# The Java FutureTask: Implementing a Memoizer


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

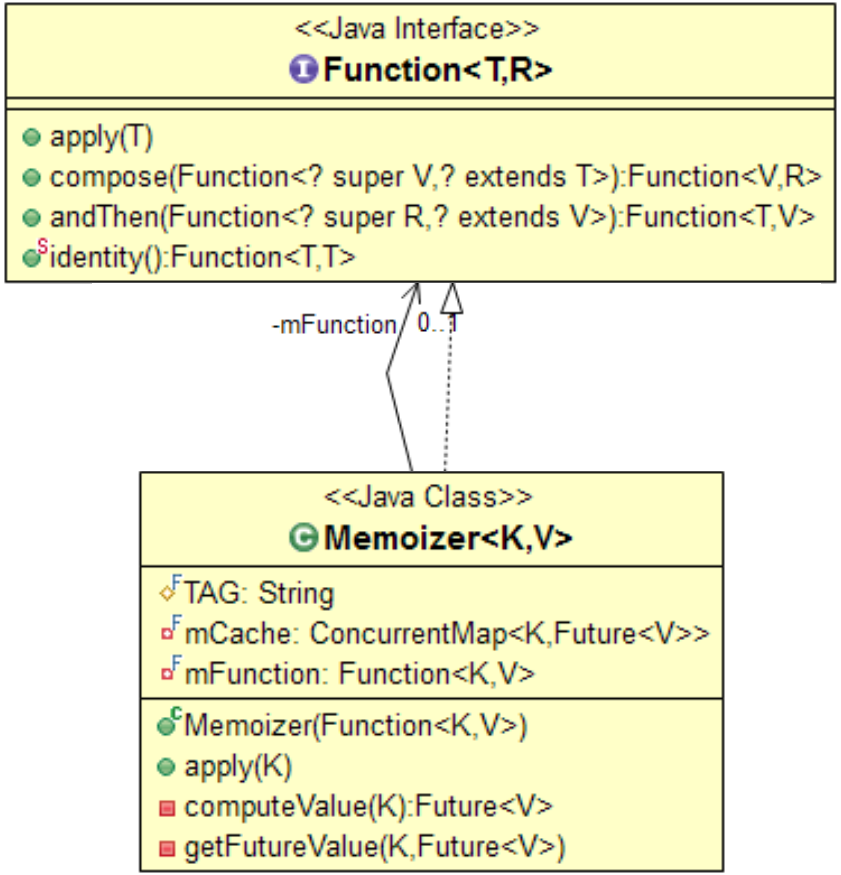
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA


# Learning Objectives in this Part of the Lesson

- Understand how Java FutureTask conveys a result from a computation running in a thread to thread(s) retrieving the result
- Recognize key methods in Java FutureTask
- Know what the Memoizer class is & why it uses FutureTask to optimize programs
- Learn how to implement the Memoizer with FutureTask





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

# Implementing the Memoizer with FutureTask

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final ConcurrentMap<K, Future<V>> mCache =
    new ConcurrentHashMap<>();



  private final Function<K, V> mFunction;



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

  ...
```

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

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final ConcurrentMap<K, Future<V>> mCache =
    new ConcurrentHashMap<>();

  private final Function<K, V> mFunction;

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

  ...
```

*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 with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final ConcurrentMap<K, Future<V>> mCache =
    new ConcurrentHashMap<>();

  private final Function<K, V> mFunction;

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

  ...
```

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

See docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final ConcurrentMap<K, Future<V>> mCache =
    new ConcurrentHashMap<>();


  private final Function<K, V> mFunction;


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

  ...
```

*A Future is used to ensure that the (expensive) function's only called once*

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

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final ConcurrentMap<K, Future<V>> mCache =
    new ConcurrentHashMap<>();

  private final Function<K, V> mFunction;

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

  ...
```

*This function produces a value based on the key*

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
  private final ConcurrentMap<K, Future<V>> mCache =
    new ConcurrentHashMap<>();



  private final Function<K, V> mFunction;


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

  ...
```

*Constructor initializes the mFunction field*

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

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

    Future<V> future = mCache.get(key);

    if (future == null)
       future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

*Returns the value associated with the key in cache*

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

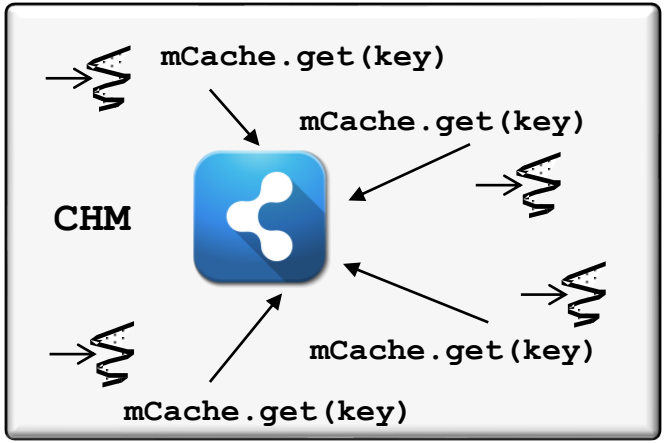
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {

    Future<V> future = mCache.get(key);

    if (future == null)
        future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

*Try to find the key in the cache*




Multiple threads may simultaneously call get() for the same (non-existent) key

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

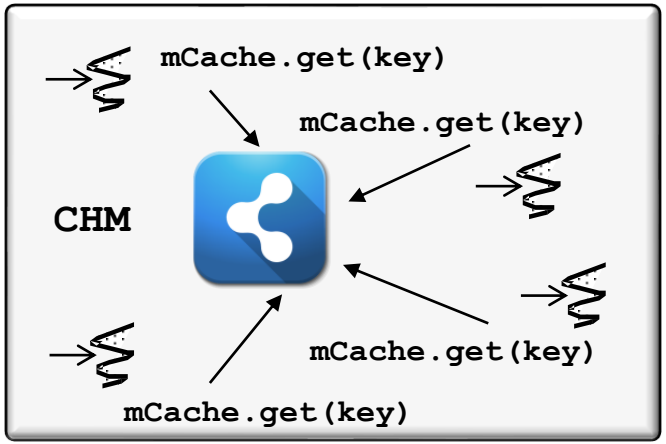
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {

    Future<V> future = mCache.get(key);

    if (future == null)
       future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

*This implementation uses ConcurrentHashMap features that were available prior to Java 8*




See docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html#get

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

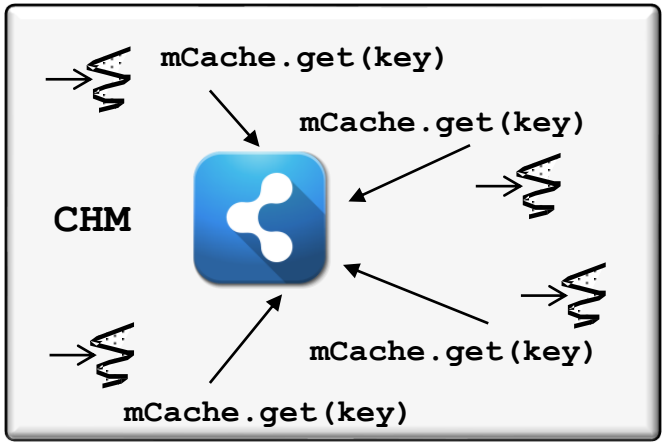
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {

    Future<V> future = mCache.get(key);

    if (future == null)
        future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

*Multiple threads might concurrently get a null future for a non-existent key*



# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

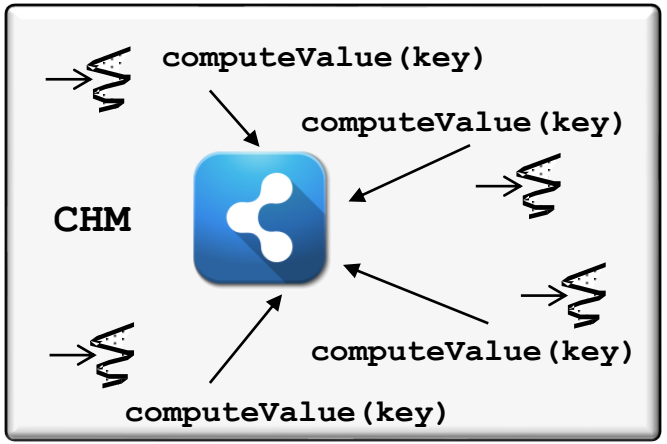
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {

    Future<V> future = mCache.get(key);

    if (future == null)
        future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

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



# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

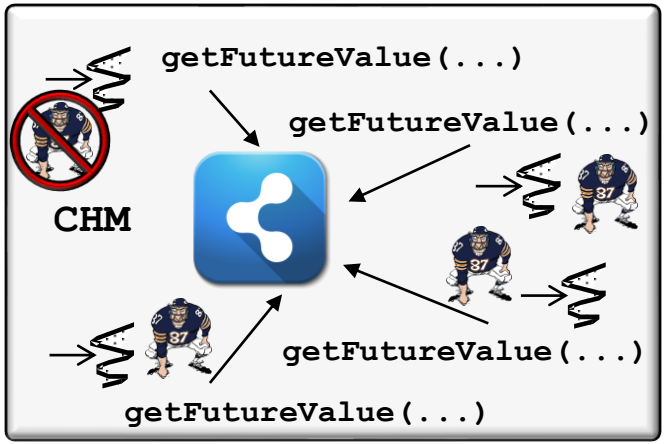
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {

    Future<V> future = mCache.get(key);

    if (future == null)
       future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

*Return the value of the future, blocking until it's computed*




"First thread in" won't block, but other threads *will* block until computation's done

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*Compute value associated with key & return a FutureTask associated with it*

```
    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {

    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```

*FutureTask's constructor is passed a callable lambda*

See docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html#FutureTask

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*FutureTask's run() method invokes callable lambda to compute value & store in cache*

```
    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```

See docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html#run()

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

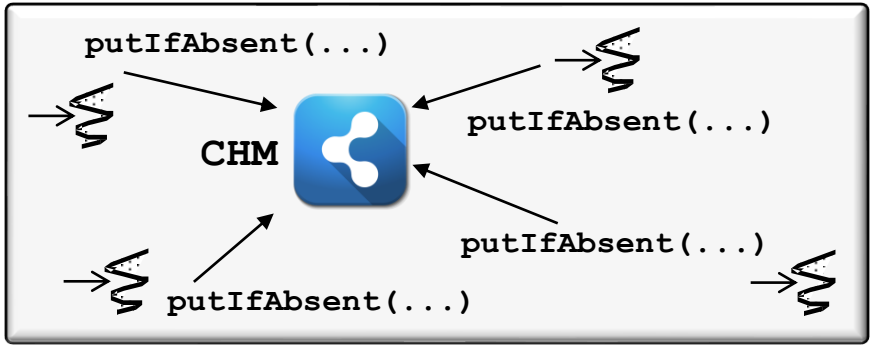
```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*Multiple threads try to atomically add futureTask to cache as value associated w/key*

```
    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```




See docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

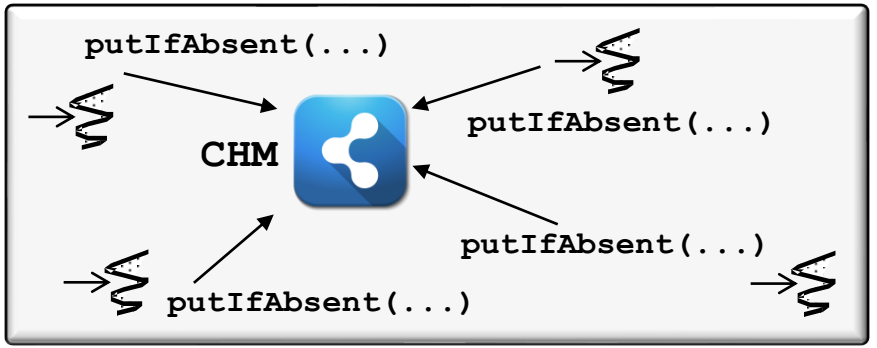
```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*Store the existing value, which is null when called the first time for a given key*

```
    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```



# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

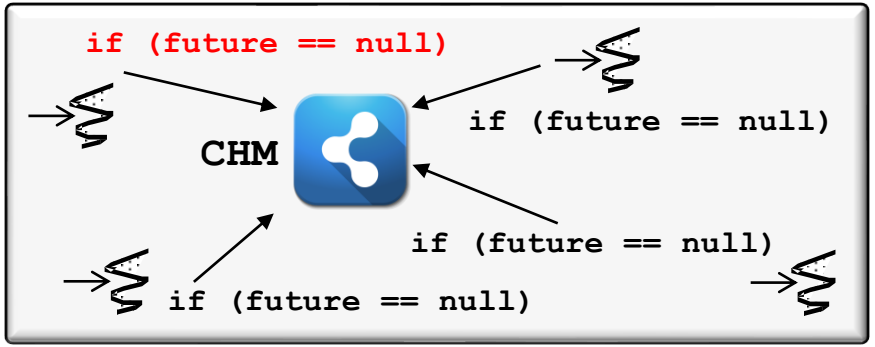
```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*A null indicates key was just added, i.e., value hasn't been computed yet*

```
    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```




Only one thread (i.e., the "first one in") should encounter future == null

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

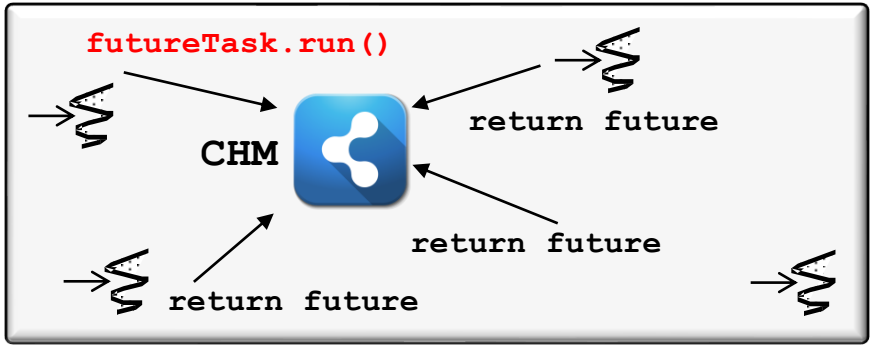
```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*Compute value & store it in the cache when computation's done*

```
   FutureTask<V> futureTask =
     new FutureTask<>(() -> mFunction.apply(key));

   Future<V> future = mCache.putIfAbsent(key, futureTask);

   if (future == null) {
     futureTask.run();
     return futureTask;
   } else return future;
 } ...
```



# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

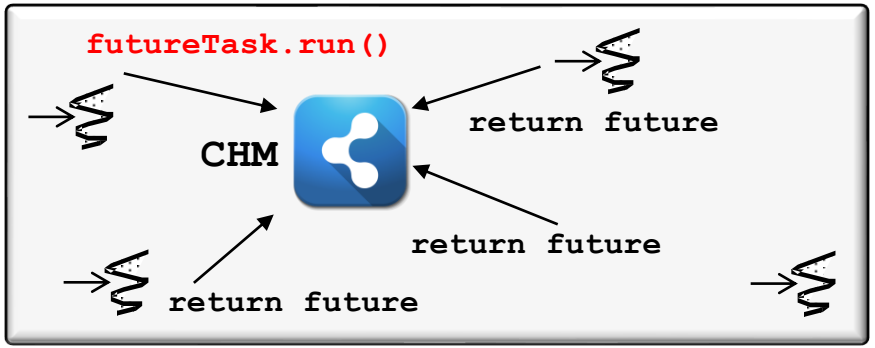
```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {
```

*run() forwards to call(), which forwards to mFunction.apply(key)*

```
   FutureTask<V> futureTask =
     new FutureTask<>(() -> mFunction.apply(key));

   Future<V> future = mCache.putIfAbsent(key, futureTask);

   if (future == null) {
     futureTask.run();
     return futureTask;
   } else return future;
 } ...
```




See earlier part of this lesson on "*Java FutureTask: Key Methods*"

# Implementing the Memoizer with FutureTask

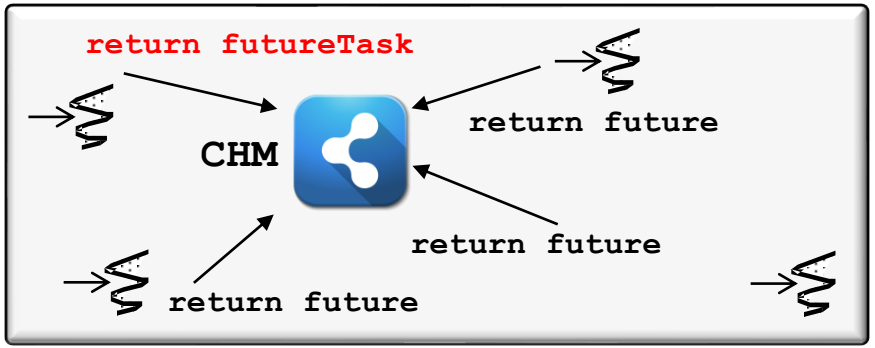
- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {


    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```




*Return a future to the task that's completed*

# Implementing the Memoizer with FutureTask

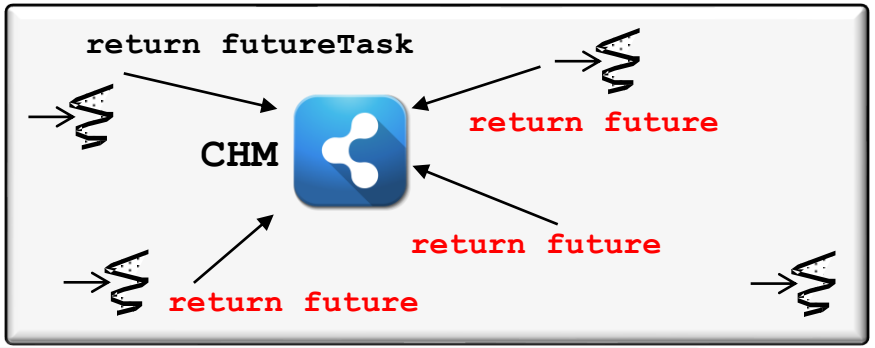
- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private Future<V> computeValue(K key) {


    FutureTask<V> futureTask =
      new FutureTask<>(() -> mFunction.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);

    if (future == null) {
      futureTask.run();
      return futureTask;
    } else return future;
  } ...
```




*If future != null then value was already in cache, so just return it*

# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

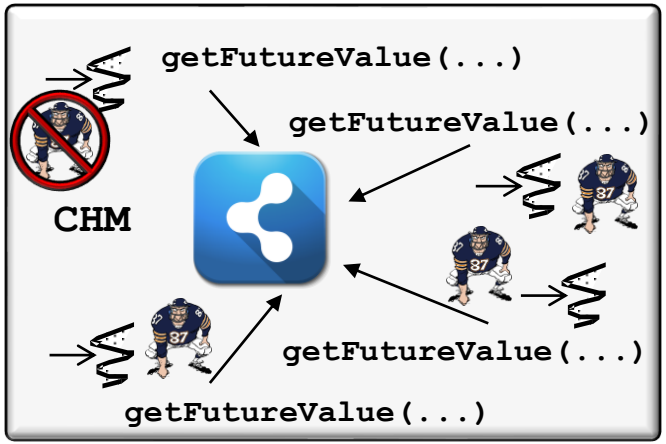
```
class Memoizer<K, V> implements Function<K, V> {
  public V apply(K key) {

    Future<V> future = mCache.get(key);

    if (future == null)
       future = computeValue(key);

    return getFutureValue(key, future);
  }
  ...
```

*Return the value of the future, blocking until it's computed*




"First thread in" won't block, but other threads *will* block until computation's done

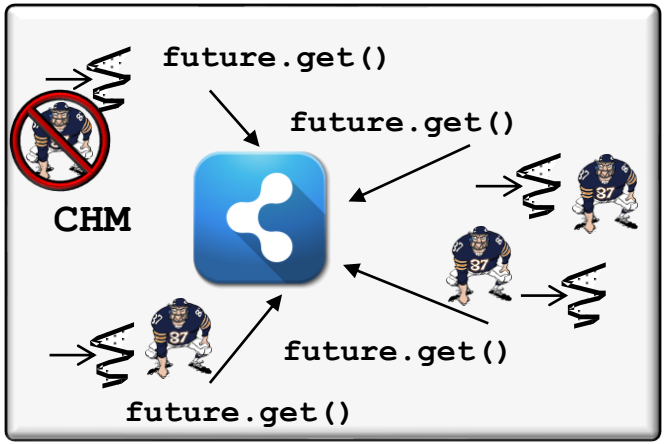
# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private V getFutureValue(K key, Future<V> future) {
```

*Return the value of the future, blocking until it's computed*

```
    ...
      return future.get();
    ...
  } ...
```

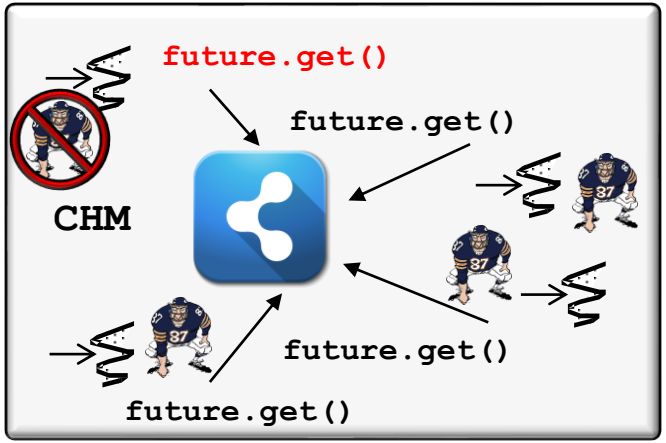


# Implementing the Memoizer with FutureTask

- Memoizer uses FutureTask to ensure a computation only runs once

```
class Memoizer<K, V> implements Function<K, V> {
 private V getFutureValue(K key, Future<V> future) {
    ...
      return future.get();
    ...
  } ...
```

*Get the result of the future, which blocks if the future hasn't finished running*




"First thread in" won't block, but other threads *will* block until computation's done

# End of Java FutureTask: Implementing a Memoizer