Motivating the Need for
Java 8 Completable Futures (Part 1)

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

• Motivate the need for Java futures

```
<<Java Interface>>

1 Future<V>

• cancel(boolean):boolean
• isCancelled():boolean
• isDone():boolean
• get()
• get(long, TimeUnit)
```

```
Interface Future<V>

Type Parameters:
V - The result type returned by this Future's get method

All Known Subinterfaces:
Response<T>, RunnableFuture<V>, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:
CompletableFuture, CountedCompleter, ForkJoinTask, FutureTask, RecursiveAction, RecursiveTask, SwingWorker

public interface Future<V>

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.
```

Java futures provide the foundation for Java 8 completable futures
Motivating the Need for Futures
Motivating the Need for Futures

- Thus far, behaviors running in aggregate operations have all been *synchronous*.
Motivating the Need for Futures

• Thus far, behaviors running in aggregate operations have all been *synchronous*
• i.e., a behavior borrows the thread of its caller until its computation(s) finish
Motivating the Need for Futures

- Thus far, behaviors running in aggregate operations have all been *synchronous*.
- i.e., a behavior borrows the thread of its caller until its computation(s) finish.
Motivating the Need for Futures

- Synchronous calls have pros & cons
Pros of synchronous calls:

“Intuitive” since they map cleanly onto conventional two-way method patterns.

See www.iro.umontreal.ca/~keller/Layla/remote.pdf
Motivating the Need for Futures

- Cons of synchronous calls:
- May not leverage all the parallelism available in multi-core systems

See www.ibm.com/developerworks/library/j-jvmc3
Motivating the Need for Futures

- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
  - Blocking threads incur overhead
    - e.g., due to context switching, synchronization, data movement, & memory management

See www.ibm.com/developerworks/library/j-jvmc3
Motivating the Need for Futures

- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
  - Blocking threads incur overhead
  - Selecting right number of threads is hard
Motivating the Need for Futures

- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
  - Blocking threads incur overhead
  - Selecting right number of threads is hard

Efficient Performance  
Efficient Resource Utilization
Cons of synchronous calls:

- May not leverage all the parallelism available in multi-core systems
- Blocking threads incur overhead
- Selecting right number of threads is hard

Motivating the Need for Futures

CALLER

searchForWord$_1$

return result$_1$

searchForWord$_2$

return result$_2$

searchForWord$_3$

return return$_3$

CALLEE

Efficient Performance

Efficient Resource Utilization
Motivating the Need for Futures

- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
  - Blocking threads incur overhead
  - Selecting right number of threads is hard

Particularly tricky for I/O-bound programs that need more threads to run efficiently
Motivating the Need for Futures

- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
  - Synchronous calls may need to (dynamically) change the size of the common fork-join pool

See dzone.com/articles/think-twice-using-java-8
Motivating the Need for Futures

• An alternative approach uses asynchronous (async) calls & Java futures

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html
Motivating the Need for Futures

- An alternative approach uses asynchronous (async) calls & Java futures
- Async calls return a future & continue running the computation in the background

See en.wikipedia.org/wiki/Asynchrony_(computer_programming)
Motivating the Need for Futures

- An alternative approach uses asynchronous (async) calls & Java futures
- Async calls return a future & continue running the computation in the background
- A future is a proxy that represents the result of an async computation

See en.wikipedia.org/wiki/Futures_and_promises
Motivating the Need for Futures

- An alternative approach uses asynchronous (async) calls & Java futures
  - Async calls return a future & continue running the computation in the background

- A future is a proxy that represents the result of an async computation
  - e.g., McDonald’s vs Wendy’s

```
12

McDonald's

Wendy's
```

CALLER

```
searchForWord_1

future_1

searchForWord_2

future_2

searchForWord_3

future_3
```

CALLEE
Motivating the Need for Futures

- An alternative approach uses asynchronous (async) calls & Java futures
  - Async calls return a future & continue running the computation in the background
  - A future is a proxy that represents the result of an async computation
  - When the computation completes the future is triggered & the caller can get the result
Motivating the Need for Futures

An alternative approach uses asynchronous (async) calls & Java futures:

- Async calls return a future & continue running the computation in the background.
- A future is a proxy that represents the result of an async computation.
- When the computation completes the future is triggered & the caller can get the result.
- get() returns a result via blocking, polling, or time-bounded blocking.

See [www.nurkiewicz.com/2013/02/javautilconcurrentfuture-basics.html](http://www.nurkiewicz.com/2013/02/javautilconcurrentfuture-basics.html)
• An alternative approach uses asynchronous (async) calls & Java futures
  • Async calls return a future & continue running the computation in the background
  • A future is a proxy that represents the result of an async computation
  • When the computation completes the future is triggered & the caller can get the result
    • get() returns a result via blocking, polling, or time-bounded blocking
  • Results can occur in a different order than the original calls were made
End of Motivating the Need for Java 8 Completable Futures (Part 1)