Motivating the Need for Java 8 Completable Futures (Part 3)

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 by understanding the pros & cons of synchrony & asynchrony

• Know how Java futures provide the foundation for completable futures in Java 8

• Motivate the need for Java 8 completable futures by recognizing the pros & cons with Java futures

<<Java Interface>>

Future&lt;V&gt;

- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long, TimeUnit)
Motivating the Need for Completable Futures
Motivating the Need for Completable Futures

• Pros & cons of async calls with Java futures
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
- May leverage parallelism more effectively with fewer threads

Input Strings to Search

Search Words
"do", "re", "mi", "fa", "so", "la", "ti", "do"

ExecutorCompletionService
Completion Queue

1. submit(task)
2. offer(runnable)
3. take()
4. run(runnable)
5. add()
6. take()
7. take()

See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchTaskGang
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
- May leverage parallelism more effectively with fewer threads, e.g.,
- Queue async computations for execution in a pool of threads

```java
mCompletionService.submit(() ->
    searchForWord(word, input));
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#submit
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads, e.g.,
  - Queue async computations for execution in a pool of threads
  - Automatically tune # of threads

```java
mCompletionService
    .submit(() ->
        searchForWord(word, input));
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool)
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads, e.g.,
    - Queue async computations for execution in a pool of threads
    - Automatically tune # of threads
    - Results can be taken from queue of completed futures

```java
Future<SearchResults> resultF = mCompletionService.take();
resultF.get().print();
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#take](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#take)
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads
  - Can block until the result of an async two-way task is available

```java
String f1 = "62675744/15668936";
String f2 = "609136/913704";

Future<BigFraction> f =
    commonPool().submit(() -> {
        BigFraction bf1 =
            new BigFraction(f1);
        BigFraction bf2 =
            new BigFraction(f2);
        return bf1.multiply(bf2);
    });

...;

BigFraction result = f.get();
```
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads
  - Can block until the result of an async two-way task is available
  - Can also poll or time-block

```java
String f1 = "62675744/15668936";
String f2 = "609136/913704";

Future<BigFraction> f =
    commonPool().submit(() -> {
        BigFraction bf1 = new BigFraction(f1);
        BigFraction bf2 = new BigFraction(f2);
        return bf1.multiply(bf2);
    });

...  
BigFraction result = f.get(n, MILLISECONDS);
```
Motivating the Need for Completable Futures

- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads
  - Can block until the result of an async two-way task is available
  - Can be canceled & tested to see if a task is done

```java
String f1 = "62675744/15668936";
String f2 = "609136/913704";

Future<BigFraction> f = 
    commonPool().submit(() -> {
        BigFraction bf1 = 
            new BigFraction(f1);
        BigFraction bf2 = 
            new BigFraction(f2);
        return bf1.multiply(bf2);
    });

... if (!f.isDone() || !f.isCancelled()) 
    f.cancel();
```
Motivating the Need for Completable Futures

- Cons of async calls with Java futures
  - Limited feature set

<<Java Interface>>

Future\(<V>\)

- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long, TimeUnit)
Motivating the Need for Completable Futures

• Cons of async calls with Java futures
  • Limited feature set
    • *Cannot* be completed explicitly
      • e.g., additional mechanisms like FutureTask are needed

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

- Cons of async calls with Java futures
  - Limited feature set
    - Cannot be completed explicitly
    - Cannot be chained fluently to handle async results

See [en.wikipedia.org/wiki/Fluent_interface](en.wikipedia.org/wiki/Fluent_interface)
Motivating the Need for Completable Futures

• Cons of async calls with Java futures
• Limited feature set
  • Cannot be completed explicitly
  • Cannot be chained fluently to handle async results
  • Cannot be triggered reactively
    • i.e., must (timed-)wait or poll

String f1 = "62675744/15668936";
String f2 = "609136/913704"

Future<BigFraction> f =
    commonPool().submit(() -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
    });
...
BigFraction result = f.get();
    // f.get(10, MILLISECONDS);
    // f.get(0, 0);
Motivating the Need for Completable Futures

• Cons of async calls with Java futures
  • Limited feature set
    • Cannot be completed explicitly
    • Cannot be chained fluently to handle async results
    • Cannot be triggered reactively
      • i.e., must (timed-)wait or poll

String f1 = "62675744/15668936";
String f2 = "609136/913704";

Future<BigFraction> f =
  commonPool().submit(() -> {
    BigFraction bf1 =
      new BigFraction(f1);
    BigFraction bf2 =
      new BigFraction(f2);
    return bf1.multiply(bf2);
  });

BigFraction result = f.get();
// f.get(10, MILLISECONDS);
// f.get(0, 0);

"open mouth, insert foot"

Nearly always the wrong thing to do!!

See crondev.blog/2017/01/23/timeouts-with-java-8-completablefuture-youre-probably-doing-it-wrong
Motivating the Need for Completable Futures

• Cons of async calls with Java futures
• Limited feature set
  • *Cannot* be completed explicitly
  • *Cannot* be chained fluently to handle async results
  • *Cannot* be triggered reactively
  • *Cannot* be treated efficiently as a *collection* of futures

```java
Future<BigFraction> future1 = commonPool().submit(() -> {
  ...
});

Future<BigFraction> future2 = commonPool().submit(() -> {
  ...
});

... future1.get();
future2.get();
```

Can’t wait efficiently for the completion of whichever async computation finishes first
Motivating the Need for Completable Futures

- Cons of async calls with Java futures
  - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently to handle async results
    - *Cannot* be triggered reactively
    - *Cannot* be treated efficiently as a *collection* of futures

In general, it’s awkward & inefficient to “compose” multiple futures
End of Motivating the Need for Java 8 Completable Futures (Part 3)