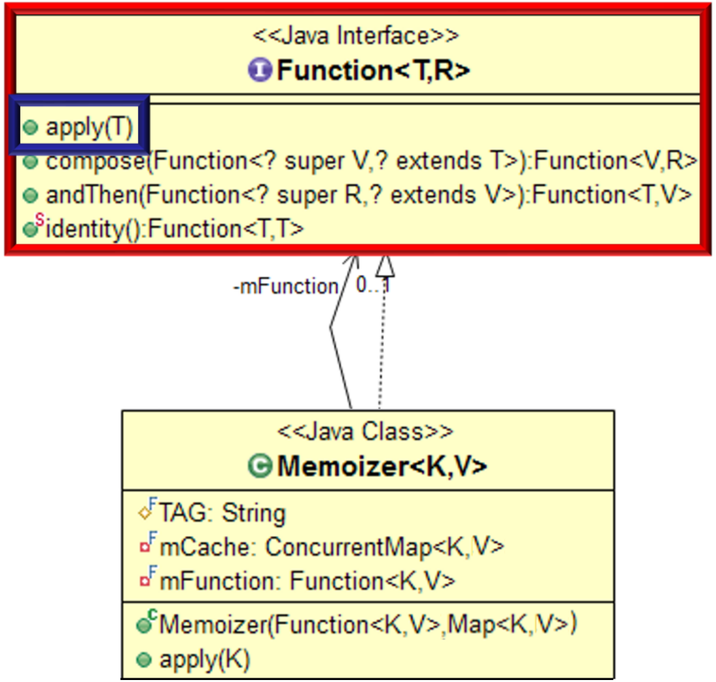
See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

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

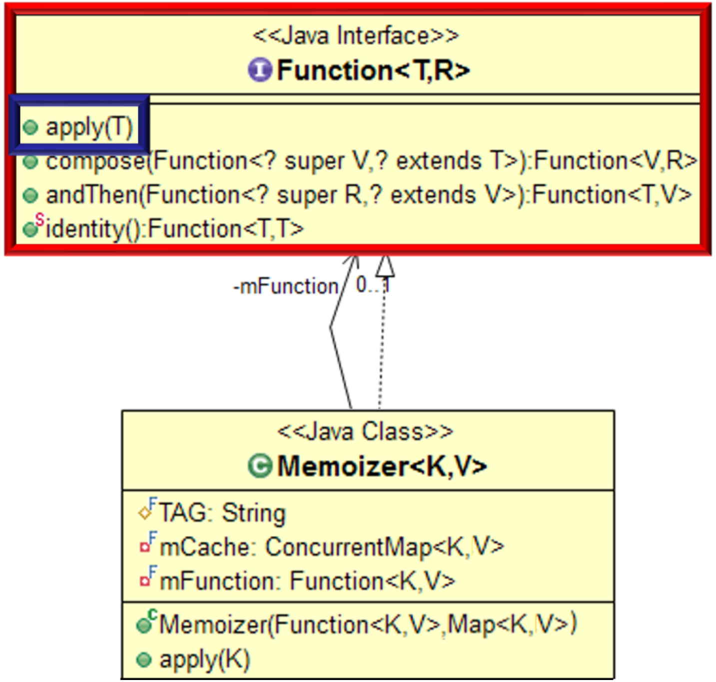
```
Function<Long, Long> func =
   doMemoization
     ? new Memoizer<>
         (PrimeCheckers::isPrime,
          new ConcurrentHashMap());
     : PrimeCheckers::isPrime;
...
new PrimeCallable(randomNumber, func)); ...
```

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




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

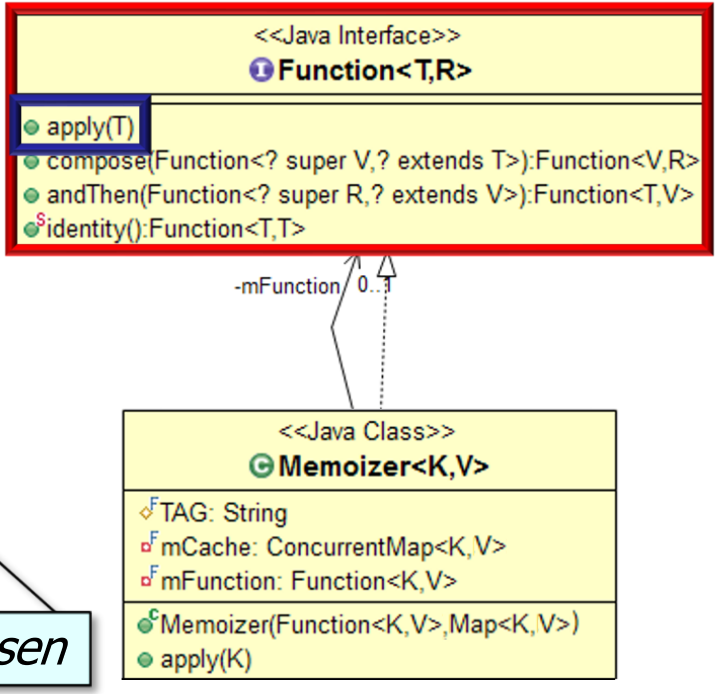
Don't use memoizer

```
...
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

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



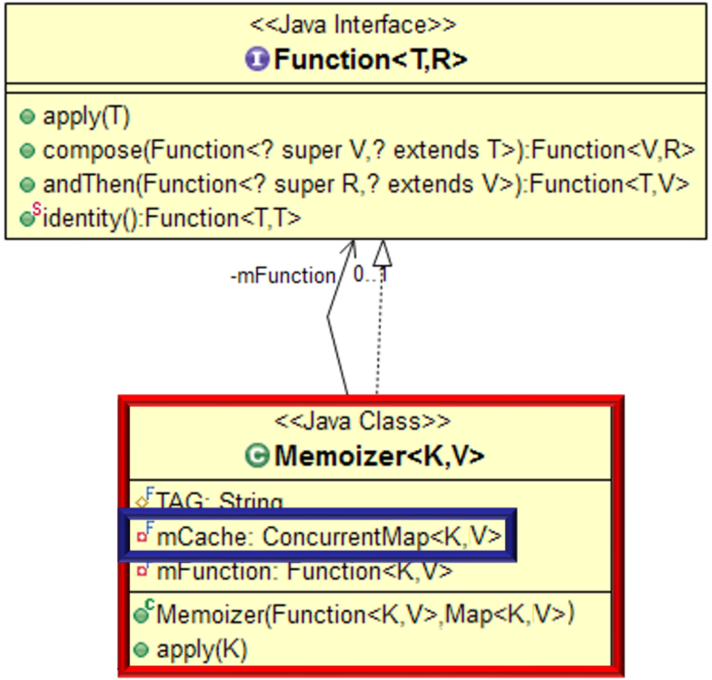
`func` *is identical, regardless of which branch is chosen*

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

See upcoming part of this lesson on "*Application to PrimeChecker App*"

Designing a Memoizer with ConcurrentHashMap

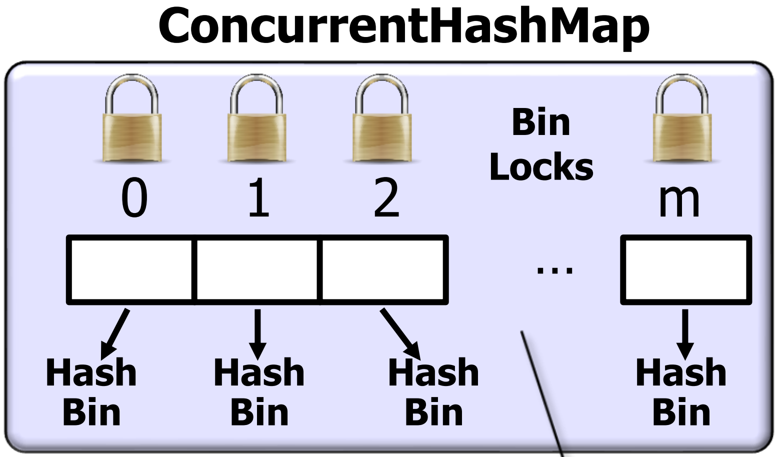
• Memoizer uses a ConcurrentHashMap to minimize synchronization overhead



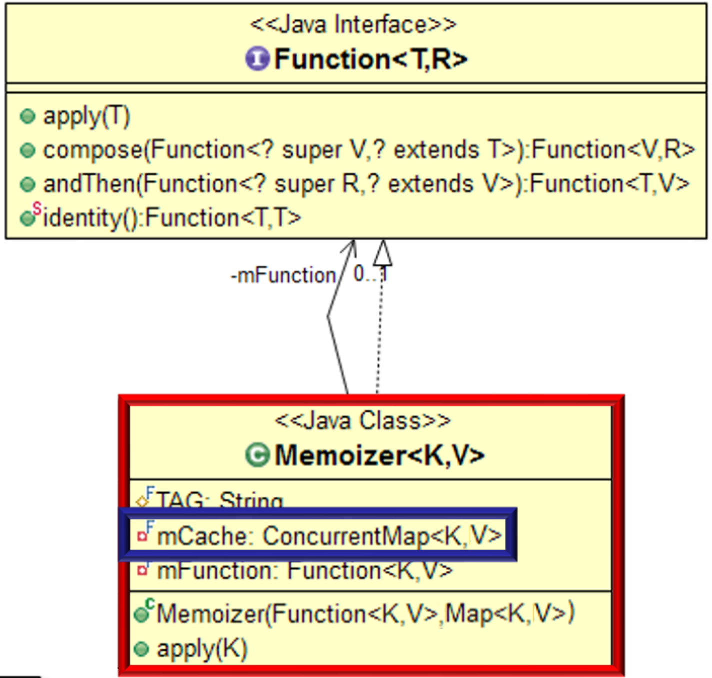
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

Designing a Memoizer with ConcurrentHashMap

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



Contention is low due to use of multiple locks

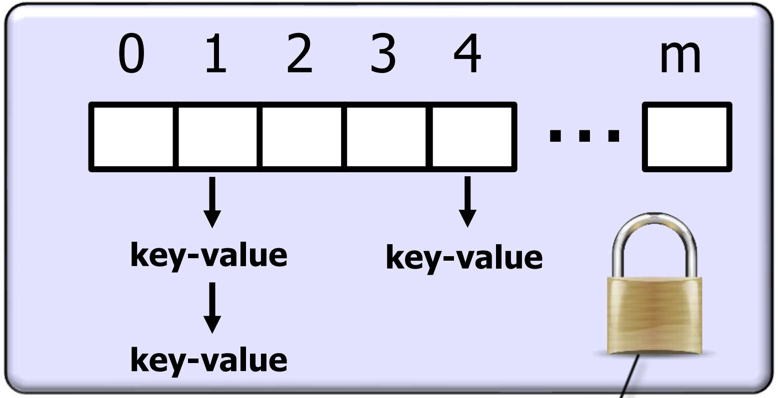


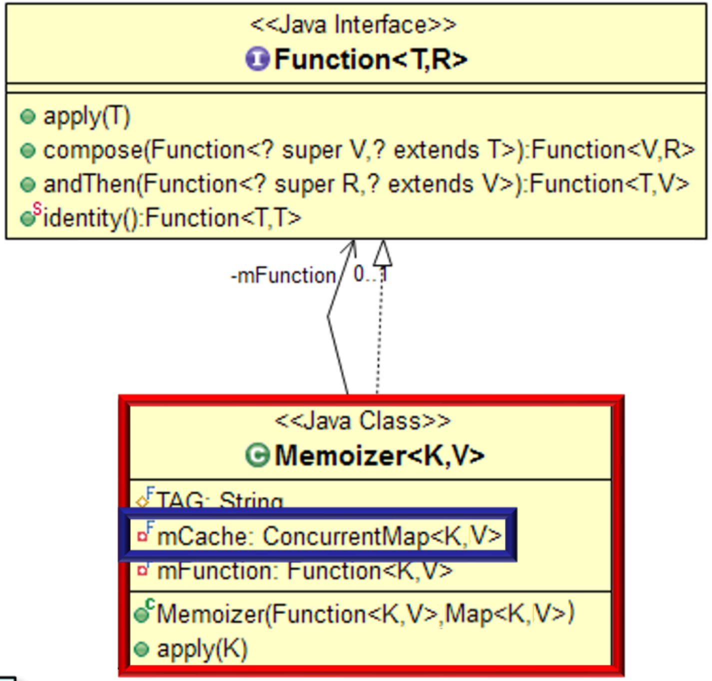
See www.ibm.com/developerworks/java/library/j-jtp08223

Designing a Memoizer with ConcurrentHashMap

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

SynchronizedMap



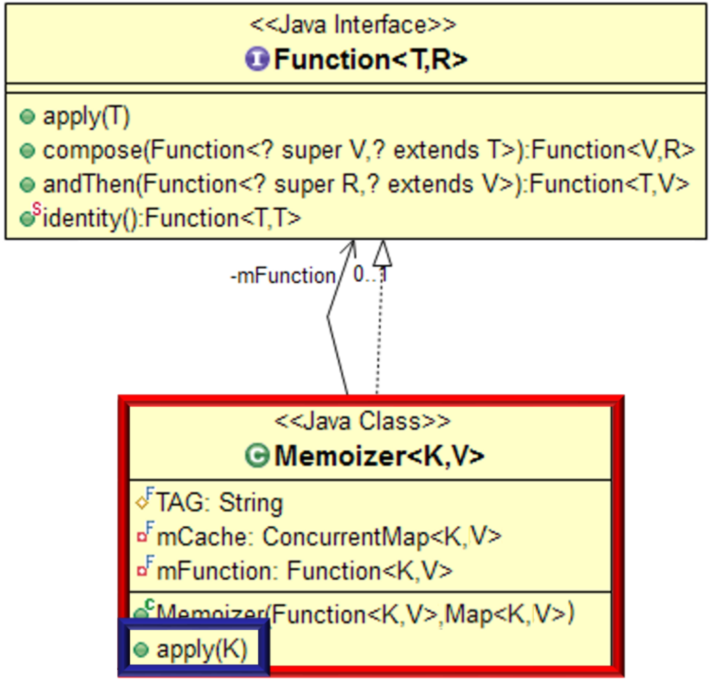


Contention is higher due to use of one lock

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

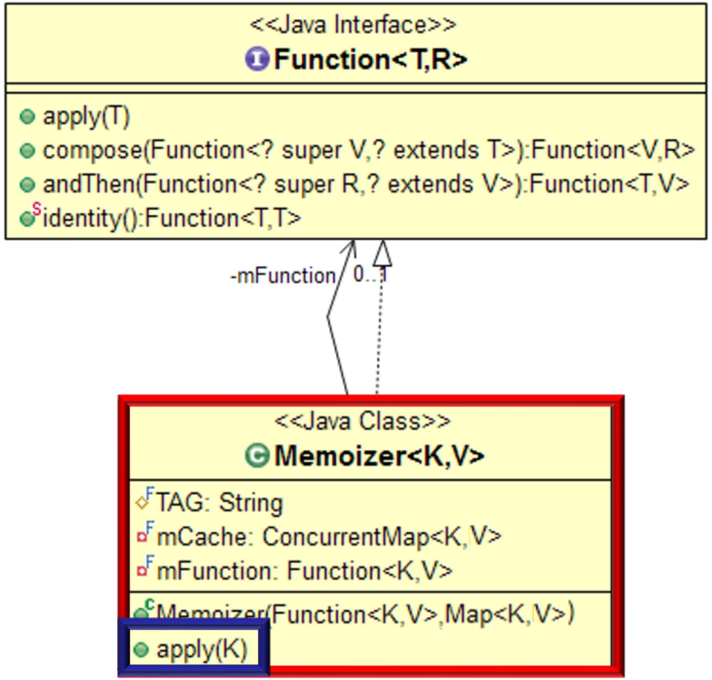


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent

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

```
return map.computeIfAbsent
   (key,
    k -> mappingFunc(k));
```

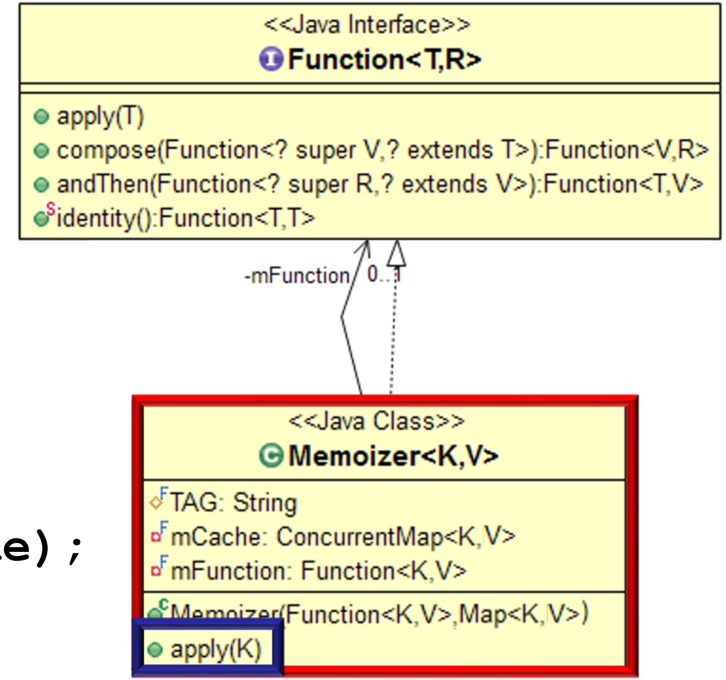


See dig.cs.illinois.edu/papers/checkThenAct.pdf

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

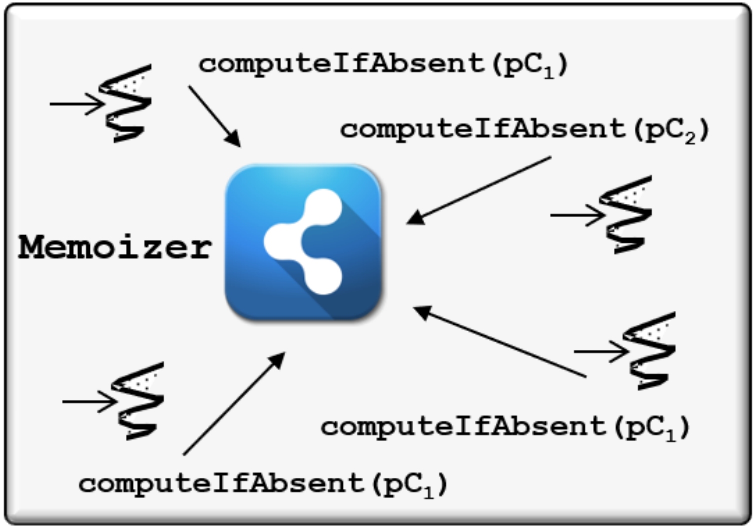
```
V value = map.get(key);
if (value == null) {
   value = mappingFunc.apply(key);
   if (value != null) map.put(key, value);
}
return value;
```



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 Designing a Memoizer for Use With the Java ExecutorCompletionService