The Java FutureTask

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 how the Java FutureTask conveys the result from the thread executing a computation to thread(s) retrieving the result
Learning Objectives in this Lesson

- Understand how the Java FutureTask conveys the result from the thread executing a computation to thread(s) retrieving the result.
- Know how the Memoizer class uses FutureTask to optimize programs by caching results of expensive function calls & returning cached results when the same inputs occur again.
Overview of Java FutureTask
FutureTask conveys the result from the thread executing a computation to thread(s) retrieving the result.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html
Overview of Java FutureTask

- FutureTask implements RunnableFuture & provides several capabilities

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html
Overview of Java FutureTask

- FutureTask implements RunnableFuture & provides several capabilities, e.g.
  - Start & cancel a computation that runs asynchronously

```
Java Interface
Runnable
- run(): void

Java Interface
RunnableFuture<V>
- run(): void

Java Interface
Future<V>
- cancel(boolean): boolean
- isCancelled(): boolean
- isDone(): boolean
- get()
- get(long, TimeUnit)

Java Class
FutureTask<V>
- FutureTask(Callable<V>)
- FutureTask(Runnable, V)
- isCancelled(): boolean
- isDone(): boolean
- cancel(boolean): boolean
- get()
- get(long, TimeUnit)
- run(): void
```

FutureTask computations are often run by a Java ThreadPoolExecutor
Overview of Java FutureTask

- FutureTask implements RunnableFuture & provides several capabilities, e.g.
  - Start & cancel a computation that runs asynchronously
  - Query to see if computation is complete or has been cancelled
FutureTask implements RunnableFuture & provides several capabilities, e.g.:

- Start & cancel a computation that runs asynchronously
- Query to see if computation is complete or has been cancelled
- Retrieve the result of the computation
## Overview of Java FutureTask

- **Key methods in FutureTask**

  ```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask()
      (Callable<V> callable) {...}

    public void run() {...}

    public boolean isCancelled() {...}

    public boolean isDone() {...}

    public boolean cancel(boolean mayInterruptIfRunning) {...}

    public V get() {...}

    protected void done() {...}

    ...
  }
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html)
Overview of Java FutureTask

- Key methods in FutureTask, e.g.
  - Constructs a future task with a Callable to run later
  - The run() method invokes call() on the Callable param

```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask(Callable<V> callable) {...}
    public void run() {...}
    public boolean isCancelled() {...}
    public boolean isDone() {...}
    public boolean cancel(boolean mayInterruptIfRunning) {...}
    public V get() {...}
    protected void done() {...}
    ...
}
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html
Overview of Java `FutureTask`

- Key methods in `FutureTask`, e.g.
  - Constructs a future task with a `Callable` to run later
  - Runs the task
    - i.e., forward to the `call()` method on the `Callable`

```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask (Callable<V> callable) {...}
    public void run() {...}
    public boolean isCancelled() {...}
    public boolean isDone() {...}
    public boolean cancel(boolean mayInterruptIfRunning) {...}
    public V get() {...}
    protected void done() {...}
    ...
}
```

Sets the future task to the result of its computation unless it’s been cancelled
Overview of Java FutureTask

- Key methods in FutureTask, e.g.
  - Constructs a future task with a Callable to run later
  - Runs the task
  - Checks whether the future task has finished normally or was cancelled

```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask(
        Callable<V> callable) {...}
    public void run() {...}
    public boolean isCancelled() {...}
    public boolean isDone() {...}
    public boolean cancel(boolean mayInterruptIfRunning) {...}
    public V get() {...}
    protected void done() {...}
    ...
```
Overview of Java FutureTask

- Key methods in FutureTask, e.g.
  - Constructs a future task with a Callable to run later
  - Runs the task
  - Checks whether the future task has finished normally or was cancelled
  - Cancels a future task

```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask
        (Callable<V> callable) {...}

    public void run() {...}

    public boolean isCancelled() {...}

    public boolean isDone() {...}

    public boolean cancel(boolean mayInterruptIfRunning) {...}

    public V get() {...}

    protected void done() {...}
    ...
```
Overview of Java FutureTask

- Key methods in FutureTask, e.g.
  - Constructs a future task with a Callable to run later
  - Runs the task
  - Checks whether the future task has finished normally or was cancelled
  - Cancels a future task
  - Gets the result

```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask (Callable<V> callable) {...}
    public void run() {...}
    public boolean isCancelled() {...}
    public boolean isDone() {...}
    public boolean cancel(boolean mayInterruptIfRunning) {...}
    public V get() {...}
    protected void done() {...}
    ...
}
```

This call blocks until the task is finished its computations – there’s also a timed version
Overview of Java FutureTask

- Key methods in FutureTask, e.g.
  - Constructs a future task with a Callable to run later
  - Runs the task
  - Checks whether the future task has finished normally or was cancelled
  - Cancels a future task
  - Gets the result
  - Hook method called back after run() returns

```java
class FutureTask<V> implements RunnableFuture<V> {
    public FutureTask (Callable<V> callable) {...}
    public void run() {...}
    public boolean isCancelled() {...}
    public boolean isDone() {...}
    public boolean cancel(boolean mayInterruptIfRunning) {...}
    public V get() {...}
    protected void done() {...}
    ...
}
```

Subclasses may override this method to invoke completion callbacks or perform bookkeeping
Motivating FutureTask with a Memoizer
Motivating FutureTask with a Memoizer

- Memoization is optimization technique used to speed up programs by caching the results of expensive function calls & returning the cached result when the same inputs occur again

See en.wikipedia.org/wiki/Memoization
Motivating FutureTask with a Memoizer

- This class defines a "memoizing" cache that maps a key to the value produced by a function.
Motivating FutureTask with a Memoizer

• This class defines a "memoizing" cache that maps a key to the value produced by a function
• If a value has been computed for a key it is returned rather than calling the function to compute it again
This class defines a "memoizing" cache that maps a key to the value produced by a function.

- If a value has been computed for a key it is returned rather than calling the function to compute it again.
- A ConcurrentHashMap is used to minimize synchronization overhead.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html)
This class defines a "memoizing" cache that maps a key to the value produced by a function.

- If a value has been computed for a key it is returned rather than calling the function to compute it again.
  - A ConcurrentHashMap is used to minimize synchronization overhead.
- computeValue() uses a FutureTask to ensure only one call to the function is run when a key & value are first added to the memoizing cache.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html
Motivating FutureTask with a Memoizer

- This class defines a "memoizing" cache that maps a key to the value produced by a function
  - If a value has been computed for a key it is returned rather than calling the function to compute it again
    - A ConcurrentHashMap is used to minimize synchronization overhead
  - computeValue() uses a FutureTask to ensure only one call to the function is run when a key & value are first added to the memoizing cache

Only one computation per key should be performed even if the multiple threads call computeValue() for the same key
Motivating FutureTask with a Memoizer

- This class defines a "memoizing" cache that maps a key to the value produced by a function.
  - If a value has been computed for a key it is returned rather than calling the function to compute it again.
- By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected.

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html
Applying FutureTask to the Memoizer Class
The Memoizer uses FutureTask to ensure computation only run 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> function) {
        mFunction = function;
    }

    ...

See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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> function) {
        mFunction = function;
    }

    ...
}
```

Memoizer can be used transparently whenever a Function is expected
The Memoizer uses FutureTask to ensure computation only run once

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

    private final Function<K, V> mFunction;

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

    ...

}
Applying FutureTask to the Memoizer Class

• The Memoizer uses FutureTask to ensure computation only run once

```java
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> function) {
        mFunction = function;
    }

    ...
}
```

A Future is used to ensure that the (expensive) function's only called once
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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> function) {
        mFunction = function;
    }

    ...  
```

*This function produces a value based on the key*
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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> function) {
        mFunction = function;
    }

    // Constructor initializes the function field
    ...
}
```
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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*
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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 might concurrently call get() for the same (non-existent) key
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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

Multiple threads might concurrently get a null future for a non-existent key
The Memoizer uses FutureTask to ensure computation only run once.

```java
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.
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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;
    }
}
```

*Compute the value associated with the key & return a FutureTask associated with it*
The Memoizer uses FutureTask to ensure computation only run 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;
    }
}

Create a FutureTask whose run() method forwards to the callable lambda, which computes the value & store it in the cache.
The Memoizer uses FutureTask to ensure computation only run once

```java
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;
    }
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent)
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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;
    }
```

*This call returns the existing value, which will be null when it’s called the first time for a given key*
The Memoizer uses FutureTask to ensure computation only run once.

```java
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;
    }
```

A null indicates the key was just added (it's the "first time in"), which indicates the value hasn't been computed yet.

Only one thread (i.e., the “first one in”) should encounter future == null.
The Memoizer uses FutureTask to ensure computation only run 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;
    }

This run() method forwards to mFunction.apply(key)
The Memoizer uses FutureTask to ensure computation only run 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
Applying FutureTask to the Memoizer Class

- The Memoizer uses FutureTask to ensure computation only run once

```java
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 the value was already in the cache, so just return it.
The Memoizer uses FutureTask to ensure computation only run once.

class Memoizer<K, V> implements Function<K, V> {

    ... 
    private V getFutureValue(K key, Future<V> future) {

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

}
The Memoizer uses FutureTask to ensure computation only run 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
    }
}

The “first thread in” will *not* block here, but any other threads *will* block until the (expensive) computation has completed.
Applying the Memoizer to Check for Prime #’s
• By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.

• PrimeCallable determines if a # is prime

```java
class PrimeCallable implements Callable<PrimeResult> {
    long mPrimeCandidate;
    mFunction<Long, Long> mPrimeChecker;

    PrimeCallable(Long primeCandidate,
                  Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }

    PrimeResult call() {
        return new PrimeResult(mPrimeCandidate, mPrimeChecker
                                 .apply(mPrimeCandidate));
    }
...}
```
By implementing `Function` an instance of `Memoizer` can be used transparently whenever a `Function` is expected, e.g.

PrimeCallable determines if a # is prime

```java
class PrimeCallable implements Callable<PrimeResult> {
    long mPrimeCandidate;
    mFunction<Long, Long> mPrimeChecker;

    PrimeCallable(Long primeCandidate,
                  Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }

    PrimeResult call() {
        return new PrimeResult(mPrimeCandidate, mPrimeChecker
                                .apply(mPrimeCandidate));
    }
}
```

The function that checks primes is passed as a param & stored in a field
Applying the Memoizer to Check for Prime #’s

- By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.
- PrimeCallable determines if a # is prime

```java
class PrimeCallable implements Callable<PrimeResult> {
    long mPrimeCandidate;
    mFunction<Long, Long> mPrimeChecker;

    PrimeCallable(Long primeCandidate,
                   Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }

    PrimeResult call() {
        return new PrimeResult(mPrimeCandidate, mPrimeChecker.apply(mPrimeCandidate));
    }
}
```

The call() hook method applies the function, which returns 0 if it’s prime or smallest factor if it’s not.
Applying the Memoizer to Check for Prime #'s

- By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.
- PrimeCallable determines if a # is prime

```java
class PrimeCallable implements Callable<PrimeResult> {
    long mPrimeCandidate;
    mFunction<Long, Long> mPrimeChecker;

    PrimeCallable(Long primeCandidate, Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }

    PrimeResult call() {
        return new PrimeResult(mPrimeCandidate, mPrimeChecker.apply(mPrimeCandidate));
    }

    ...
```

The PrimeResult tuple matches the prime number candidate with the result of checking for primality.
Applying the Memoizer to Check for Prime #’s

• By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.
  • PrimeCallable determines if a # is prime
  • MainActivity uses PrimeCallable

... 

Function<Long, Long> primeMemoizer = new Memoizer<

(PrimeCheckers::bruteForceChecker);

List<Future<PrimeResult>> futures =

new Random()

.longs(count, 0, Long.MAX_VALUE)

.mapToObj(ranNum -> new PrimeCallable

(ranNum, primeMemoizer))

.map(mRetainedState.mExecutorService::submit)

.collect(toList()); ...
Applying the Memoizer to Check for Prime #’s

- By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.
  - PrimeCallable determines if a # is prime
- MainActivity uses PrimeCallable

```
... 
Function<Long, Long> primeMemoizer = new Memoizer<>(
    (PrimeCheckers::bruteForceChecker);

List<Future<PrimeResult>> futures =
    new Random()
    .longs(count, 0, Long.MAX_VALUE)
    .mapToObj(ranNum -> new PrimeCallable
        (ranNum, primeMemoizer))
    .map(mRetainedState.mExecutorService::submit)
    .collect(toList()); ...
```
By implementing `Function`, an instance of `Memoizer` can be used transparently whenever a `Function` is expected, e.g.

- `PrimeCallable` determines if a number is prime
- `MainActivity` uses `PrimeCallable`

```java
Function<Long, Long> primeMemoizer = new Memoizer<>
    (PrimeCheckers::bruteForceChecker);

List<Future<PrimeResult>> futures =
    new Random()
        .longs(count, 0, Long.MAX_VALUE)
        .mapToObj(ranNum -> new PrimeCallable
            (ranNum, primeMemoizer))
        .map(mRetainedState.mExecutorService::submit)
        .collect(toList()); ...
```

Transforms random numbers into `PrimeCallable`s
By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.

- PrimeCallable determines if a # is prime
- MainActivity uses PrimeCallable

```java
Function<Long, Long> primeMemoizer = new Memoizer<>((PrimeCheckers::bruteForceChecker));

List<Future<PrimeResult>> futures =
    new Random()
    .longs(count, 0, Long.MAX_VALUE)
    .mapToObj(ranNum -> new PrimeCallable(ranNum, primeMemoizer))
    .map(mRetainedState.mExecutorService::submit)
    .collect(toList()); ...
```

Memoizer transparently caches results of prime checking

See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java
• By implementing Function an instance of Memoizer can be used transparently whenever a Function is expected, e.g.
  • PrimeCallable determines if a # is prime
  • MainActivity uses PrimeCallable

... 
Function<Long, Long> primeMemoizer
    = new Memoizer<>((PrimeCheckers::bruteForceChecker);

List<Future<PrimeResult>> futures =
    new Random()
        .longs(count, 0, Long.MAX_VALUE)
            .mapToObj(ranNum -> new PrimeCallable(ranNum, primeMemoizer))
                .map(mRetainedState.mExecutorService::submit)
                    .collect(toList()); ...

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit

Submit a value-returning (two-way) task for execution & returns a future that's used to represent the pending results of the task
Applying the Memoizer to Check for Prime #'s

- By implementing `Function` an instance of `Memoizer` can be used transparently whenever a `Function` is expected, e.g.
  - `PrimeCallable` determines if a # is prime
  - `MainActivity` uses `PrimeCallable`

```java
Function<Long, Long> primeMemoizer = new Memoizer<>()
    (PrimeCheckers::bruteForceChecker);

List<Future<PrimeResult>> futures = new Random()
    .longs(count, 0, Long.MAX_VALUE)
    .mapToObj(ranNum -> new PrimeCallable
        (ranNum, primeMemoizer))
    .map(mRetainedState.mExecutorService::submit)
    .collect(toList()); ...

Collects the results into a list of futures to `PrimeResults`
Java 8 Changes to ConcurrentHashMap
Java 8 Changes to ConcurrentHashMap

- Java 8’s ConcurrentHashMap has a `computeIfAbsent()` method that eliminates the need for FutureTask in many circumstances.

```java
public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
```

If the specified key is not already associated with a value, attempts to compute its value using the given mapping function and enters it into this map unless null. The entire method invocation is performed atomically, so the function is applied at most once per key. Some attempted update operations on this map by other threads may be blocked while computation is in progress, so the computation should be short and simple, and must not attempt to update any other mappings of this map.

**Specified by:**
- `computeIfAbsent` in interface `ConcurrentMap<K,V>`
- `computeIfAbsent` in interface `Map<K,V>`

**Parameters:**
- `key` - key with which the specified value is to be associated
- `mappingFunction` - the function to compute a value

**Returns:**
- the current (existing or computed) value associated with the specified key, or null if the computed value is null

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#computeIfAbsent)
Java 8 Changes to ConcurrentHashMap

- Java 8’s ConcurrentHashMap has a computeIfAbsent() method that eliminates the need for FutureTask in many circumstances.

```java
private Future<V> computeValue(K key) {
    FutureTask<V> ft = new FutureTask<>(() -> mF.apply(key));

    Future<V> future = mCache.putIfAbsent(key, futureTask);
    if (future != null) return future;
    else { futureTask.run(); return futureTask; }
}
```

```java
public V apply(final K key) {
    return mCache.computeIfAbsent(key, mFunction::apply);
}
```

All threads block if the value has not yet been completed by the first task, & after it's completed, those blocked threads will unblock & any future threads calling the method will not block either.
Java 8 Changes to ConcurrentHashMap

Java 8’s ConcurrentHashMap has a computeIfAbsent() method that eliminates the need for FutureTask in many circumstances.

```java
private Future<V>computeValue(K key) {
    FutureTask<V> ft = new FutureTask<>(() -> mF.apply(key));
    Future<V> future = mCache.putIfAbsent(key, futureTask);
    if (future != null) return future;
    else { futureTask.run(); return futureTask; }
}
```

vs.

```java
public V apply(final K key) {
    return mCache.computeIfAbsent(key, mFunction::apply);
}
```

All threads block if the value has not yet been completed by the first task, & after it's completed, those blocked threads will unblock & any future threads calling the method will not block either.

See [ashkrit.blogspot.com/2014/12/what-is-new-in-java8-concurrenthashmap.html](ashkrit.blogspot.com/2014/12/what-is-new-in-java8-concurrenthashmap.html)
End of Java FutureTask