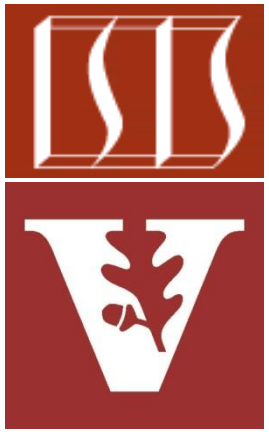
# Java Concurrent Collections: Designing a Memoizer with ConcurrentHashMap

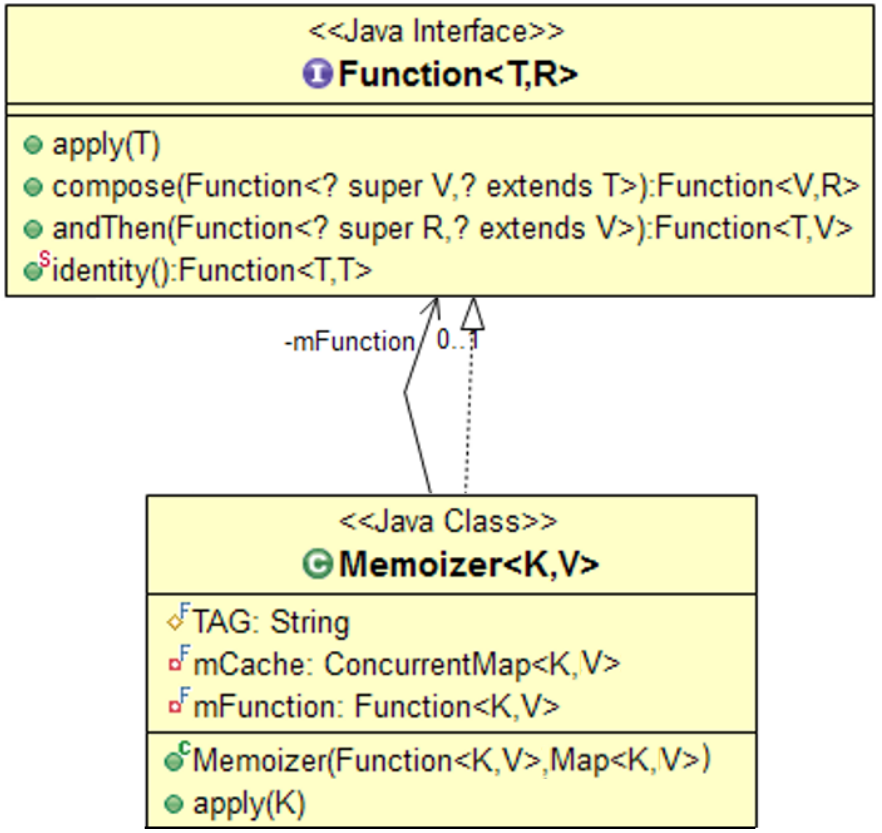



Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

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 optimization technique used to speed up programs

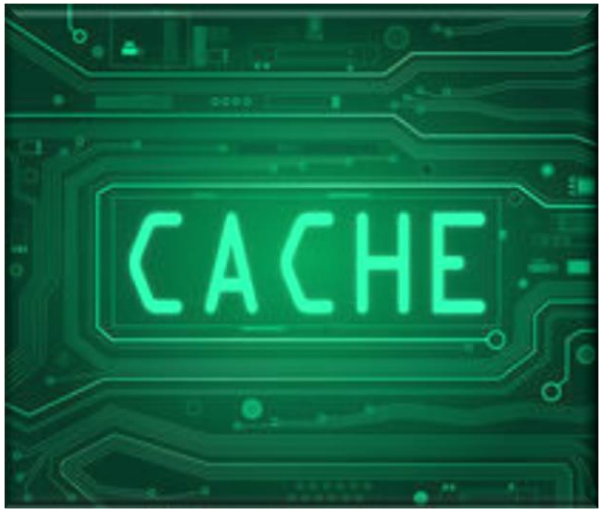

See en.wikipedia.org/wiki/Memoization

# Overview of Memoization

- Memoization is optimization technique used to speed up programs
  - It caches the results of expensive function calls

```
Value computeIfAbsent(Key key) {
   1. If key doesn't exist in map then
      perform a long-running computation
      associated with key & store the
      resulting value via the key
   2. Return value associated with key
}
```

Memoizer

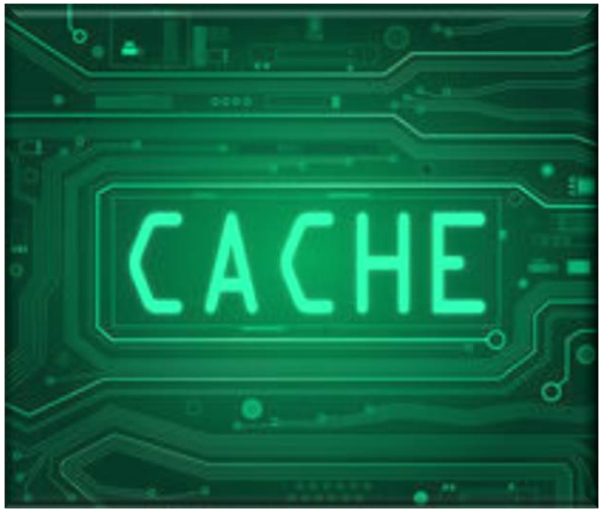


# Overview of Memoization

- 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

```
Value computeIfAbsent(Key key) {
   1. If key already exists in map return
      cached value associated w/key
}
```

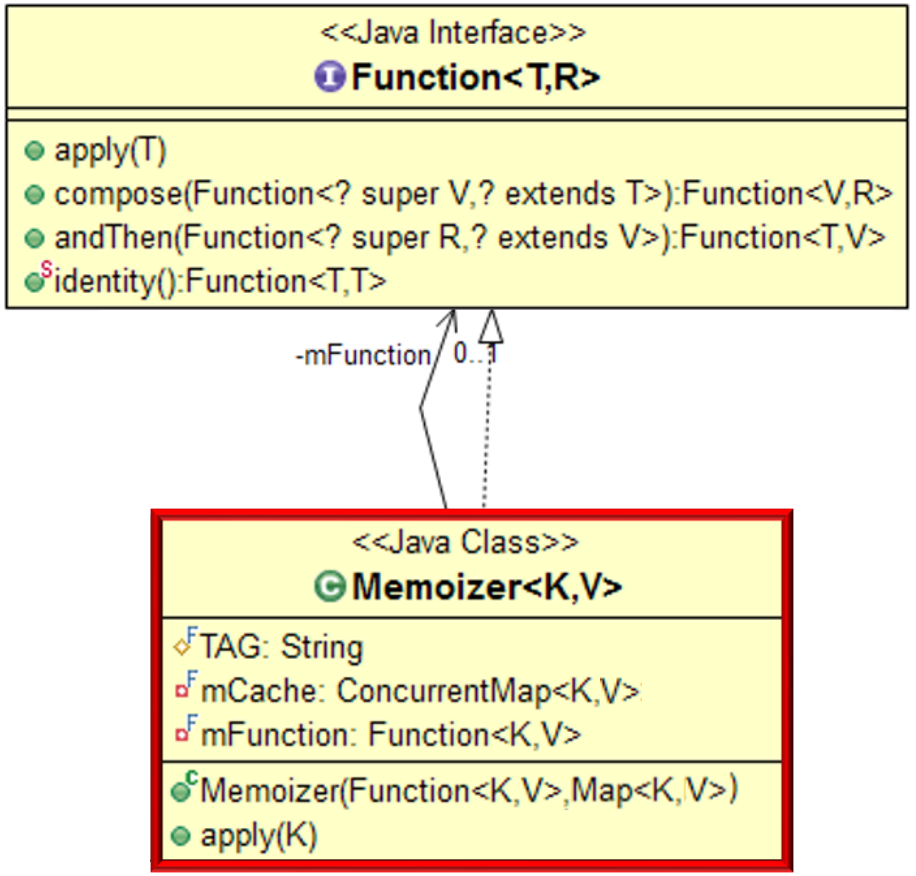
Memoizer



# Designing a Memoizer with ConcurrentHashMap

# Designing a Memoizer with ConcurrentHashMap

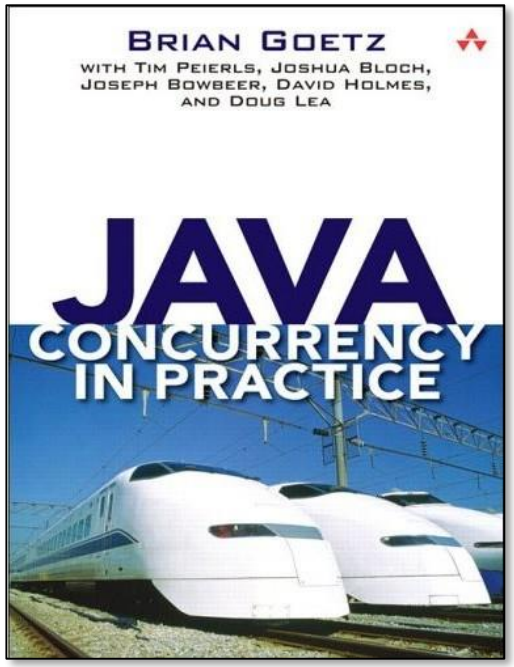
- Memoizer defines a cache that returns a value produced by applying a (long-running) function to a key

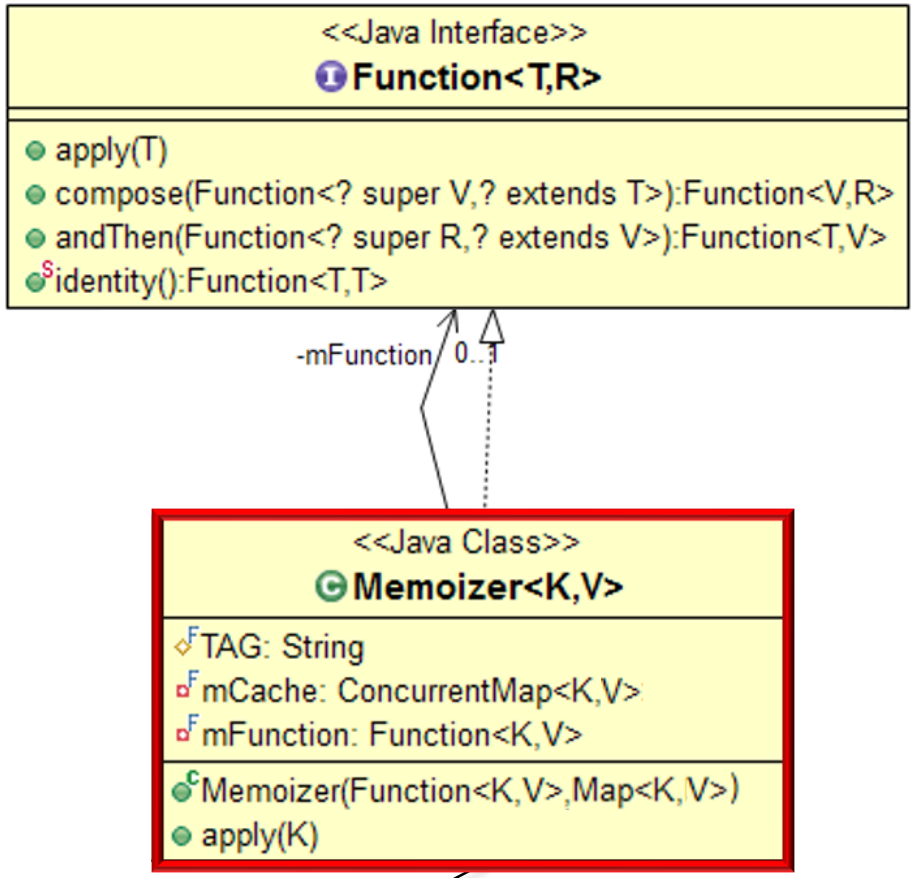



See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java

# Designing a Memoizer with ConcurrentHashMap

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



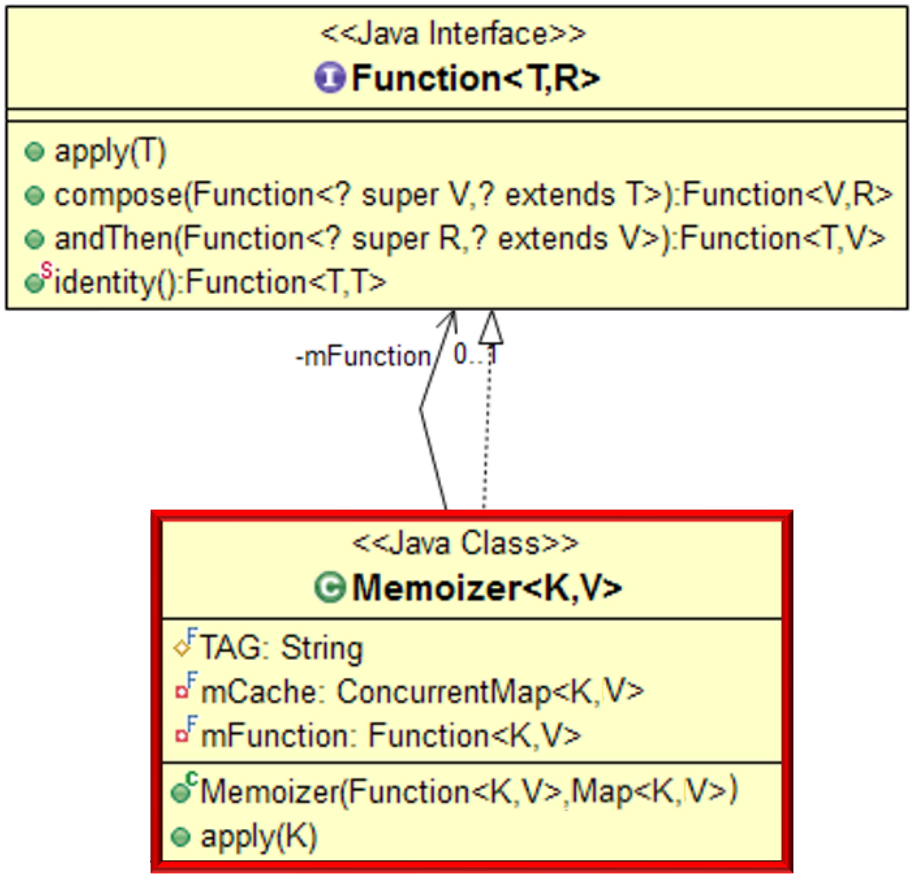




*This class is based on "Java Concurrency in Practice" by Brian Goetz et al.*

See jcip.net

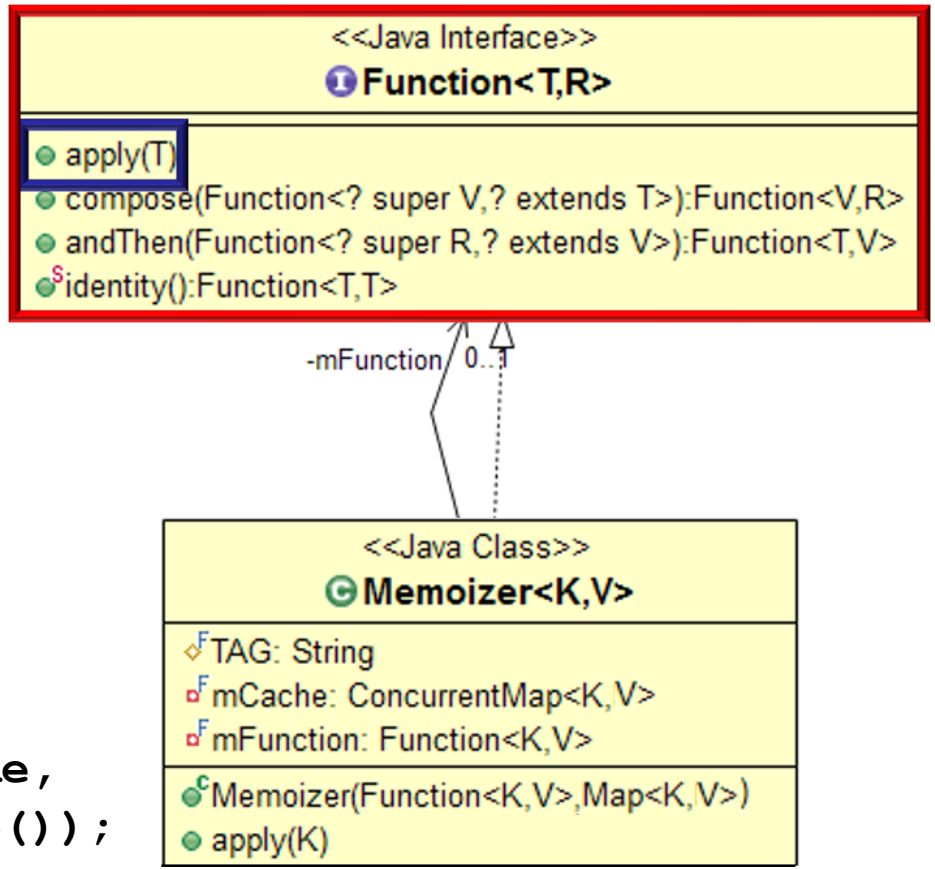
# Designing a Memoizer with ConcurrentHashMap

- 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

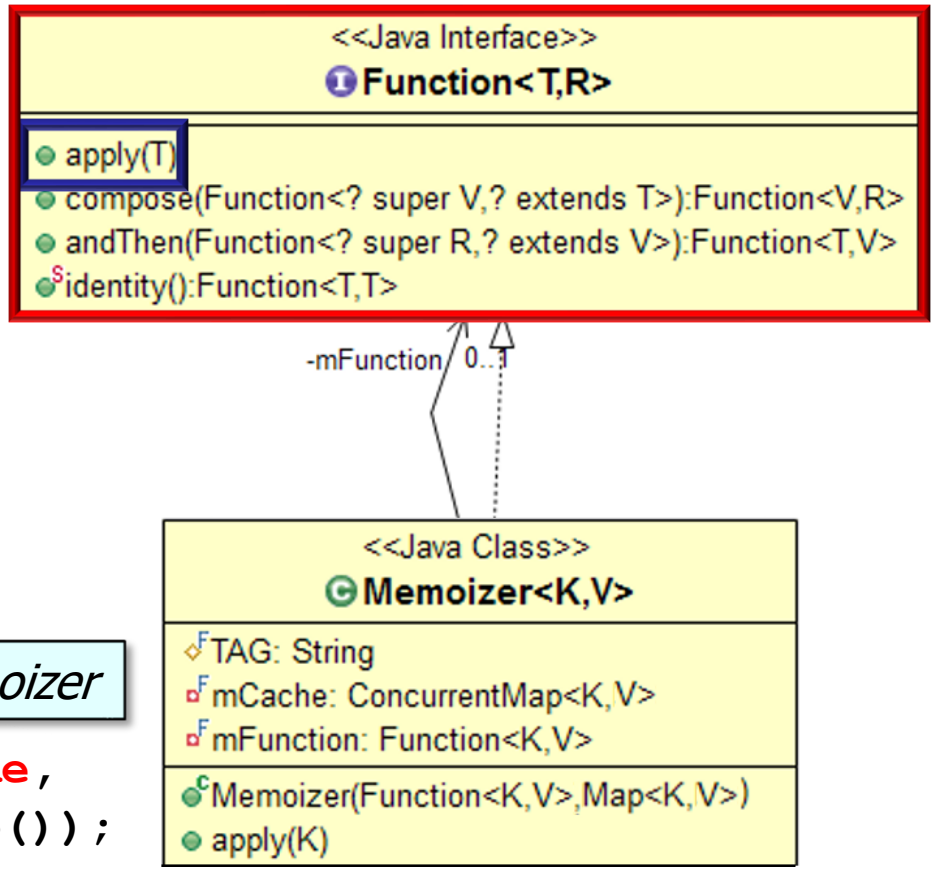



```
Function<Long, Long> func =
  doMemoization
    ? new Memoizer<>
        (PrimeCheckers::isPrime,
         new ConcurrentHashMap());
    : PrimeCheckers::isPrime;

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

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

# Designing a Memoizer with ConcurrentHashMap

- 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




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

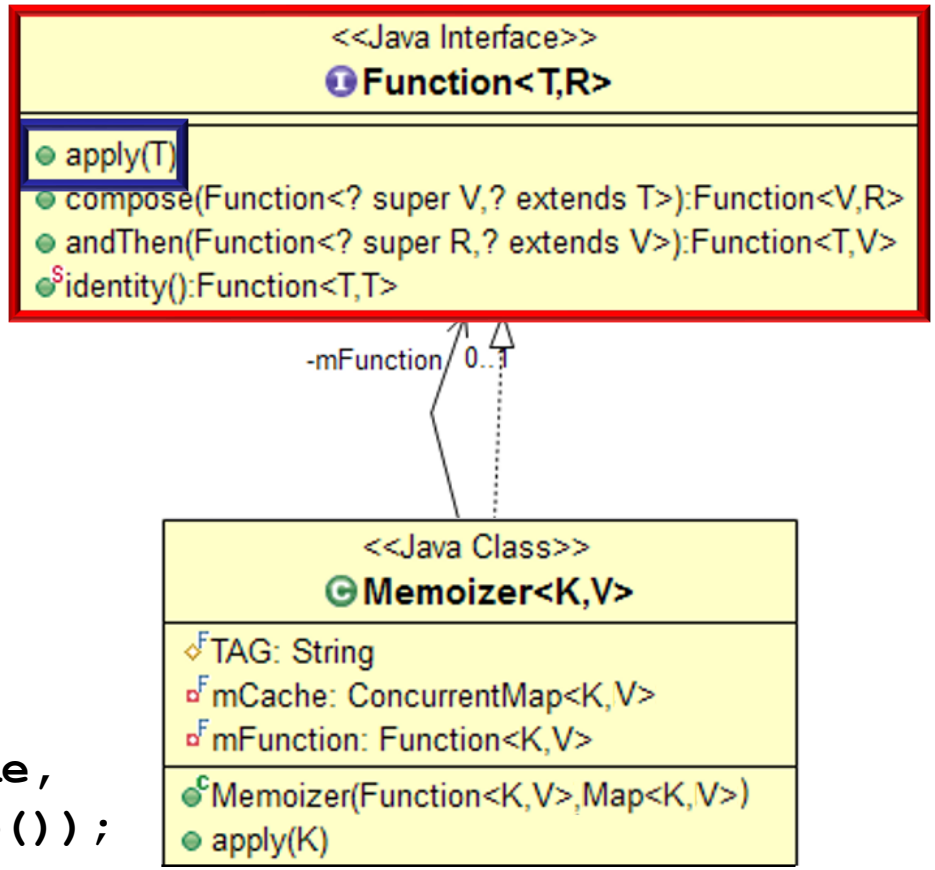
*Use memoizer*

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

See 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




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

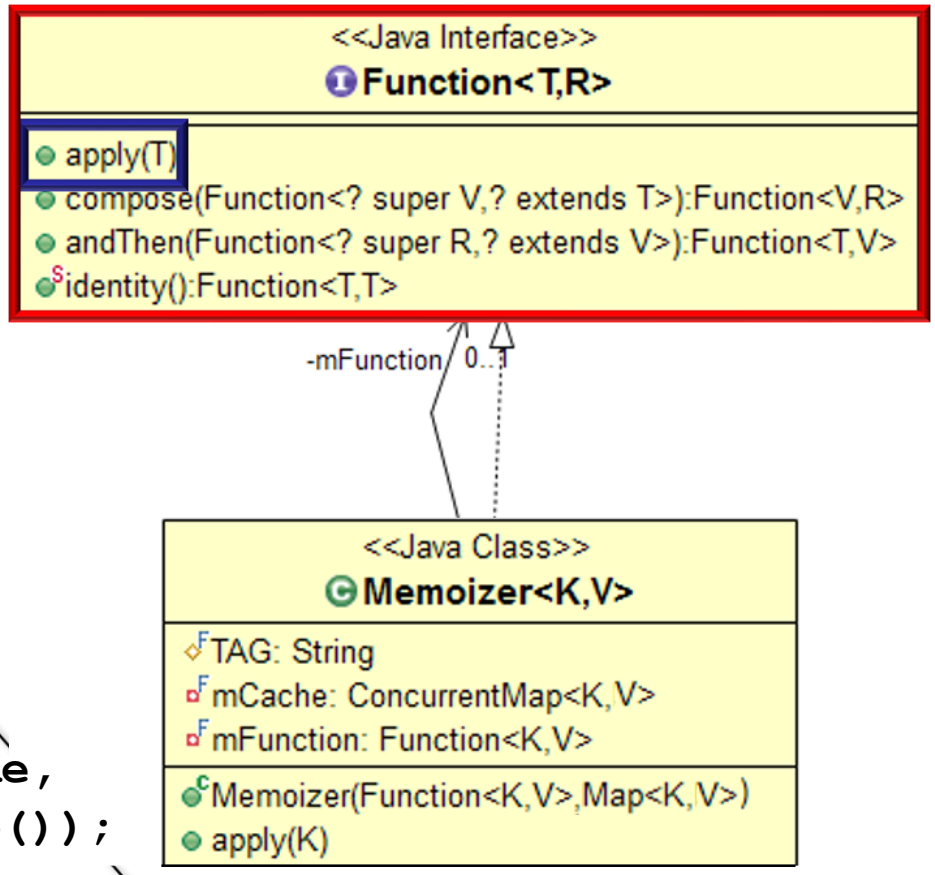
*Don't use memoizer*

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

See 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



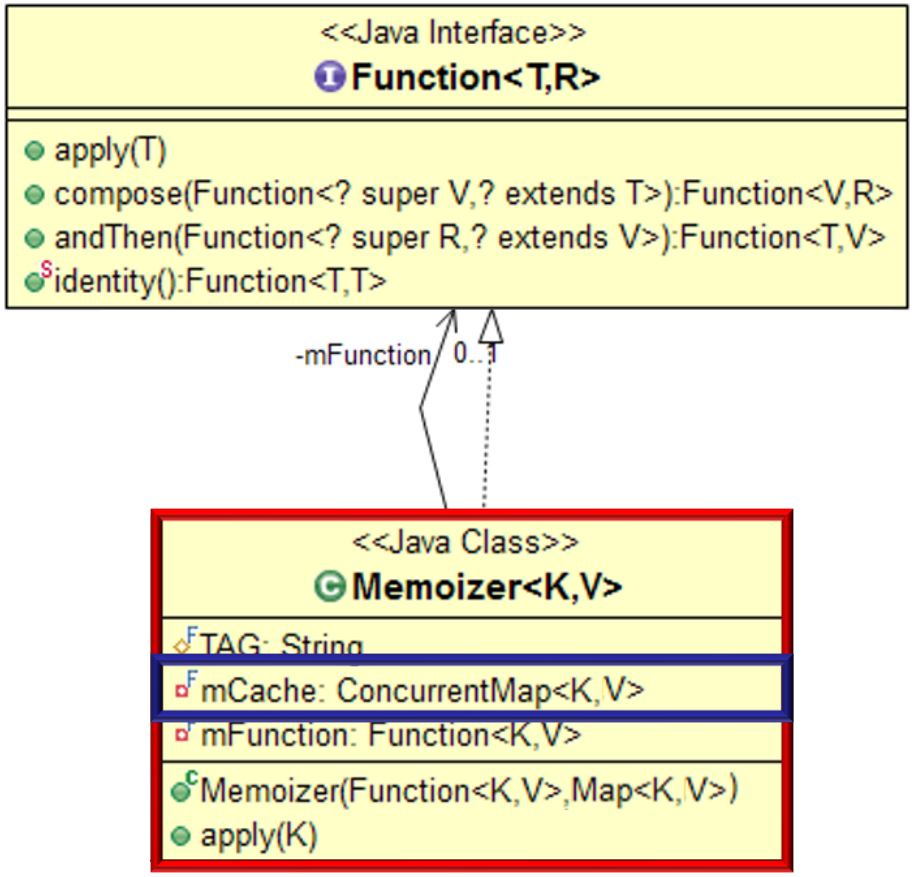

```
Function<Long, Long> func =
   doMemoization
      ? new Memoizer<>
          (PrimeCheckers::isPrime,
           new ConcurrentHashMap());
      : PrimeCheckers::isPrime;

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

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

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

# Designing a Memoizer with ConcurrentHashMap

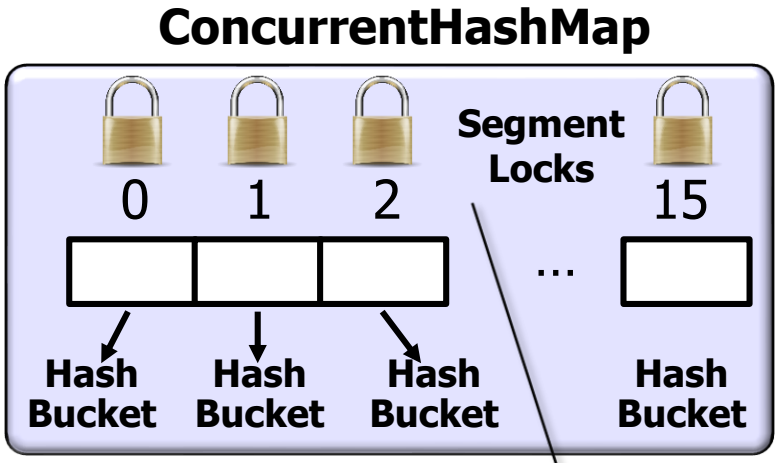
- Memoizer uses a ConcurrentHashMap to minimize synchronization overhead



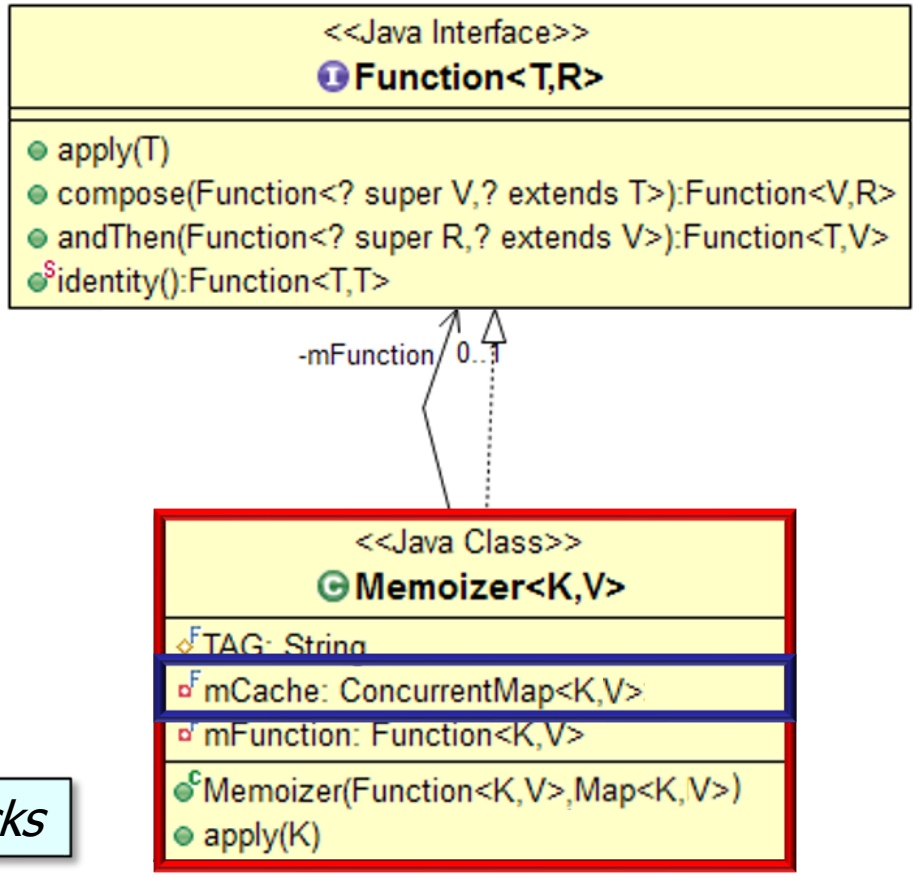

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

# Designing a Memoizer with ConcurrentHashMap

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



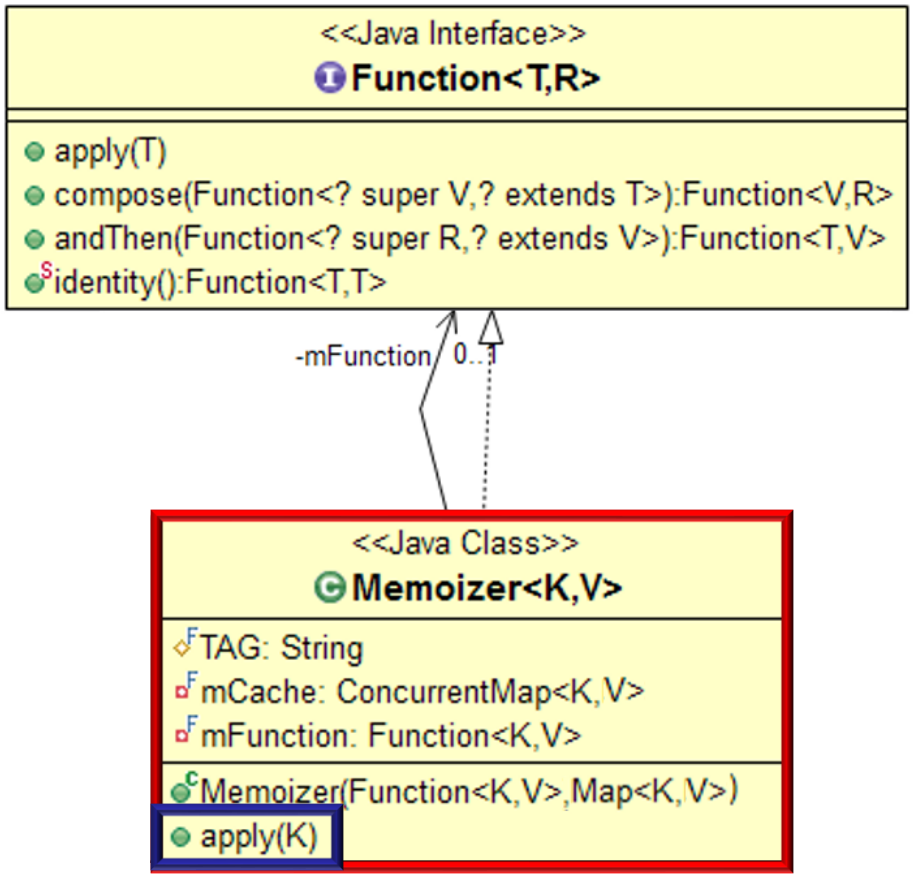

*Contention is low due to use of multiple locks*




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

# 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

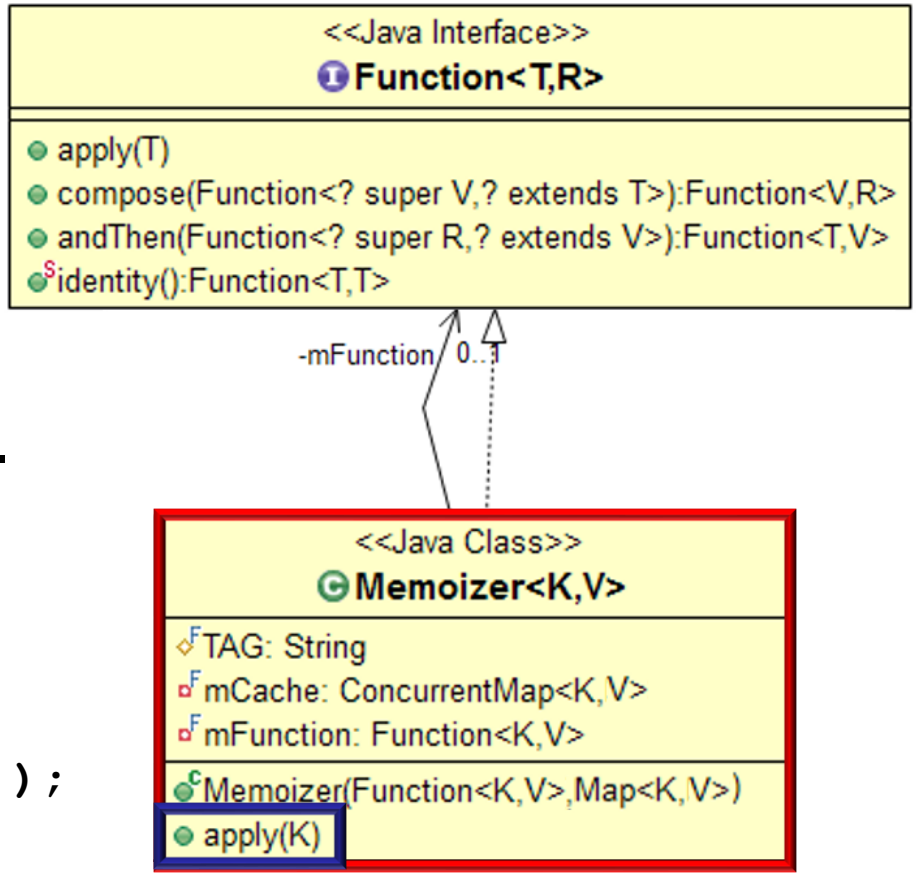



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent

# 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

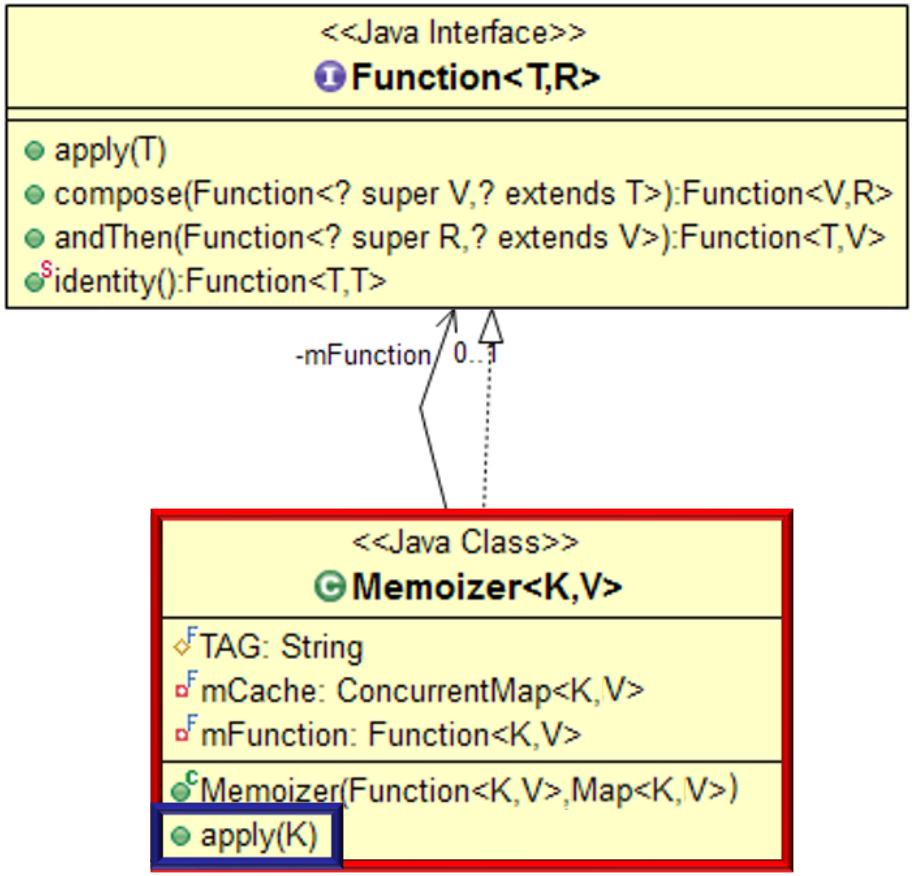
```
return map.computeIfAbsent
   (key,
    k -> new V(mappingFunc(k)));
```




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 Java (non-atomic/-optimized) code

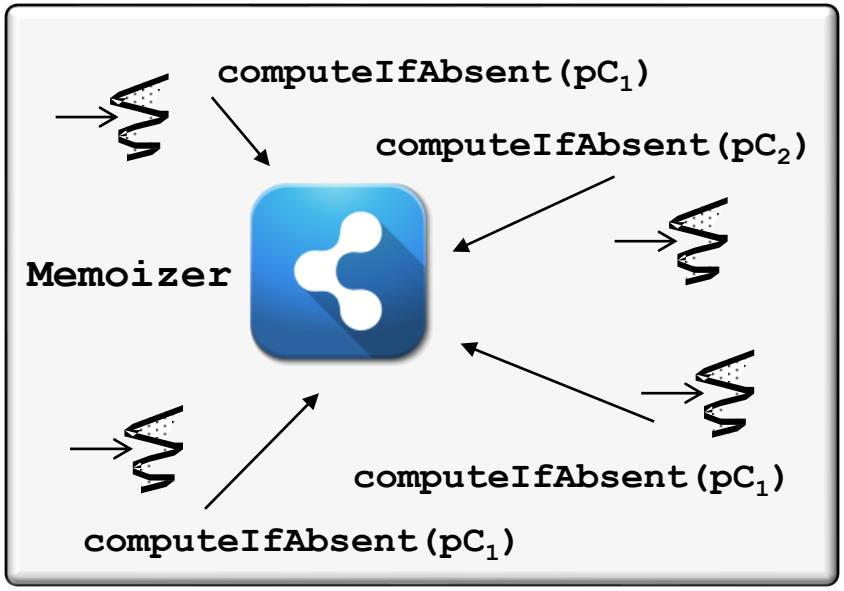



```
V value = map.get(key);
if (value == null) {
  value = mappingFunc.apply(key);
  if (value != null) map.put(key, value);
}
return value;
```

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

# Designing a Memoizer with ConcurrentHashMap

- Memoizer 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 call computeIfAbsent() using the same key*

# End of JavaConcurrent Collections: Designing a Memoizer with ConcurrentHashMap