Evaluating the Pros & Cons of Java Futures

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

- Understand how to program with Java futures

- Motivate the need for Java completable futures by evaluating the pros & cons with Java futures

<<Java Interface>>
① Future<V>

- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
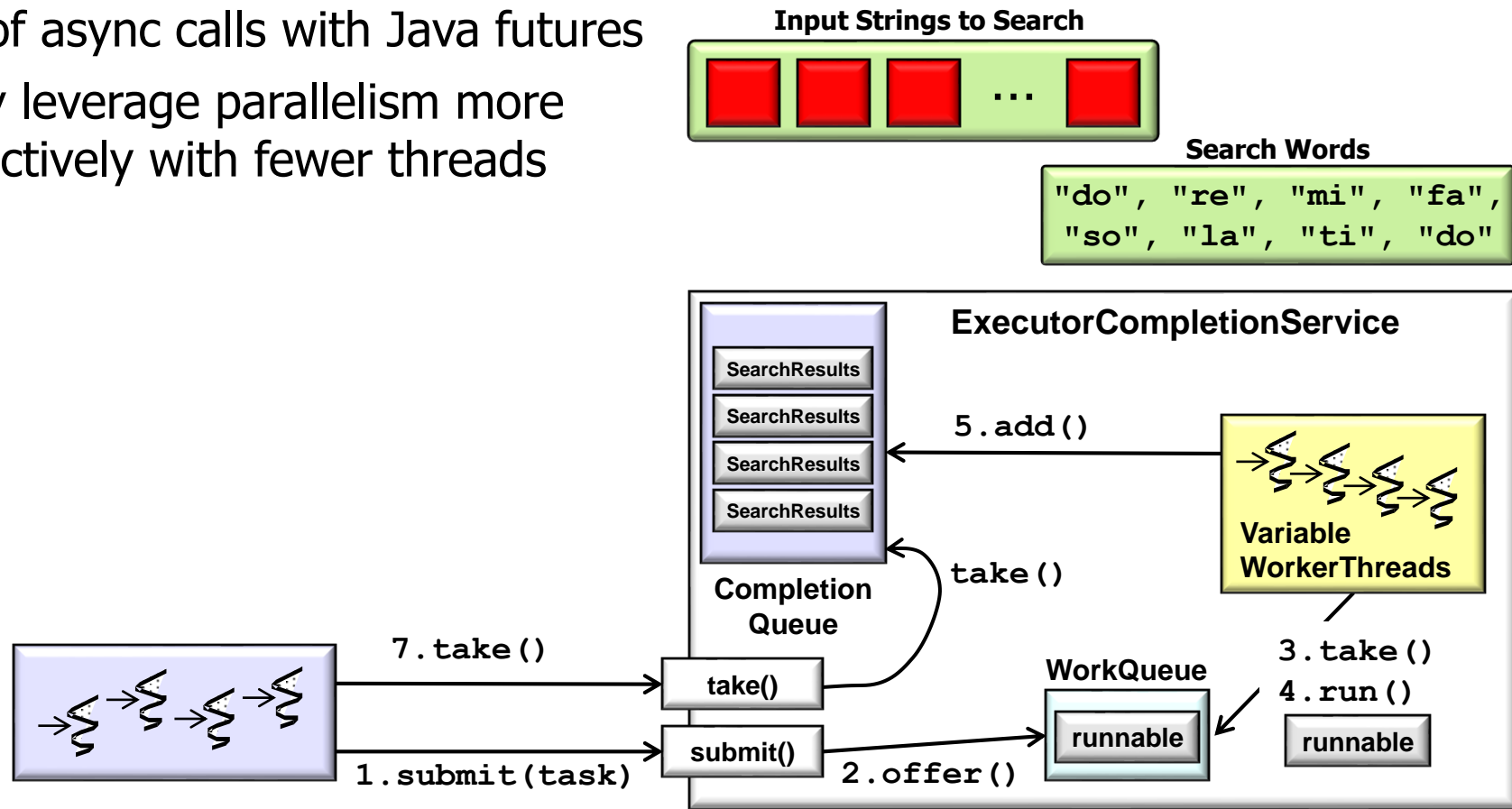- get()
- get(long,TimeUnit)

LIMITED

The Pros of Java Futures

- Pros of async calls with Java futures

The Pros of Java 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

SearchResults
SearchResults
SearchResults
SearchResults

Completion Queue

Variable WorkerThreads

`5.add()`

`take()`

`7.take()`

`take()`

`3.take()`

`4.run()`

WorkQueue

runnable

runnable

`submit()`

`1.submit(task)`

`2.offer()`

See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchTaskGang
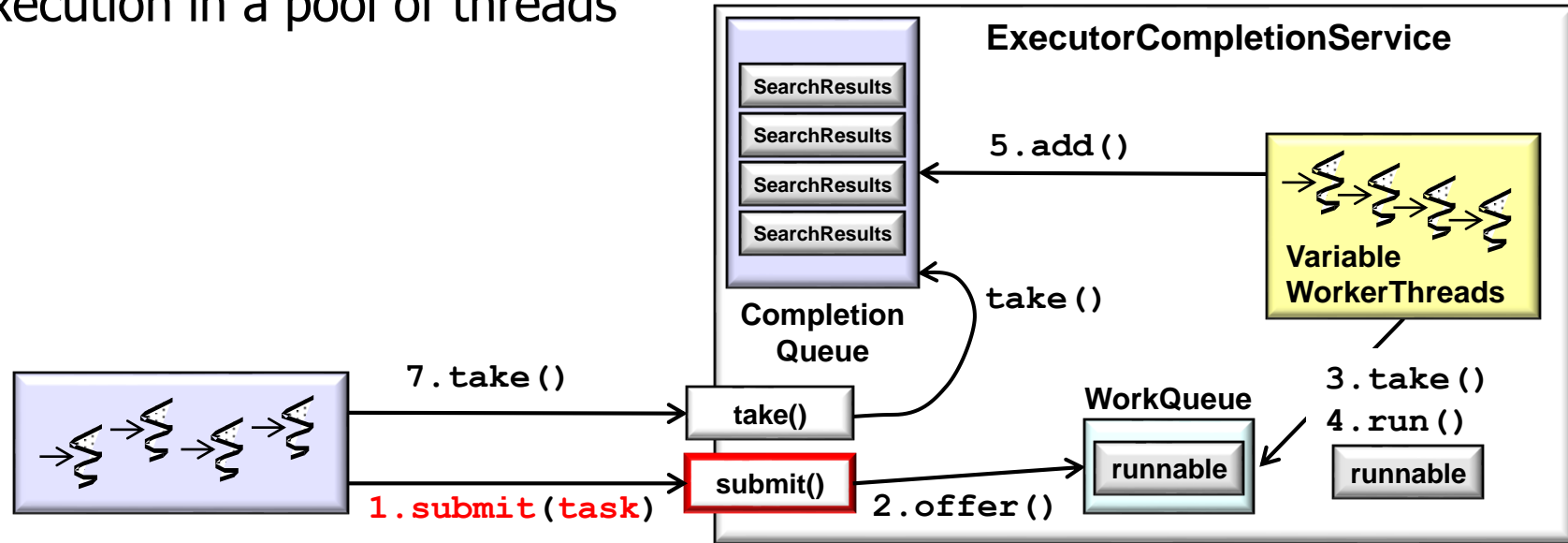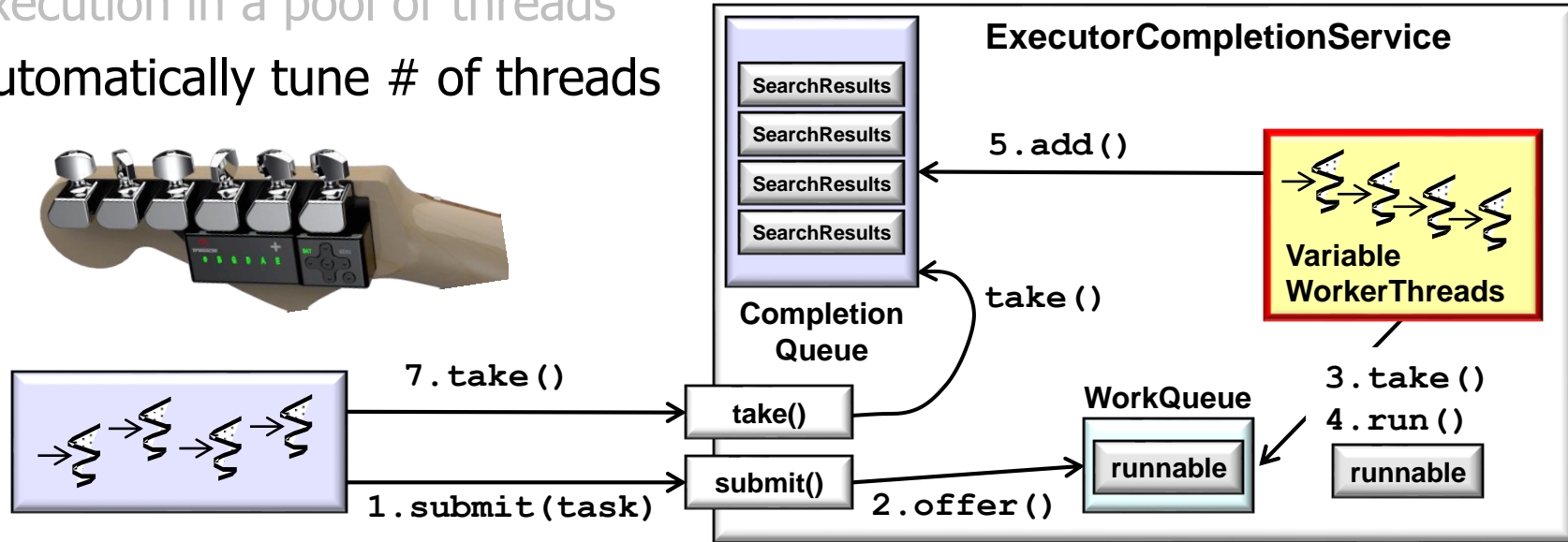
The Pros of Java Futures

- 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

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

The Pros of Java 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

```
mCompletionService
    .submit(() ->
        searchForWord(word,
                      input));
```
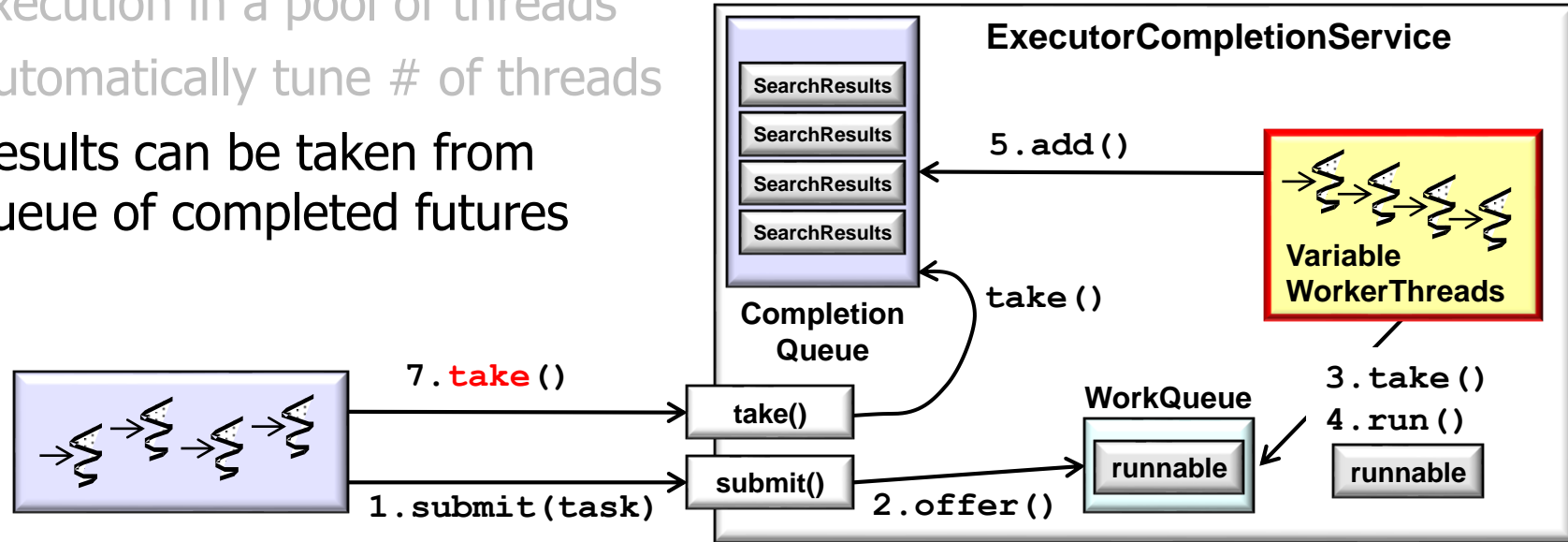
The Pros of Java Futures

- 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

```
Future<SearchResults> resultF =
    mCompletionService.take();
```
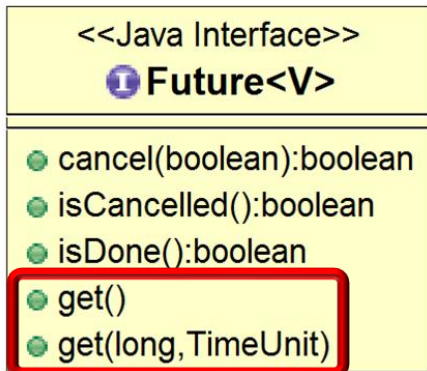
take() blocks, but get() doesn't

```
resultF.get().print()
```

The Pros of Java 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 Interface>>
Future<V>

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

```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();
```
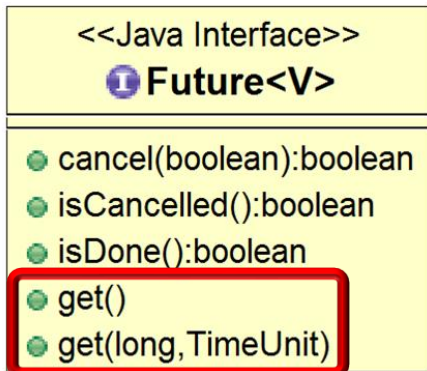
The Pros of Java Futures

- 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 Interface>>
Future<V>

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

```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);
```
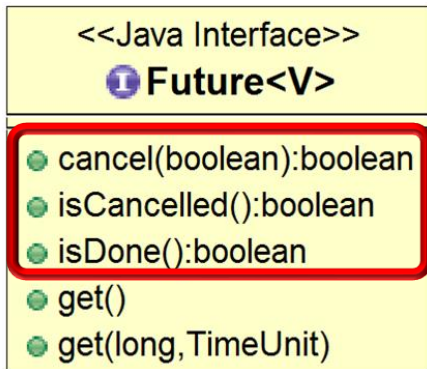
The Pros of Java Futures

- 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 Interface>>
Future<V>

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

```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();
```

The Cons of Java Futures

The Cons of Java Futures

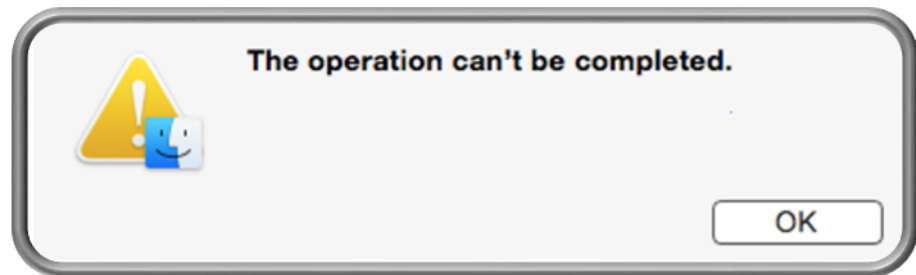- Cons of async calls with Java futures

The Cons of Java Futures

- Cons of async calls with Java futures

  - Limited feature set

The Cons of Java Futures

- Cons of async calls with Java futures

  - Limited feature set

    - *Cannot* be completed explicitly

      - e.g., additional mechanisms like FutureTask are needed

The operation can't be completed.

OK

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

The Cons of Java 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

The Cons of Java 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

TRIGGER WARNING

```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();
   // f.get(10, MILLISECONDS);
   // f.get(0, 0);
```

The Cons of Java 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

"open mouth, insert foot"

Nearly always the wrong thing to do!!

```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();
// f.get(10, MILLISECONDS);
// f.get(0, 0);
```

See crondev.blog/2017/01/23/timeouts-with-java-8-completablefuture-youre-probably-doing-it-wrong

The Cons of Java 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

```
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*

The Cons of Java 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

The Cons of Java Futures

- These limitations with Java futures motivate the need for the Java completable futures framework!

Class CompletableFuture<T>

java.lang.Object
    java.util.concurrent.CompletableFuture<T>

All Implemented Interfaces:

CompletionStage<T>, Future<T>

---

public class **CompletableFuture<T>**
extends Object
implements Future<T>, CompletionStage<T>

A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

See lesson on "*Overcoming Limitations with Java Futures via Java Completable Futures*"

End of Evaluating the Pros & Cons of Java Futures