

# Overview of Reactive Programming with Java 8 Completable Futures

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Lesson

---

- Know what topics we'll cover



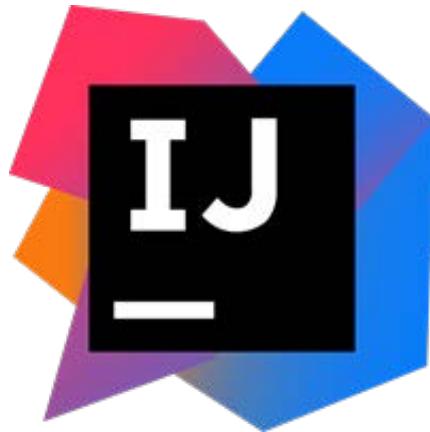
## ***CompletableFuture***



# Learning Objectives in this Lesson

---

- Know what topics we'll cover
- Learn where to find Java 8 & relevant IDEs



# Learning Objectives in this Lesson

---

- Know what topics we'll cover
- Learn where to find Java 8 & relevant IDEs
- Be aware of other digital learning resources



# Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8 & relevant IDEs
- Be aware of other digital learning resources
- Be able to locate examples of Java 8 programs

USE THE  
SOURCE/LIVE  
SOURCE/LIVE



Branch: master   New pull request			Create new file	Upload files	Find file	Clone or download
douglascraigschmidt	updates					Latest commit a67fd89 35 minutes ago
BarrierTaskGang	Updates					10 months ago
BuggyQueue	updates					a day ago
BusySynchronizedQueue	updates					4 months ago
DeadlockQueue	Refactored					3 years ago
ExpressionTree	Updates					10 months ago
Factorials	update					5 days ago
ImageStreamGang	updates					22 days ago
ImageTaskGangApplication	Updates					10 months ago
Java8	update					3 days ago
PalantirManagerApplication						7 months ago
PingPongApplication						10 months ago
PingPongWrong						3 years ago
SearchStreamForkJoin						35 minutes ago
SearchStreamGang						3 hours ago
SearchStreamSpliterator						37 minutes ago
SearchTaskGang						4 months ago
SimpleAtomicLong						2 years ago
SimpleBlockingQueue	Updates					4 months ago
SimpleSearchStream	update					6 days ago
ThreadJoinTest	updates					22 days ago
ThreadedDownloads	Updates					10 months ago
UserOrDaemonExecutor	Refactored					3 years ago
UserOrDaemonRunnable	Updates.					2 years ago
UserOrDaemonThread	Updates					10 months ago
UserThreadInterrupted	Update					2 years ago
.gitattributes	Committed.					3 years ago
.gitignore	Updates					10 months ago
README.md	updates					21 days ago

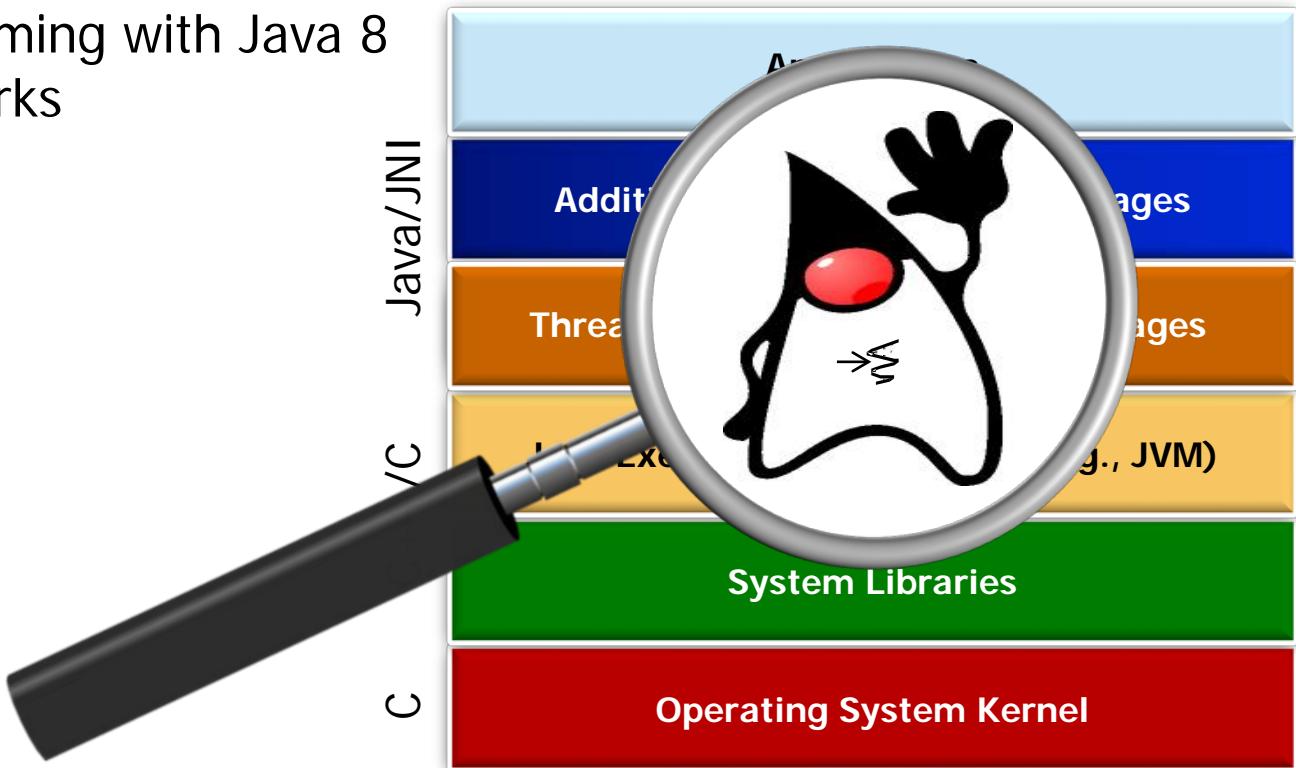
*We'll review many of  
these examples, so  
feel free to clone or  
download this repo!*

---

# Overview of this Course

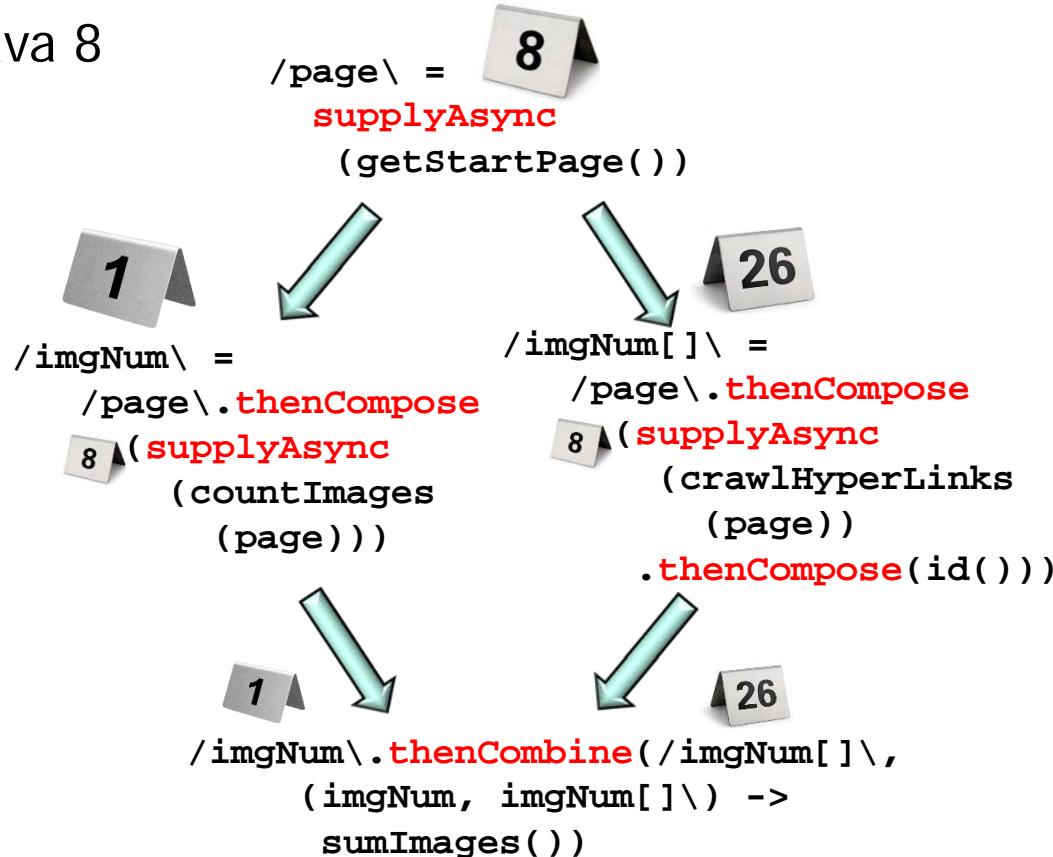
# Overview of this Course

- We focus on programming with Java 8 concurrency frameworks



# Overview of this Course

- We focus on programming with Java 8 concurrency frameworks, e.g.
  - **Competable futures**
    - Support dependent functions that trigger upon completion of async operations

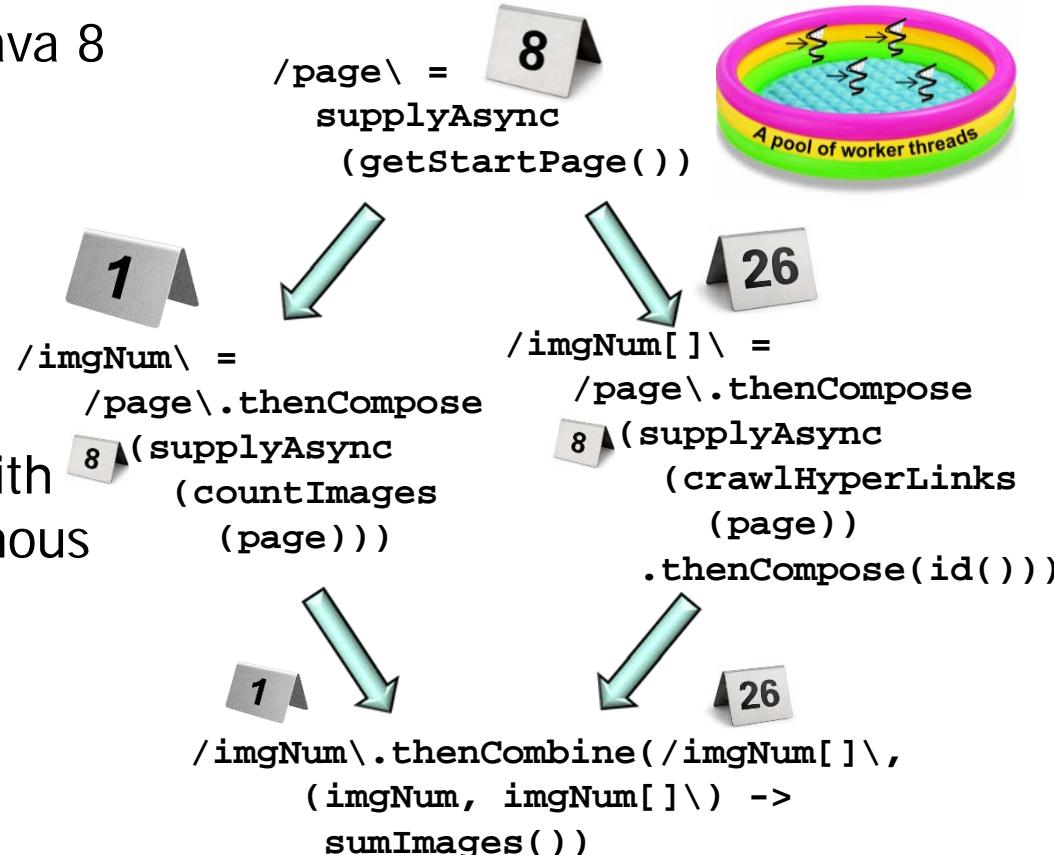


# Overview of this Course

- We focus on programming with Java 8 concurrency frameworks, e.g.

- **Competable futures**

- Support dependent functions that trigger upon completion of asynchronous operations
- Can be used in conjunction with thread pools to run asynchronous operations concurrently



# Overview of this Course

---

- We'll assume you're familiar with core Java 8 functional programming concepts & features
  - e.g., lambda expressions, method references, & functional interfaces



# Overview of this Course

---

- We'll assume you're familiar with core Java 8 functional programming concepts & features
  - e.g., lambda expressions, method references, & functional interfaces



These features are the foundation for Java 8's concurrency/parallelism frameworks

# Overview of this Course

- There are several related Live Training courses

Programming with Java 8 Lambdas and Streams



DOUGLAS SCHMIDT

October 17, 2017

9:00am – 12:00pm CDT

Scalable Programming with Java 8 Parallel Streams



DOUGLAS SCHMIDT

October 19, 2017

9:00am – 12:00pm CDT

No credit card required

START YOUR FREE TRIAL

115 spots available

Registration closes October 18, 2017 5:00 PM

See [www.dre.vanderbilt.edu/~schmidt/DigitalLearning](http://www.dre.vanderbilt.edu/~schmidt/DigitalLearning)

# Overview of this Course

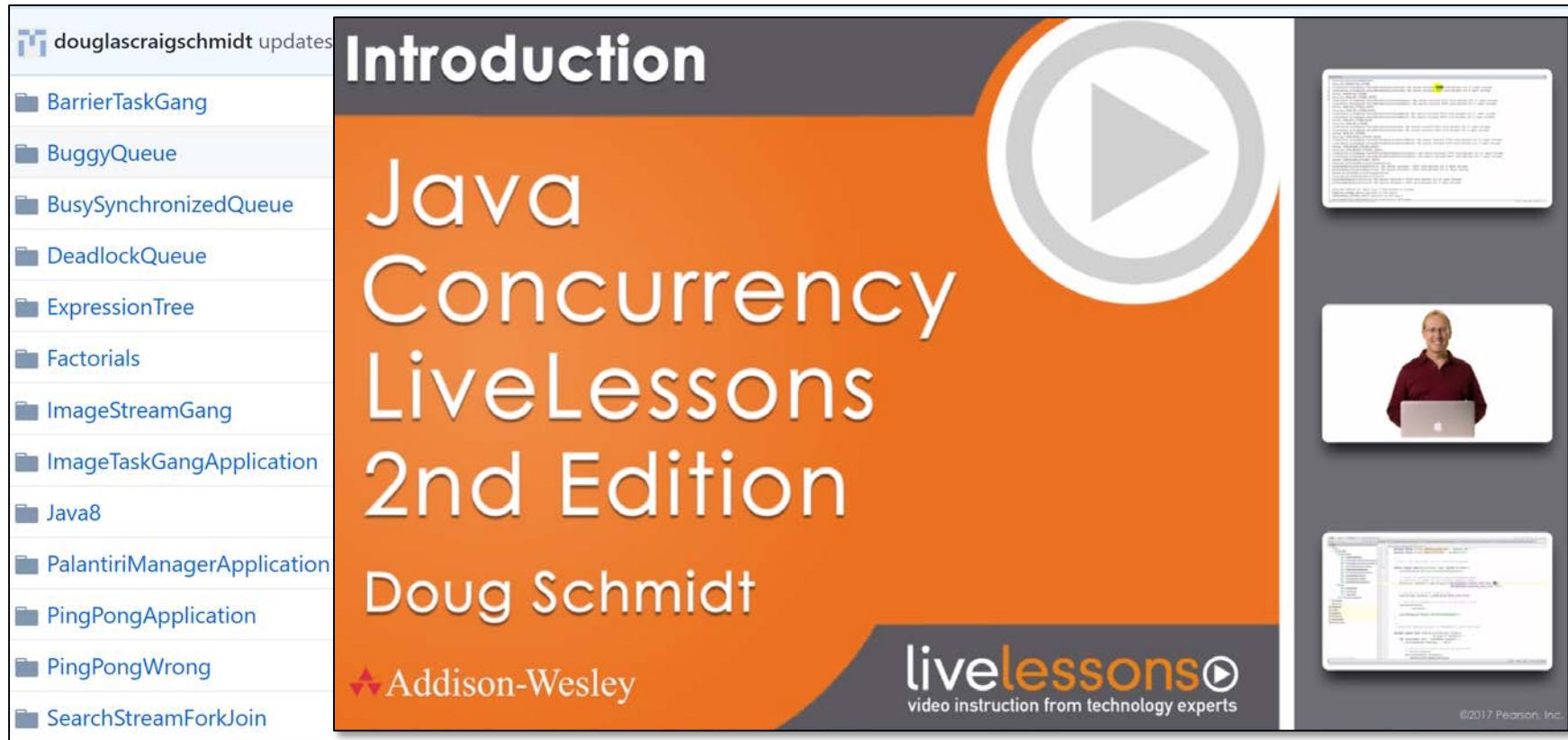
- All Java 8 concurrent examples we cover are available on github

douglascraigschmidt	updates	Latest commit a67fd89 38 minutes ago
BarrierTaskGang	Updates	10 months ago
BuggyQueue	updates	a day ago
BusySynchronizedQueue	updates	4 months ago
DeadlockQueue	Refactored	3 years ago
ExpressionTree	Updates	10 months ago
Factorials	update	5 days ago
ImageStreamGang	updates	22 days ago
ImageTaskGangApplication	Updates	10 months ago
Java8	update	3 days ago
PalantiriManagerApplication	Updates	7 months ago
PingPongApplication	Updates	10 months ago
PingPongWrong	Refactored	3 years ago
SearchStreamForkJoin	updates	38 minutes ago

See [www.github.com/douglascraigschmidt/LiveLessons](https://www.github.com/douglascraigschmidt/LiveLessons)

# Overview of this Course

- Examples we don't cover in this course are covered in my LiveLessons course



The image shows the cover of the book "Java Concurrency LiveLessons 2nd Edition" by Doug Schmidt. The cover is orange with white text. At the top left is a sidebar with a blue icon and the text "douglasraigschmidt updates" followed by a list of topics: BarrierTaskGang, BuggyQueue, BusySynchronizedQueue, DeadlockQueue, ExpressionTree, Factorials, ImageStreamGang, ImageTaskGangApplication, Java8, PalantiriManagerApplication, PingPongApplication, PingPongWrong, and SearchStreamForkJoin. The main title "Introduction" is at the top in a dark grey bar. Below it is a large play button icon. The central text reads "Java Concurrency LiveLessons 2nd Edition" in large, bold, white font. Below that is the author's name "Doug Schmidt". At the bottom left is the Addison-Wesley logo. On the right side of the cover, there are three screenshots: one showing code in a terminal window, one showing a video player interface with a smiling man, and one showing a complex Java concurrency diagram.

See [www.dre.vanderbilt.edu/~schmidt/LiveLessons/CPiJava](http://www.dre.vanderbilt.edu/~schmidt/LiveLessons/CPiJava)

# Overview of this Course

- There's also a Facebook group dedicated to discussing Java 8-related topics

The screenshot shows a Facebook group page. At the top, there's a blue header bar with the Facebook logo, a search bar, and a profile picture for 'Douglas'. Below the header, the group name 'Java Concurrency LiveLessons' is displayed in large white text on an orange background. To the right of the group name is a small video thumbnail showing a man with a laptop. On the left side, there's a sidebar with links for 'Concurrent Programming in Java', 'Public Group', 'Discussion', 'Members', 'Events', 'Photos', 'Group Insights', and 'Manage Group'. Below the sidebar is a search bar and a 'Shortcuts' section with links to 'Concurrent Programmin...' and 'Design Patterns in J...'. The main content area has buttons for 'Write Post', 'Add Photo/Video', 'Live Video', and 'More'. There's also a text input field with placeholder text 'Write something...'. At the bottom, there are buttons for 'Photo/Video', 'Poll', 'Feeling/Activ...', and '...'. On the right side, there's a 'ADD MEMBERS' section with a search bar and a list of 'MEMBERS' with 1,647 members, each represented by a small profile picture.

See [www.facebook.com/groups/153248779024074](http://www.facebook.com/groups/153248779024074)

# Overview of this Course

- My website has more content related to Java 8 concurrent programming



## Digital Learning Offerings

[Douglas C. Schmidt](#) ([d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu))  
Associate Chair of [Computer Science and Engineering](#),  
[Professor](#) of Computer Science, and Senior Researcher  
in the [Institute for Software Integrated Systems](#) (ISIS)  
at [Vanderbilt University](#)



## O'Reilly LiveTraining Courses

- [Programming with Java 8 Lambdas and Streams](#), October 17, 2017 9:00am-12:00pm central time
- [Scalable Programming with Java 8 Parallel Streams](#), October 19, 2017 9:00am-12:00pm central time
- [Reactive Programming with Java 8 Completable Futures](#), October 23, 2017, 9:00am-12:00pm central time

## Pearson LiveLessons Courses

- [Java Concurrency](#)
- [Design Patterns in Java](#)

## Coursera MOOCs

- [Android App Development](#) Coursera Specialization
- [Pattern-Oriented Software Architecture](#) (POSA)

## Vanderbilt University Courses

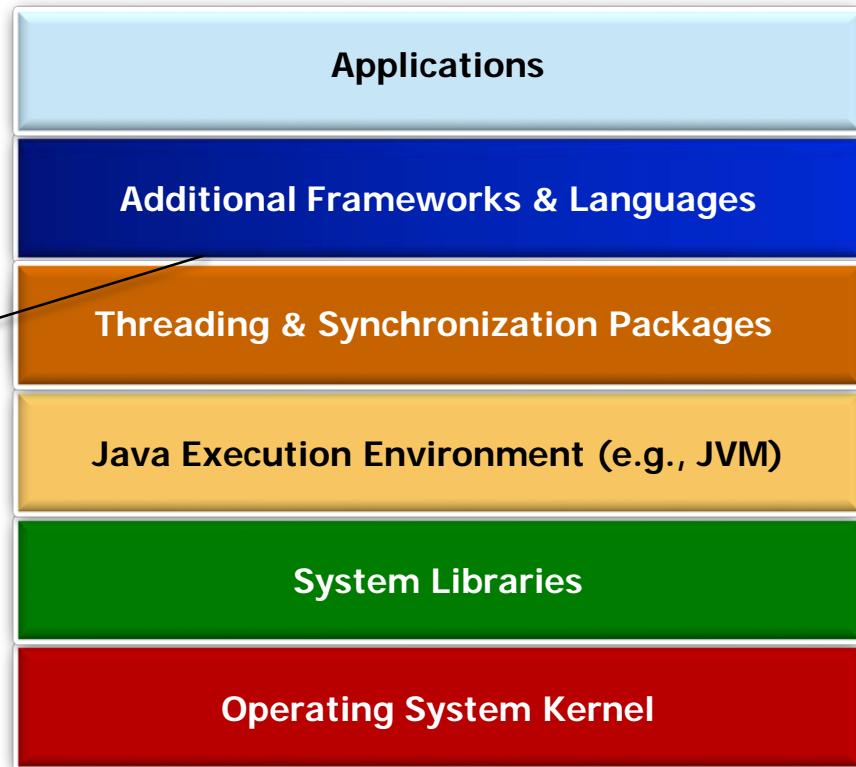
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 891: Introduction to Concurrent and Parallel Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 892: Concurrent Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with Java](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Concurrent Java Network Programming in Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with C++](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Systems Programming for Android](#)

See [www.dre.vanderbilt.edu/~schmidt/DigitalLearning](http://www.dre.vanderbilt.edu/~schmidt/DigitalLearning)

# Overview of this Course

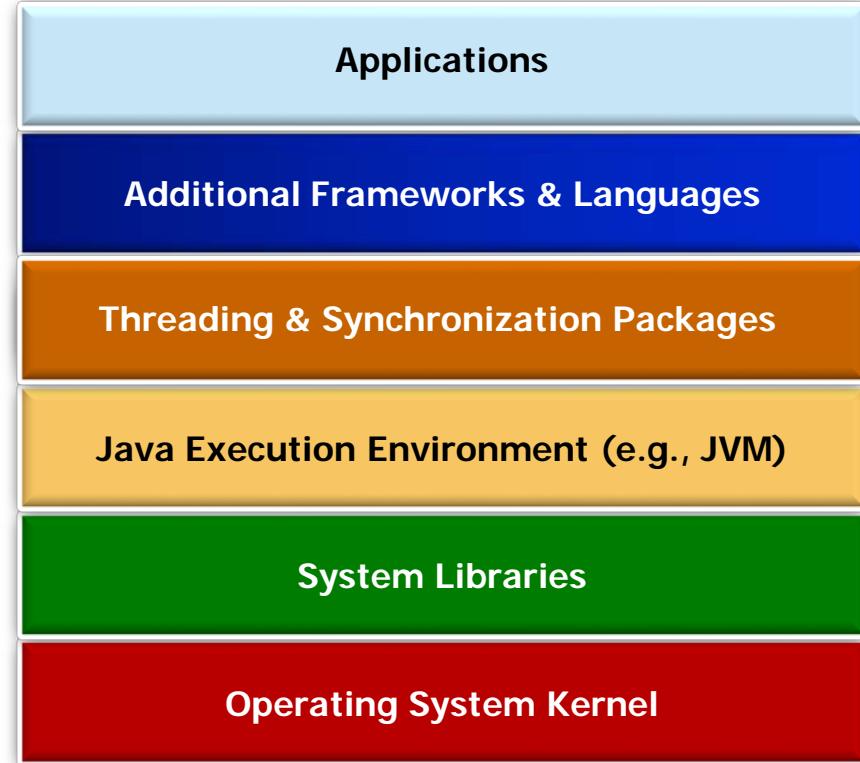
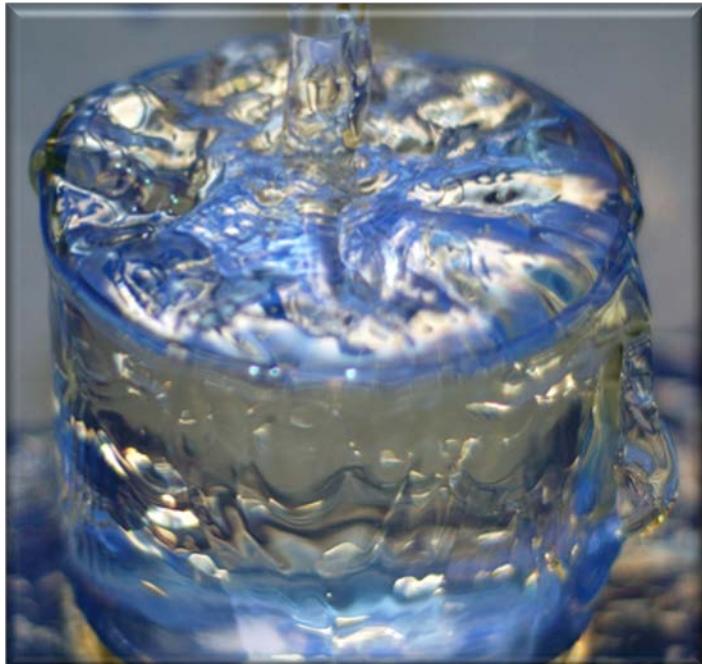
- There are also other Java-based & JVM-related frameworks & languages for concurrency

e.g., RxJava, Android, Java 9 Flow APIs, Scala, Kotlin, etc.



# Overview of this Course

- There are also other Java-based & JVM-related frameworks & languages for concurrency



Very interesting, but beyond the scope of this course!

---

# Accessing Java 8 Features & Functionality

# Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features

Overview   Downloads   Documentation   Community   Technologies   Training

## Java SE Downloads

 DOWNLOAD  DOWNLOAD

Java Platform (JDK) 8u101 / 8u102      NetBeans with JDK 8

### Java Platform, Standard Edition

**Java SE 8u101 / 8u102**  
Java SE 8u101 includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release. Java SE 8u102 is a patch-set update, including all of 8u101 plus additional features (described in the release notes).  
[Learn more](#)

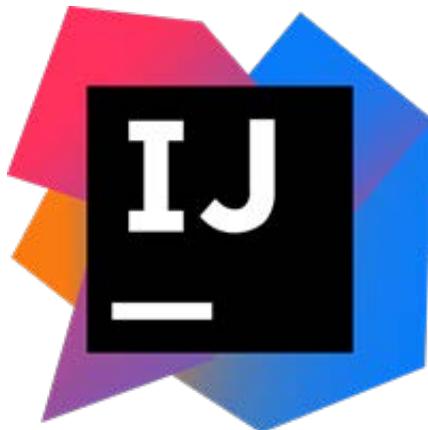
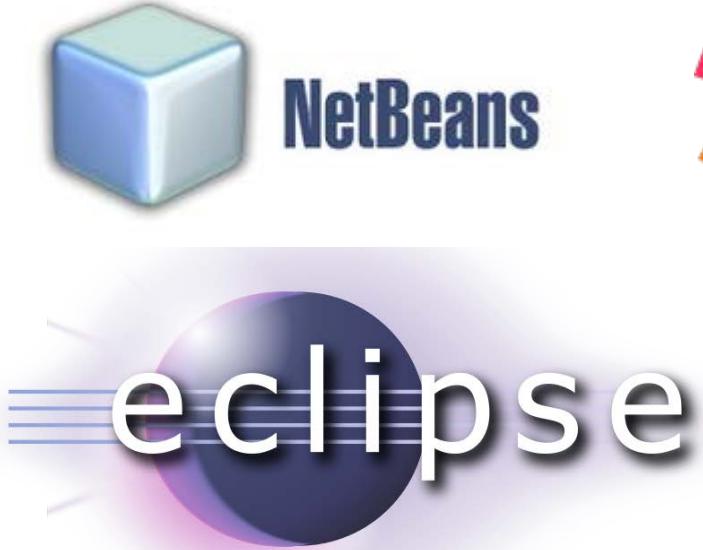
- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
  - JDK ReadMe
  - JRE ReadMe

JDK DOWNLOAD  
Server JRE DOWNLOAD  
JRE DOWNLOAD



# Accessing Java 8 Features & Functionality

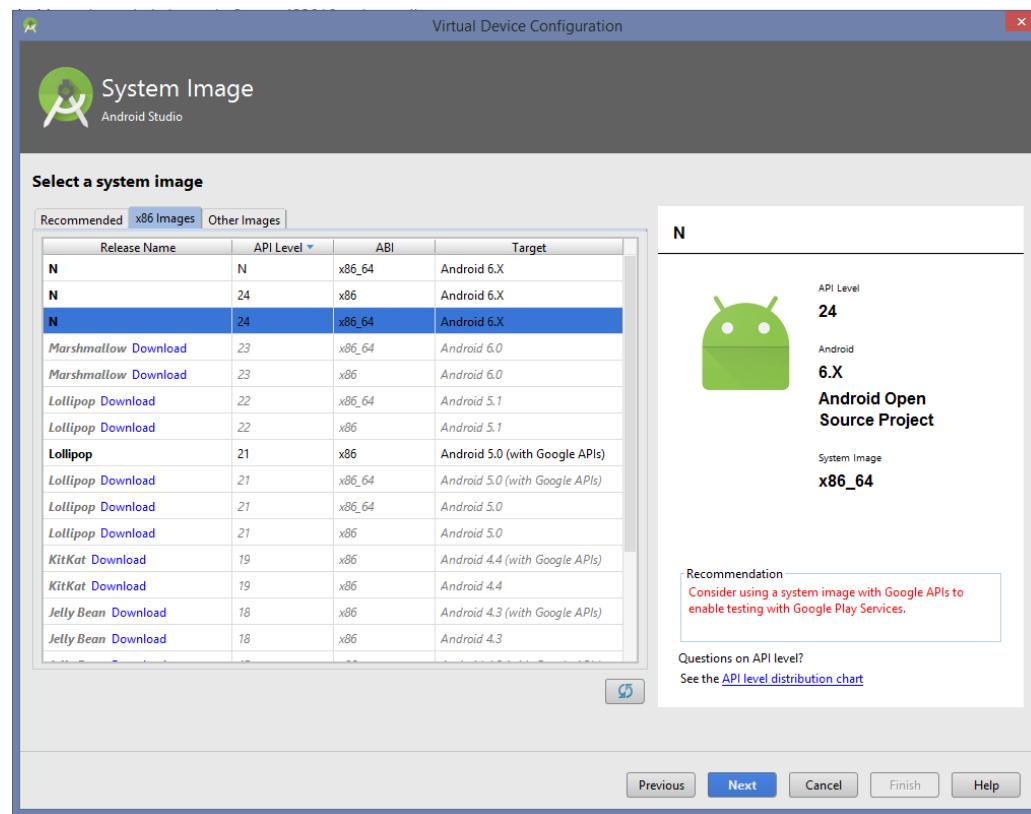
- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
  - IntelliJ & Eclipse are popular Java 8 IDEs



See [www.eclipse.org/downloads](http://www.eclipse.org/downloads) & [www.jetbrains.com/idea/download](http://www.jetbrains.com/idea/download)

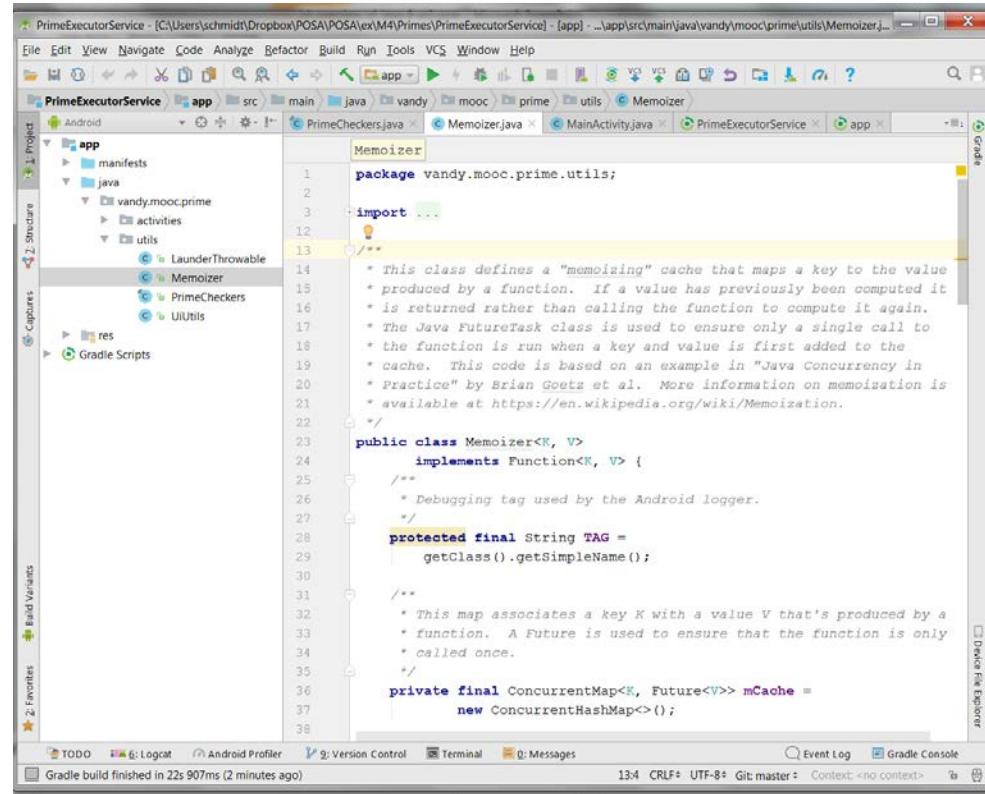
# Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
  - IntelliJ & Eclipse are popular Java 8 IDEs
  - Most Java 8 features are supported by Android API level 24 (& beyond)



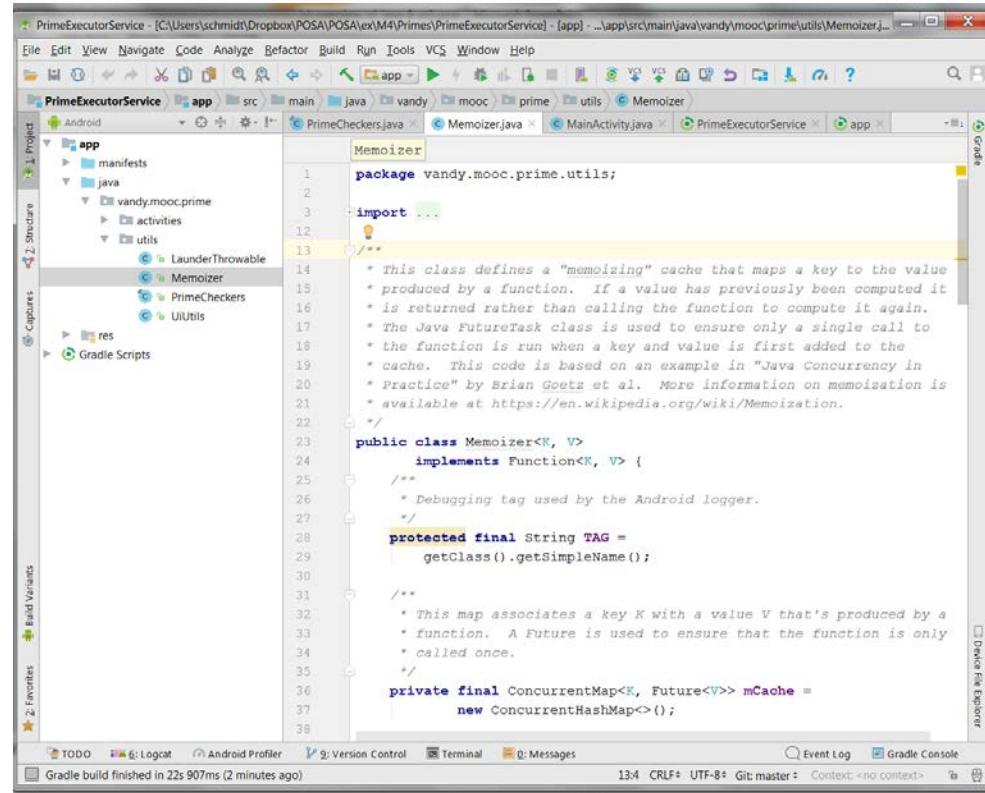
# Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
  - IntelliJ & Eclipse are popular Java 8 IDEs
  - Most Java 8 features are supported by Android API level 24 (& beyond)
    - A subset of Java 8 features are available in earlier Android releases, as well



# Accessing Java 8 Features & Functionality

- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
  - IntelliJ & Eclipse are popular Java 8 IDEs
  - Most Java 8 features are supported by Android API level 24 (& beyond)
    - A subset of Java 8 features are available in earlier Android releases, as well
  - Make sure to get Android Studio 3.x or later!



See [developer.android.com/studio/preview/features/java8-support.html](https://developer.android.com/studio/preview/features/java8-support.html)

# Accessing Java 8 Features & Functionality

- Java 8 source code is available online

- For browsing

[grepcode.com/file/repository](http://grepcode.com/file/repository).

[grepcode.com/java/root/jdk/](http://grepcode.com/java/root/jdk/)

[openjdk/8-b132/java](http://openjdk/8-b132/java)

- For downloading

[jdk8.java.net/download.html](http://jdk8.java.net/download.html)

The screenshot shows the Java.net website with the URL <http://java.net/projects/jdk8>. The page features a header with the Java.net logo and navigation links for Login, Register, and Help. A sidebar on the left contains links for Downloads, Feedback Forum, OpenJDK, and Planet JDK. The main content area is titled "JDK 8 Project" with the subtitle "Building the next generation of the JDK platform". It includes sections for "JDK 8 snapshot builds" (with a list of download links for 8u40 early access, source code, official Java SE 8 reference implementations, and build test results), "Feedback" (instructions for using the Project Feedback forum), and a "We Want Contributions!" section encouraging users to contribute to the platform.

Java.net  
The Source for Java Technology Collaboration

Login | Register | Help

JDK 8

Downloads  
Feedback Forum  
OpenJDK  
Planet JDK

**JDK 8 Project**  
Building the next generation of the JDK platform

**JDK 8 snapshot builds**

- Download 8u40 early access snapshot builds
- Source code (instructions)
- Official Java SE 8 Reference Implementations
- Early Access Build Test Results (instructions)

**Feedback**

Please use the [Project Feedback](#) forum if you have suggestions for or encounter issues using JDK 8.

If you find bugs in a release, please submit them using the usual [Java SE bug reporting channels](#), not with the Issue tracker accompanying this project. Be sure to include complete version information from the output of the `java -version` command.

We Want Contributions!

Frustrated with a bug that never got fixed? Have a great idea for improving the Java SE platform? See how to contribute for information on making contributions to the platform.

---

# End of Course Overview

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

Douglas C. Schmidt

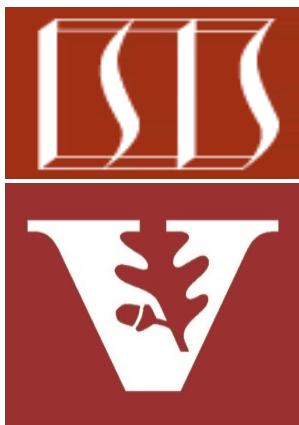
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

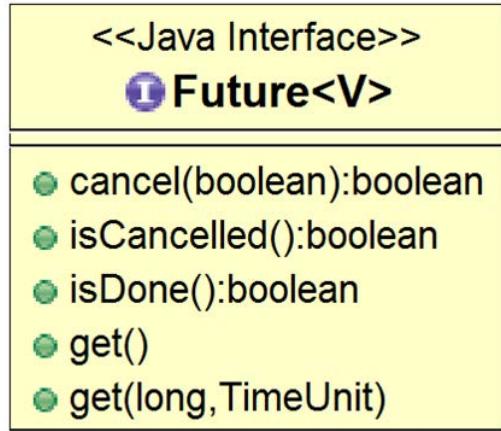
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Motivate the need for Java futures



## 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:

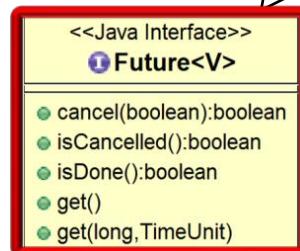
`CompletableFuture, CountedCompleter, ForkJoinTask, FutureTask, RecursiveAction, RecursiveTask, SwingWorker`

### 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.

# Learning Objectives in this Part of the Lesson

- Motivate the need for Java futures



Java futures provide the foundation for Java 8 completable futures

# Learning Objectives in this Part of the Lesson

---

- Motivate the need for Java futures
- Know how to program using Java futures

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

Callable<BigFraction> call =
    () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};

Future<BigFraction> future =
    commonPool().submit(call);
BigFraction result = future.get();

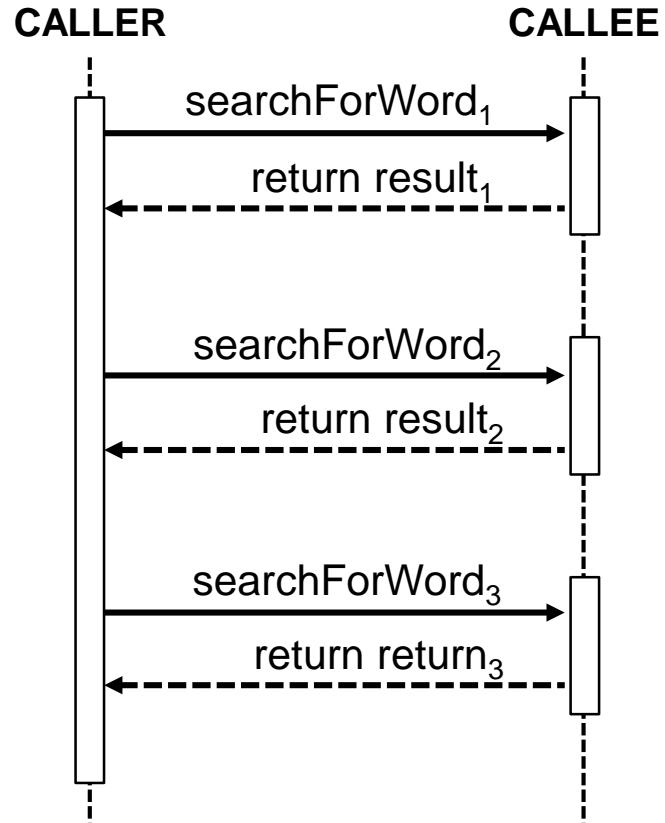
...
```

---

# Motivating the Need for Futures

# Motivating the Need for Futures

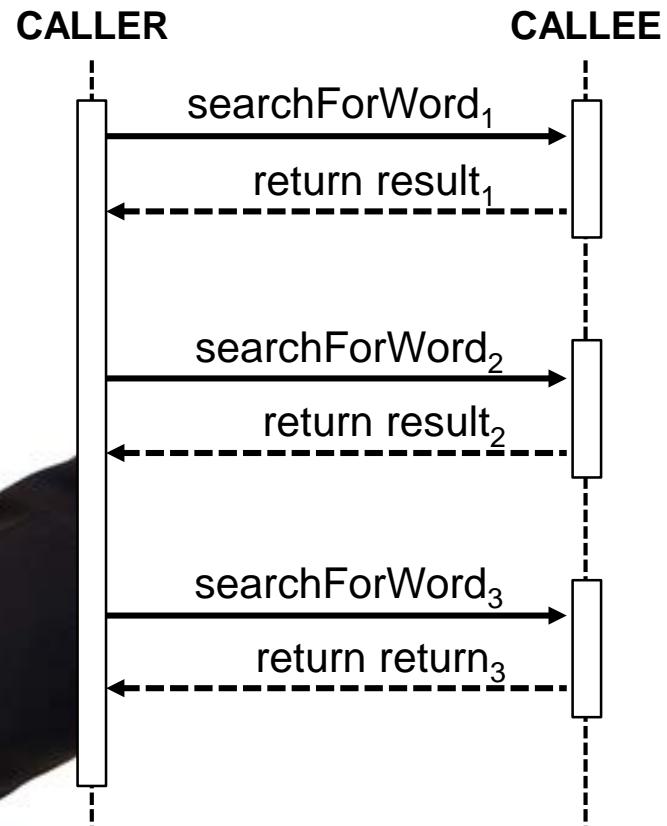
- Method calls in typical Java programs are largely *synchronous*



e.g., calls on Java collections & behaviors in Java 8 stream aggregate operations

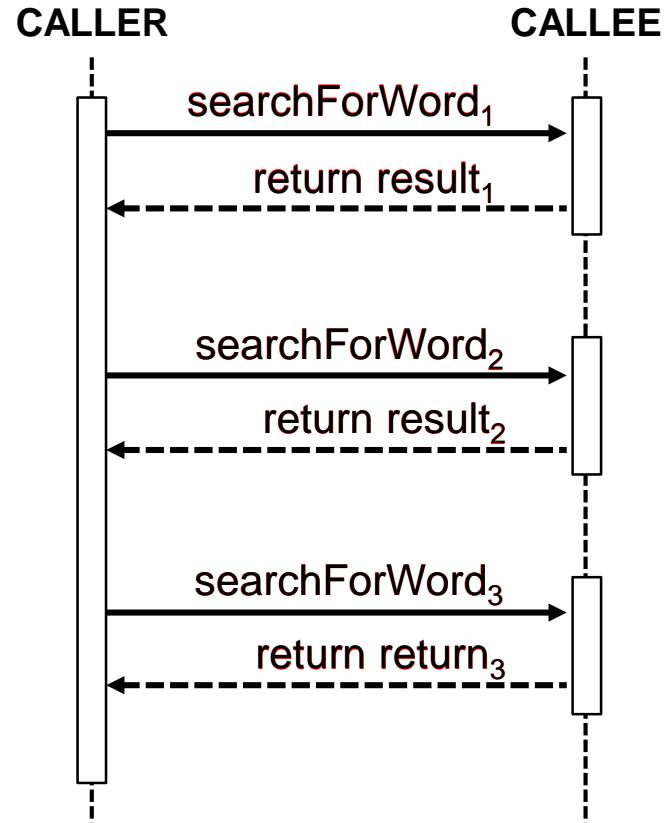
# Motivating the Need for Futures

- Method calls in typical Java programs are largely *synchronous*
  - i.e., a behavior borrows the thread of its caller until its computation(s) finish



# Motivating the Need for Futures

- Method calls in typical Java programs are largely *synchronous*
  - i.e., a behavior borrows the thread of its caller until its computation(s) finish



# Motivating the Need for Futures

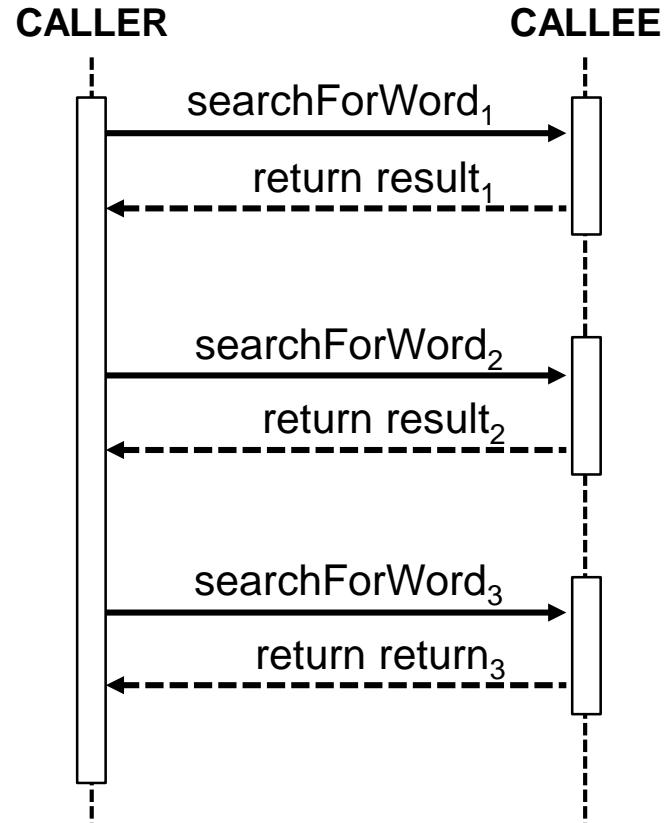
---

- Synchronous calls have pros & cons



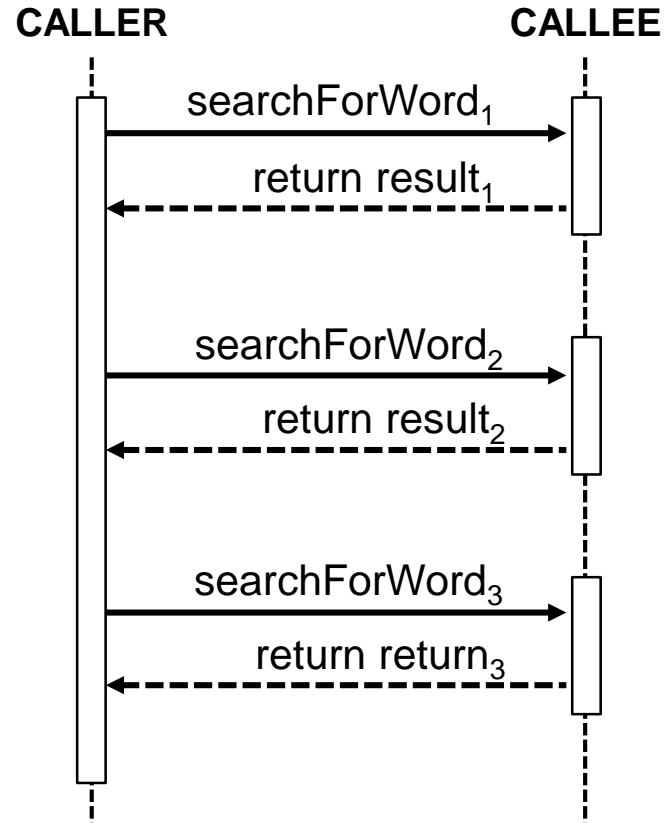
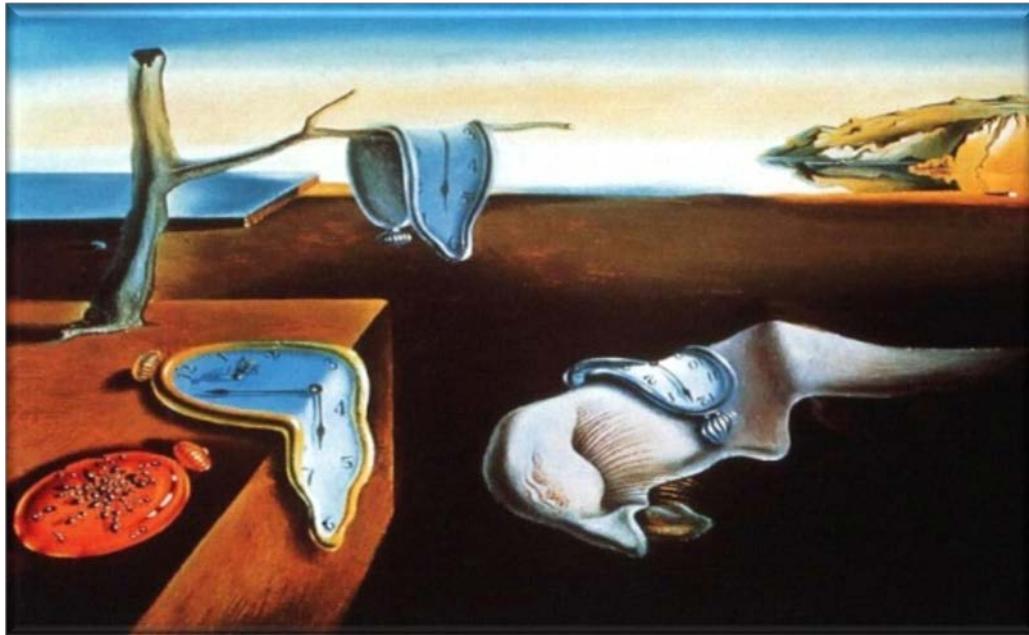
# Motivating the Need for Futures

- Pros of synchronous calls:
  - “Intuitive” since they map cleanly onto conventional two-way method patterns



# Motivating the Need for Futures

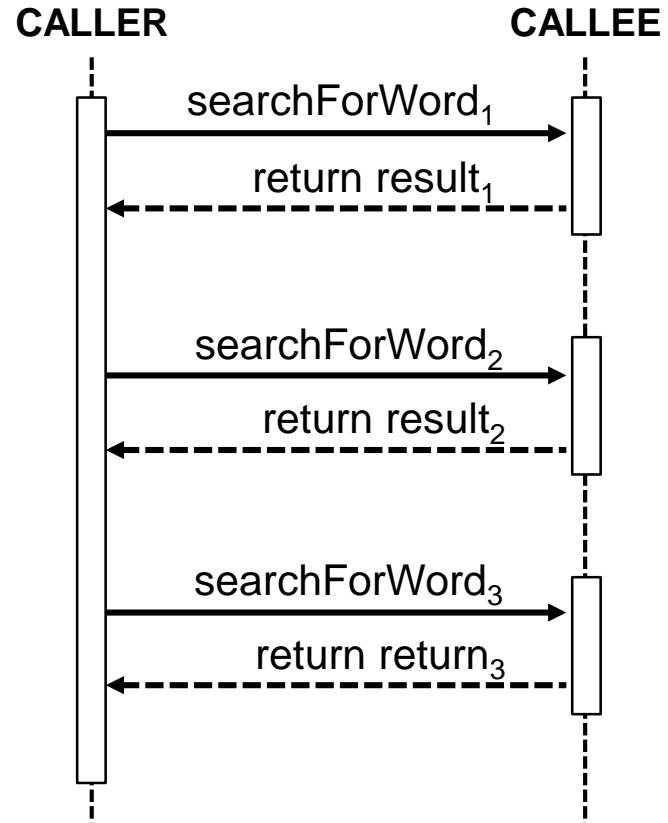
- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems



See [www.ibm.com/developerworks/library/j-jvmc3](http://www.ibm.com/developerworks/library/j-jvmc3)

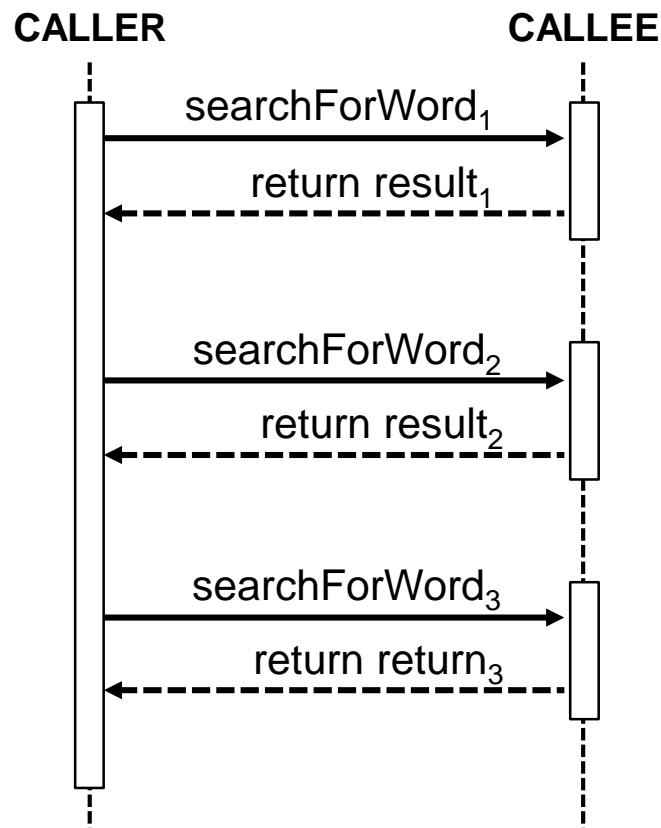
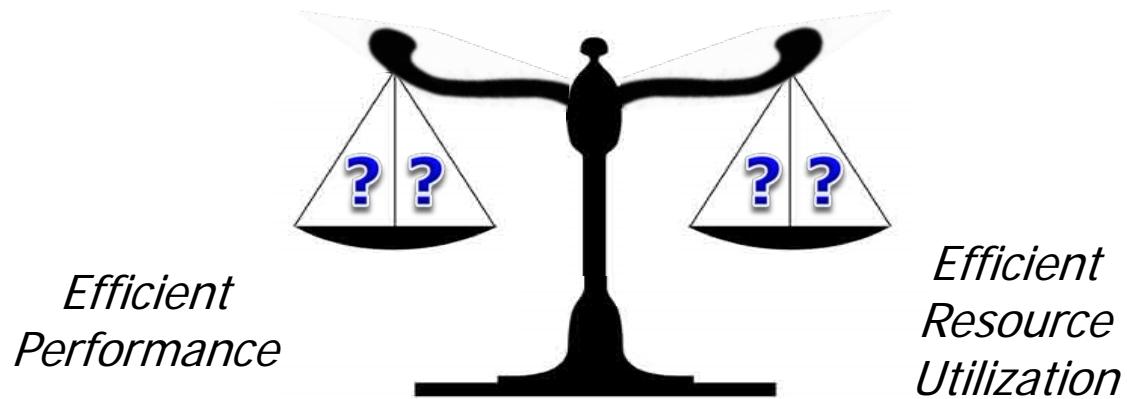
# Motivating the Need for Futures

- 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



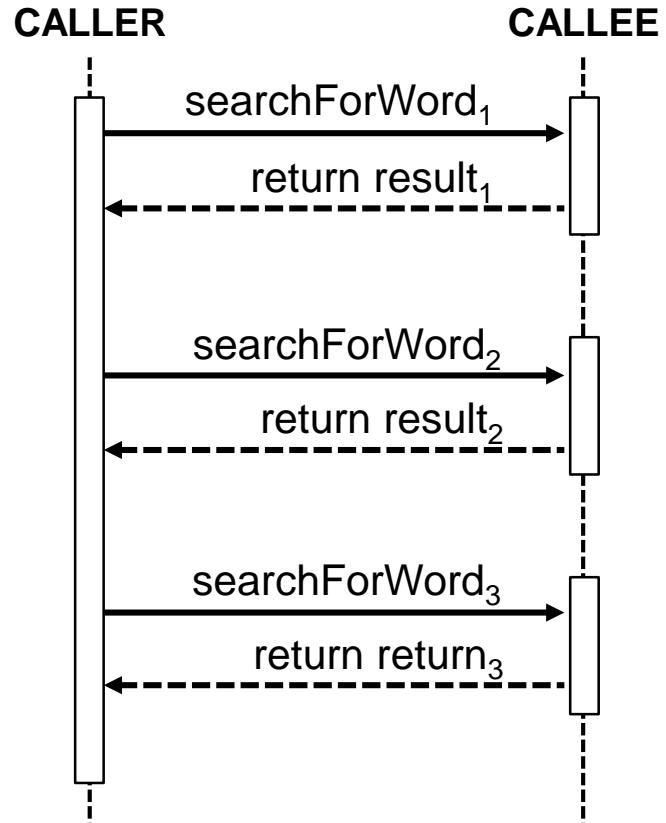
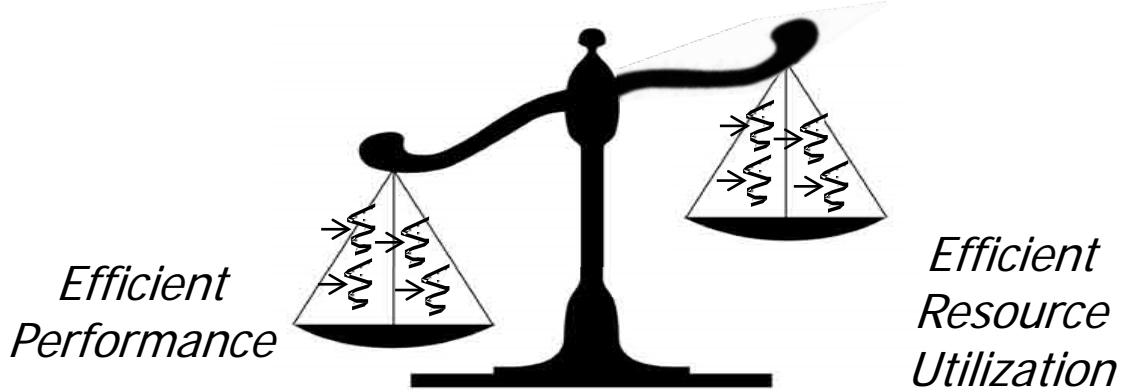
# Motivating the Need for Futures

- 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



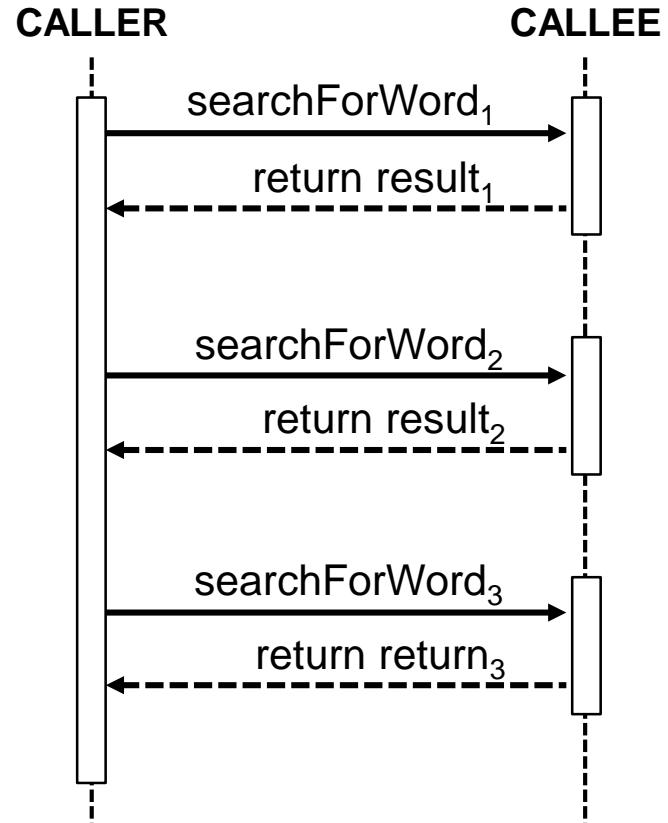
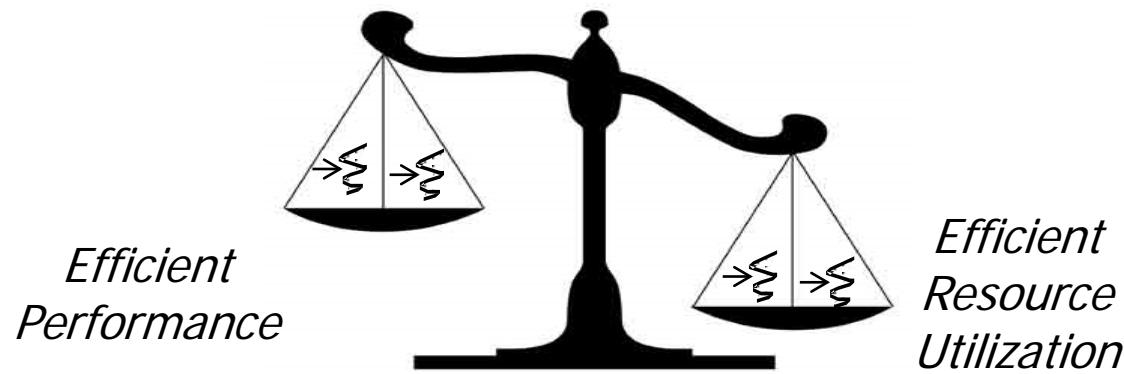
# Motivating the Need for Futures

- 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



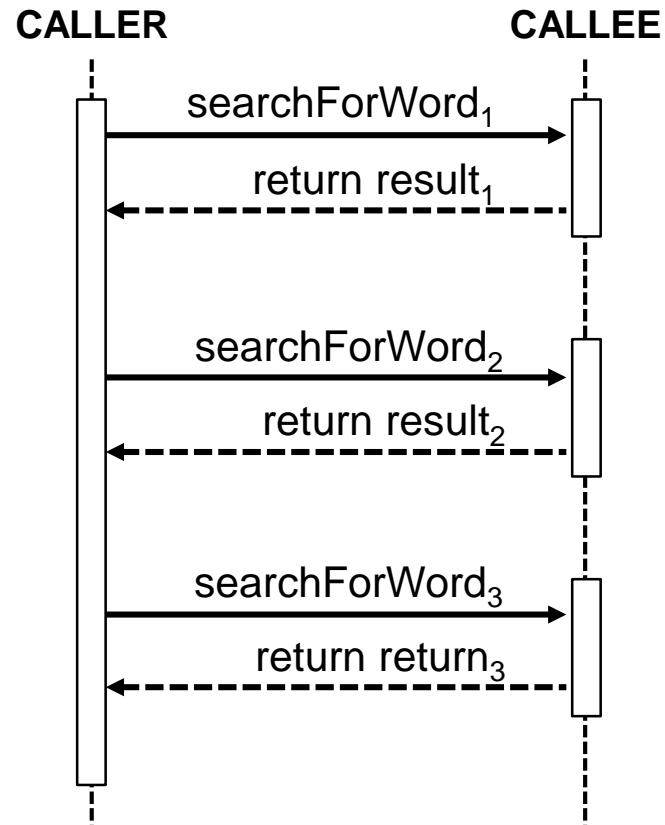
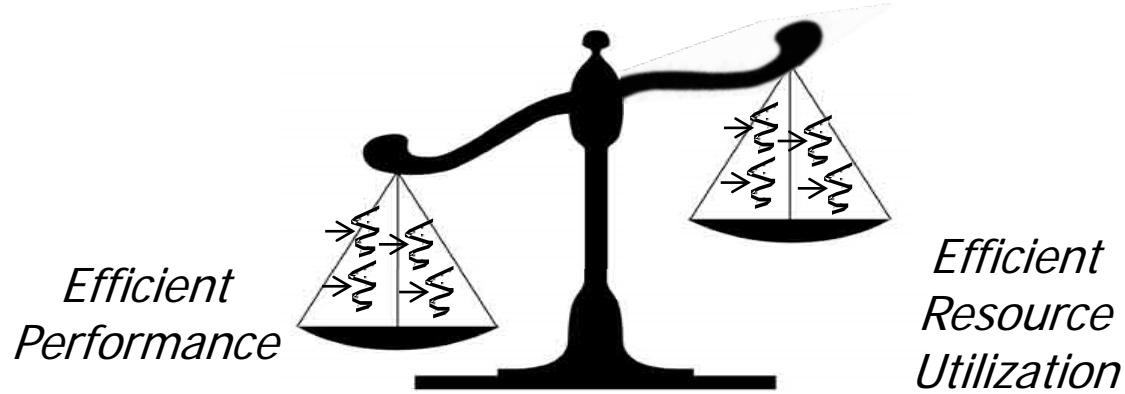
# Motivating the Need for Futures

- 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

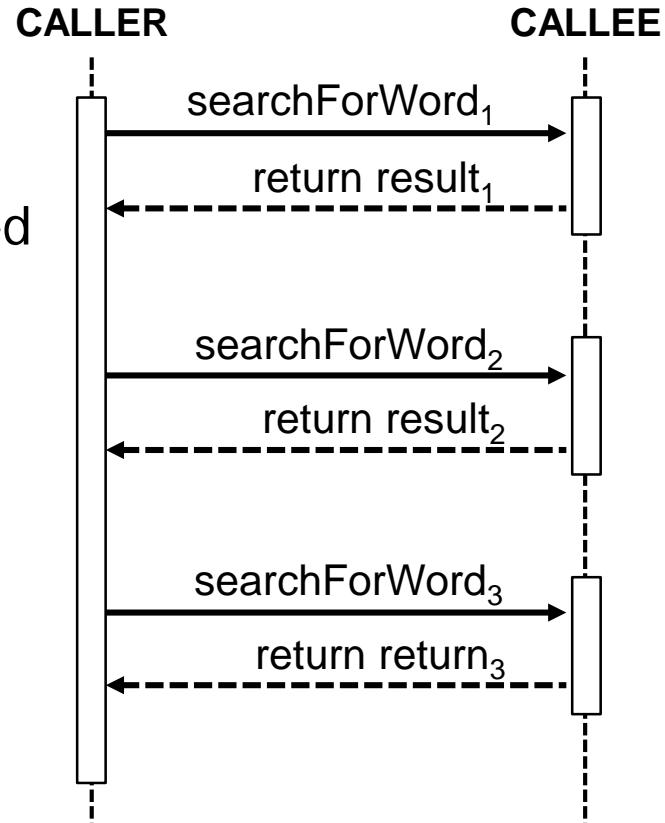
- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
    - Blocking threads incur overhead
  - Selecting right number of threads is hard



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

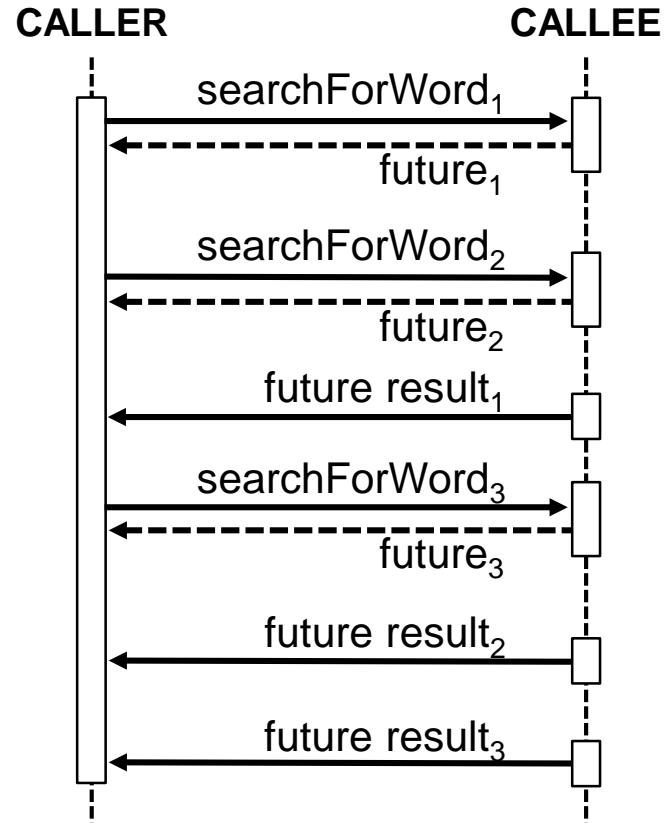
# Motivating the Need for Futures

- Cons of synchronous calls:
  - May not leverage all the parallelism available in multi-core systems
  - Synchronous calls in Java 8 parallel may need to change the common fork-join pool size



# Motivating the Need for Futures

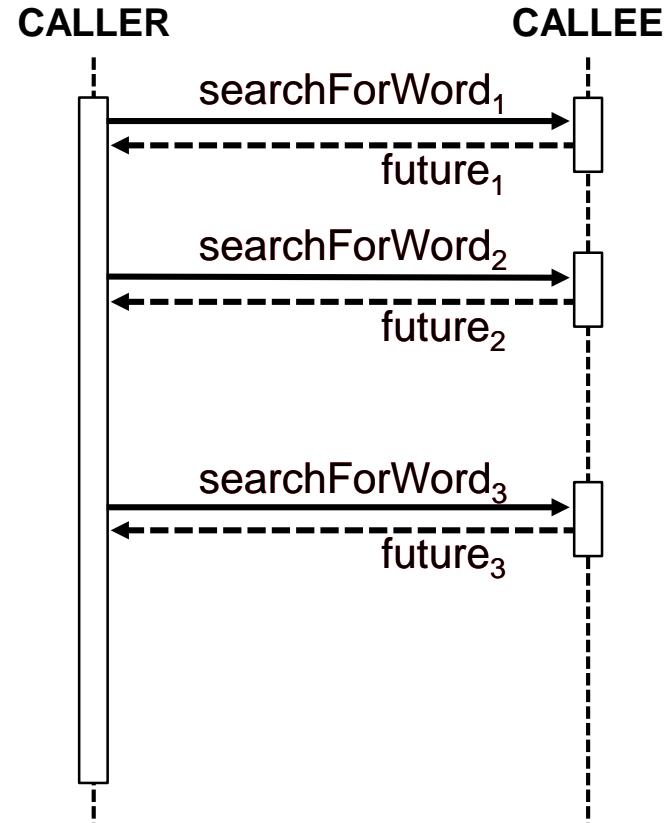
- An alternative approach uses asynchronous (async) calls & Java futures



See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)

# Motivating the Need for Futures

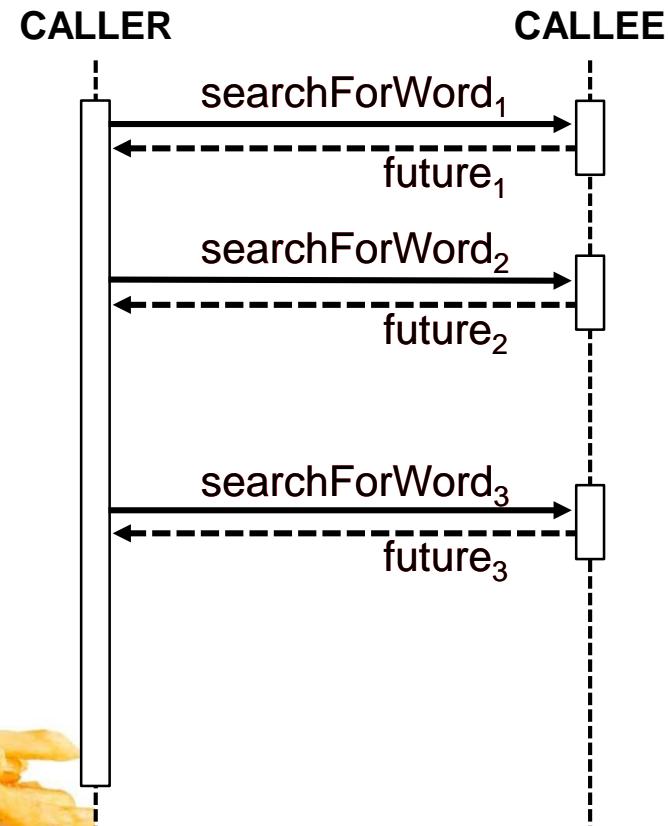
- An alternative approach uses asynchronous (async) calls & Java futures
  - Async calls return a future & continue running the computation in the background



See [en.wikipedia.org/wiki/Asynchrony\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Asynchrony_(computer_programming))

# Motivating the Need for Futures

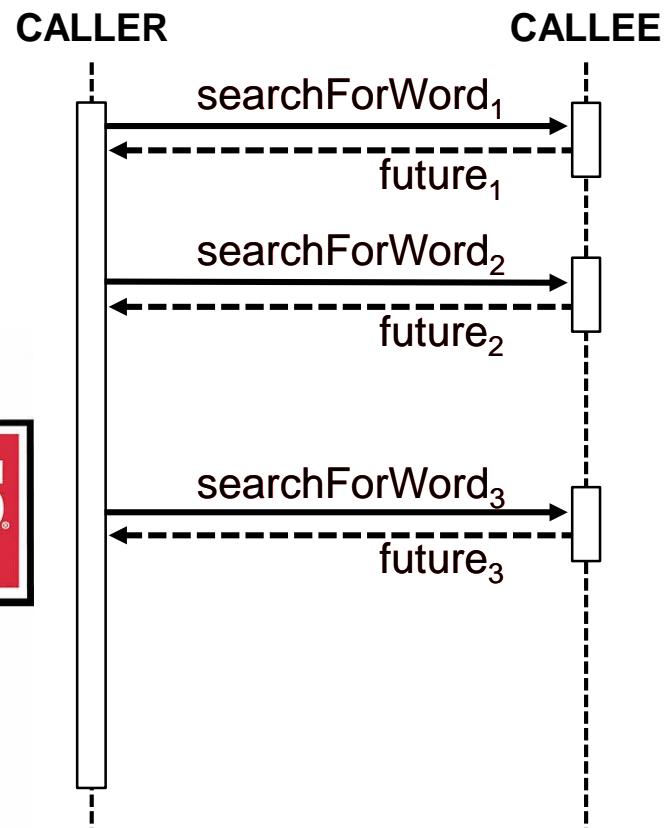
- An alternative approach uses asynchronous (async) calls & Java futures
  - Async calls return a future & continue running the computation in the background
  - A future is a proxy that represents the result of an async computation



See [en.wikipedia.org/wiki/Futures\\_and\\_promises](https://en.wikipedia.org/wiki/Futures_and_promises)

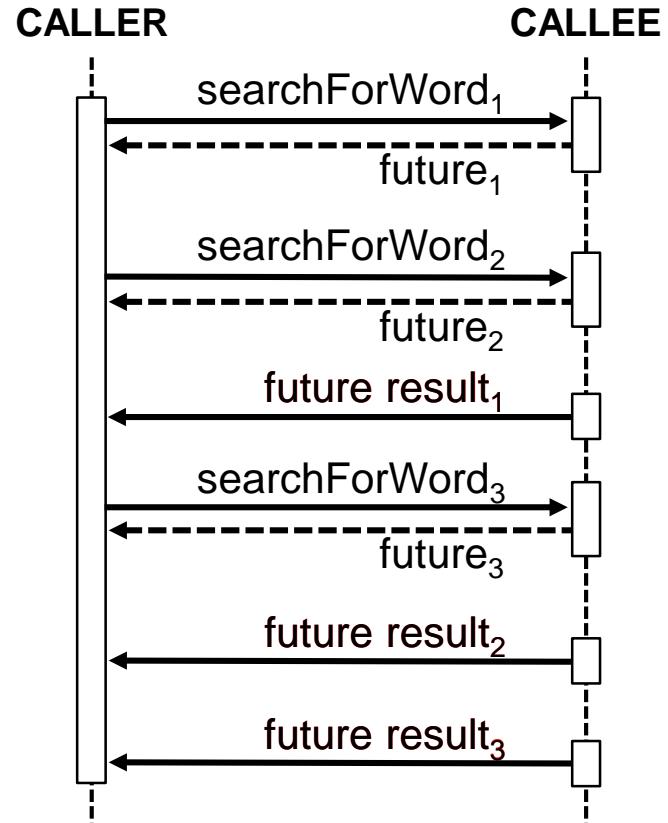
# Motivating the Need for Futures

- 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



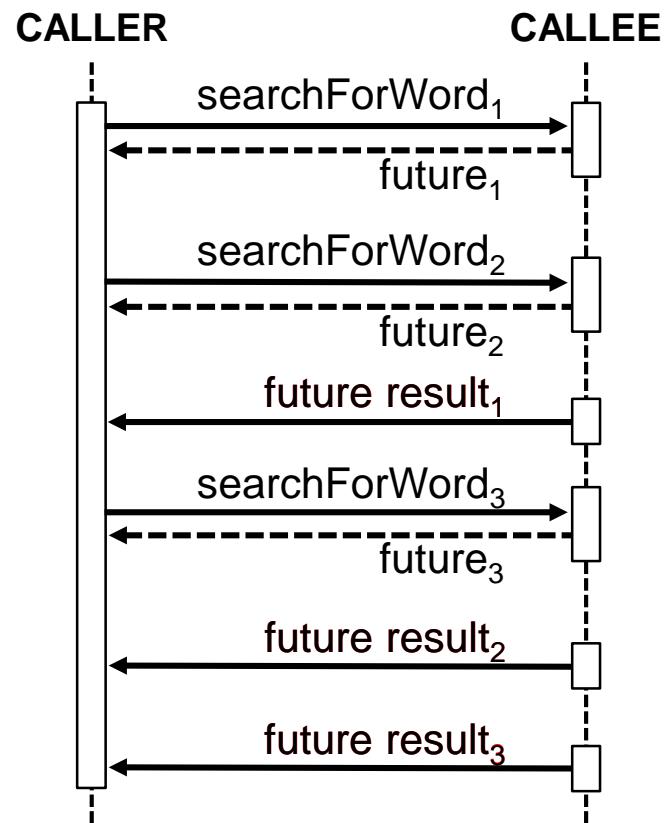
# Motivating the Need for Futures

- 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



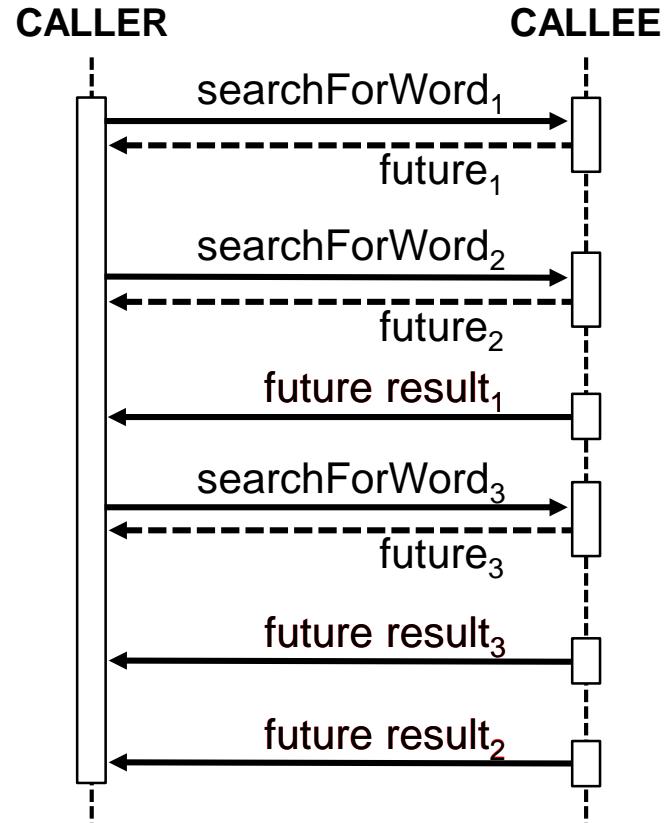
# Motivating the Need for Futures

- 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



# 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
  - Results can occur in a different order than the original calls were made



---

# Programming with Java Futures

# Programming with Java Futures

---

- Example of using Java Future via a Callable & the common fork-join pool

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

Callable<BigFraction> call = () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};

Future<BigFraction> future =
    commonPool().submit(call);
BigFraction result = future.get();

...
```

# Programming with Java Futures

- Example of using Java Future via a Callable & the common fork-join pool

*Callable is a task that returns a result via a single method with no arguments*

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

Callable<BigFraction> call = () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};
```

```
Future<BigFraction> future =
    commonPool().submit(call);
BigFraction result = future.get();

...
```

# Programming with Java Futures

- Example of using Java Future via a Callable & the common fork-join pool

*Java 8 enables the initialization of a Callable via a lambda expression*

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

Callable<BigFraction> call = () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};
```

```
Future<BigFraction> future =
    commonPool().submit(call);
BigFraction result = future.get();

...
```

See “Overview of Java 8 Lambda Expressions & Method References”

# Programming with Java Futures

- Example of using Java Future via a Callable & the common fork-join pool

*Submits a value-returning task for execution in the common fork-join pool*

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

Callable<BigFraction> call = () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};

Future<BigFraction> future =
    commonPool().submit(call);
BigFraction result = future.get();

...
```

# Programming with Java Futures

- Example of using Java Future via a Callable & the common fork-join pool

*submit() returns a future representing the pending results of the task*

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

Callable<BigFraction> call = () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};
```

```
Future<BigFraction> future =
commonPool().submit(call);
BigFraction result = future.get();

...
```

# Programming with Java Futures

- Example of using Java Future via a Callable & the common fork-join pool

*get() blocks if necessary for the computation to complete & then retrieves its result*

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

Callable<BigFraction> call = () -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);
    return bf1.multiply(bf2);
};

Future<BigFraction> future =
    commonPool().submit(call);
BigFraction result = future.get();

...
```

---

# End of Motivating the Need for Java 8 CompletableFuture Futures (Part 1)

# Motivating the Need for Java 8 Completable Futures (Part 2)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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
- Motivate the need for Java 8 completable futures

<<Java Interface>>

**I Future<V>**

---

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



---

# Motivating the Need for Completable Futures

# Motivating the Need for Completable Futures

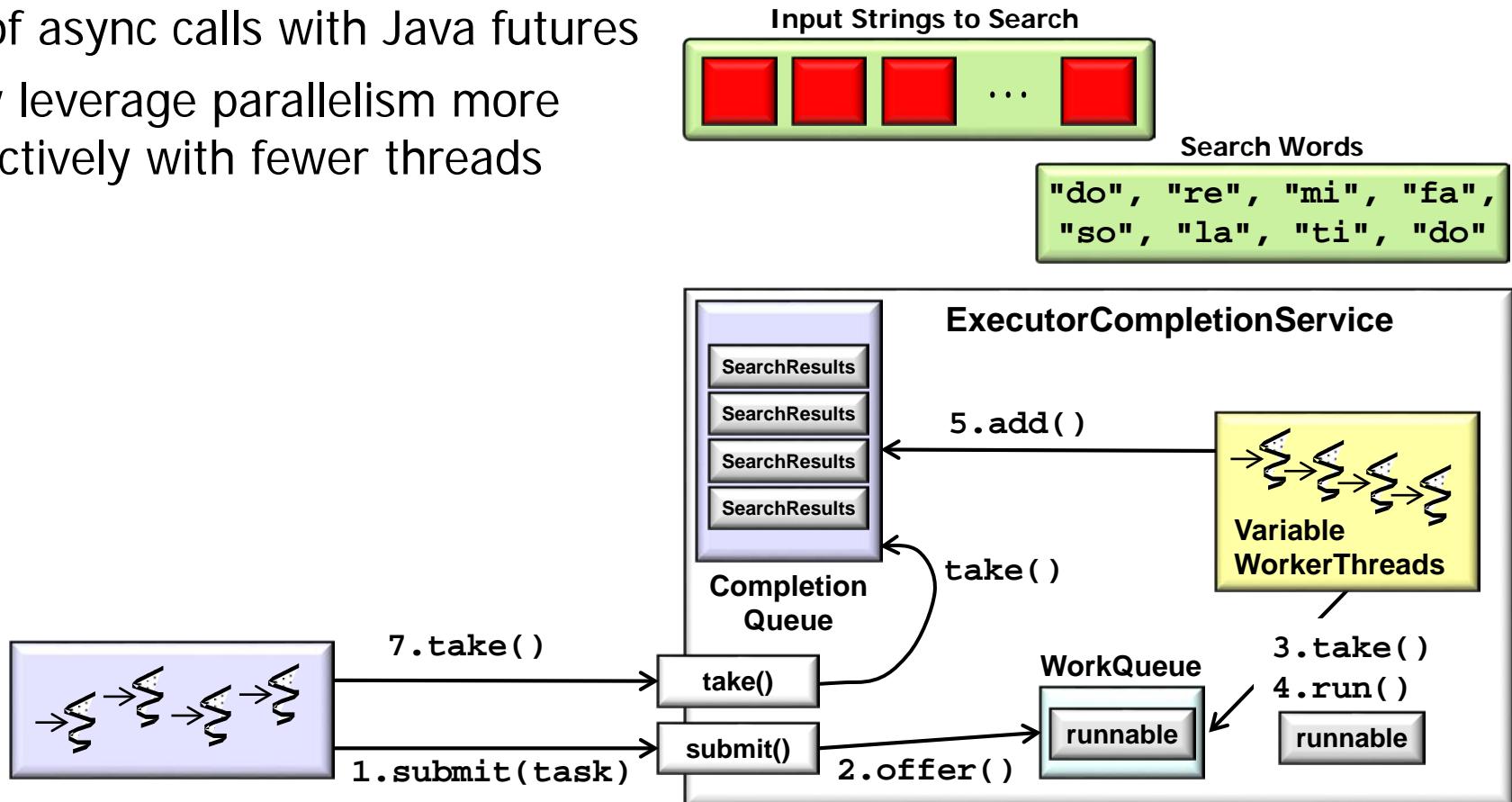
---

- 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

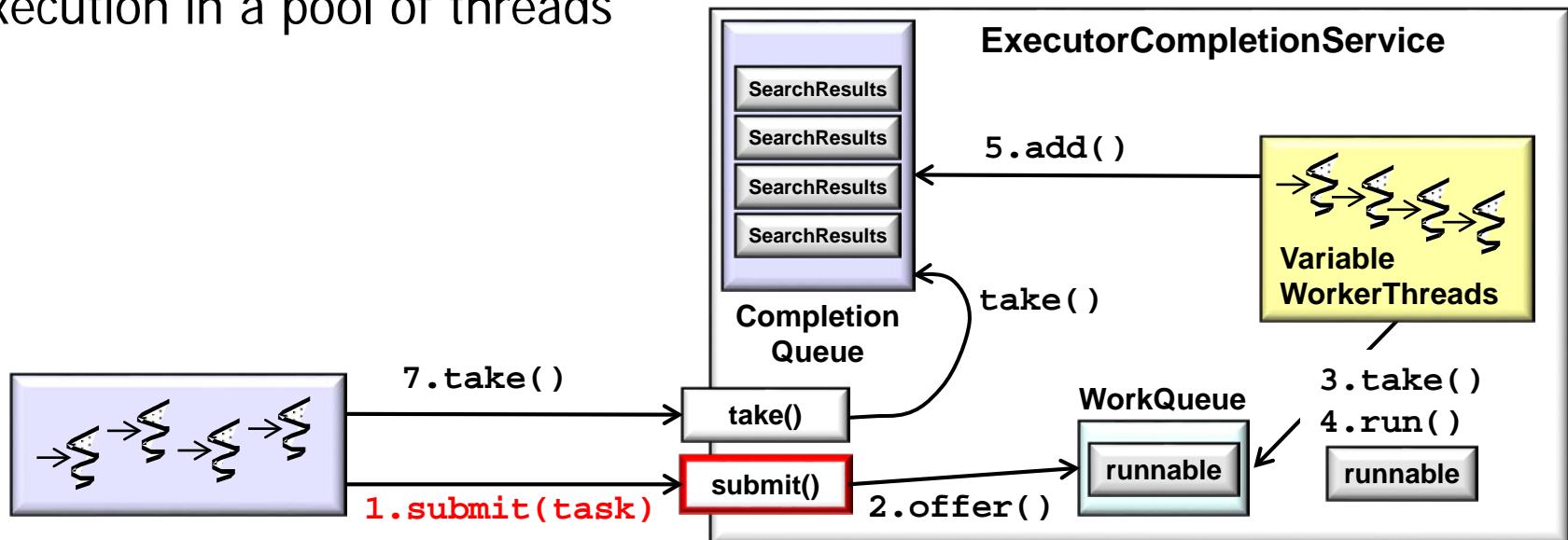


# Motivating the Need for Completable Futures

- 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

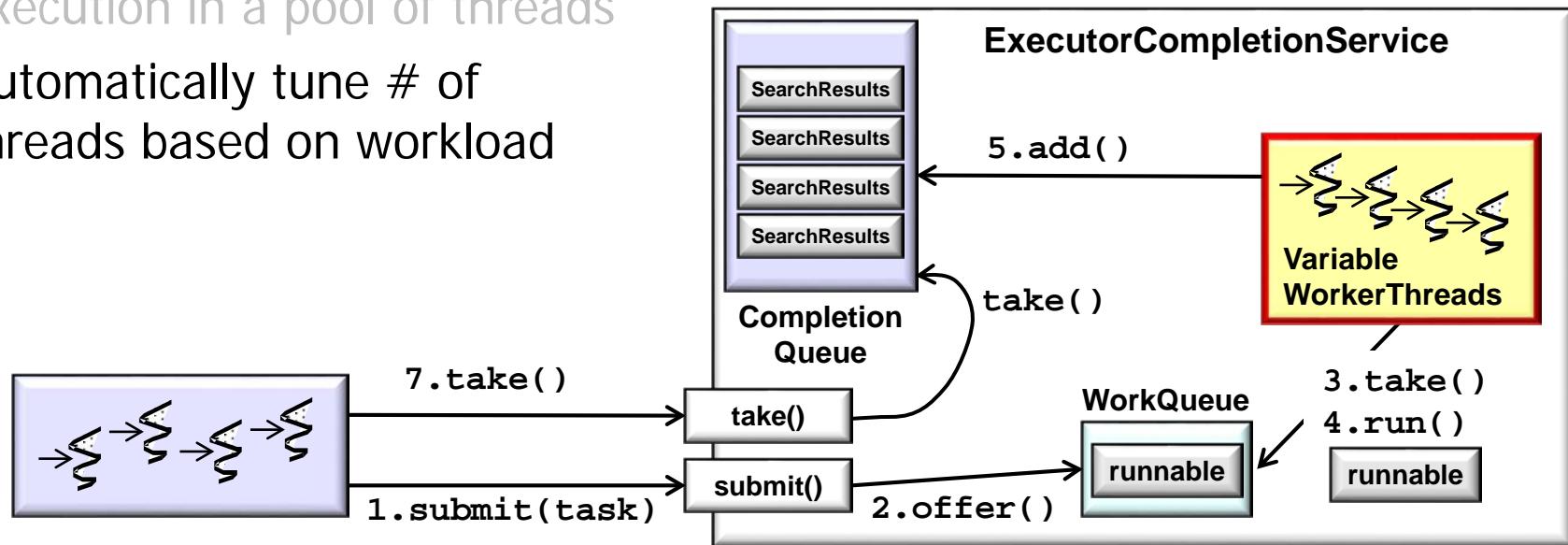
mCompletionService

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



# Motivating the Need for Completable Futures

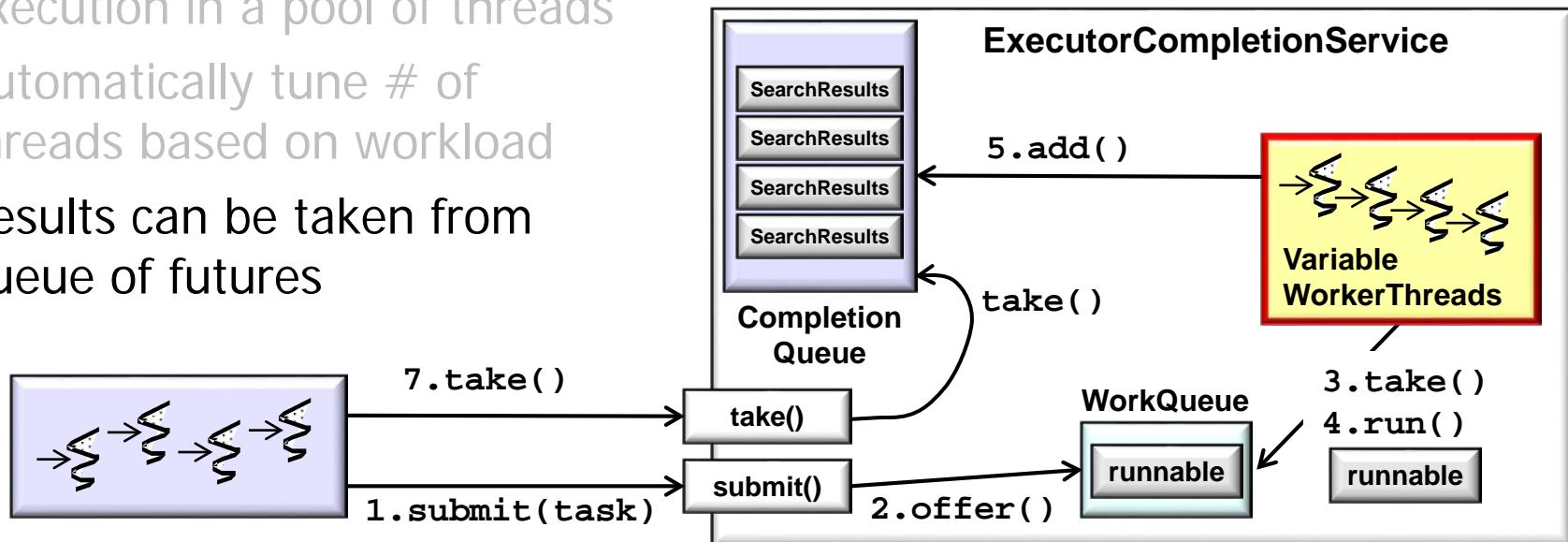
- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads, e.g.,
  - Queue async computations for execution in a pool of threads
  - Automatically tune # of threads based on workload



# Motivating the Need for Completable Futures

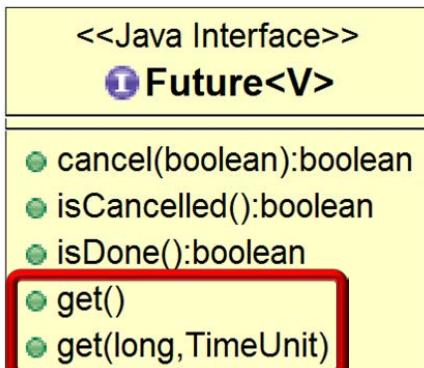
- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads, e.g.,
  - Queue async computations for execution in a pool of threads
  - Automatically tune # of threads based on workload
  - Results can be taken from queue of futures

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



# 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

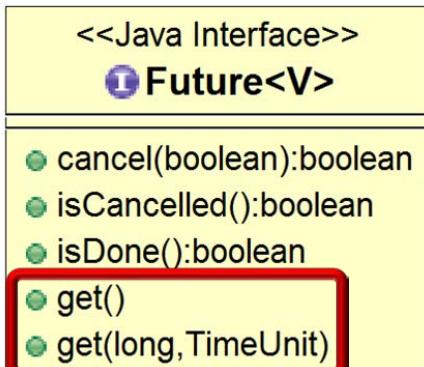


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

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 timeout or time-block

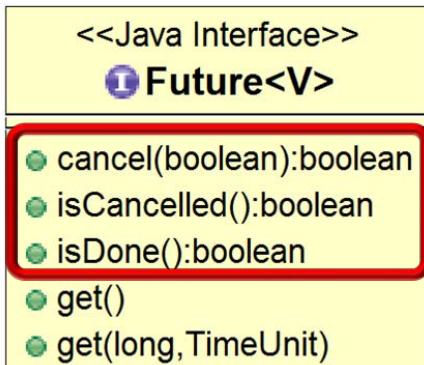


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

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

# 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



```
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

---

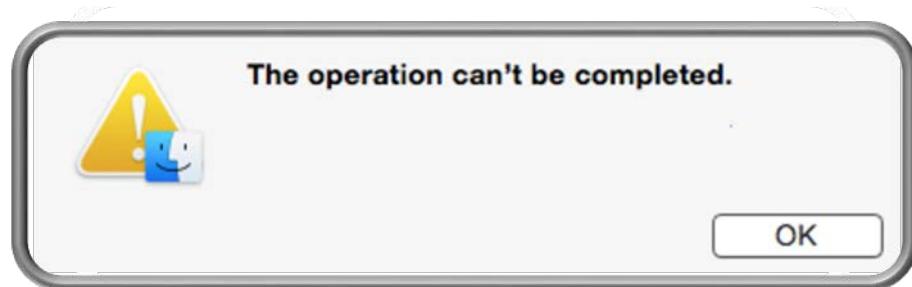
- Cons of async calls with Java futures
  - Limited feature set

<<Java Interface>>	
 Future<V>	
<ul style="list-style-type: none"><li>● cancel(boolean):boolean</li><li>● isCancelled():boolean</li><li>● isDone():boolean</li><li>● get()</li><li>● get(long, TimeUnit)</li></ul>	

LIMITED

# 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](https://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



# 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



"open  
mouth,  
insert  
foot"

*Nearly always  
the wrong  
thing to do!!*

```
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

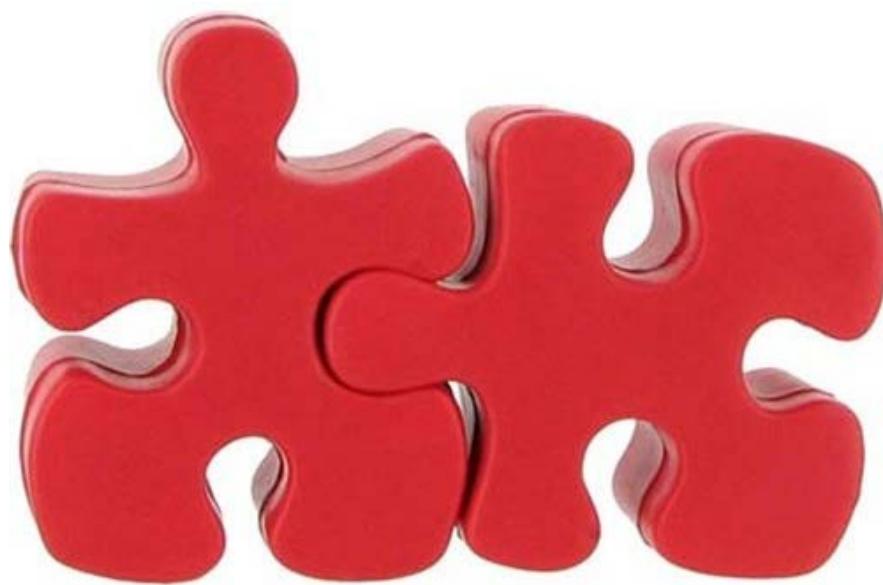
- Cons of async calls with Java futures
    - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently to handle async results
    - *Cannot* be triggered reactively
    - *Cannot* be treated efficiently as a *collection* of futures
- ```
Future<BigFraction> future1 =  
    commonPool().submit(() -> {  
        ... });  
  
Future<BigFraction> future2 =  
    commonPool().submit(() -> {  
        ... });  
  
...  
future1.get();  
future2.get();
```

*Can't wait efficiently for just  
the first futures to complete*

# 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 CompletableFuture Futures (Part 2)

# Overview of Java 8 CompletableFuture

## (Part 1)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

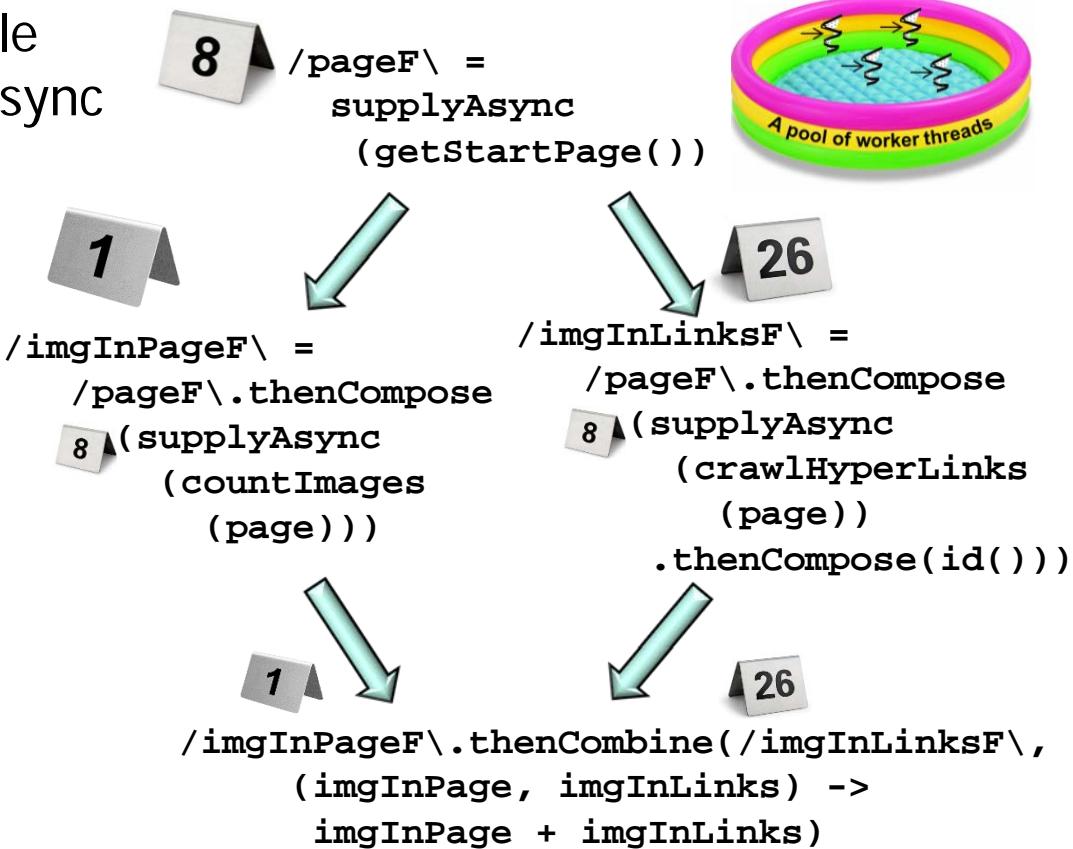
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Know how the Java 8 completable futures framework provides an async concurrent programming model



# Learning Objectives in this Part of the Lesson

---

- Know how the Java 8 completable futures framework provides an async concurrent programming model
- Recognize Java 8 completable futures overcome limitations with Java futures



---

# Overview of Completable Futures

# Overview of Completable Futures

- The Java 8 completable future framework provides an async concurrent programming model

## 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>
```

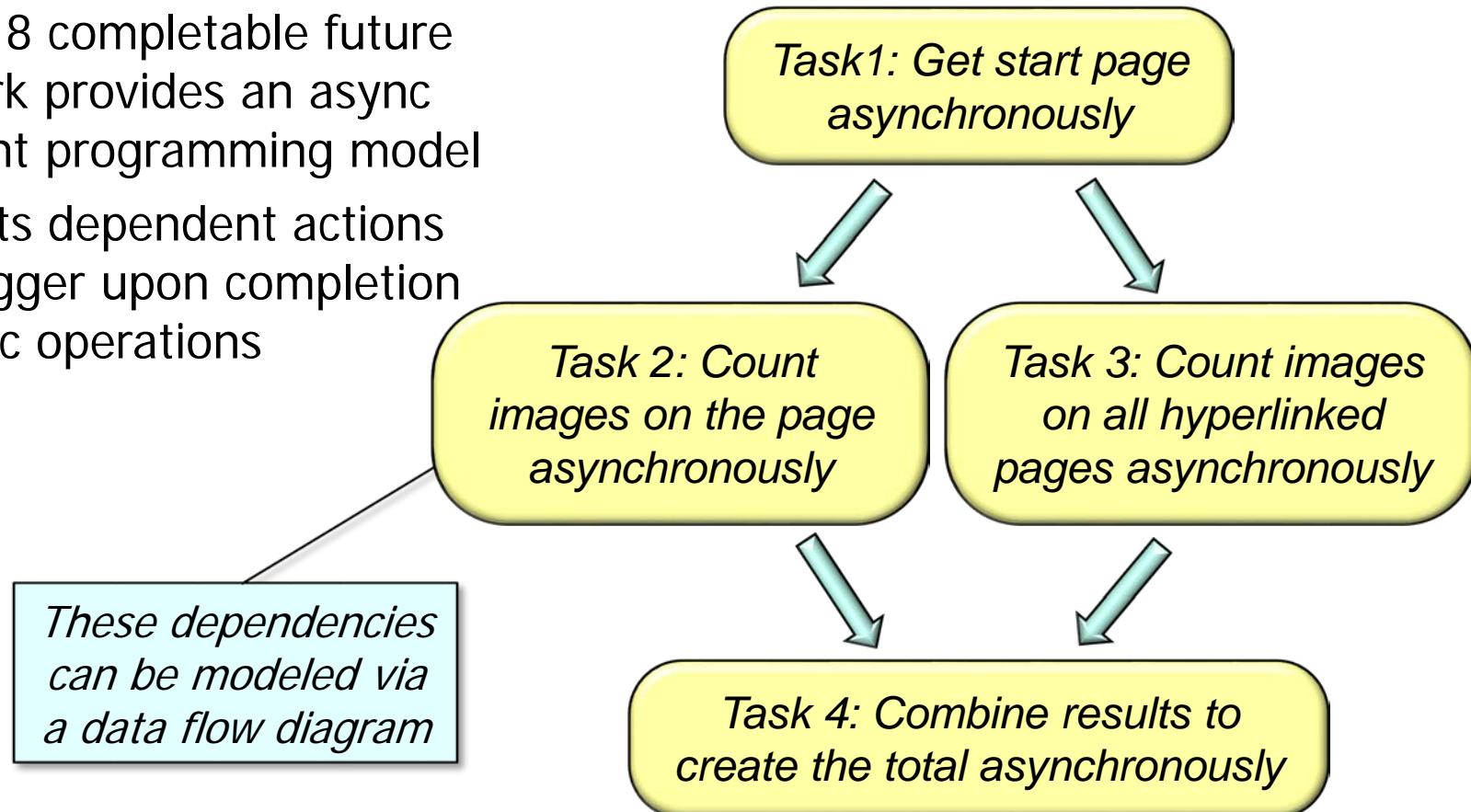
A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.

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

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

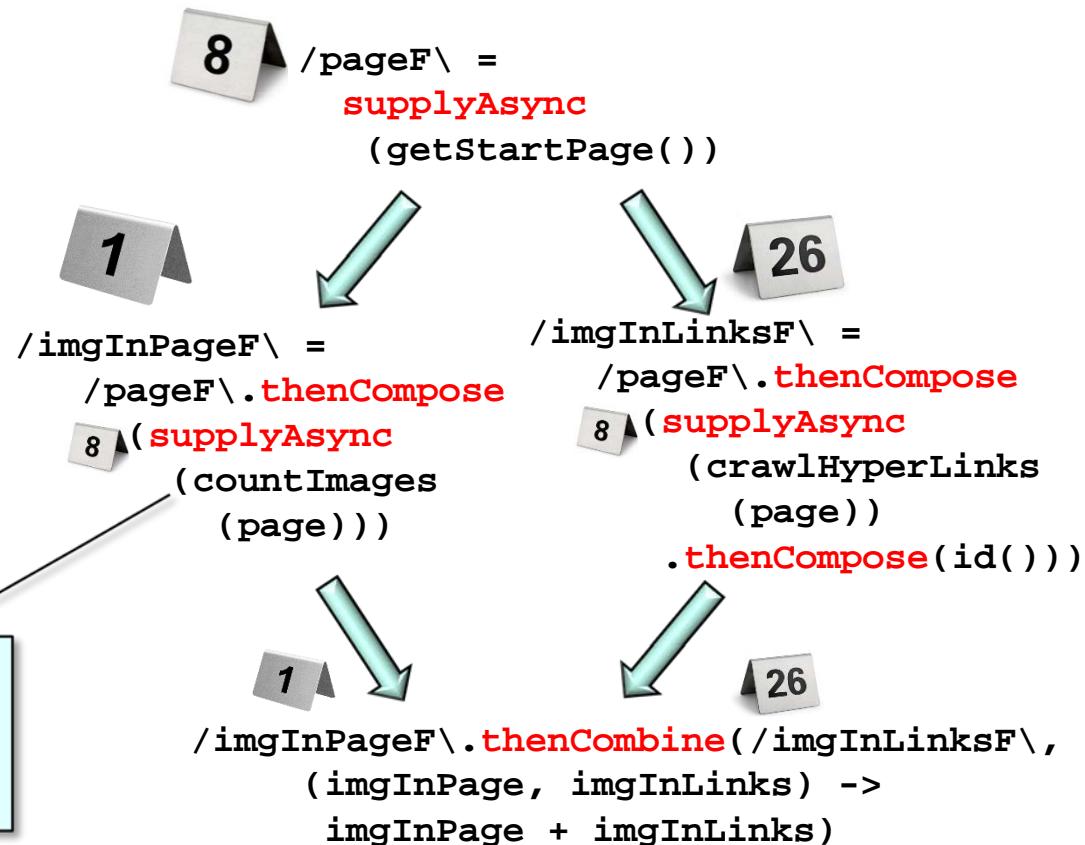
# Overview of Completable Futures

- The Java 8 completable future framework provides an async concurrent programming model
  - Supports dependent actions that trigger upon completion of async operations



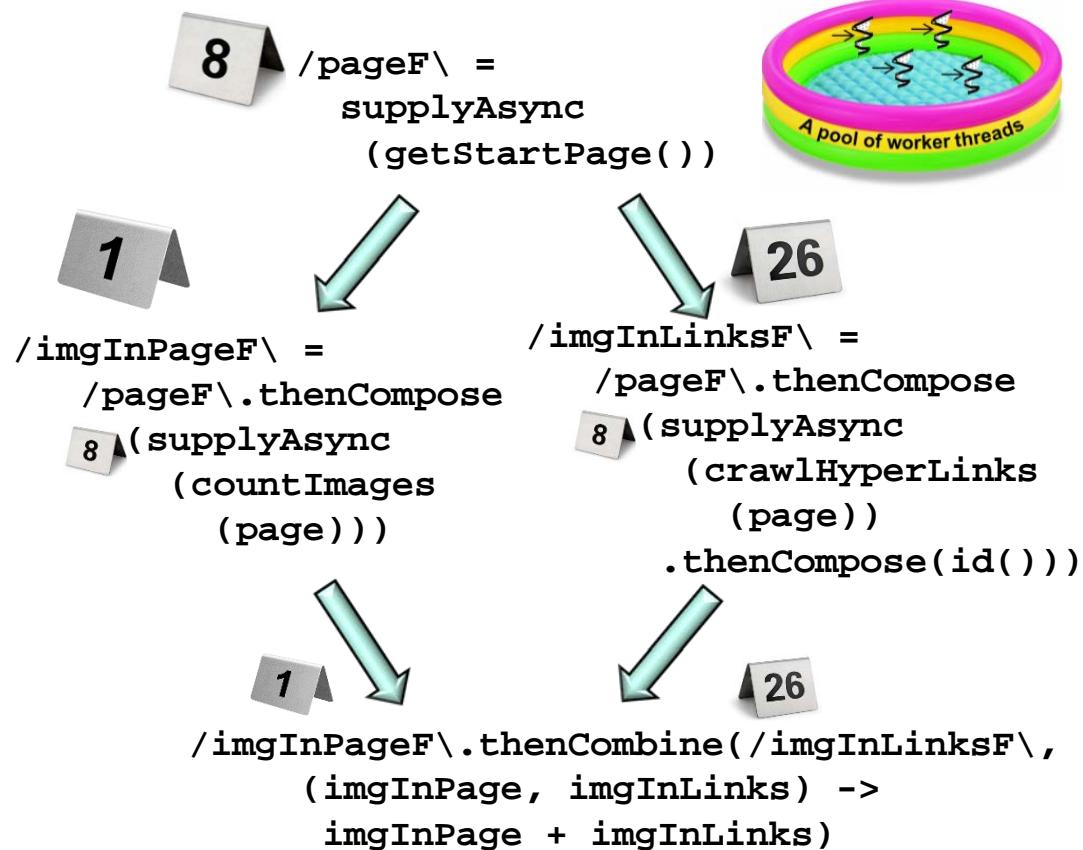
# Overview of Completable Futures

- The Java 8 completable future framework provides an async concurrent programming model
  - Supports dependent actions that trigger upon completion of async operations



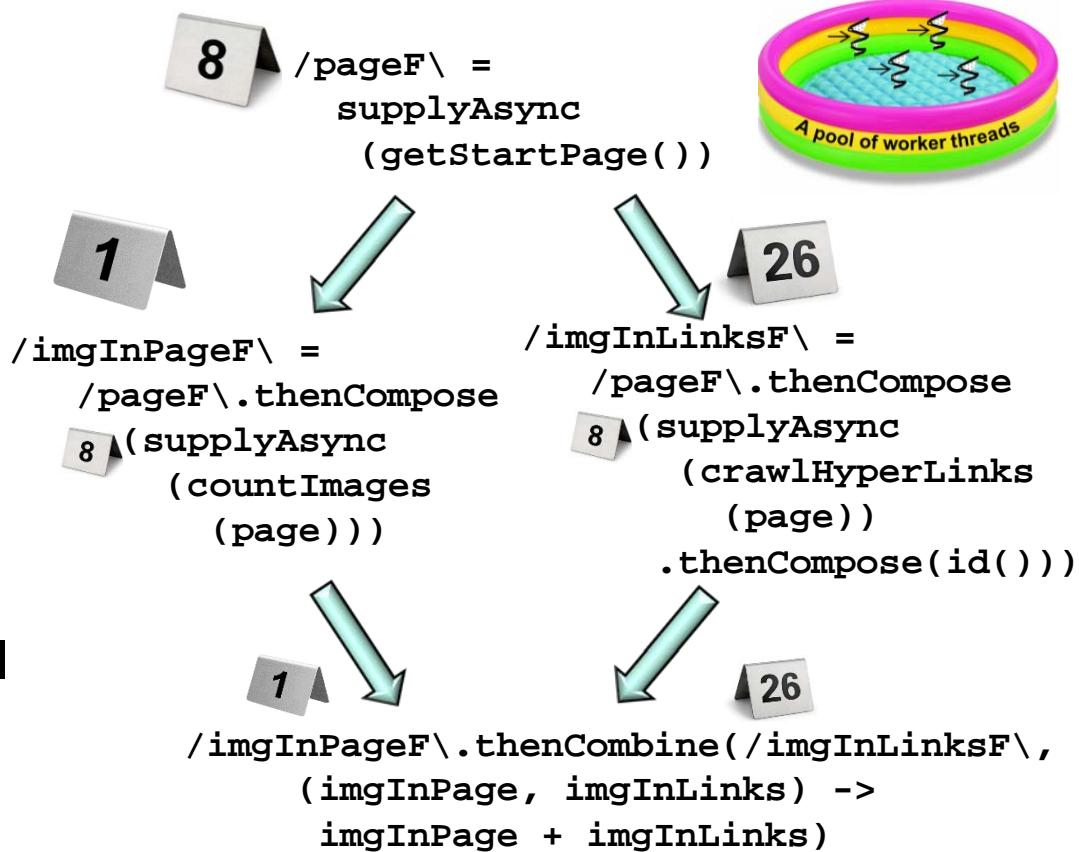
# Overview of Completable Futures

- The Java 8 completable future framework provides an async concurrent programming model
  - Supports dependent actions that trigger upon completion of async operations
  - Async operations can run concurrently in thread pools



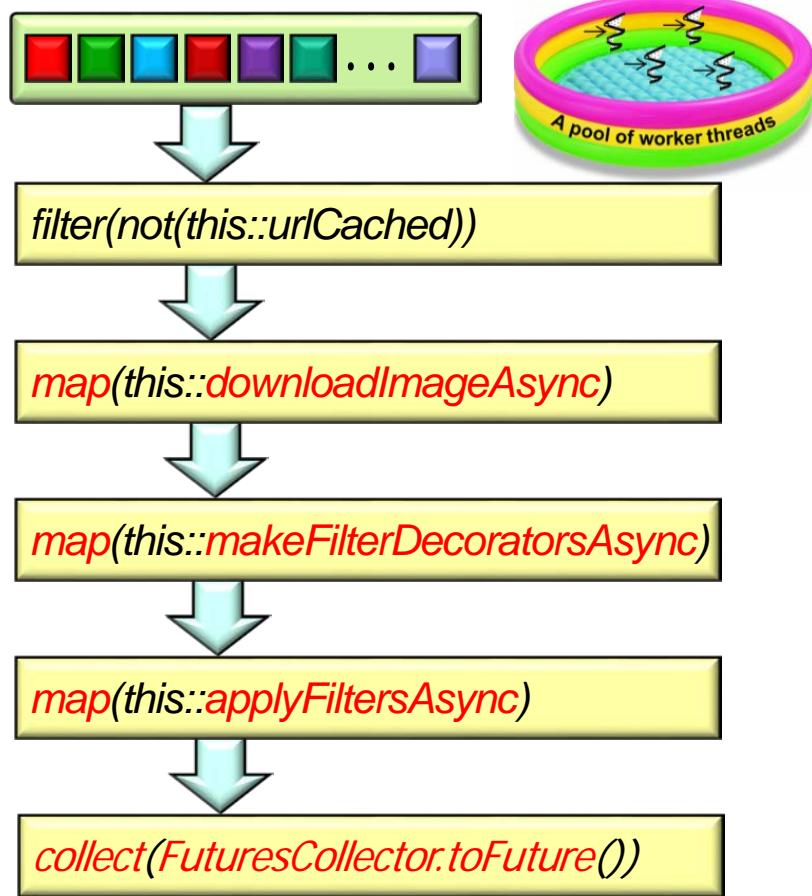
# Overview of Completable Futures

- The Java 8 completable future framework provides an async concurrent programming model
  - Supports dependent actions that trigger upon completion of async operations
  - Async operations can run concurrently in thread pools
    - Either the common fork-join pool or a user-designed pool



# Overview of Completable Futures

- Java 8 completable futures & streams can be combined to good effects!!



# Overview of Completable Futures

- The Java 8 completable futures framework often eliminates the need for synchronization or explicit threading when developing concurrent apps!



Alleviates many accidental & inherent complexities of concurrent programming

---

# Overcoming Limitations with Java Futures

# Overcoming Limitations with Java Futures

---

- The completable future framework overcomes Java future limitations



# Overcoming Limitations with Java Futures

- The completable future framework overcomes Java future limitations
  - Can be completed explicitly



you complete me

```
CompletableFuture<...> future =  
    new CompletableFuture<>();
```

```
new Thread (() -> {  
    ...  
    future.complete(...);  
}).start();
```

*After complete() is done  
calls to join() will unblock*

```
...  
System.out.println(future.join());
```

# Overcoming Limitations with Java Futures

- The completable future framework overcomes Java future limitations
  - Can be completed explicitly*
  - Can be chained together fluently to handle async results efficiently*



`CompletableFuture`

```
.supplyAsync(reduceFraction)  
.thenApply(BigFraction  
          ::toMixedString)  
.thenAccept(System.out::println);
```

*The action of each “completion stage” is triggered when the future from the previous stage completes asynchronously*

# Overcoming Limitations with Java Futures

- The completable future framework overcomes Java future limitations
  - *Can* be completed explicitly
  - *Can* be chained together fluently to handle async results efficiently
  - *Can* be triggered reactively/efficiently as a *collection* of futures w/out undue overhead



```
CompletableFuture<List<BigFraction>> futureToList =  
    Stream  
        .generate(generator)  
        .limit(sMAX_FRACTIONS)  
        .map(reduceFractions)  
        .collect(FuturesCollector  
            .toFutures());  
  
futureToList  
    .thenAccept(printList);
```

*Print out the results after all async fraction reductions have completed*

# Overcoming Limitations with Java Futures

- The completable future framework overcomes Java future limitations
  - *Can* be completed explicitly
  - *Can* be chained together fluently to handle async results efficiently
  - *Can* be triggered reactively/efficiently as a *collection* of futures w/out undue overhead



```
CompletableFuture<List  
<BigFraction>> futureToList =  
Stream  
.generate(generator)  
.limit(sMAX_FRACTIONS)  
.map(reduceFractions)  
.collect(FuturesCollector  
.toFutures());  
  
futureToList  
.thenAccept(printList);
```

*Completable futures can also be combined with Java 8 streams*

---

# End of Overview of Java 8 CompletableFuture (Part 1)

# Overview of Java 8 CompletableFuture

## (Part 2)

Douglas C. Schmidt

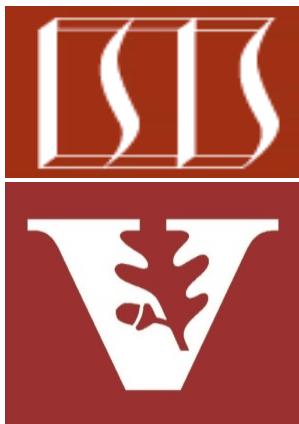
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

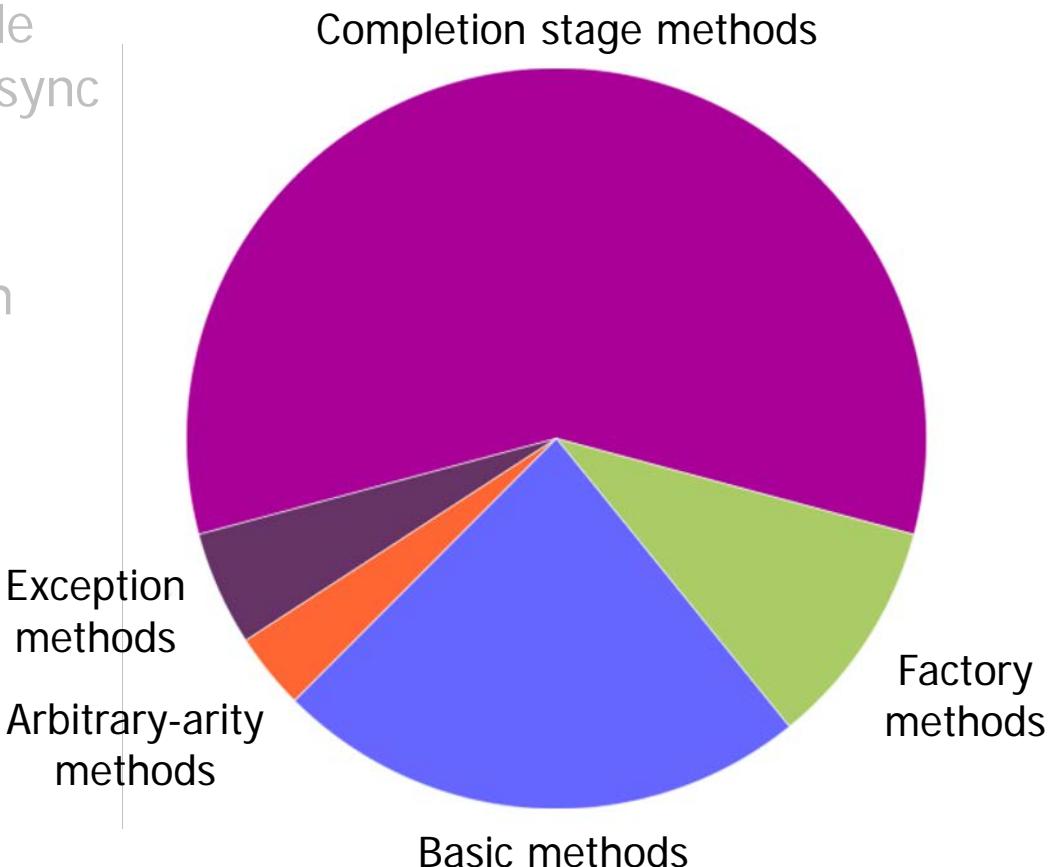
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Know how the Java 8 completable futures framework provides an async concurrent programming model
- Recognize Java 8 completable futures overcome limitations with Java futures
- Be able to group methods in the Java 8 completable future API



---

# Grouping the Java 8 CompletableFuture API

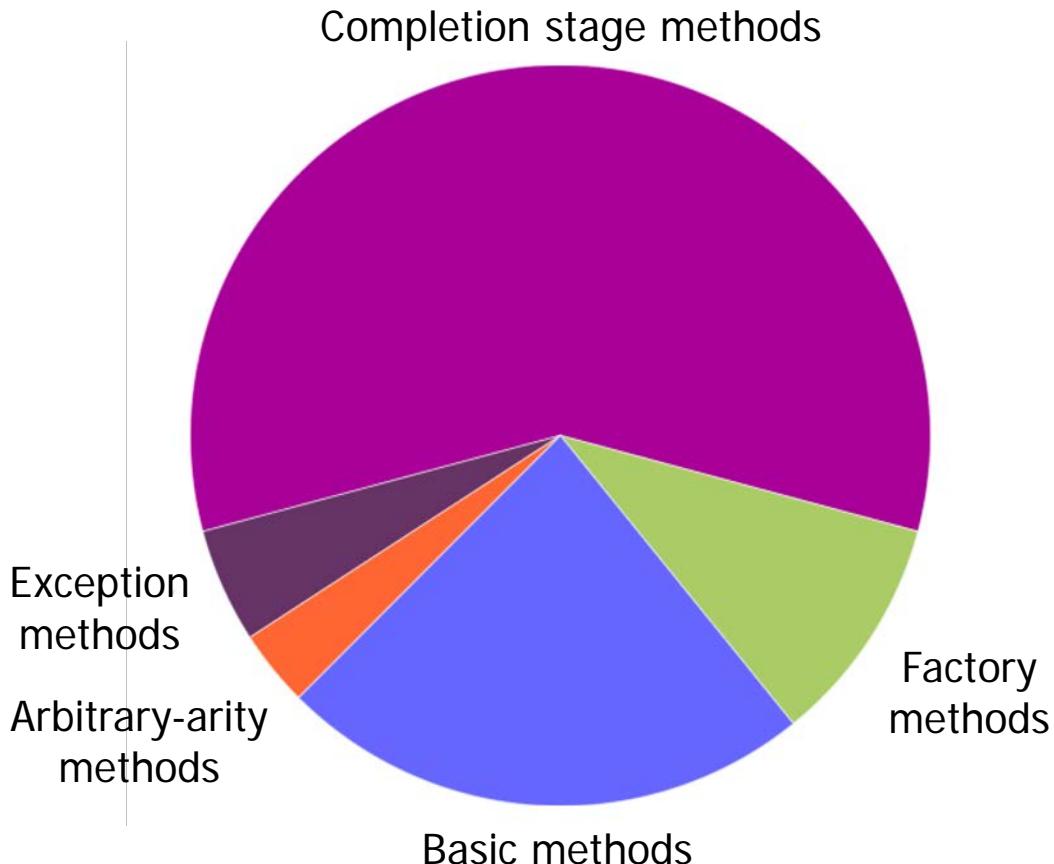
# Grouping the Java 8 CompletableFuture API

- The entire completable future framework resides in one class with 60+ methods!!!

| <<Java Class>>                                                               |  |
|------------------------------------------------------------------------------|--|
| CompletableFuture<T>                                                         |  |
| CompletableFuture()                                                          |  |
| cancel(boolean):boolean                                                      |  |
| isCancelled():boolean                                                        |  |
| isDone():boolean                                                             |  |
| get()                                                                        |  |
| get(long,TimeUnit)                                                           |  |
| join()                                                                       |  |
| complete(T):boolean                                                          |  |
| supplyAsync(Supplier<U>):CompletableFuture<U>                                |  |
| supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                       |  |
| runAsync(Runnable):CompletableFuture<Void>                                   |  |
| runAsync(Runnable,Executor):CompletableFuture<Void>                          |  |
| completedFuture(U):CompletableFuture<U>                                      |  |
| thenApply(Function<?>):CompletableFuture<U>                                  |  |
| thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |  |
| thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |  |
| thenCompose(Function<?>):CompletableFuture<U>                                |  |
| whenComplete(BiConsumer<?>):CompletableFuture<T>                             |  |
| allOf(CompletableFuture[]<?>):CompletableFuture<Void>                        |  |
| anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                      |  |

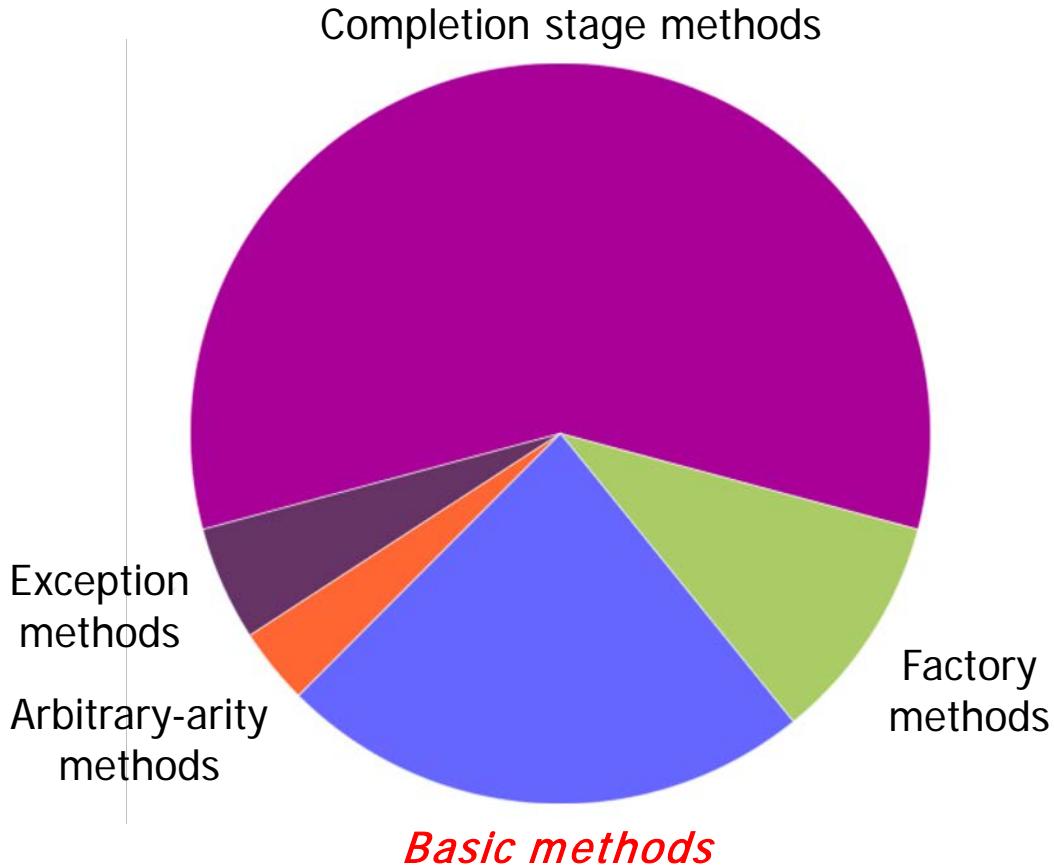
# Grouping the Java 8 CompletableFuture API

- The entire completable future framework resides in one class with 60+ methods!!!
  - It therefore helps to have a “birds-eye” view of this class



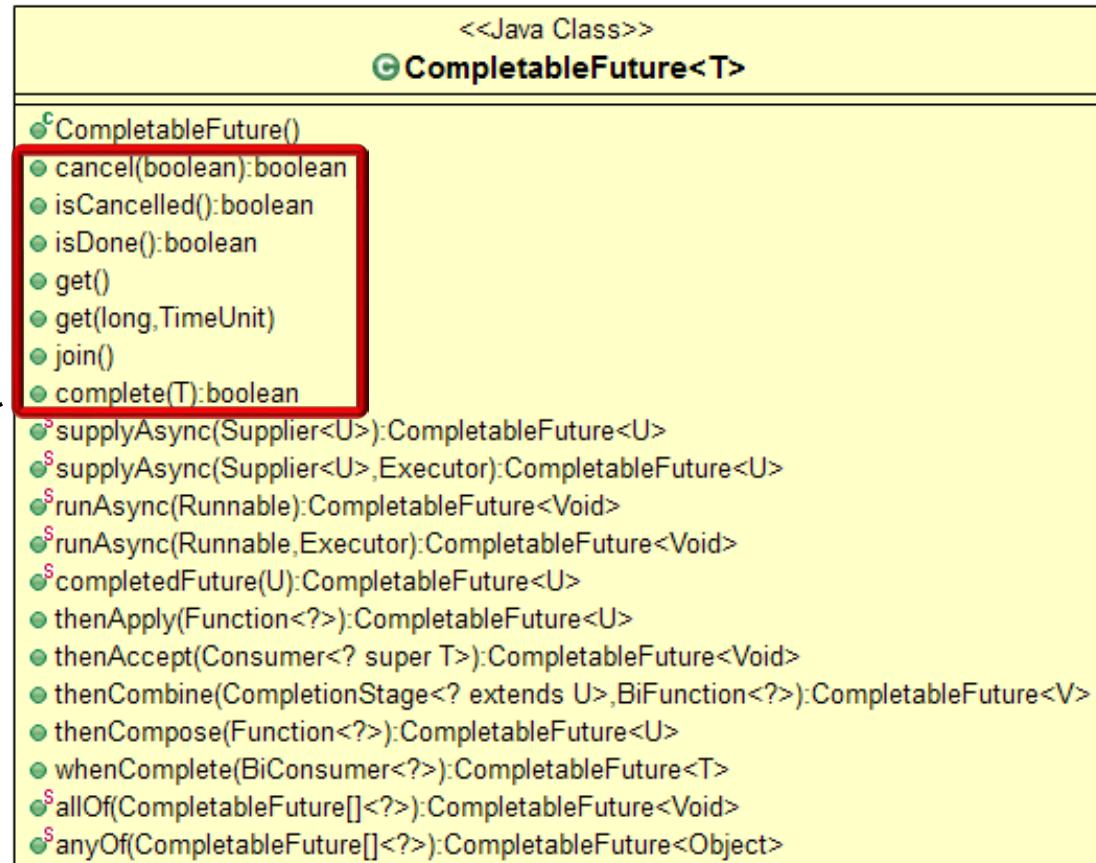
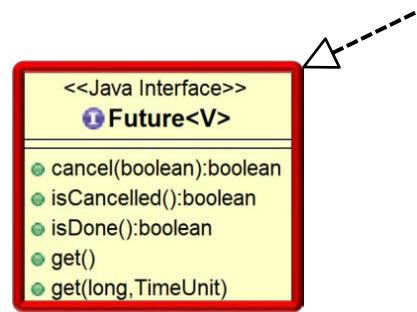
# Grouping the Java 8 CompletableFuture API

- Some completable future features are basic



# Grouping the Java 8 CompletableFuture API

- Some completable future features are basic
  - e.g., the Java Future API + some simple enhancements



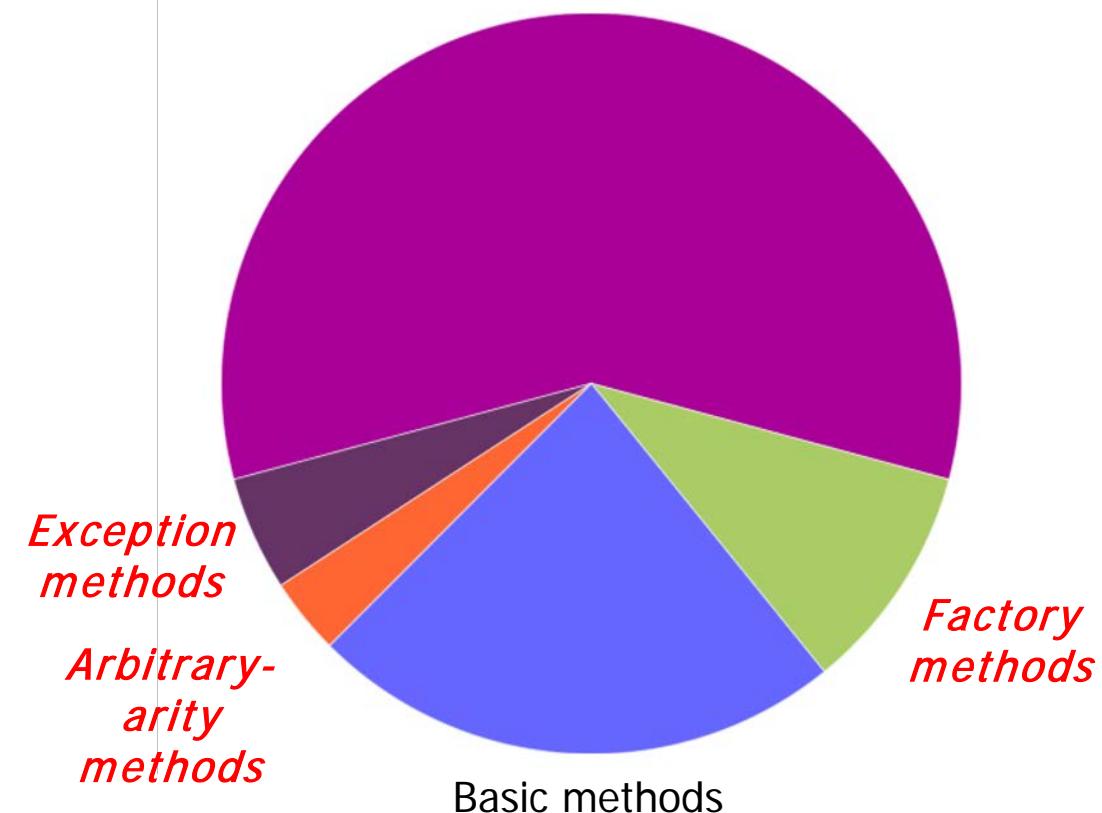
Only slightly better than the conventional Future interface

# Grouping the Java 8 CompletableFuture API

- Other completable future features are more advanced



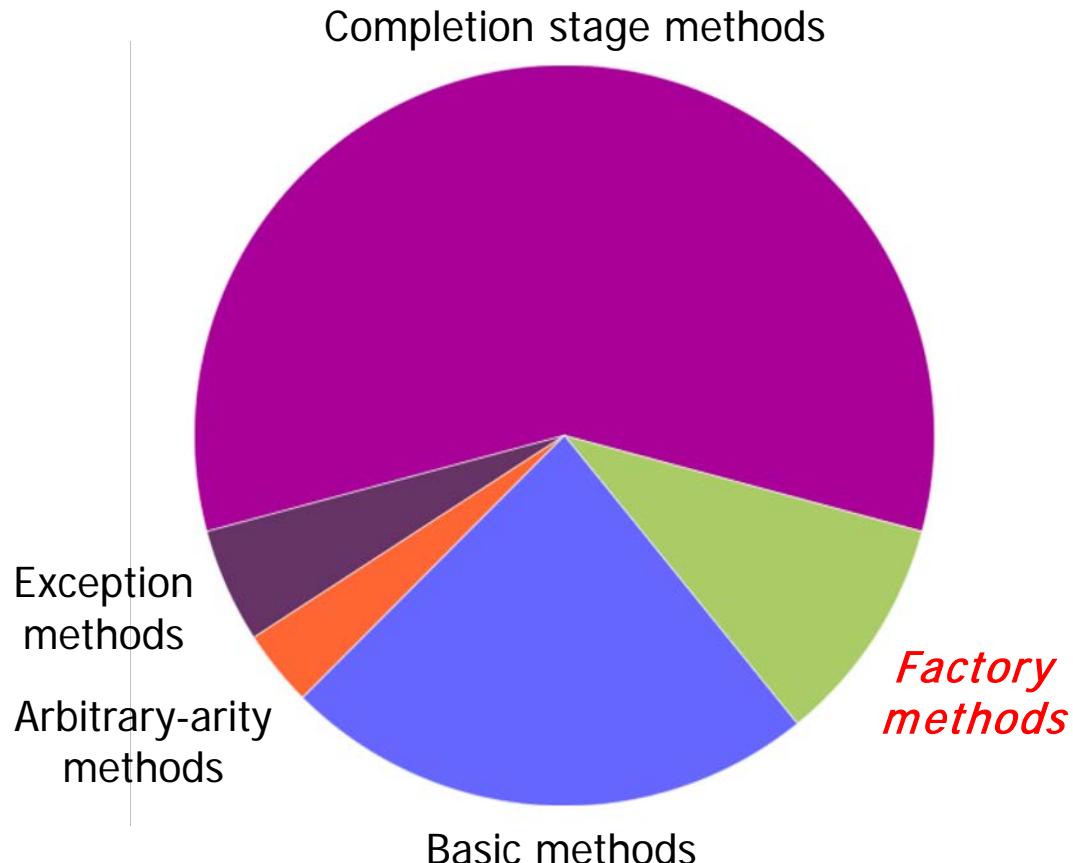
*Completion stage methods*



# Grouping the Java 8 CompletableFuture API

---

- Other completable future features are more advanced
  - Factory methods



# Grouping the Java 8 CompletableFuture API

- Other completable future features are more advanced
  - Factory methods
    - Initiate async two-way or one-way computations

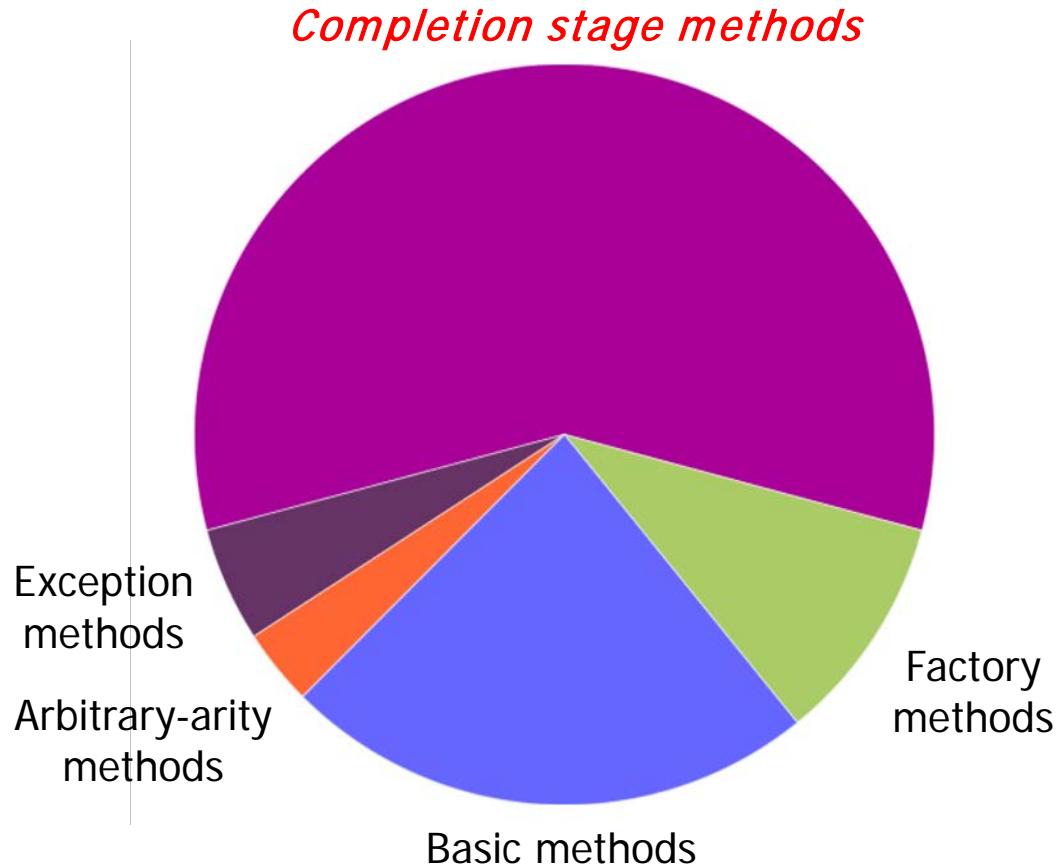
<<Java Class>>

**CompletableFuture<T>**

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• CompletableFuture()</li><li>• cancel(boolean):boolean</li><li>• isCancelled():boolean</li><li>• isDone():boolean</li><li>• get()</li><li>• get(long,TimeUnit)</li><li>• join()</li><li>• complete(T):boolean</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <ul style="list-style-type: none"><li><b>s</b> supplyAsync(Supplier&lt;U&gt;):CompletableFuture&lt;U&gt;</li><li><b>s</b> supplyAsync(Supplier&lt;U&gt;,Executor):CompletableFuture&lt;U&gt;</li><li><b>s</b> runAsync(Runnable):CompletableFuture&lt;Void&gt;</li><li><b>s</b> runAsync(Runnable,Executor):CompletableFuture&lt;Void&gt;</li><li>• completedFuture(U):CompletableFuture&lt;U&gt;</li><li>• thenApply(Function&lt;?&gt;):CompletableFuture&lt;U&gt;</li><li>• thenAccept(Consumer&lt;? super T&gt;):CompletableFuture&lt;Void&gt;</li><li>• thenCombine(CompletionStage&lt;? extends U&gt;,BiFunction&lt;?&gt;):CompletableFuture&lt;V&gt;</li><li>• thenCompose(Function&lt;?&gt;):CompletableFuture&lt;U&gt;</li><li>• whenComplete(BiConsumer&lt;?&gt;):CompletableFuture&lt;T&gt;</li><li><b>s</b> allOf(CompletableFuture[]&lt;?&gt;):CompletableFuture&lt;Void&gt;</li><li><b>s</b> anyOf(CompletableFuture[]&lt;?&gt;):CompletableFuture&lt;Object&gt;</li></ul> |

# Grouping the Java 8 CompletableFuture API

- Other completable future features are more advanced
  - Factory methods
  - Completion stage methods



# Grouping the Java 8 CompletableFuture API

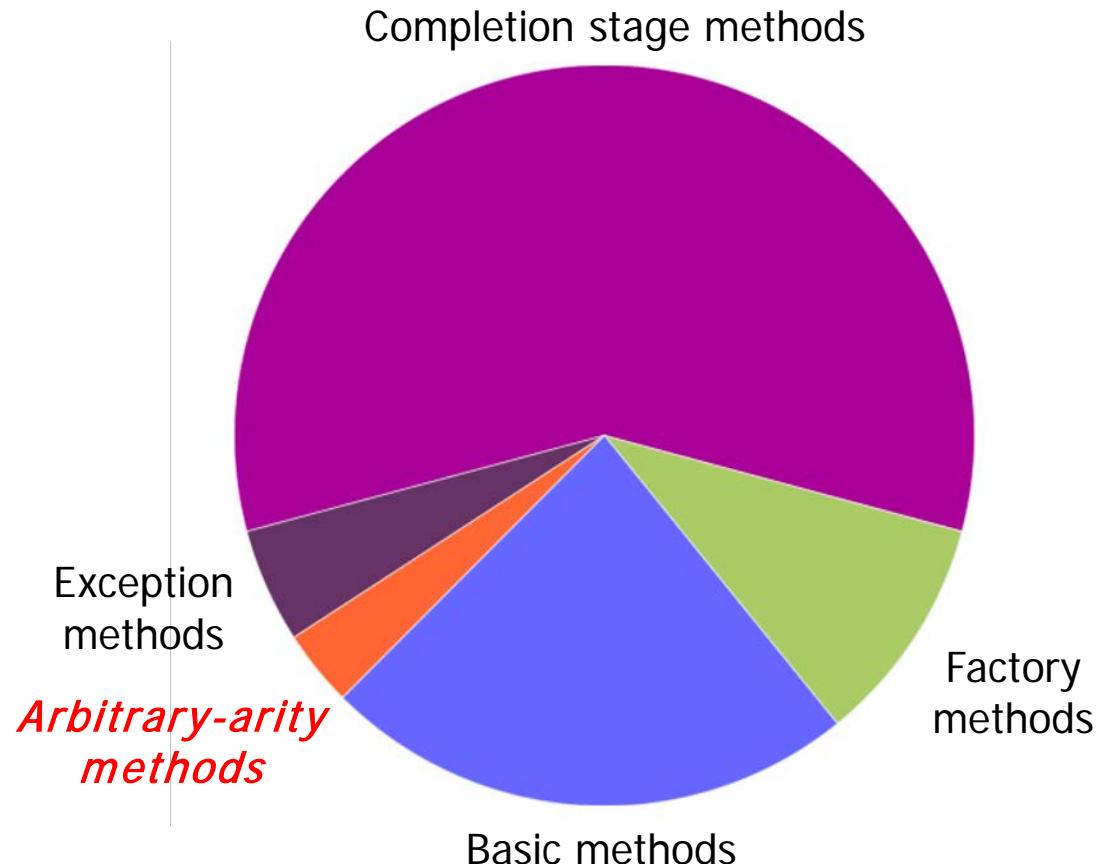
- Other completable future features are more advanced
  - Factory methods
  - Completion stage methods
    - Chain together actions that perform async result processing & composition



| <<Java Class>>       |                                                                              |
|----------------------|------------------------------------------------------------------------------|
| CompletableFuture<T> |                                                                              |
| •                    | CompletableFuture()                                                          |
| •                    | cancel(boolean):boolean                                                      |
| •                    | isCancelled():boolean                                                        |
| •                    | isDone():boolean                                                             |
| •                    | get()                                                                        |
| •                    | get(long,TimeUnit)                                                           |
| •                    | join()                                                                       |
| •                    | complete(T):boolean                                                          |
| •                    | supplyAsync(Supplier<U>):CompletableFuture<U>                                |
| •                    | supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                       |
| •                    | runAsync(Runnable):CompletableFuture<Void>                                   |
| •                    | runAsync(Runnable,Executor):CompletableFuture<Void>                          |
| •                    | completedFuture(U):CompletableFuture<U>                                      |
| •                    | thenApply(Function<?>):CompletableFuture<U>                                  |
| •                    | thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |
| •                    | thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |
| •                    | thenCompose(Function<?>):CompletableFuture<U>                                |
| •                    | whenComplete(BiConsumer<?>):CompletableFuture<T>                             |
| •                    | allOf(CompletableFuture[]<?>):CompletableFuture<Void>                        |
| •                    | anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                      |

# Grouping the Java 8 CompletableFuture API

- Other completable future features are more advanced
  - Factory methods
  - Completion stage methods
  - “Arbitrary-arity” methods that process futures in bulk



# Grouping the Java 8 CompletableFuture API

- Other completable future features are more advanced
  - Factory methods
  - Completion stage methods
  - “Arbitrary-arity” methods that process futures in bulk
    - Combine multiple futures into a single future

| <<Java Class>>                                                               |  |
|------------------------------------------------------------------------------|--|
| CompletableFuture<T>                                                         |  |
| CompletableFuture()                                                          |  |
| cancel(boolean):boolean                                                      |  |
| isCancelled():boolean                                                        |  |
| isDone():boolean                                                             |  |
| get()                                                                        |  |
| get(long,TimeUnit)                                                           |  |
| join()                                                                       |  |
| complete(T):boolean                                                          |  |
| supplyAsync(Supplier<U>):CompletableFuture<U>                                |  |
| supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                       |  |
| runAsync(Runnable):CompletableFuture<Void>                                   |  |
| runAsync(Runnable,Executor):CompletableFuture<Void>                          |  |
| completedFuture(U):CompletableFuture<U>                                      |  |
| thenApply(Function<?>):CompletableFuture<U>                                  |  |
| thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |  |
| thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |  |
| thenCompose(Function<?>):CompletableFuture<U>                                |  |
| whenComplete(BiConsumer<?>):CompletableFuture<T>                             |  |
| allOf(CompletableFuture[]<?>):CompletableFuture<Void>                        |  |
| anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                      |  |

---

# End of Overview of Java 8 CompletableFuture (Part 2)

# Overview of Java 8 CompletableFuture

## (Part 3)

Douglas C. Schmidt

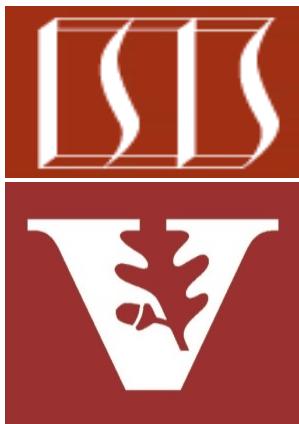
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

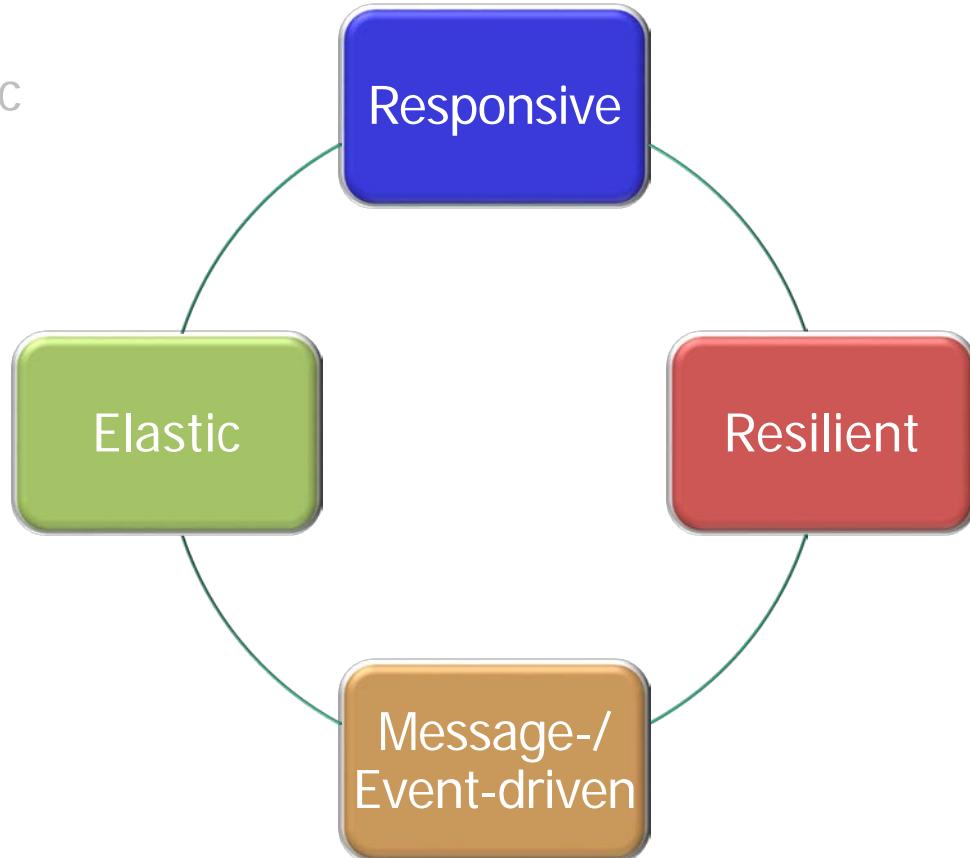
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Know how the Java 8 completable futures framework provides an async concurrent programming model
- Recognize Java 8 completable futures overcome limitations with Java futures
- Be able to group methods in the Java 8 completable future API
- Understand the relationship between Java 8 completable futures & reactive programming

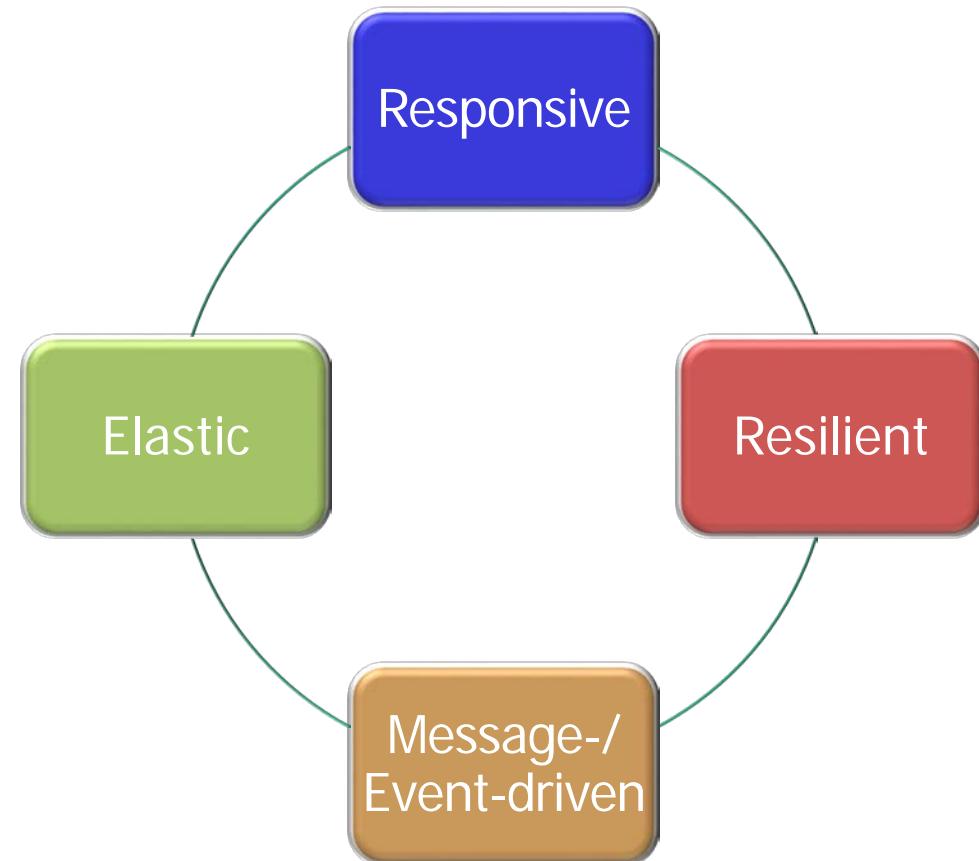


---

# Java 8 CompletableFuture & Reactive Programming

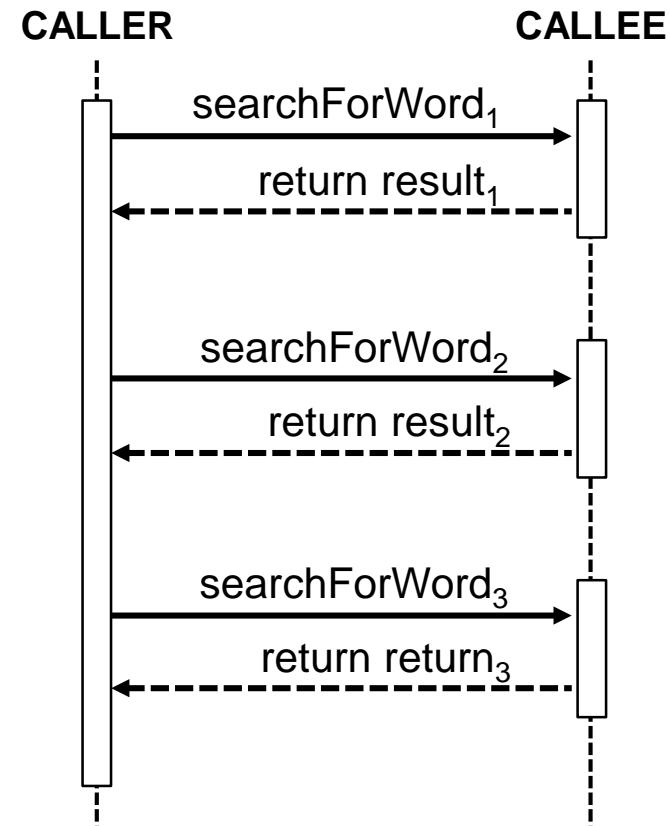
# Java 8 Completable Futures & Reactive Programming

- Java 8 completable futures map nicely onto key principles of reactive programming



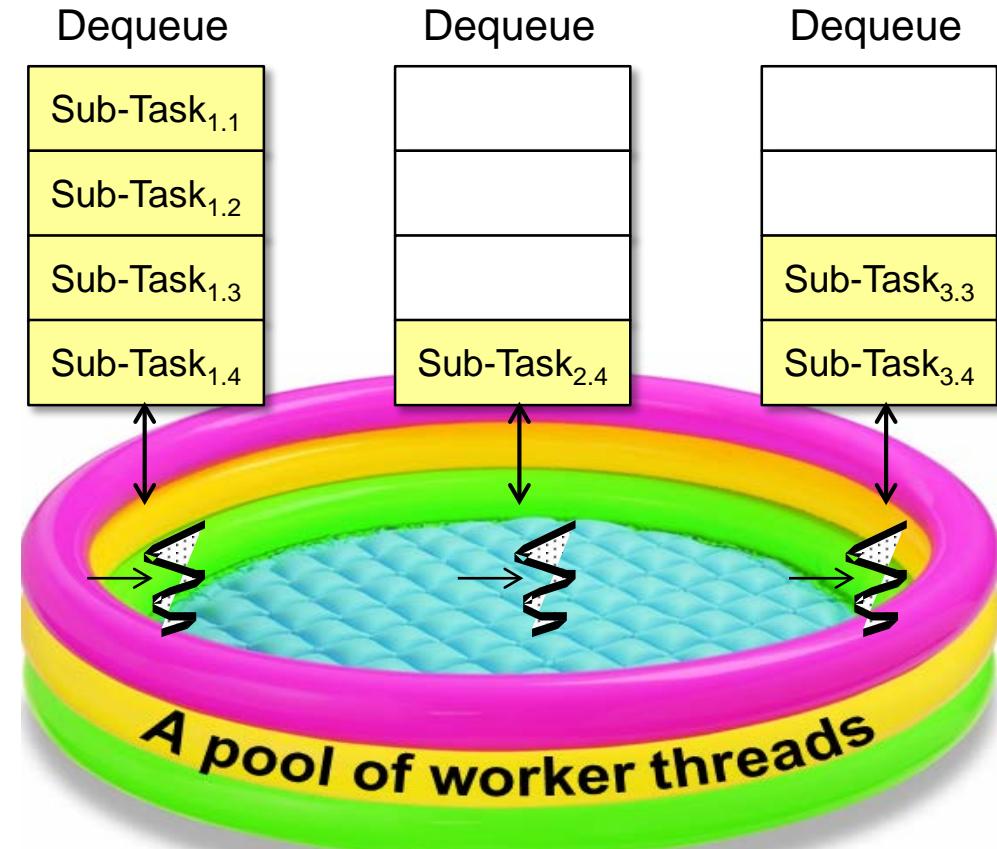
# Java 8 Completable Futures & Reactive Programming

- Java 8 completable futures map nicely onto key principles of reactive programming, e.g.
  - Avoid blocking threads
    - Blocking underutilizes cores, impedes inherent parallelism, & complicates program structure



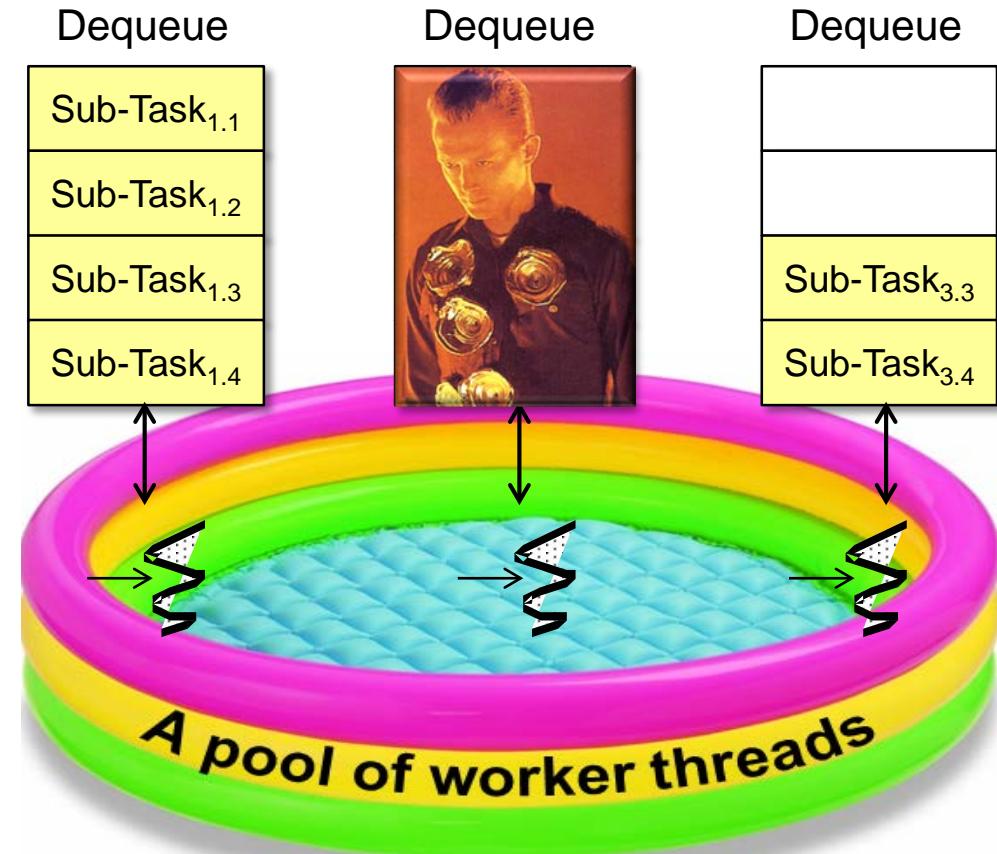
# Java 8 Completable Futures & Reactive Programming

- Java 8 completable futures map nicely onto key principles of reactive programming, e.g.
  - Avoid blocking threads
  - Avoid changing threads
    - Incurs excessive context switching, synchronization, memory/cache management overhead



# Java 8 Completable Futures & Reactive Programming

- Java 8 completable futures map nicely onto key principles of reactive programming, e.g.
  - Avoid blocking threads
  - Avoid changing threads
  - Avoid crippling failures
    - Failure of some operations shouldn't bring the whole system down



# Java 8 Completable Futures & Reactive Programming

- Reactive programming in Java 9 is supported via “Reactive Streams” & the Flow API

## Class Flow

java.lang.Object  
java.util.concurrent.Flow

```
public final class Flow  
extends Object
```

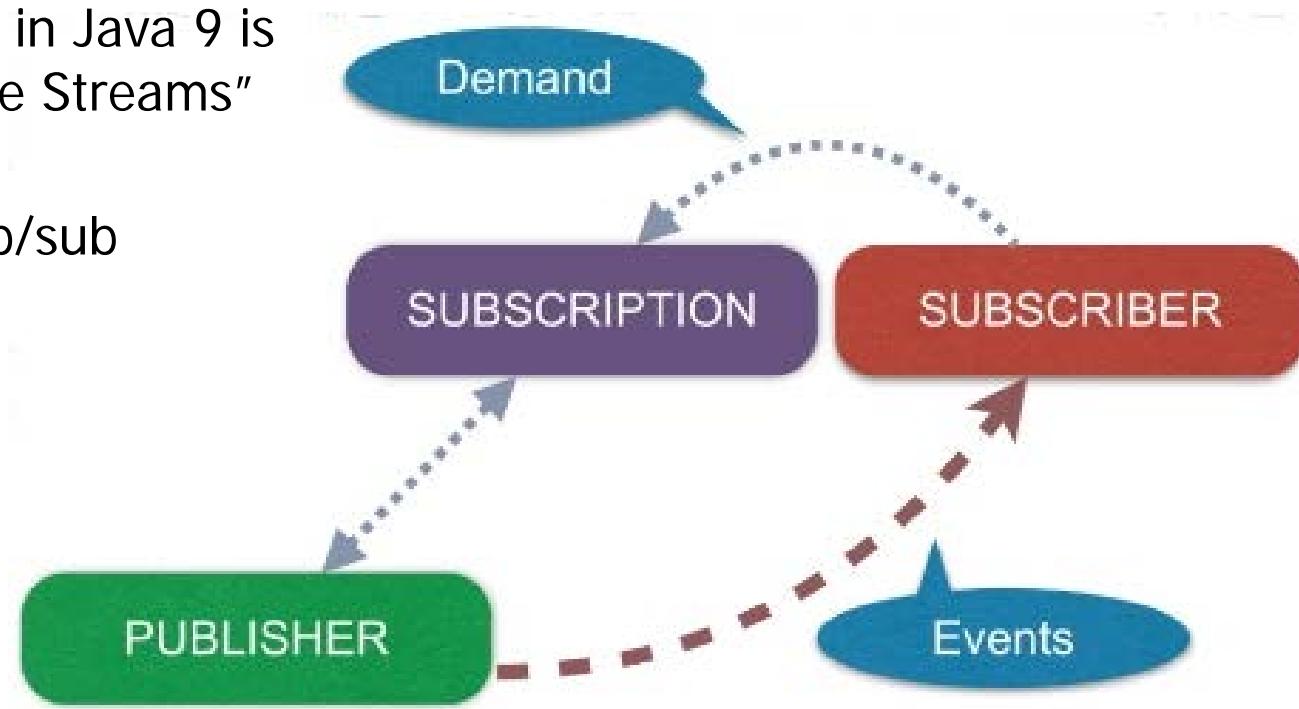
Interrelated interfaces and static methods for establishing flow-controlled components in which Publishers produce items consumed by one or more Subscribers, each managed by a Subscription.

These interfaces correspond to the reactive-streams specification. They apply in both concurrent and distributed asynchronous settings: All (seven) methods are defined in void "one-way" message style. Communication relies on a simple form of flow control (method `Flow.Subscription.request(long)`) that can be used to avoid resource management problems that may otherwise occur in "push" based systems.

**Examples.** A `Flow.Publisher` usually defines its own `Flow.Subscription` implementation; constructing one in method `subscribe` and issuing it to the calling `Flow.Subscriber`. It publishes items to the subscriber asynchronously, normally using an Executor. For example, here is a very simple publisher that only issues (when requested) a single TRUE item to a single subscriber. Because the subscriber receives only a single item, this class does not use buffering and ordering control required in most implementations (for example `SubmissionPublisher`).

# Java 8 Completable Futures & Reactive Programming

- Reactive programming in Java 9 is supported via “Reactive Streams” & the Flow API
  - Adds support for pub/sub patterns



---

# End of Overview of Java 8 CompletableFuture (Part 3)

# Overview of Basic Java 8 CompletableFuture Features (Part 1)

Douglas C. Schmidt

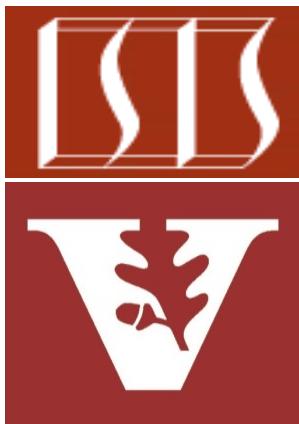
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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 the basic completable futures features



## 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.

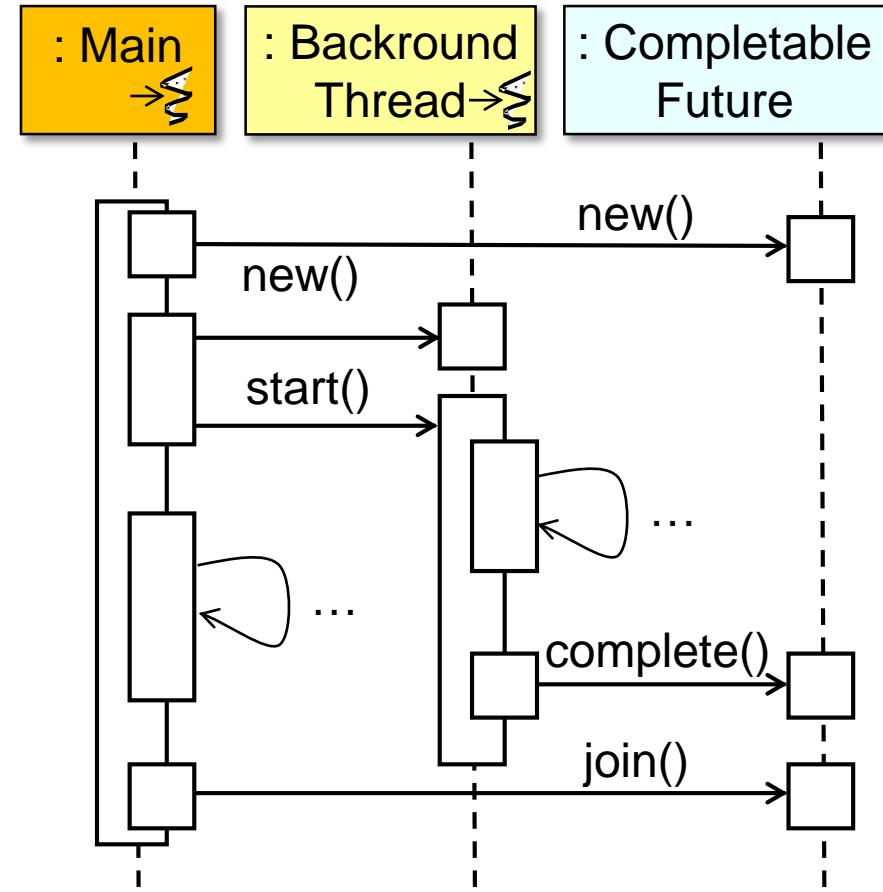
In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

---

# Basic Completable Future Features

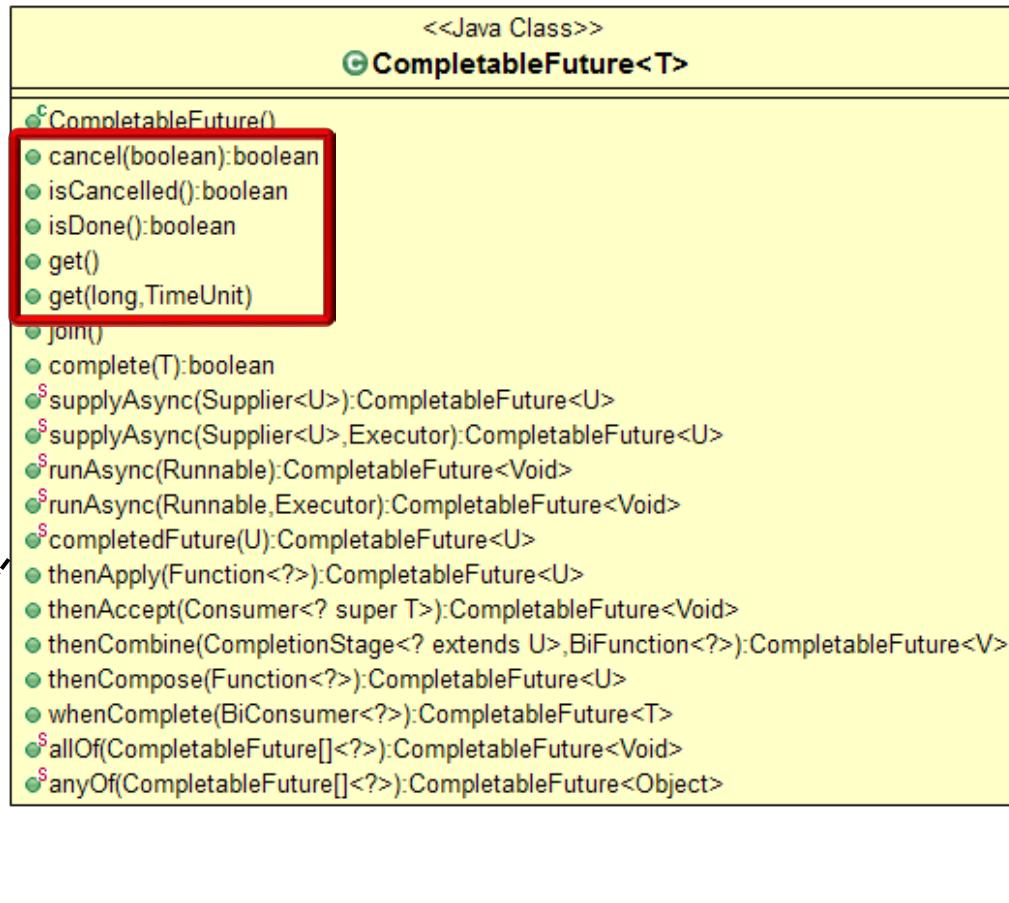
# Basic Completable Future Features

- Basic completable future features



# Basic Completable Future Features

- Basic completable future features
  - Support the Future API



See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)

# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Can (time-) block & poll

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

CompletableFuture<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);
```

# Basic Completable Future Features

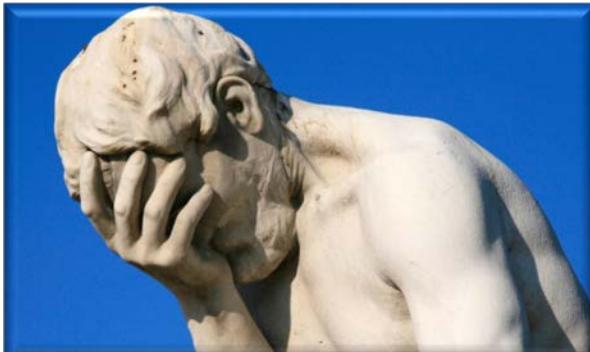
- Basic completable future features
  - Support the Future API
    - Can (time-) block & poll
    - Can be cancelled & tested if canceled/done

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

CompletableFuture<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();
```

# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
    - Can (time-) block & poll
    - Can be cancelled & tested if canceled/done
    - cancel() doesn't interrupt the computation by default..

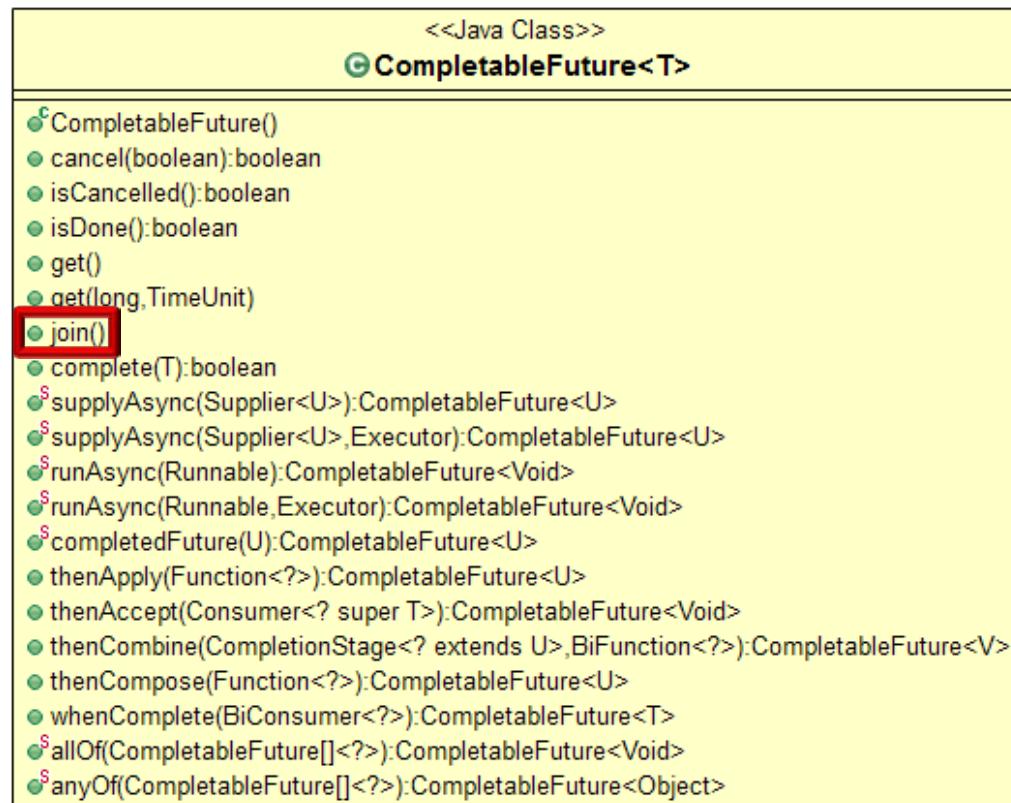


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

CompletableFuture<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();
```

# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method



# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
    - Behaves like get() *without* using checked exceptions

**futures**

```
.stream()
.map(Future::join)
.collect(toList())
```

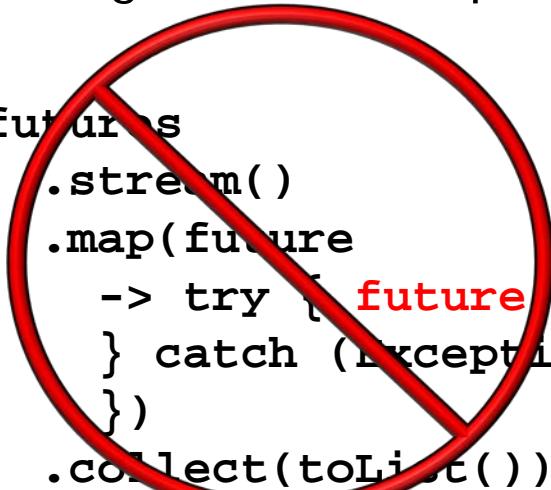
<<Java Class>>

**CompletableFuture<T>**

|                                                                                |
|--------------------------------------------------------------------------------|
| • CompletableFuture()                                                          |
| • cancel(boolean):boolean                                                      |
| • isCancelled():boolean                                                        |
| • isDone():boolean                                                             |
| • get()                                                                        |
| • get(long,TimeUnit)                                                           |
| • <b>join()</b>                                                                |
| • complete(T):boolean                                                          |
| • supplyAsync(Supplier<U>):CompletableFuture<U>                                |
| • supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                       |
| • runAsync(Runnable):CompletableFuture<Void>                                   |
| • runAsync(Runnable,Executor):CompletableFuture<Void>                          |
| • completedFuture(U):CompletableFuture<U>                                      |
| • thenApply(Function<?>):CompletableFuture<U>                                  |
| • thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |
| • thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |
| • thenCompose(Function<?>):CompletableFuture<U>                                |
| • whenComplete(BiConsumer<?>):CompletableFuture<T>                             |
| • allOf(CompletableFuture[]<?>):CompletableFuture<Void>                        |
| • anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                      |

# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
    - Behaves like get() *without* using checked exceptions



```
futures
    .stream()
    .map(future
        -> try {
            future.get();
        } catch (Exception e){
        })
    .collect(toList())
```

«Java Class»

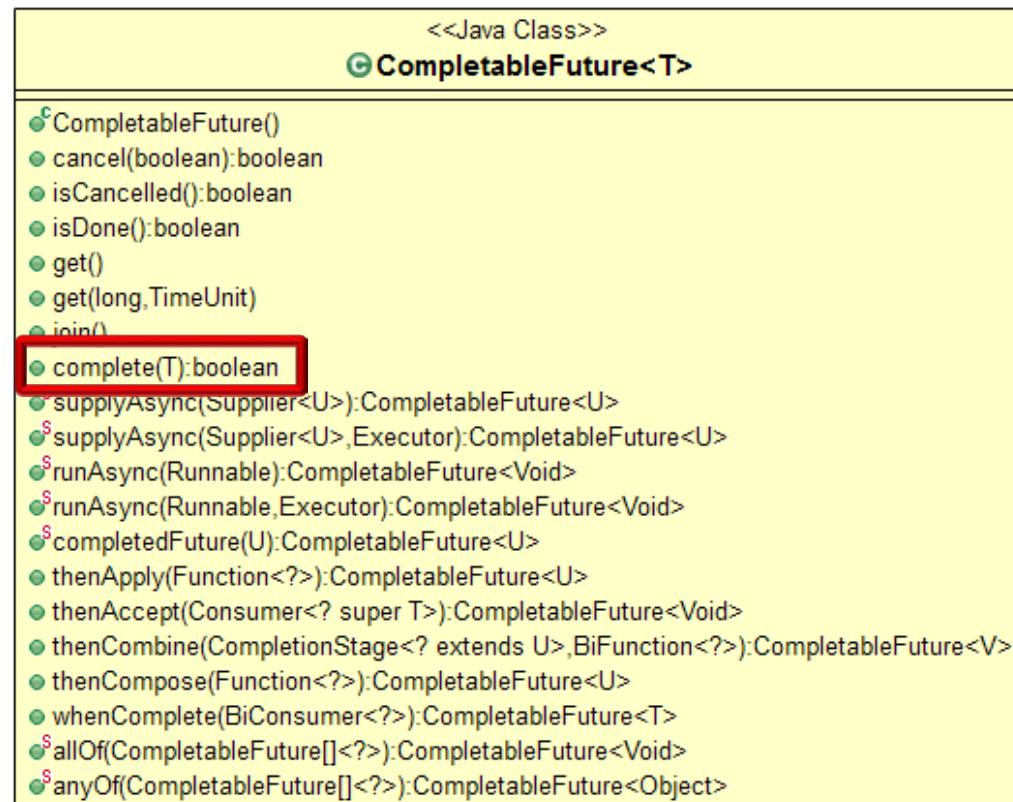
**CompletableFuture<T>**

|                                                                                |
|--------------------------------------------------------------------------------|
| • CompletableFuture()                                                          |
| • cancel(boolean):boolean                                                      |
| • isCancelled():boolean                                                        |
| • isDone():boolean                                                             |
| • get()                                                                        |
| • get(long,TimeUnit)                                                           |
| • <b>join()</b>                                                                |
| • complete(T):boolean                                                          |
| • \$ supplyAsync(Supplier<U>):CompletableFuture<U>                             |
| • \$ supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                    |
| • \$ runAsync(Runnable):CompletableFuture<Void>                                |
| • \$ runAsync(Runnable,Executor):CompletableFuture<Void>                       |
| • \$ completedFuture(U):CompletableFuture<U>                                   |
| • thenApply(Function<?>):CompletableFuture<U>                                  |
| • thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |
| • thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |
| • thenCompose(Function<?>):CompletableFuture<U>                                |
| • whenComplete(BiConsumer<?>):CompletableFuture<T>                             |
| • \$ allOf(CompletableFuture[]<?>):CompletableFuture<Void>                     |
| • \$ anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                   |

Mixing checked exceptions & Java 8 streams is ugly..

# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly



# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - i.e., sets result returned by get()/join() to a given value

```
CompletableFuture<...> future =  
    new CompletableFuture<>();  
  
new Thread (() -> {  
    ...  
    future.complete(...);  
}).start();  
  
...  
System.out.println(future.join());
```

*After complete() is done  
calls to join() will unblock*

# Basic Completable Future Features

- Basic completable future features
  - Support the Future API
  - Define a join() method
  - Can be completed explicitly
    - i.e., sets result returned by get()/join() to a given value

*A completable future can be initialized to a value/constant*

```
CompletableFuture<...> future =  
    new CompletableFuture<>();  
  
CompletableFuture<Long> zero =  
    CompletableFuture  
        .completedFuture(0L);  
  
new Thread (() -> {  
    ...  
    future.complete(zero.join());  
}).start();  
  
...  
System.out.println(future.join());
```

---

End of Overview of  
Basic Java 8 Completable  
Future Features (Part 1)

# Overview of Basic Java 8 CompletableFuture Features (Part 2)

Douglas C. Schmidt

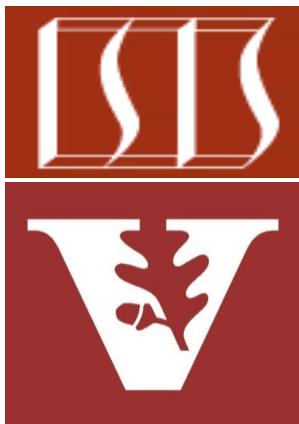
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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 the basic completable futures features
- Know how to apply these basic features to multiply big fractions

|                                              |
|----------------------------------------------|
| <<Java Class>>                               |
| <b>BigFraction</b><br>(default package)      |
| mNumerator: BigInteger                       |
| mDenominator: BigInteger                     |
| S valueOf(Number):BigFraction                |
| S valueOf(Number,Number):BigFraction         |
| S valueOf(String):BigFraction                |
| S valueOf(Number,Number,boolean):BigFraction |
| S reduce(BigFraction):BigFraction            |
| F getNumerator():BigInteger                  |
| F getDenominator():BigInteger                |
| S add(Number):BigFraction                    |
| S subtract(Number):BigFraction               |
| S multiply(Number):BigFraction               |
| S divide(Number):BigFraction                 |
| S gcd(Number):BigFraction                    |
| S toMixedString():String                     |

# Learning Objectives in this Part of the Lesson

---

- Understand the basic completable futures features
- Know how to apply these basic features to multiply big fractions
- Recognize limitations with these basic features

**LIMITED**

## 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.

In addition to these and related methods for directly manipulating status and results, `CompletableFuture` implements interface `CompletionStage` with the following policies:

---

# Applying Basic Completable Future Features

# Applying Basic Completable Future Features

- We show how to apply basic completable future features in the context of BigFraction

|                                                |
|------------------------------------------------|
| <<Java Class>>                                 |
| <b>G BigFraction</b><br>(default package)      |
| ↳ F mNumerator: BigInteger                     |
| ↳ F mDenominator: BigInteger                   |
| ↳ S valueOf(Number):BigFraction                |
| ↳ S valueOf(Number,Number):BigFraction         |
| ↳ S valueOf(String):BigFraction                |
| ↳ S valueOf(Number,Number,boolean):BigFraction |
| ↳ S reduce(BigFraction):BigFraction            |
| ↳ G getNumerator():BigInteger                  |
| ↳ G getDenominator():BigInteger                |
| ↳ G add(Number):BigFraction                    |
| ↳ G subtract(Number):BigFraction               |
| ↳ G multiply(Number):BigFraction               |
| ↳ G divide(Number):BigFraction                 |
| ↳ G gcd(Number):BigFraction                    |
| ↳ G toMixedString():String                     |

# Applying Basic Completable Future Features

- We show how to apply basic completable future features in the context of BigFraction
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator

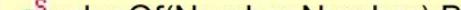
<<Java Class>>

**G BigFraction**  
(default package)

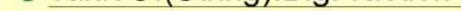
 

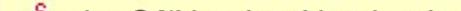
 

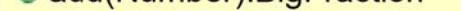
 

<img alt="Red box highlighting mNumerator and mDenominator" data-bbox="595 35

# Applying Basic Completable Future Features

- We show how to apply basic completable future features in the context of BigFraction
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating “reduced” fractions, e.g.
    - $44/55 \rightarrow 4/5$
    - $12/24 \rightarrow 1/2$
    - $144/216 \rightarrow 2/3$

<<Java Class>>

**G BigFraction**  
(default package)

**mNumerator: BigInteger**  
**mDenominator: BigInteger**

**S valueOf(Number):BigFraction**  
**S valueOf(Number,Number):BigFraction**  
**S valueOf(String):BigFraction**

**S valueOf(Number,Number,boolean):BigFraction**  
**S reduce(BigFraction):BigFraction**  
**S getNumerator():BigInteger**  
**S getDenominator():BigInteger**  
**S add(Number):BigFraction**  
**S subtract(Number):BigFraction**  
**S multiply(Number):BigFraction**  
**S divide(Number):BigFraction**  
**S gcd(Number):BigFraction**  
**S toMixedString():String**

# Applying Basic Completable Future Features

- We show how to apply basic completable future features in the context of BigFraction
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating “reduced” fractions
  - Factory methods for creating “non-reduced” fractions (& then reducing them)
    - e.g.,  $12/24 \rightarrow 1/2$

|                                                       |
|-------------------------------------------------------|
| <<Java Class>>                                        |
| <b>G BigFraction</b>                                  |
| (default package)                                     |
| ↳ mNumerator: BigInteger                              |
| ↳ mDenominator: BigInteger                            |
| ↳ <u>S valueOf(Number):BigFraction</u>                |
| ↳ <u>S valueOf(Number,Number):BigFraction</u>         |
| ↳ <u>S valueOf(String):BigFraction</u>                |
| ↳ <u>S valueOf(Number,Number,boolean):BigFraction</u> |
| ↳ <u>S reduce(BigFraction):BigFraction</u>            |
| ↳ getNumerator():BigInteger                           |
| ↳ getDenominator():BigInteger                         |
| ↳ add(Number):BigFraction                             |
| ↳ subtract(Number):BigFraction                        |
| ↳ multiply(Number):BigFraction                        |
| ↳ divide(Number):BigFraction                          |
| ↳ gcd(Number):BigFraction                             |
| ↳ toMixedString():String                              |

# Applying Basic Completable Future Features

- We show how to apply basic completable future features in the context of BigFraction
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating “reduced” fractions
  - Factory methods for creating “non-reduced” fractions (& then reducing them)
  - Arbitrary-precision fraction arithmetic
    - e.g.,  $18/4 \times 2/3 = 3$

|                                                                                     |
|-------------------------------------------------------------------------------------|
| <p>&lt;&lt;Java Class&gt;&gt;</p> <p><b>G BigFraction</b><br/>(default package)</p> |
| <p>▫ mNumerator: BigInteger</p>                                                     |
| <p>▫ mDenominator: BigInteger</p>                                                   |
| <p>§ <u>valueOf(Number):BigFraction</u></p>                                         |
| <p>§ <u>valueOf(Number,Number):BigFraction</u></p>                                  |
| <p>§ <u>valueOf(String):BigFraction</u></p>                                         |
| <p>§ <u>valueOf(Number,Number,boolean):BigFraction</u></p>                          |
| <p>§ <u>reduce(BigFraction):BigFraction</u></p>                                     |
| <p>§ <u>getNumerator():BigInteger</u></p>                                           |
| <p>§ <u>getDenominator():BigInteger</u></p>                                         |
| <p>§ <u>add(Number):BigFraction</u></p>                                             |
| <p>§ <u>subtract(Number):BigFraction</u></p>                                        |
| <p>§ <u>multiply(Number):BigFraction</u></p>                                        |
| <p>§ <u>divide(Number):BigFraction</u></p>                                          |
| <p>§ <u>gcd(Number):BigFraction</u></p>                                             |
| <p>§ <u>toMixedString():String</u></p>                                              |

# Applying Basic Completable Future Features

- We show how to apply basic completable future features in the context of BigFraction
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator & denominator
  - Factory methods for creating “reduced” fractions
  - Factory methods for creating “non-reduced” fractions (& then reducing them)
  - Arbitrary-precision fraction arithmetic
  - Create a mixed fraction from an improper fraction
    - e.g.,  $18/4 \rightarrow 4 \frac{1}{2}$

|                                                       |
|-------------------------------------------------------|
| <<Java Class>>                                        |
| <b>G BigFraction</b>                                  |
| (default package)                                     |
| ↳ mNumerator: BigInteger                              |
| ↳ mDenominator: BigInteger                            |
| ↳ <u>S valueOf(Number):BigFraction</u>                |
| ↳ <u>S valueOf(Number,Number):BigFraction</u>         |
| ↳ <u>S valueOf(String):BigFraction</u>                |
| ↳ <u>S valueOf(Number,Number,boolean):BigFraction</u> |
| ↳ <u>S reduce(BigFraction):BigFraction</u>            |
| ↳ <u>G getNumerator():BigInteger</u>                  |
| ↳ <u>G getDenominator():BigInteger</u>                |
| ↳ add(Number):BigFraction                             |
| ↳ subtract(Number):BigFraction                        |
| ↳ multiply(Number):BigFraction                        |
| ↳ divide(Number):BigFraction                          |
| ↳ gcd(Number):BigFraction                             |
| ↳ <b>toMixedString():String</b>                       |

# Applying Basic Completable Future Features

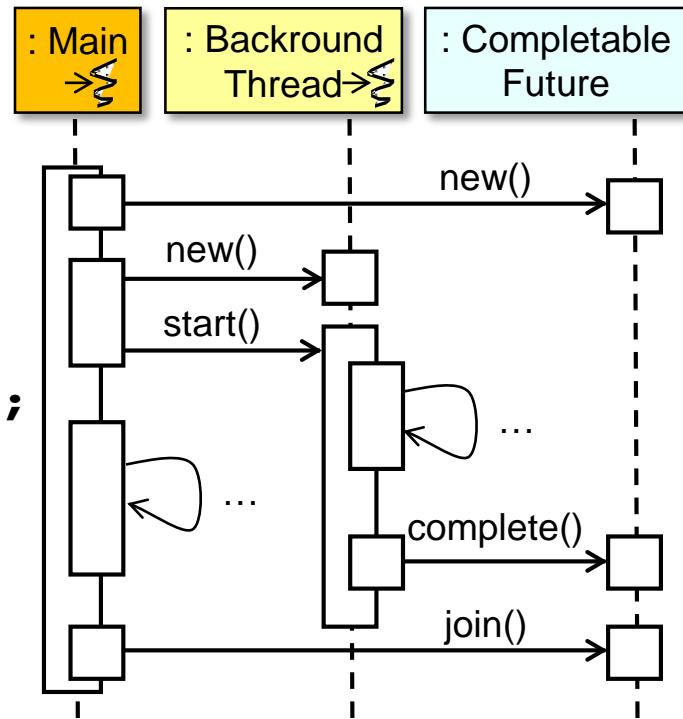
- Multiplying big fractions w/a completable future

```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");  
  
    future.complete(bf1.multiply(bf2));  
}).start();
```

...

```
System.out.println(future.join().toMixedString());
```



See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex8](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex8)

# Applying Basic Completable Future Features

- Multiplying big fractions w/a completable future

```
CompletableFuture<BigFraction> future
```

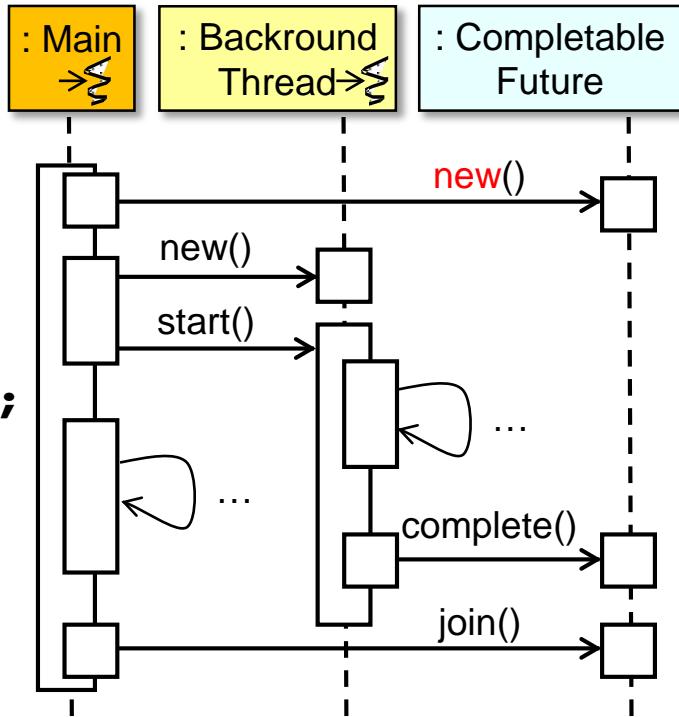
```
    = new CompletableFuture<>();
```

```
new Thread (() -> {      Make "empty" future
    BigFraction bf1 =
        new BigFraction("62675744/15668936");
    BigFraction bf2 =
        new BigFraction("609136/913704");

    future.complete(bf1.multiply(bf2));
}).start();
```

```
...
```

```
System.out.println(future.join().toMixedString());
```



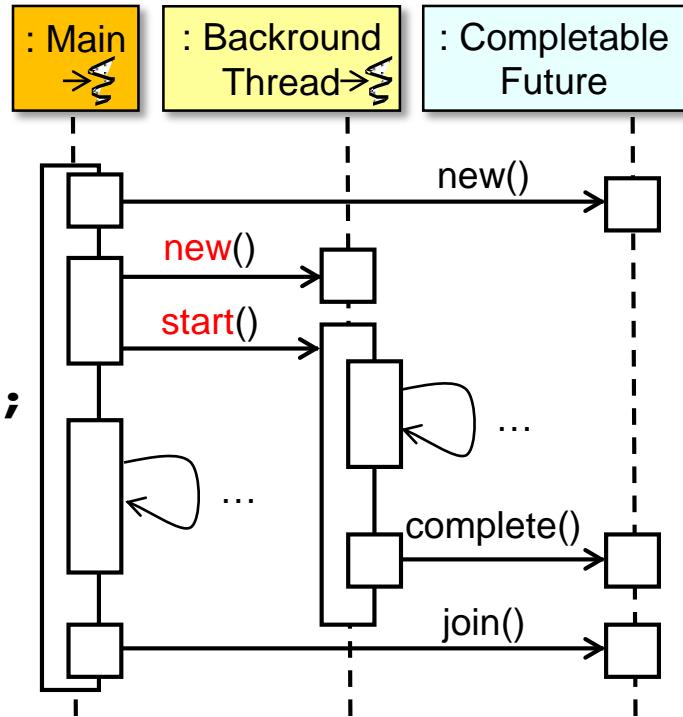
# Applying Basic Completable Future Features

- Multiplying big fractions w/a completable future

```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");  
  
    future.complete(bf1.multiply(bf2));  
}).start();  
...  
System.out.println(future.join().toMixedString());
```

*Start computation in  
a background thread*



# Applying Basic Completable Future Features

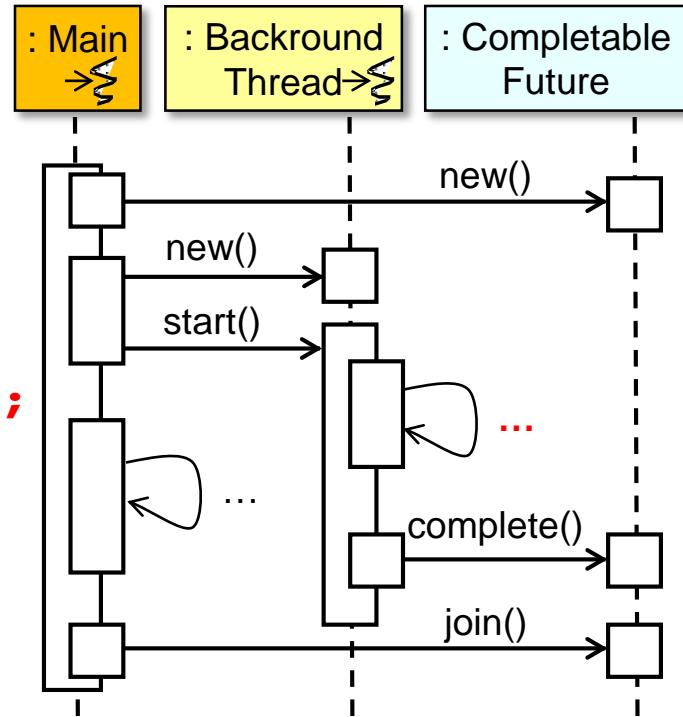
- Multiplying big fractions w/a completable future

```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");  
  
    future.complete(bf1.multiply(bf2));  
}).start();
```

...

```
System.out.println(future.join().toMixedString());
```



*The computation multiplies BigIntegers*

See [docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html](https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html)

# Applying Basic Completable Future Features

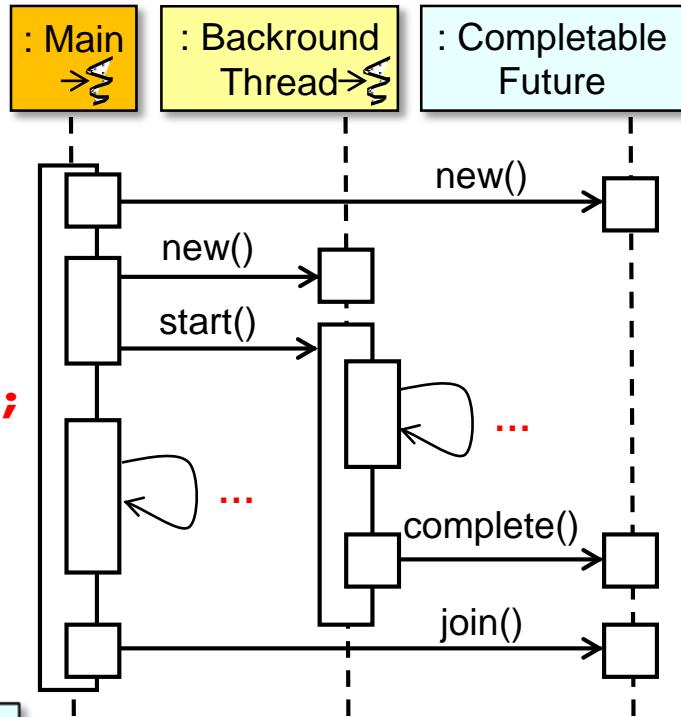
- Multiplying big fractions w/a completable future

```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");  
  
    future.complete(bf1.multiply(bf2));  
}).start();
```

... *These computations run concurrently*

```
System.out.println(future.join().toMixedString());
```



# Applying Basic Completable Future Features

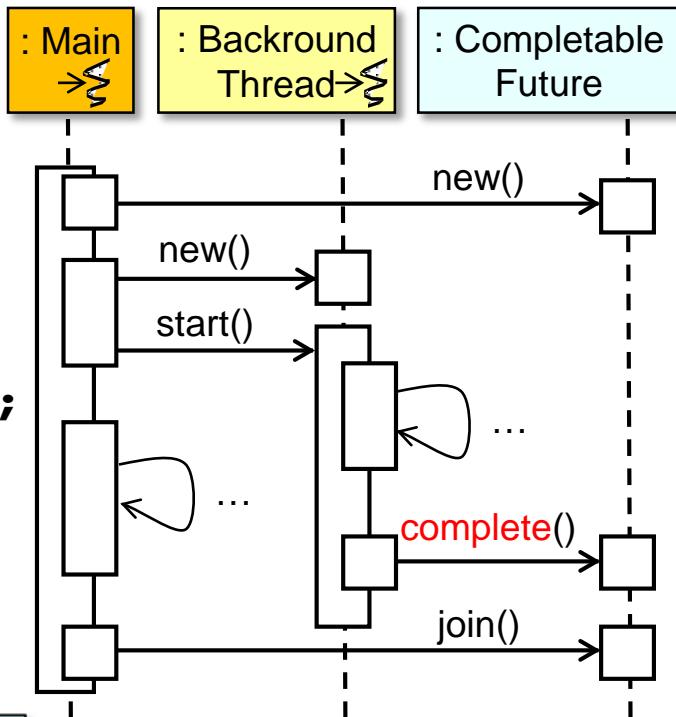
- Multiplying big fractions w/a completable future

```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");  
  
    future.complete(bf1.multiply(bf2));  
}).start();
```

*Explicitly complete the future w/result*

```
...  
System.out.println(future.join().toMixedString());
```



# Applying Basic Completable Future Features

- Multiplying big fractions w/a completable future

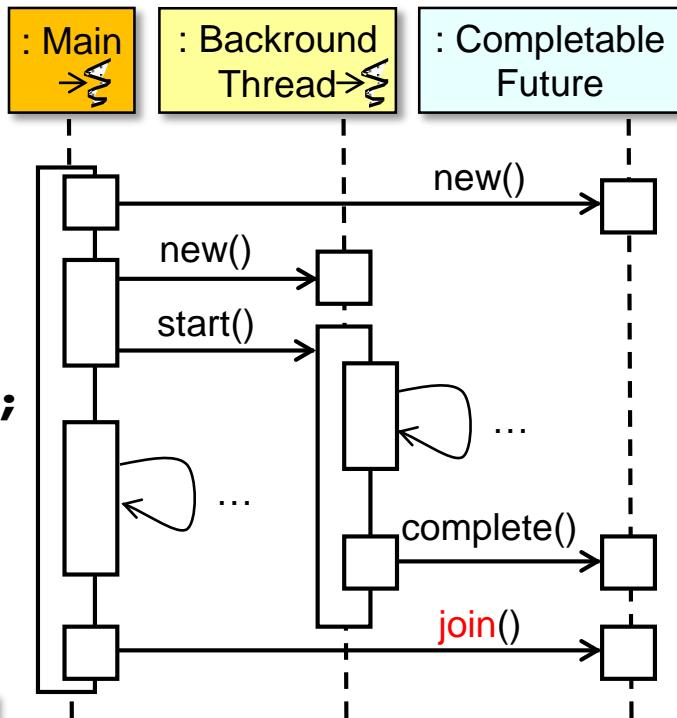
```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");  
  
    future.complete(bf1.multiply(bf2));  
}).start();
```

*join() blocks until result is computed*

...

```
System.out.println(future.join().toMixedString());
```



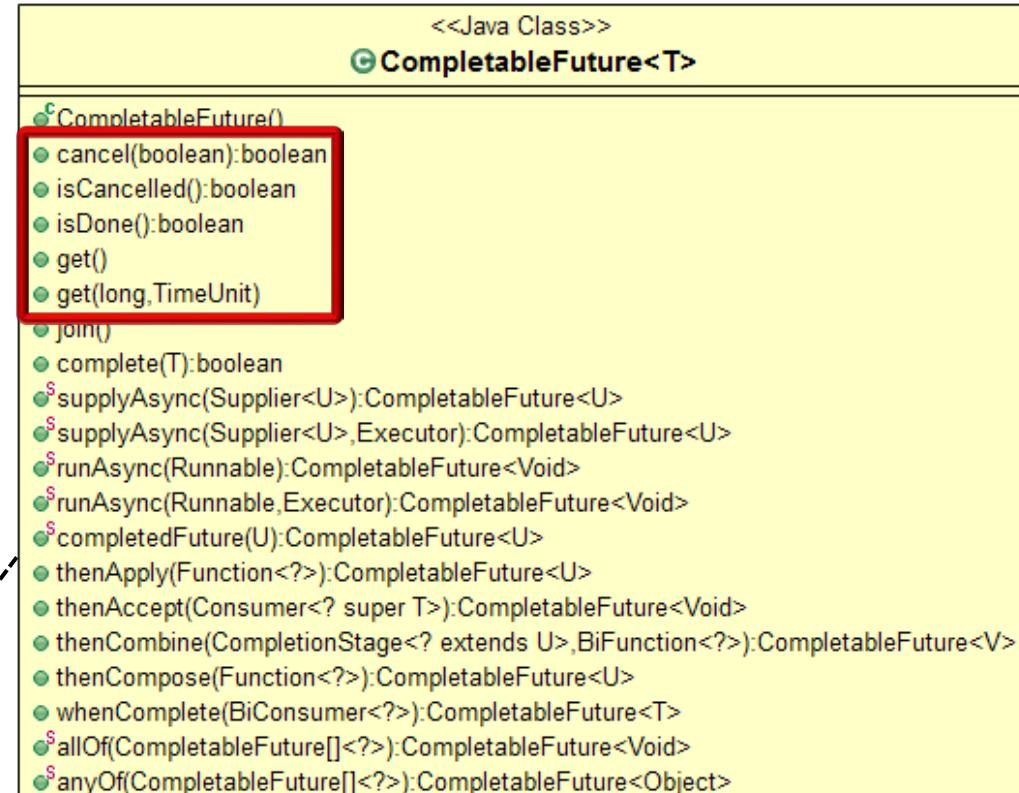
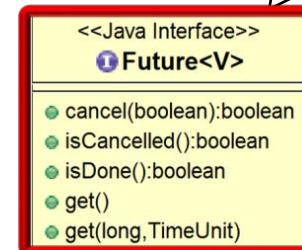
---

# Limitations with Basic Completable Futures Features

# Limitations with Basic Completable Futures Features

- Basic completable future features have similar limitations as futures
  - *Cannot* be chained fluently to handle async results
  - *Cannot* be triggered reactively
  - *Cannot* be treated efficiently as a *collection* of futures

LIMITED



# Limitations with Basic Completable Futures Features

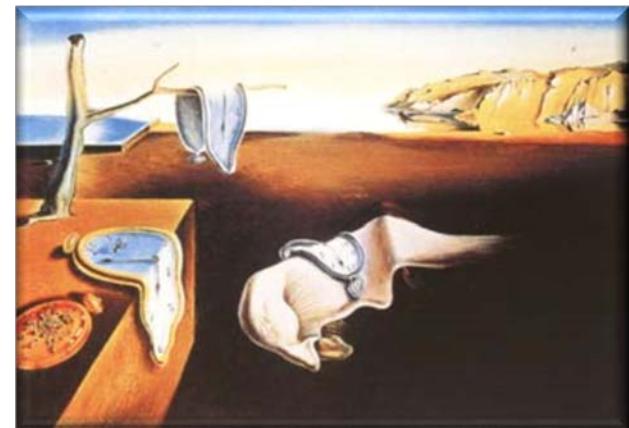
- e.g., `join()` blocks until the future is completed..

```
CompletableFuture<BigFraction> future  
= new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");
```

```
    future.complete(bf1.multiply(bf2));  
}).start();
```

```
...  
System.out.println(future.join().toMixedString());
```



*This blocking call underutilizes cores & increases overhead*

# Limitations with Basic Completable Futures Features

- e.g., `join()` blocks until the future is completed..

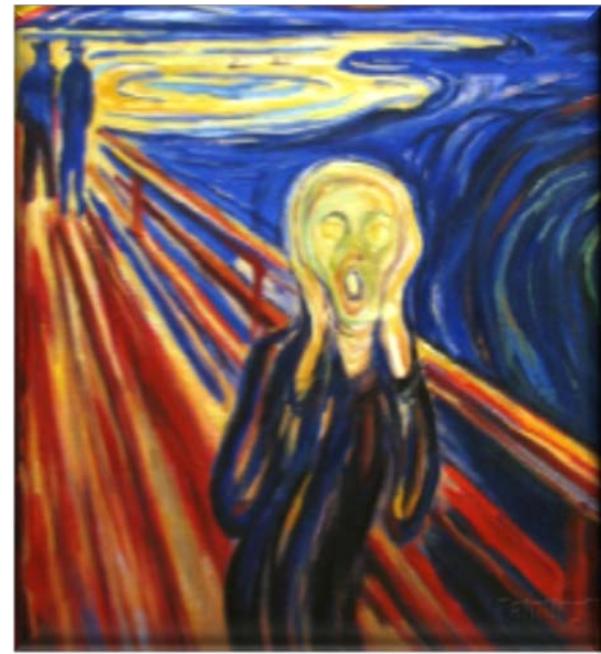
```
CompletableFuture<BigFraction> future  
= new CompletableFuture<>();
```

```
new Thread (() -> {  
    BigFraction bf1 =  
        new BigFraction("62675744/15668936");  
    BigFraction bf2 =  
        new BigFraction("609136/913704");
```

```
    future.complete(bf1.multiply(bf2));  
}).start();
```

...

```
System.out.println(future.join(1, SECONDS).toMixedString());
```



Using a timeout to bound the blocking duration is still inefficient & error-prone

# Limitations with Basic Completable Futures Features

- We therefore need to leverage the advanced features of completable futures



## 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.

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

---

End of Overview of  
Basic Java 8 Completable  
Future Features (Part 2)

# Overview of Advanced Java 8

# CompletableFuture Features (Part 1)

Douglas C. Schmidt

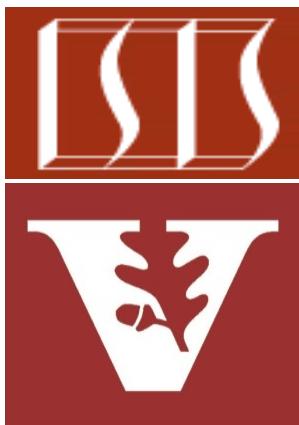
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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 advanced features of completable futures



## 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.

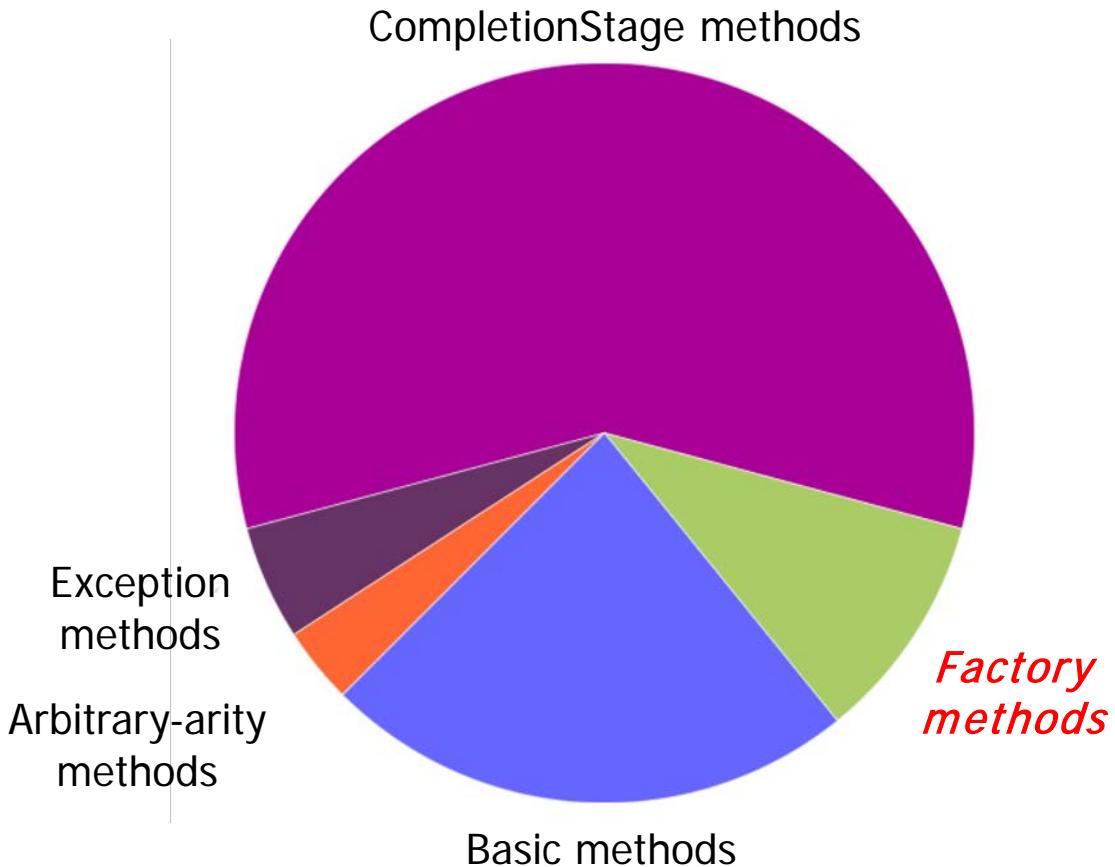
When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface CompletionStage with the following policies:

# Learning Objectives in this Part of the Lesson

---

- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async computations



---

# Factory Methods Initiate Async Computations

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations

«Java Class»

**CompletableFuture<T>**

- CompletableFuture()
- cancel(boolean):boolean
- isCancelled():boolean
- isDone():boolean
- get()
- get(long, TimeUnit)
- join()
- complete(T):boolean
- supplyAsync(Supplier<U>):CompletableFuture<U>**
- supplyAsync(Supplier<U>, Executor):CompletableFuture<U>**
- runAsync(Runnable):CompletableFuture<Void>**
- runAsync(Runnable, Executor):CompletableFuture<Void>**
- completedFuture(U):CompletableFuture<U>
- thenApply(Function<?>):CompletableFuture<U>
- thenAccept(Consumer<? super T>):CompletableFuture<Void>
- thenCombine(CompletionStage<? extends U>, BiFunction<?>):CompletableFuture<V>
- thenCompose(Function<?>):CompletableFuture<U>
- whenComplete(BiConsumer<?>):CompletableFuture<T>
- allOf(CompletableFuture[]<?>):CompletableFuture<Void>**
- anyOf(CompletableFuture[]<?>):CompletableFuture<Object>**

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value



«Java Class»

**CompletableFuture<T>**

CompletableFuture()

cancel(boolean):boolean

isCancelled():boolean

isDone():boolean

get()

get(long,TimeUnit)

join()

complete(T):boolean

supplyAsync(Supplier<U>):CompletableFuture<U>

supplyAsync(Supplier<U>,Executor):CompletableFuture<U>

runAsync(Runnable):CompletableFuture<Void>

runAsync(Runnable,Executor):CompletableFuture<Void>

completedFuture(U):CompletableFuture<U>

thenApply(Function<?>):CompletableFuture<U>

thenAccept(Consumer<? super T>):CompletableFuture<Void>

thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>

thenCompose(Function<?>):CompletableFuture<U>

whenComplete(BiConsumer<?>):CompletableFuture<T>

allOf(CompletableFuture[]<?>):CompletableFuture<Void>

anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
    - `supplyAsync()` allows two-way calls via a supplier

| Methods                       | Params                          | Returns                                                                      | Behavior                                             |
|-------------------------------|---------------------------------|------------------------------------------------------------------------------|------------------------------------------------------|
| <code>supply<br/>Async</code> | <code>Supplier</code>           | <code>CompletableFuture&lt;T&gt;</code> with result of <code>Supplier</code> | Asynchronously run supplier in common fork/join pool |
| <code>supply<br/>Async</code> | <code>Supplier, Executor</code> | <code>CompletableFuture&lt;T&gt;</code> with result of <code>Supplier</code> | Asynchronously run supplier in given executor pool   |

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
  - `supplyAsync()` allows two-way calls via a supplier
  - Can be passed params & returns a value

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

CompletableFuture<BigFraction> future
    = CompletableFuture
        .supplyAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);

            return bf1.multiply(bf2);
        });

```

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
  - `supplyAsync()` allows two-way calls via a supplier
  - Can be passed params & returns a value

*Params are passed as "effectively final" objects to the supplier lambda*

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

CompletableFuture<BigFraction> future
    = CompletableFuture
        .supplyAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);

            return bf1.multiply(bf2);
        });

```

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
    - `supplyAsync()` allows two-way calls via a supplier
    - `runAsync()` enables one-way calls via a Runnable

| Methods                    | Params                          | Returns                                    | Behavior                                             |
|----------------------------|---------------------------------|--------------------------------------------|------------------------------------------------------|
| <code>run<br/>Async</code> | <code>Runnable</code>           | <code>CompletableFuture&lt;Void&gt;</code> | Asynchronously run runnable in common fork/join pool |
| <code>run<br/>Async</code> | <code>Runnable, Executor</code> | <code>CompletableFuture&lt;Void&gt;</code> | Asynchronously run runnable in given executor pool   |

# Factory Methods Initiate Async Computations

---

- Four factory methods initiate async computations
  - These computations may or may not return a value
    - `supplyAsync()` allows two-way calls via a supplier
    - `runAsync()` enables one-way calls via a `Runnable`
    - Can be passed params, but returns no values

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

CompletableFuture<Void> future
= CompletableFuture
    .runAsync(() -> {
    BigFraction bf1 =
        new BigFraction(f1);
    BigFraction bf2 =
        new BigFraction(f2);

    System.out.println
        (bf1.multiply(bf2)
         .toMixedString());
});
```

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
    - `supplyAsync()` allows two-way calls via a supplier
    - `runAsync()` enables one-way calls via a `Runnable`
    - Can be passed params, but returns no values

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

CompletableFuture<Void> future
    = CompletableFuture
        .runAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);

            System.out.println
                (bf1.multiply(bf2)
                    .toMixedString());
        });

```

*"Void" is not a value!*

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
  - Async functionality runs in a thread pool



| <<Java Class>>                                                               |  |
|------------------------------------------------------------------------------|--|
| CompletableFuture<T>                                                         |  |
| CompletableFuture()                                                          |  |
| cancel(boolean):boolean                                                      |  |
| isCancelled():boolean                                                        |  |
| isDone():boolean                                                             |  |
| get()                                                                        |  |
| get(long,TimeUnit)                                                           |  |
| join()                                                                       |  |
| complete(T):boolean                                                          |  |
| supplyAsync(Supplier<U>):CompletableFuture<U>                                |  |
| supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                       |  |
| runAsync(Runnable):CompletableFuture<Void>                                   |  |
| runAsync(Runnable,Executor):CompletableFuture<Void>                          |  |
| completedFuture(U):CompletableFuture<U>                                      |  |
| thenApply(Function<?>):CompletableFuture<U>                                  |  |
| thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |  |
| thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |  |
| thenCompose(Function<?>):CompletableFuture<U>                                |  |
| whenComplete(BiConsumer<?>):CompletableFuture<T>                             |  |
| allOf(CompletableFuture[]<?>):CompletableFuture<Void>                        |  |
| anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                      |  |

# Factory Methods Initiate Async Computations

- Four factory methods initiate async computations
  - These computations may or may not return a value
  - Async functionality runs in a thread pool



| <<Java Class>>                                                               |  |
|------------------------------------------------------------------------------|--|
| CompletableFuture<T>                                                         |  |
| CompletableFuture()                                                          |  |
| cancel(boolean):boolean                                                      |  |
| isCancelled():boolean                                                        |  |
| isDone():boolean                                                             |  |
| get()                                                                        |  |
| get(long,TimeUnit)                                                           |  |
| join()                                                                       |  |
| complete(T):boolean                                                          |  |
| supplyAsync(Supplier<U>):CompletableFuture<U>                                |  |
| supplyAsync(Supplier<U>,Executor):CompletableFuture<U>                       |  |
| runAsync(Runnable):CompletableFuture<void>                                   |  |
| runAsync(Runnable,Executor):CompletableFuture<Void>                          |  |
| completedFuture(U):CompletableFuture<U>                                      |  |
| thenApply(Function<?>):CompletableFuture<U>                                  |  |
| thenAccept(Consumer<? super T>):CompletableFuture<Void>                      |  |
| thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V> |  |
| thenCompose(Function<?>):CompletableFuture<U>                                |  |
| whenComplete(BiConsumer<?>):CompletableFuture<T>                             |  |
| allOf(CompletableFuture[]<?>):CompletableFuture<Void>                        |  |
| anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                      |  |

This thread pool defaults to common fork-join pool, but can be given explicitly

---

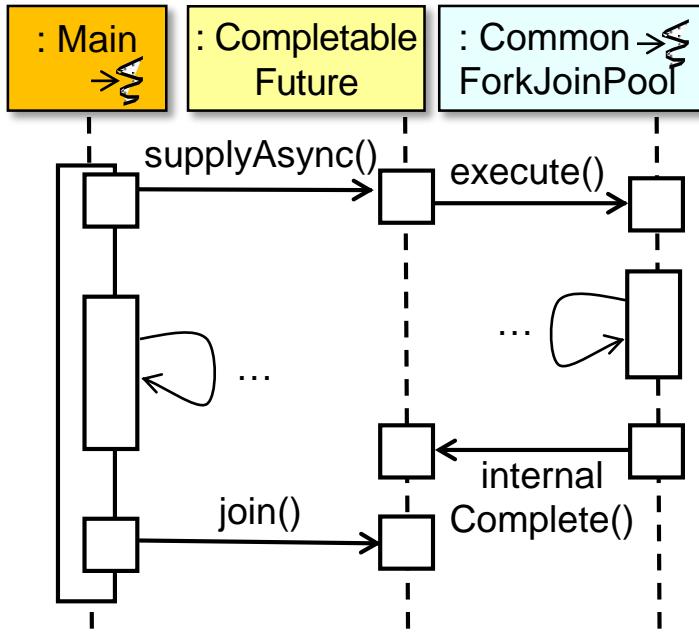
# Applying Completable Future Factory Methods

# Applying CompletableFuture Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
    CompletableFuture
        .supplyAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);

            return bf1.multiply(bf2);
        });
...
System.out.println(future.join().toMixedString());
```



# Applying CompletableFuture Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
```

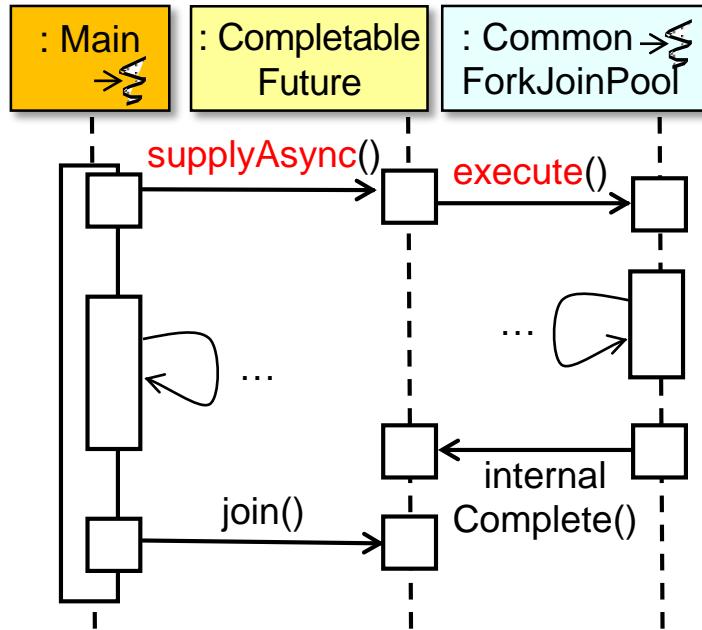
```
String f2 = "609136/913704";
```

```
CompletableFuture<BigFraction> future =
```

```
    CompletableFuture
```

```
        .supplyAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);
```

```
            return bf1.multiply(bf2);
        });
    ...
System.out.println(future.join().toMixedString());
```



*Schedule computation for execution on the fork-join pool*

# Applying CompletableFuture Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
```

```
String f2 = "609136/913704";
```

```
CompletableFuture<BigFraction> future =
```

```
    CompletableFuture
```

```
        .supplyAsync(() -> {
```

```
            BigFraction bf1 =
```

```
                new BigFraction(f1);
```

```
            BigFraction bf2 =
```

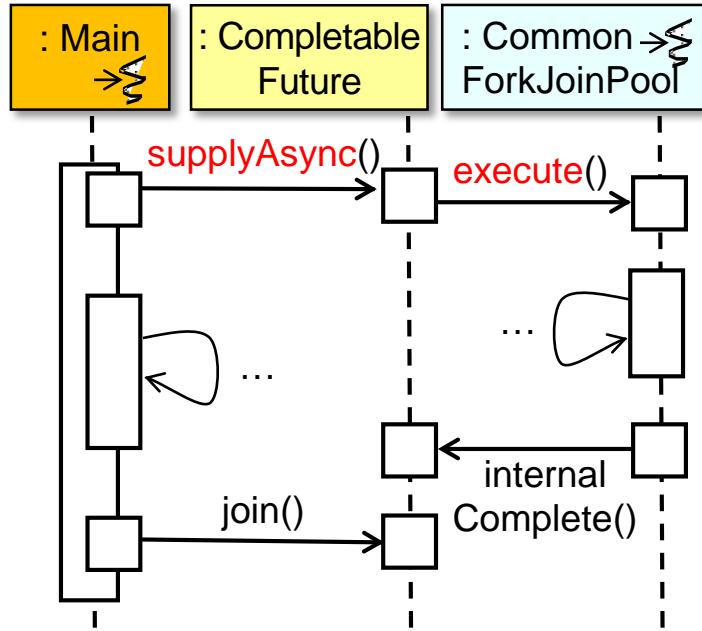
```
                new BigFraction(f2);
```

```
            return bf1.multiply(bf2);
```

```
});
```

```
...
```

```
System.out.println(future.join().toMixedString());
```



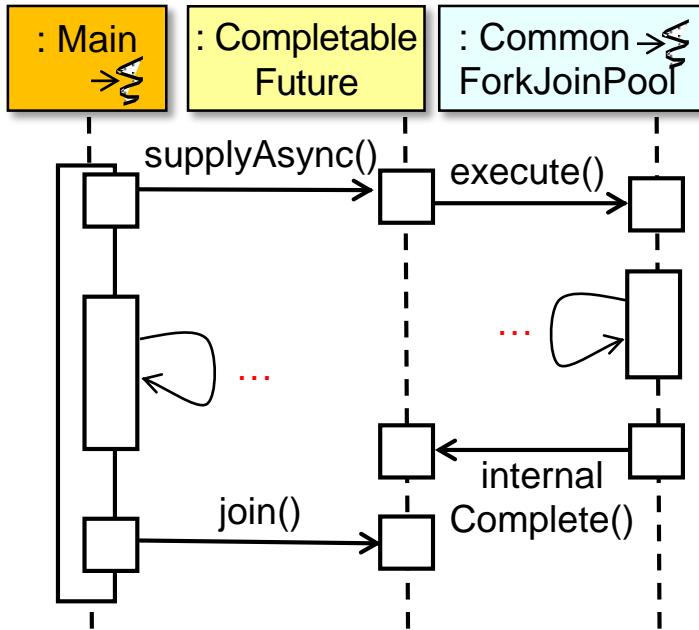
*Define a supplier lambda that multiplies two BigFractions*

# Applying CompletableFuture Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
    CompletableFuture
        .supplyAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);

            return bf1.multiply(bf2);
        });
...
System.out.println(future.join().toMixedString());
```



*These computations run concurrently*

# Applying CompletableFuture Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
```

```
String f2 = "609136/913704";
```

```
CompletableFuture<BigFraction> future =
```

```
    CompletableFuture
```

```
        .supplyAsync(() -> {
```

```
            BigFraction bf1 =
```

```
                new BigFraction(f1);
```

```
            BigFraction bf2 =
```

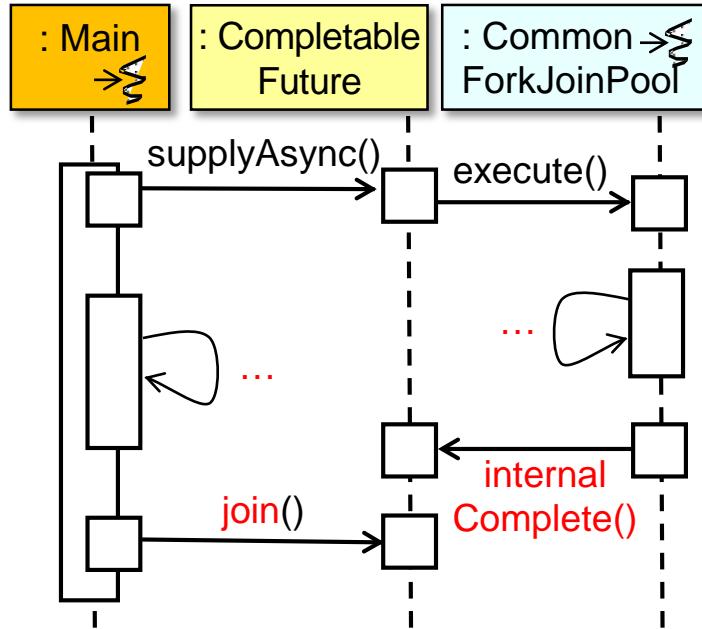
```
                new BigFraction(f2);
```

```
            return bf1.multiply(bf2);
```

```
});
```

```
...
```

```
System.out.println(future.join().toMixedString());
```



# Applying Completable Future Factory Methods

- Using supplyAsync() to multiply big fractions

```
String f1 = "62675744/15668936";
String f2 = "609136/913704";
CompletableFuture<BigFraction> future =
    CompletableFuture
        .supplyAsync(() -> {
            BigFraction bf1 =
                new BigFraction(f1);
            BigFraction bf2 =
                new BigFraction(f2);

            return bf1.multiply(bf2);
        });
...
System.out.println(future.join().toMixedString());
```



Java threads are *not* explicitly used in this example!

---

End of Overview of Advanced  
Java 8 CompletableFuture  
Features (Part 1)

# Overview of Advanced Java 8 CompletableFuture Features (Part 2)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

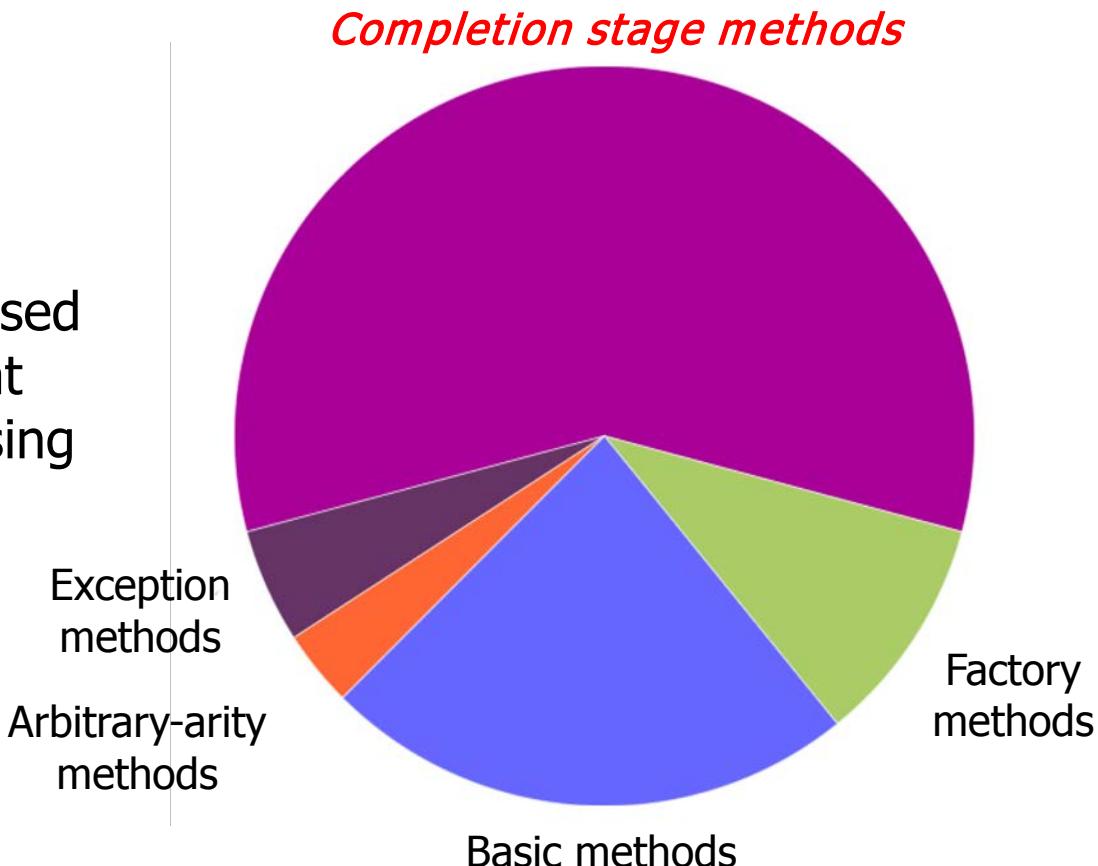
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
  - Completion stage methods used to chain together actions that perform async result processing & composition



---

# Completion Stage Methods

# Chain Actions Together

# Completion Stage Methods Chain Actions Together

- A completable future can serve as a “completion stage” for async result processing

## Interface CompletionStage<T>

All Known Implementing Classes:

CompletableFuture

```
public interface CompletionStage<T>
```

A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes. A stage completes upon termination of its computation, but this may in turn trigger other dependent stages. The functionality defined in this interface takes only a few basic forms, which expand out to a larger set of methods to capture a range of usage styles:

- The computation performed by a stage may be expressed as a Function, Consumer, or Runnable (using methods with names including *apply*, *accept*, or *run*, respectively) depending on whether it requires arguments and/or produces results. For example, `stage.thenApply(x -> square(x)).thenAccept(x -> System.out.print(x)).thenRun(() -> System.out.println())`. An additional form (*compose*) applies functions of stages themselves, rather than their results.
- One stage's execution may be triggered by completion of a single stage, or both of two stages, or either of two stages. Dependencies on a single stage are arranged using methods with prefix *then*. Those triggered by completion of *both* of two stages may *combine* their results or effects, using correspondingly named methods. Those triggered by *either* of two stages make no guarantees about which of the results or effects are used for the dependent stage's computation.

# Completion Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for async result processing
  - An action is performed on a completed async call result

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger  
        ("846122553600669882"),  
        new BigInteger  
        ("188027234133482196"),  
        false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = () ->  
    BigFraction.reduce(unreduced);
```

CompletableFuture

```
.supplyAsync(reduce)  
.thenApply(BigFraction  
    ::toMixedString)
```

...

*thenApply()'s action is triggered when future from supplyAsync() completes*

# Completion Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for async result processing
  - An action is performed on a completed async call result
  - Methods can be chained together "fluently"

*thenAccept()'s action is triggered when future from thenApply() completes*

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger  
        ("846122553600669882"),  
    new BigInteger  
        ("188027234133482196"),  
    false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = () ->  
    BigFraction.reduce(unreduced);
```

CompletableFuture

```
.supplyAsync(reduce)  
.thenApply(BigFraction  
    ::toMixedString)  
.thenAccept(System.out::println);
```

# Completion Stage Methods Chain Actions Together

---

- A completable future can serve as a “completion stage” for async result processing
  - An action is performed on a completed async call result
  - Methods can be chained together “fluently”
    - Each method registers a lambda action to apply

```
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger
        ("846122553600669882"),
    new BigInteger
        ("188027234133482196"),
    false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = () ->
    BigFraction.reduce(unreduced);
```

```
CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction
        ::toMixedString)
    .thenAccept(System.out::println);
```

# Completion Stage Methods Chain Actions Together

---

- A completable future can serve as a “completion stage” for async result processing
  - An action is performed on a completed async call result
  - Methods can be chained together “fluently”
    - Each method registers a lambda action to apply

```
BigFraction unreduced = BigFraction
    .valueOf(new BigInteger
        ("846122553600669882"),
    new BigInteger
        ("188027234133482196"),
    false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = () ->
    BigFraction.reduce(unreduced);
```

```
CompletableFuture
    .supplyAsync(reduce)
    .thenApply(BigFraction
        ::toMixedString)
    .thenAccept(System.out::println);
```

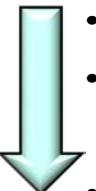
# Completion Stage Methods Chain Actions Together

- A completable future can serve as a “completion stage” for async result processing
  - An action is performed on a completed async call result
  - Methods can be chained together “fluently”
    - Each method registers a lambda action to apply
    - A lambda action is called only after the previous stage completes

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger  
        ("846122553600669882"),  
    new BigInteger  
        ("188027234133482196"),  
    false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = () ->  
    BigFraction.reduce(unreduced);
```

CompletableFuture



```
.supplyAsync(reduce)  
.thenApply(BigFraction  
    ::toMixedString)  
.thenAccept(System.out::println);
```

# Completion Stage Methods Chain Actions Together

- A completable future can serve as a “completion stage” for async result processing
  - An action is performed on a completed async call result
  - Methods can be chained together “fluently”
    - Each method registers a lambda action to apply
    - A lambda action is called only after the previous stage completes

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger  
        ("846122553600669882"),  
    new BigInteger  
        ("188027234133482196"),  
    false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = () ->  
    BigFraction.reduce(unreduced);
```

CompletableFuture

```
.supplyAsync(reduce)  
.thenApply(BigFraction  
    ::toMixedString)  
.thenAccept(System.out::println);
```

**DEFERRED**

Action is thus “deferred” until prev stage completes & fork-join thread is available

# Completion Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for async result processing
  - An action is performed on a completed async call result
  - Methods can be chained together "fluently"



```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger  
        ("846122553600669882"),  
    new BigInteger  
        ("188027234133482196"),  
    false); // Don't reduce!
```

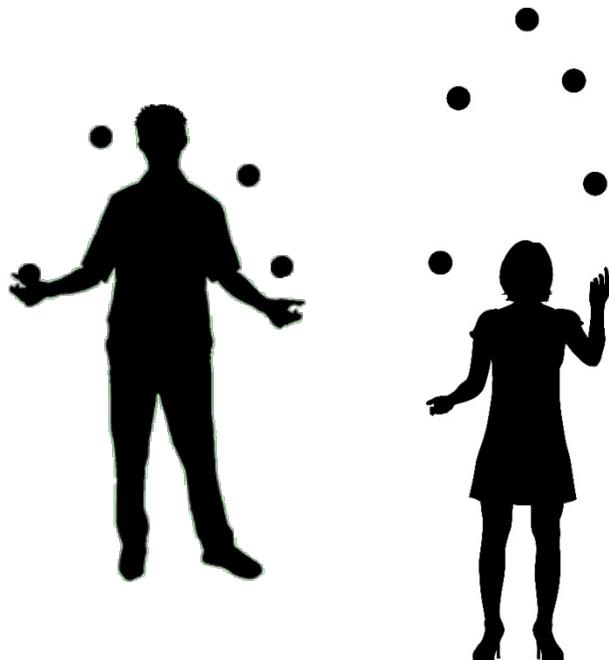
```
Supplier<BigFraction> reduce = () ->  
    BigFraction.reduce(unreduced);
```

```
CompletableFuture  
    .supplyAsync(reduce)  
    .thenApply(BigFraction  
        ::toMixedString)  
    .thenAccept(System.out::println);
```

Completion stages avoid blocking a thread until the result *must* be obtained

# Completion Stage Methods Chain Actions Together

- A completable future can serve as a "completion stage" for async result processing



| <<Java Class>>                                                                |  |
|-------------------------------------------------------------------------------|--|
| CompletableFuture<T>                                                          |  |
| CompletableFuture()                                                           |  |
| cancel(boolean):boolean                                                       |  |
| isCancelled():boolean                                                         |  |
| isDone():boolean                                                              |  |
| get()                                                                         |  |
| get(long, TimeUnit)                                                           |  |
| join()                                                                        |  |
| complete(T):boolean                                                           |  |
| supplyAsync(Supplier<U>):CompletableFuture<U>                                 |  |
| supplyAsync(Supplier<U>, Executor):CompletableFuture<U>                       |  |
| runAsync(Runnable):CompletableFuture<Void>                                    |  |
| runAsync(Runnable, Executor):CompletableFuture<Void>                          |  |
| completedFuture(U):CompletableFuture<U>                                       |  |
| thenApply(Function<?>):CompletableFuture<U>                                   |  |
| thenAccept(Consumer<? super T>):CompletableFuture<Void>                       |  |
| thenCombine(CompletionStage<? extends U>, BiFunction<?>):CompletableFuture<V> |  |
| thenCompose(Function<?>):CompletableFuture<U>                                 |  |
| whenComplete(BiConsumer<?>):CompletableFuture<T>                              |  |
| allOf(CompletableFuture[]<?>):CompletableFuture<Void>                         |  |
| anyOf(CompletableFuture[]<?>):CompletableFuture<Object>                       |  |

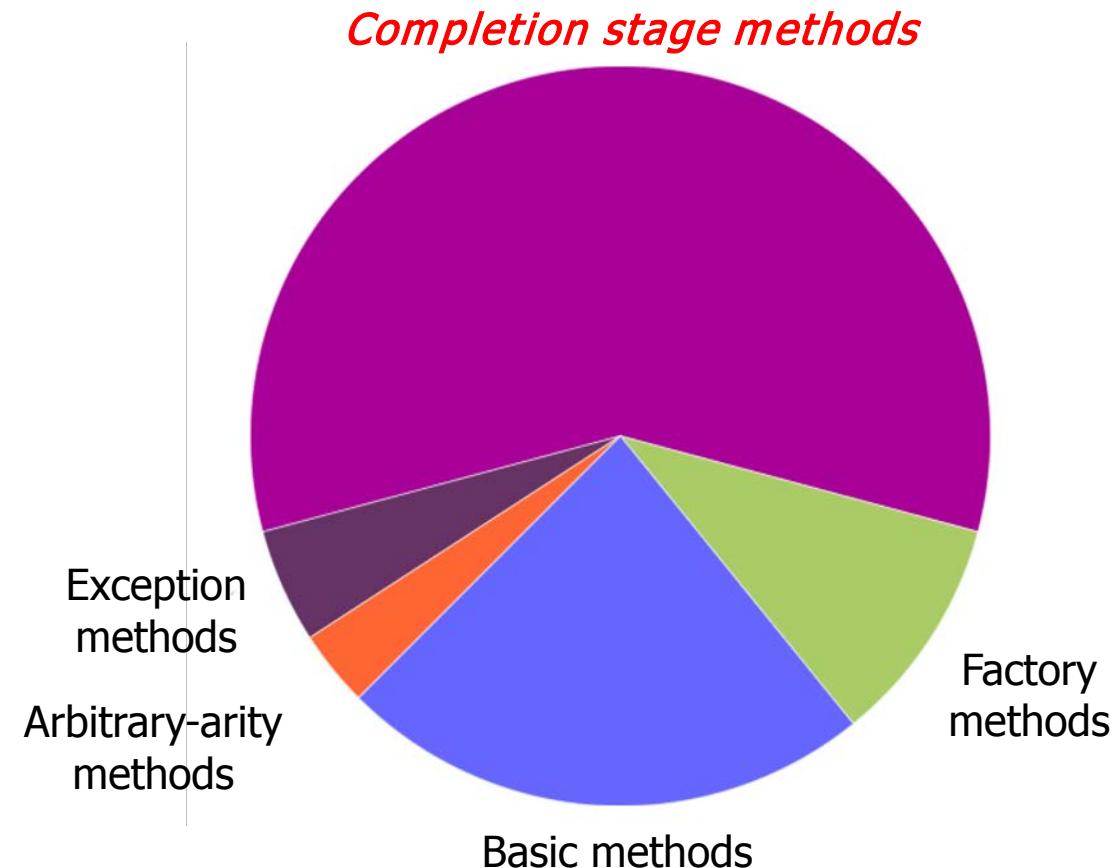
Juggling is a good analogy for completion stages!

---

# Grouping CompletableFuture Completion Stage Methods

# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage



# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage
  - Completion of a single previous stage

*These methods run in the invoking thread or the same thread as previous stage*

| Methods                              | Params                | Returns                                                                                       | Behavior                                       |
|--------------------------------------|-----------------------|-----------------------------------------------------------------------------------------------|------------------------------------------------|
| <code>thenApply<br/>(Async)</code>   | <code>Function</code> | <code>CompletableFuture&lt;Function result&gt;</code>                                         | Apply function to result of the previous stage |
| <code>thenCompose<br/>(Async)</code> | <code>Function</code> | <code>CompletableFuture&lt;Function result&gt;</code><br>directly, <i>not</i> a nested future | Apply function to result of the previous stage |
| <code>thenAccept<br/>(Async)</code>  | <code>Consumer</code> | <code>CompletableFuture&lt;Void&gt;</code>                                                    | Consumer handles result of previous stage      |
| <code>thenRun<br/>(Async)</code>     | <code>Runnable</code> | <code>CompletableFuture&lt;Void&gt;</code>                                                    | Run action w/out returning value               |

The thread that executes these methods depends on various runtime factors

# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage
  - Completion of a single previous stage

*\*Async() variants run in common fork-join pool*

| Methods                              | Params                | Returns                                        | Behavior                                                                            |
|--------------------------------------|-----------------------|------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>thenApply<br/>(Async)</code>   | <code>Function</code> | <code>CompletableFuture&lt;Function&gt;</code> | Apply function to result of the previous stage                                      |
| <code>thenCompose<br/>(Async)</code> | <code>Function</code> | <code>CompletableFuture&lt;Function&gt;</code> | Apply function to result of the previous stage directly, <i>not</i> a nested future |
| <code>thenAccept<br/>(Async)</code>  | <code>Consumer</code> | <code>CompletableFuture&lt;Void&gt;</code>     | Consumer handles result of previous stage                                           |
| <code>thenRun<br/>(Async)</code>     | <code>Runnable</code> | <code>CompletableFuture&lt;Void&gt;</code>     | Run action w/out returning value                                                    |

# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage
  - Completion of a single previous stage
  - Completion of both of 2 previous stages
    - i.e., an “and”

| Methods                           | Params         | Returns                                                | Behavior                                            |
|-----------------------------------|----------------|--------------------------------------------------------|-----------------------------------------------------|
| then<br>Combine<br>(Async)        | Bi<br>Function | CompletableFuture<BiFunction<CompletableFuture<T>, T>> | Apply bifunction to results of both previous stages |
| then<br>Accept<br>Both<br>(Async) | Bi<br>Consumer | CompletableFuture<Void>                                | BiConsumer handles results of both previous stages  |
| runAfter<br>Both<br>(Async)       | Runnable       | CompletableFuture<Void>                                | Run action when both previous stages complete       |

# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by a previous stage
  - Completion of a single previous stage
  - Completion of both of 2 previous stages
  - Completion of either of 2 previous stages
    - i.e., an “or”

| Methods                       | Params   | Returns                            | Behavior                                           |
|-------------------------------|----------|------------------------------------|----------------------------------------------------|
| applyTo<br>Either<br>(Async)  | Function | CompletableFuture<Function result> | Apply function to results of either previous stage |
| accept<br>Either<br>(Async)   | Consumer | CompletableFuture<Void>            | Consumer handles results of either previous stage  |
| runAfter<br>Either<br>(Async) | Runnable | CompletableFuture<Void>            | Run action when either previous stage completes    |

---

# Key CompletableFuture Completion Stage Methods

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage
    - `thenApply()`
- `CompletableFuture<U> thenApply  
(Function<? super T,  
? extends U> fn)  
{ ... }`

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage
    - `thenApply()`
      - Applies a function action to the previous stage's result
- `CompletableFuture<U> thenApply  
(Function<? super T,  
? extends U> fn)  
{ ... }`

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage

- thenApply()

- Applies a function action to the previous stage's result
- Returns a future containing the result of the action

```
CompletableFuture<U> thenApply  
(Function<? super T,  
? extends U> fn)  
{ ... }
```

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - `thenApply()`
    - Applies a function action to the previous stage's result
    - Returns a future containing the result of the action
    - Used for a *sync* action that returns a value, not a future

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger("..."),  
             new BigInteger("..."),  
             false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = ()  
    -> BigFraction.reduce(unreduced);
```

`CompletableFuture`

```
.supplyAsync(reduce)  
.thenApply(BigFraction  
          ::toMixedString)
```

...

e.g., `toMixedString()`  
returns a string value

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage
  - `thenApply()`
  - `thenCompose()`

```
CompletableFuture<U> thenCompose  
    (Function<? super T,  
     ? extends  
     CompletionStage<U>> fn)  
  
{ ... }
```

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage
    - `thenApply()`
    - `thenCompose()`
      - Applies a function action to the previous stage's result
- `CompletableFuture<U> thenCompose  
(Function<? super T,  
? extends CompletionStage<U>> fn)  
{ ... }`

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage

- thenApply()
- thenCompose()

- Applies a function action to the previous stage's result
- Returns a future containing result of the action directly
  - *i.e., not* a nested future

`CompletableFuture<U> thenCompose`

`(Function<? super T,`  
`? extends`

`CompletionStage<U>> fn)`

`{ ... }`

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage

- `thenApply()`

- `thenCompose()`

- Applies a function action to the previous stage's result

- Returns a future containing result of the action directly

- *i.e., not* a nested future

`CompletableFuture<U> thenCompose`

`(Function<? super T,`

`? extends`

`CompletionStage<U>> fn)`

{ ... }

$$8 + 1 = 26$$

$$8 + 1 \neq 10$$

`thenCompose()` is similar to `flatMap()` on a Stream or Optional

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - `thenApply()`
  - `thenCompose()`
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for an *async* action that returns a completable future

```
Function<BF,<br/>CompletableFuture<BF>><br/>reduceAndMultiplyFractions =<br/>unreduced -> CompletableFuture<br/>.supplyAsync<br/>((() -> BF.reduce(unreduced))<br/><br/>• thenCompose<br/>(reduced -> CompletableFuture<br/>.supplyAsync(() -><br/>reduced.multiply(...)));<br/>...<br/>
```

e.g., `supplyAsync()` returns a completable future

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage

- thenApply()
- thenCompose()
  - Applies a function action to the previous stage's result
  - Returns a future containing result of the action directly
  - Used for an *async* action that returns a completable future
  - Avoids unwieldy nesting of futures à la thenApply()

Unwieldy!

```
Function<BF, CompletableFuture<  
CompletableFuture<BF>>>  
reduceAndMultiplyFractions =  
unreduced -> CompletableFuture  
.supplyAsync  
(( ) -> BF.reduce(unreduced))  
  
.thenApply  
(reduced -> CompletableFuture  
.supplyAsync(( ) ->  
reduced.multiply(...)));  
...  
...
```

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for an *async* action that returns a completable future
    - Avoids unwieldy nesting of futures à la thenApply()

```
CompletableFuture<Integer> countF =  
    .CompletableFuture  
    .supplyAsync  
    (( ) ->  
        longRunnerReturnsCF( ))  
  
    .thenCompose  
    (Function.identity( ))  
    ...
```

*supplyAsync() will return a  
CompletableFuture to a  
CompletableFuture here!!*

Can be used to avoid calling join() when flattening nested completable futures

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
    - Applies a function action to the previous stage's result
    - Returns a future containing result of the action directly
    - Used for an *async* action that returns a completable future
    - Avoids unwieldy nesting of futures à la thenApply()

```
CompletableFuture<Integer> countF =  
    .CompletableFuture  
    .supplyAsync  
    (( ) ->  
        longRunnerReturnsCF( ))  
  
    .thenCompose  
    (Function.identity())  
    ...
```

*This idiom flattens the return value to "just" one CompletableFuture!*

Can be used to avoid calling join() when flattening nested completable futures

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage
  - `thenApply()`
  - `thenCompose()`
  - `thenAccept()`

```
CompletableFuture<Void>
    thenAccept
        (Consumer<? super T> action)
    { ... }
```

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result

```
CompletableFuture<Void>
    thenAccept
        (Consumer<? super T> action)
{ ... }
```

*This action behaves as a "callback" with a side-effect*

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void

```
CompletableFuture<Void>
    thenAccept
        (Consumer<? super T> action)
    { ... }
```

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage

- `thenApply()`
- `thenCompose()`
- `thenAccept()`

- Applies a consumer action to handle previous stage's result
- Returns a future to Void
- Often used at the end of a chain of completion stages

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger("..."),  
            new BigInteger("..."),  
            false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = ()  
    -> BigFraction.reduce(unreduced);
```

```
CompletableFuture
```

```
    .supplyAsync(reduce)  
    .thenApply(BigFraction  
              ::toMixedString)  
    .thenAccept(System.out::println);
```

*thenApply() returns a string future that thenAccept() prints when it completes*

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage

- thenApply()
- thenCompose()
- thenAccept()
  - Applies a consumer action to handle previous stage's result
  - Returns a future to Void
  - Often used at the end of a chain of completion stages

```
BigFraction unreduced = BigFraction  
    .valueOf(new BigInteger("..."),  
             new BigInteger("..."),  
             false); // Don't reduce!
```

```
Supplier<BigFraction> reduce = ()  
    -> BigFraction.reduce(unreduced);
```

CompletableFuture

```
.supplyAsync(reduce)  
.thenApply(BigFraction  
          ::toMixedString)  
.thenAccept(System.out::println);
```

*println() is a callback that has a side-effect (i.e., printing the mixed string)*

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of a single previous stage
  - thenApply()
  - thenCompose()
  - thenAccept()
    - Applies a consumer action to handle previous stage's result
    - Returns a future to Void
    - Often used at the end of a chain of completion stages
    - May lead to "callback hell!"

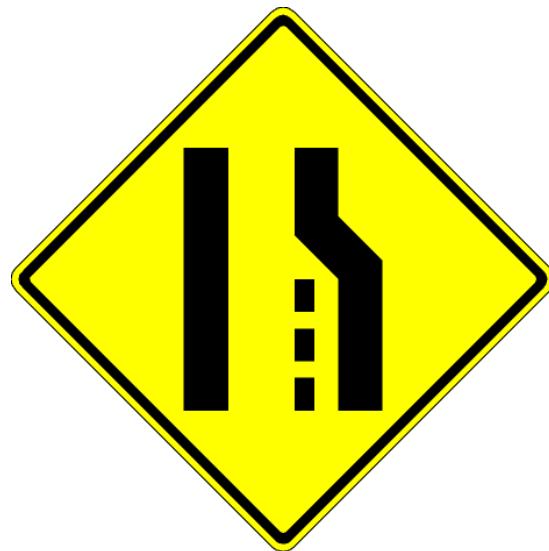


icompile.eladkarako.com

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`

```
CompletableFuture<U> thenCombine  
(CompletionStage<? Extends U>  
other,  
BiFunction<? super T,  
? super U,  
? extends V> fn)  
{ ... }
```



# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`
  - Applies a bifunction action to two previous stages' results

```
CompletableFuture<U> thenCombine  
(CompletionStage<? Extends U>  
other,  
BiFunction<? super T,  
? super U,  
? extends V> fn)  
{ ... }
```

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`
  - Applies a bifunction action to two previous stages' results
  - Returns a future containing the result of the action

```
CompletableFuture<U> thenCombine  
(CompletionStage<? Extends U>  
other,  
BiFunction<? super T,  
? super U,  
? extends V> fn)  
{ ... }
```

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`
  - Applies a bifunction action to two previous stages' results
  - Returns a future containing the result of the action

```
CompletableFuture<U> thenCombine  
(CompletionStage<? Extends U>  
other,  
BiFunction<? super T,  
? super U,  
? extends V> fn)  
{ ... }
```



thenCombine() essentially performs a “reduction”

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of both of two previous stages
  - `thenCombine()`
  - Applies a bifunction action to two previous stages' results
  - Returns a future containing the result of the action
  - Used to “join” two paths of execution

*thenCombine()'s action is triggered when its two associated futures complete*

```
CompletableFuture<BF> compF1 =  
    CompletableFuture  
.supplyAsync(() ->  
/* multiply two BFs. */);
```

```
CompletableFuture<BF> compF2 =  
    CompletableFuture  
.supplyAsync(() ->  
/* divide two BFs. */);
```

```
compF1  
.thenCombine(compF2,  
BigFraction::add)  
.thenAccept(System.out::println);
```

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of either of two previous stages
    - `acceptEither()`
- ```
CompletableFuture<Void> acceptEither  
    (CompletionStage<? Extends T>  
     other,  
     Consumer<? super T> action)  
{ ... }
```



# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of either of two previous stages
    - `acceptEither()`
      - Applies a consumer action that handles either of the previous stages' results
- `CompletableFuture<Void> acceptEither(CompletionStage<? Extends T> other,  
Consumer<? super T> action)`
- `{ ... }`

# Key CompletableFuture Completion Stage Methods

---

- Methods triggered by completion of either of two previous stages
    - `acceptEither()`
      - Applies a consumer action that handles either of the previous stages' results
      - Returns a future to Void
- `CompletableFuture<Void> acceptEither(CompletionStage<? Extends T> other,  
Consumer<? super T> action)  
{ ... }`

# Key CompletableFuture Completion Stage Methods

- Methods triggered by completion of either of two previous stages
  - acceptEither()

- Applies a consumer action that handles either of the previous stages' results
- Returns a future to Void
- Often used at the end of a chain of completion stages

*Printout sorted results from which ever sorting routine finished first*

```
CompletableFuture<List<BigFraction>>
    quickSort = CompletableFuture
        .supplyAsync(() ->
            quickSort(list));
```

```
CompletableFuture<List<BigFraction>>
    mergeSort = CompletableFuture
        .supplyAsync(() ->
            mergeSort(list));
```

```
quickSort.acceptEither
(mergeSort, results -> results
.forEach(fraction ->
    System.out.println
(fraction
.toMixedString())));
```

acceptEither() does *not* cancel the second future after the first one completes

---

# End of Overview of Advanced Java 8 CompletableFuture Features (Part 2)

# Overview of Advanced Java 8 CompletableFuture Features (Part 3)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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

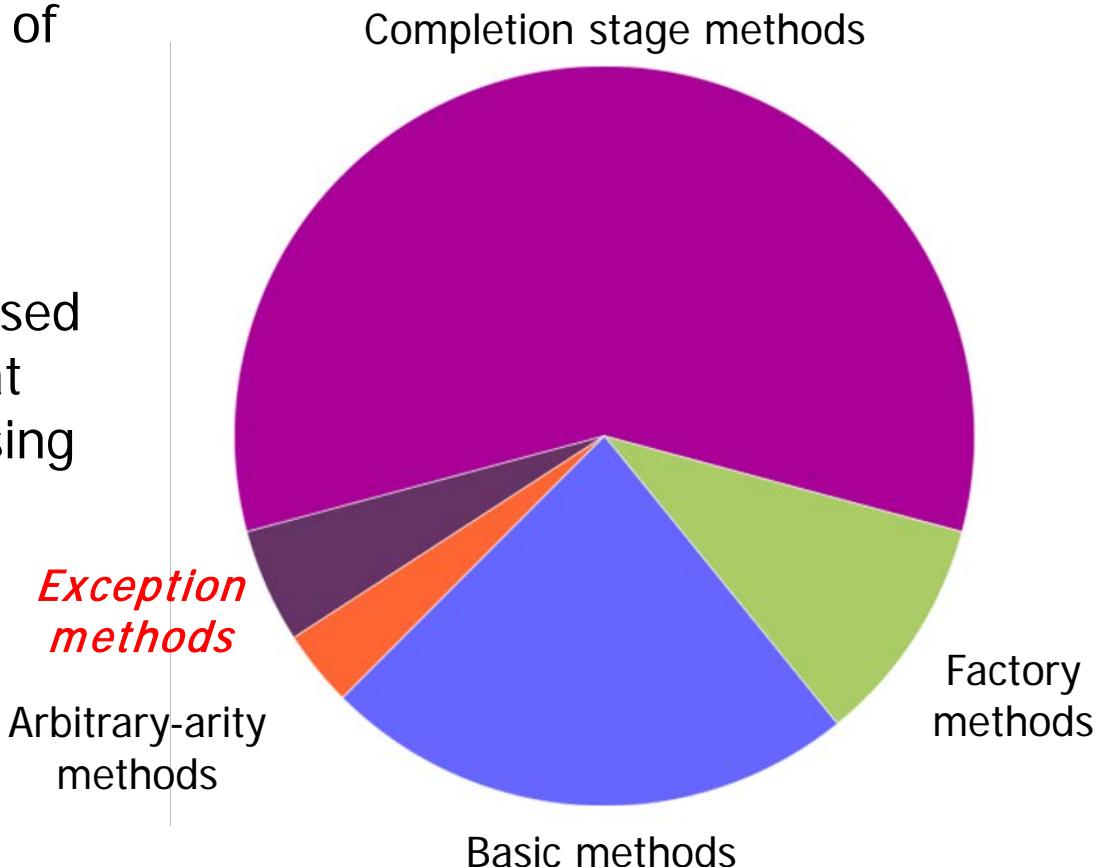
- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
  - Completion stage methods used to chain together actions that perform async result processing & composition
  - Apply completion stage methods to BigFractions

<<Java Class>>
<b>BigFraction</b> (default package)
mNumerator: BigInteger
mDenominator: BigInteger
<u>valueOf(Number):BigFraction</u>
<u>valueOf(Number,Number):BigFraction</u>
<u>valueOf(String):BigFraction</u>
<u>valueOf(Number,Number,boolean):BigFraction</u>
<u>reduce(BigFraction):BigFraction</u>
getNumerator():BigInteger
getDenominator():BigInteger
add(Number):BigFraction
subtract(Number):BigFraction
multiply(Number):BigFraction
divide(Number):BigFraction
gcd(Number):BigFraction
toMixedString():String

# Learning Objectives in this Part of the Lesson

---

- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
  - Completion stage methods used to chain together actions that perform async result processing & composition
    - Apply completion stage methods to BigFractions
    - Know how to handle runtime exceptions



---

# Applying Completable Future Completion Stage Methods

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFractions)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
                .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
                .thenCompose(reducedFrac -> CompletableFuture  
                    .supplyAsync(() -> reducedFrac  
                        .multiply(sBigFraction)));  
    ...  
}
```

*Lambda that asynchronously reduces/multiplies a big fraction*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
                .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
                .thenCompose(reducedFrac -> CompletableFuture  
                    .supplyAsync(() -> reducedFrac  
                        .multiply(sBigFraction)));  
    ...  
}
```

Asynchronously  
reduce a big fraction

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
  
            .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
  
            .thenCompose(reducedFrac -> CompletableFuture  
  
                thenCompose()  
                acts like flatMap()  
                    .supplyAsync(() -> reducedFrac  
                        .multiply(sBigFraction)));  
    ...  
}
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
  
            .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
  
            .thenCompose(reducedFrac -> CompletableFuture  
  
                Asynchronously  
                multiply big fractions  
                .supplyAsync(() -> reducedFrac  
                    .multiply(sBigFraction)));  
    ...  
}
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Consumer<List<BigFraction>> printSortedList = list -> {  
        CompletableFuture<List<BigFraction>> quickSortF =  
            CompletableFuture.supplyAsync(() -> quickSort(list));  
  
        CompletableFuture<List<BigFraction>> mergeSortF =  
            CompletableFuture.supplyAsync(() -> mergeSort(list));  
  
        quickSortF.acceptEither(mergeSortF, results ->  
            results.forEach(frac -> display(frac.toMixedString())));  
    }; ...  
}
```

*Sorts & prints a list  
of reduced fractions*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Consumer<List<BigFraction>> printSortedList = list -> {  
        CompletableFuture<List<BigFraction>> quickSortF =  
            CompletableFuture.supplyAsync(() -> quickSort(list));  
  
        Asynchronously apply quick sort & merge sort!  
        CompletableFuture<List<BigFraction>> mergeSortF =  
            CompletableFuture.supplyAsync(() -> mergeSort(list));  
  
        quickSortF.acceptEither(mergeSortF, results ->  
            results.forEach(frac -> display(frac.toMixedString()));  
    }; ...
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Consumer<List<BigFraction>> printSortedList = list -> {  
        CompletableFuture<List<BigFraction>> quickSortF =  
            CompletableFuture.supplyAsync(() -> quickSort(list));  
  
        Select whichever result finishes first..  
        CompletableFuture<List<BigFraction>> mergeSortF =  
            CompletableFuture.supplyAsync(() -> mergeSort(list));  
  
        quickSortF.acceptEither(mergeSortF, results ->  
            results.forEach(frac -> display(frac.toMixedString()));  
    }; ...
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Consumer<List<BigFraction>> printSortedList = list -> {  
        CompletableFuture<List<BigFraction>> quickSortF =  
            CompletableFuture.supplyAsync(() -> quickSort(list));  
  
        CompletableFuture<List<BigFraction>> mergeSortF =  
            CompletableFuture.supplyAsync(() -> mergeSort(list));  
  
        quickSortF.acceptEither(mergeSortF, results ->  
            results.forEach(frac -> display(frac.toMixedString()));  
    }; ...
```

If a future is already completed the action runs in the thread that registered the action (the invoking thread)

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Consumer<List<BigFraction>> printSortedList = list -> {  
        CompletableFuture<List<BigFraction>> quickSortF =  
            CompletableFuture.supplyAsync(() -> quickSort(list));  
  
        Otherwise, the action runs in the thread in which the previous stage ran  
        CompletableFuture<List<BigFraction>> mergeSortF =  
            CompletableFuture.supplyAsync(() -> mergeSort(list));  
  
        quickSortF.acceptEither(mergeSortF, results ->  
            results.forEach(frac -> display(frac.toMixedString()));  
    }; ...
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the `testFractionMultiplications()` method that multiplies big fractions using a stream of `CompletableFuture`s

```
static void testFractionMultiplications() {  
    ...  
    Consumer<List<BigFraction>> printSortedList = list -> {  
        CompletableFuture<List<BigFraction>> quickSortF =  
            CompletableFuture.supplyAsync(() -> quickSort(list));  
  
        CompletableFuture<List<BigFraction>> mergeSortF =  
            CompletableFuture.supplyAsync(() -> mergeSort(list));  
  
        quickSortF.acceptEither(mergeSortF, results ->  
            results.forEach(frac -> display(frac.toMixedString()));  
    }; ...
```

`acceptEither()` does *not* cancel the second future after the first one completes

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

*Generate a bounded # of large & random fractions*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
BigFraction makeBigFraction(Random random, boolean reduced) {  
    BigInteger numerator =  
        new BigInteger(150000, random);  
  
    BigInteger denominator =  
        numerator.divide(BigInteger.valueOf(random.nextInt(10)  
            + 1));  
  
    return BigFraction.valueOf(numerator,  
        denominator,  
        reduced);  
}
```

*Factory method that creates  
a large & random fraction*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

*Reduce & multiply these fractions asynchronously*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

*Return a future to a list of  
big fractions being reduced  
& multiplied asynchronously*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

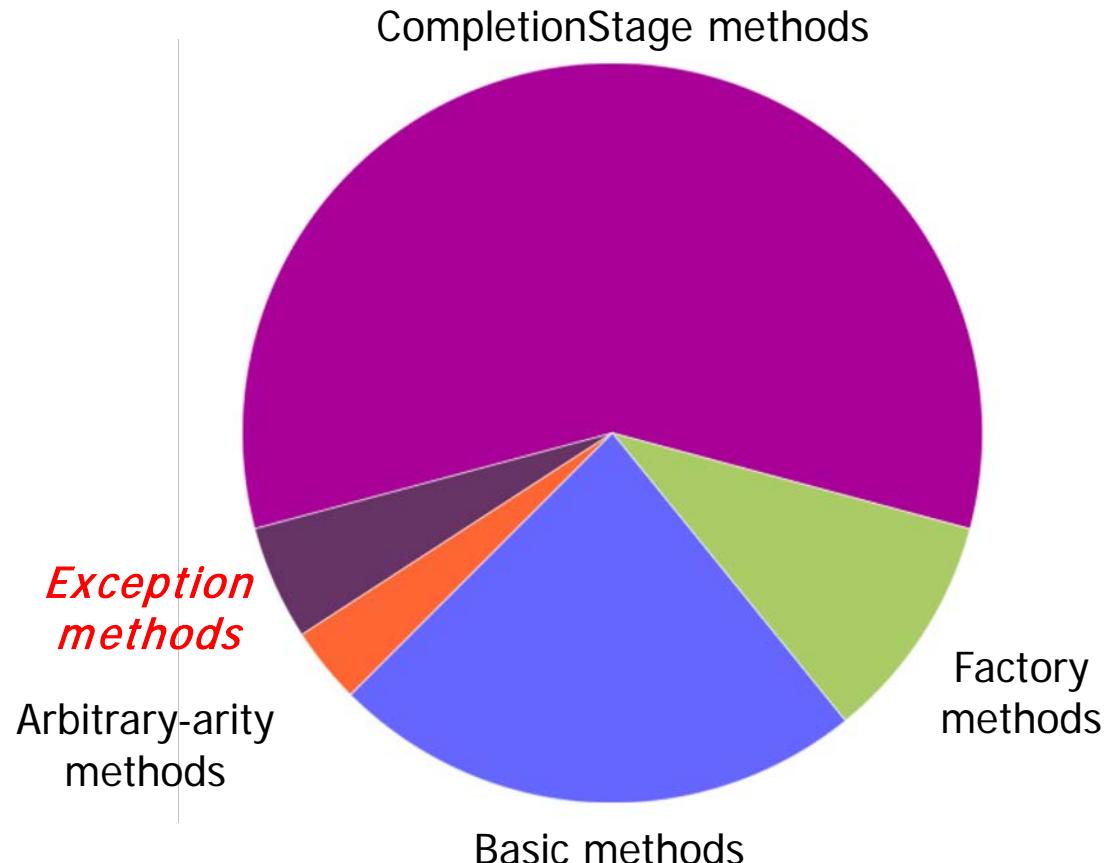
*Sort & print results when all  
async computations complete*

---

# Handling Runtime Exceptions in Completion Stages

# Handling Runtime Exceptions in Completion Stages

- Completion stage methods handle runtime exceptions



# Handling Runtime Exceptions in Completion Stages

- Completion stage methods handle runtime exceptions

Methods	Params	Returns	Behavior
<code>whenComplete(Async)</code>	<code>BiConsumer&lt;CompletableFuture&lt;T&gt;, Throwable&gt;</code>	<code>CompletableFuture&lt;T&gt;</code>	Handle outcome of a stage, whether a result value or an exception
<code>handle(Async)</code>	<code>BiFunction&lt;CompletableFuture&lt;T&gt;, T, CompletableFuture&lt;U&gt;&gt;</code>	<code>CompletableFuture&lt;U&gt;</code>	Handle outcome of a stage & return new value
<code>exceptionally</code>	<code>Function&lt;Throwable, CompletableFuture&lt;T&gt;&gt;</code>	<code>CompletableFuture&lt;T&gt;</code>	When exception occurs, replace exception with result value

# Handling Runtime Exceptions in Completion Stages

- Using the handle() method to handle exceptional or normal completions

```
CompletableFuture
```

```
.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))

.handle((fraction, ex) -> {
    if (fraction == null)
        return BigFraction.ZERO;
    else
        return fraction.multiply(sBigReducedFraction);
})

.thenAccept(fraction ->
    System.out.println(fraction.toMixedString()));
```

*Handle outcome  
of previous stage*

# Handling Runtime Exceptions in Completion Stages

- Using the handle() method to handle exceptional or normal completions

```
CompletableFuture
```

```
.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))

.handle((fraction, ex) -> {
    if (fraction == null)
        return BigFraction.ZERO;
    else
        return fraction.multiply(sBigReducedFraction);
})

.thenAccept(fraction ->
    System.out.println(fraction.toMixedString()));
```

*The exception path*

# Handling Runtime Exceptions in Completion Stages

- Using the handle() method to handle exceptional or normal completions

```
CompletableFuture
```

```
.supplyAsync(() ->
    BigFraction.valueOf(100, denominator))

.handle((fraction, ex) -> {
    if (fraction == null)
        return BigFraction.ZERO;
    else
        return fraction.multiply(sBigReducedFraction);
})

.thenAccept(fraction ->
    System.out.println(fraction.toMixedString()));
```

*The “normal” path*

# Handling Runtime Exceptions in Completion Stages

- Using the exceptionally() method to handle exceptional or normal completions

```
CompletableFuture
```

```
.supplyAsync(() ->  
    BigFraction.valueOf(100, denominator))  
  
.thenApply(fraction ->  
    fraction.multiply(sBigReducedFraction))  
  
.exceptionally(ex -> BigFraction.ZERO)  
  
.thenAccept(fraction ->  
    System.out.println(fraction.toMixedString()));
```

*Convert an exception to a 0 result*

---

# End of Overview of Advanced Java 8 CompletableFuture Features (Part 3)

# Overview of Advanced Java 8

# CompletableFuture Features (Part 4)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

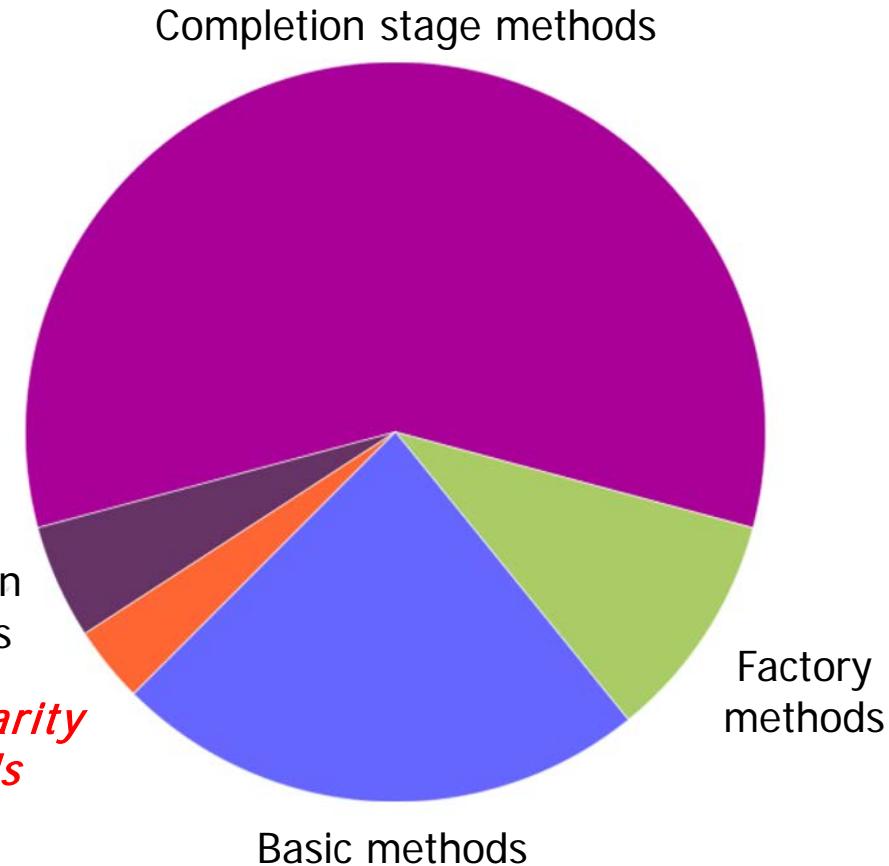
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
  - Completion stage methods used to chain together actions that perform async result processing & composition
- Arbitrary-arity methods that process futures in bulk

*Arbitrary-arity  
methods*



---

# Arbitrary-Arity Methods Process Futures in Bulk

# Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods are triggered after completion of some/all of  $n$  futures

Methods	Params	Returns	Behavior
allOf	Varargs	CompletableFuture<Void>	Return a future that completes when all futures in params complete
anyOf	Varargs	CompletableFuture<Void>	Return a future that completes when any future in params complete

These “arbitrary-arity” methods are hard to program without using wrappers

# Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods are triggered after completion of some/all of  $n$  futures

<<Java Class>>

**CompletableFuture<T>**

- `CompletableFuture()`
- `cancel(boolean):boolean`
- `isCancelled():boolean`
- `isDone():boolean`
- `get()`
- `get(long,TimeUnit)`
- `join()`
- `complete(T):boolean`
- `supplyAsync(Supplier<U>):CompletableFuture<U>`
- `supplyAsync(Supplier<U>,Executor):CompletableFuture<U>`
- `runAsync(Runnable):CompletableFuture<Void>`
- `runAsync(Runnable,Executor):CompletableFuture<Void>`
- `completedFuture(U):CompletableFuture<U>`
- `thenApply(Function<?>):CompletableFuture<U>`
- `thenAccept(Consumer<? super T>):CompletableFuture<Void>`
- `thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>`
- `thenCompose(Function<?>):CompletableFuture<U>`
- `whenComplete(BiConsumer<?>):CompletableFuture<T>`
- `allOf(CompletableFuture[]<?>):CompletableFuture<Void>`
- `anyOf(CompletableFuture[]<?>):CompletableFuture<Object>`

See [en.wikipedia.org/wiki/Arity](https://en.wikipedia.org/wiki/Arity)

# Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods are triggered after completion of some/all of  $n$  futures
  - Can wait for any or all completable futures in an array to complete

«Java Class»

**CompletableFuture<T>**

- `CompletableFuture()`
- `cancel(boolean):boolean`
- `isCancelled():boolean`
- `isDone():boolean`
- `get()`
- `get(long,TimeUnit)`
- `join()`
- `complete(T):boolean`
- `supplyAsync(Supplier<U>):CompletableFuture<U>`
- `supplyAsync(Supplier<U>,Executor):CompletableFuture<U>`
- `runAsync(Runnable):CompletableFuture<Void>`
- `runAsync(Runnable,Executor):CompletableFuture<Void>`
- `completedFuture(U):CompletableFuture<U>`
- `thenApply(Function<?>):CompletableFuture<U>`
- `thenAccept(Consumer<? super T>):CompletableFuture<Void>`
- `thenCombine(CompletionStage<? extends U>,BiFunction<?>):CompletableFuture<V>`
- `thenCompose(Function<?>):CompletableFuture<U>`
- `whenComplete(BiConsumer<?>):CompletableFuture<T>`
- `allOf(CompletableFuture[]<?>):CompletableFuture<Void>`
- `anyOf(CompletableFuture[]<?>):CompletableFuture<Object>`

# Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods are triggered after completion of some/all of  $n$  futures
  - Can wait for any or all completable futures in an array to complete

We focus on `allOf()`, which is like `thenCombine()` on steroids!

<<Java Class>>	
CompletableFuture<T>	
•	CompletableFuture()
•	cancel(boolean):boolean
•	isCancelled():boolean
•	isDone():boolean
•	get()
•	get(long, TimeUnit)
•	join()
•	complete(T):boolean
•	supplyAsync(Supplier<U>):CompletableFuture<U>
•	supplyAsync(Supplier<U>, Executor):CompletableFuture<U>
•	runAsync(Runnable):CompletableFuture<Void>
•	runAsync(Runnable, Executor):CompletableFuture<Void>
•	completedFuture(U):CompletableFuture<U>
•	thenApply(Function<?>):CompletableFuture<U>
•	thenAccept(Consumer<? super T>):CompletableFuture<Void>
•	thenCombine(CompletionStage<? extends U>, BiFunction<?>):CompletableFuture<V>
•	thenCompose(Function<?>):CompletableFuture<U>
•	whenComplete(BiConsumer<?>):CompletableFuture<T>
•	allOf(CompletableFuture[]<?>):CompletableFuture<Void>
•	anyOf(CompletableFuture[]<?>):CompletableFuture<Object>

# Arbitrary-Arity Methods Process Futures in Bulk

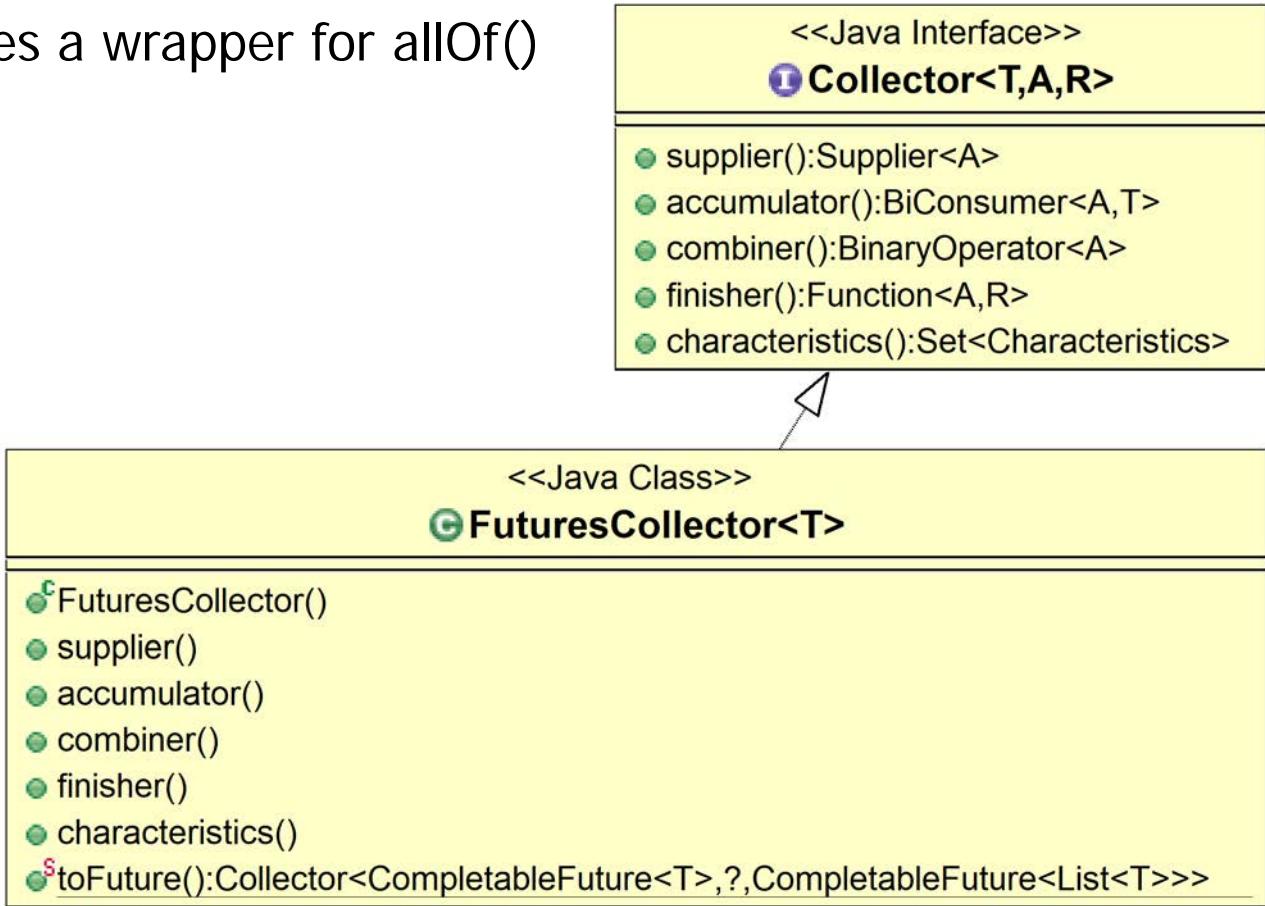
---

- FuturesCollector is used to return a completable future to a list of big fractions that are being reduced and multiplied asynchronously

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

# Arbitrary-Arity Methods Process Futures in Bulk

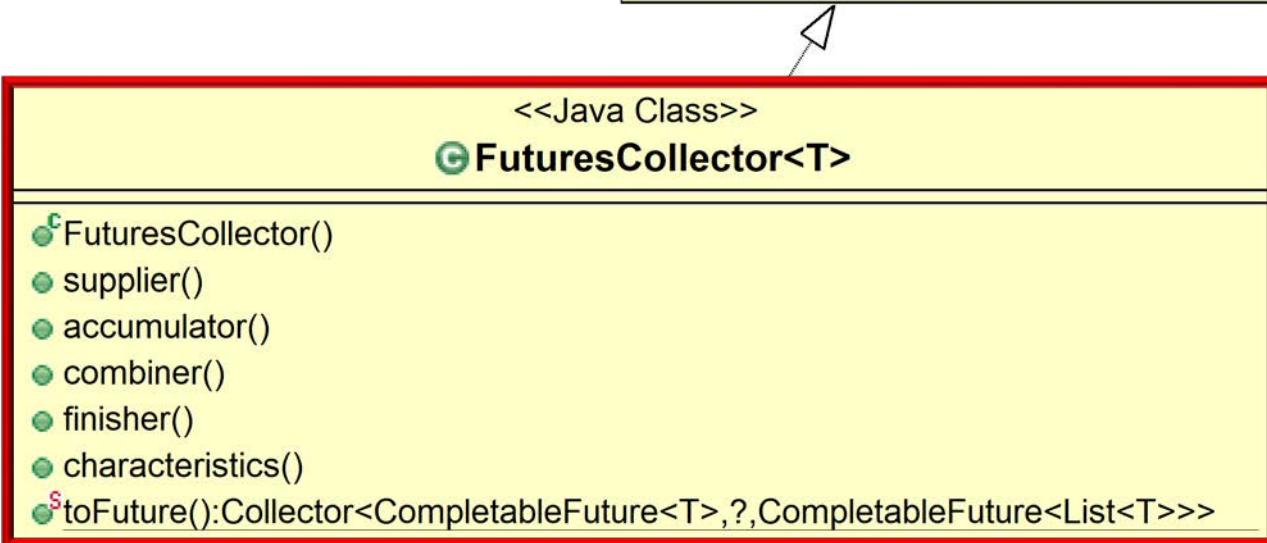
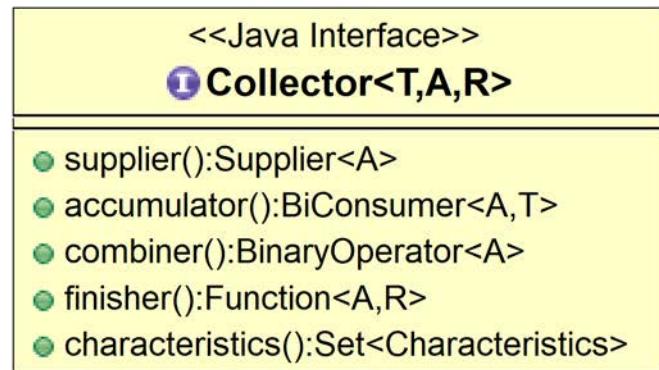
- FuturesCollector provides a wrapper for allOf()



See [Java8/ex8/utils/FuturesCollector.java](#)

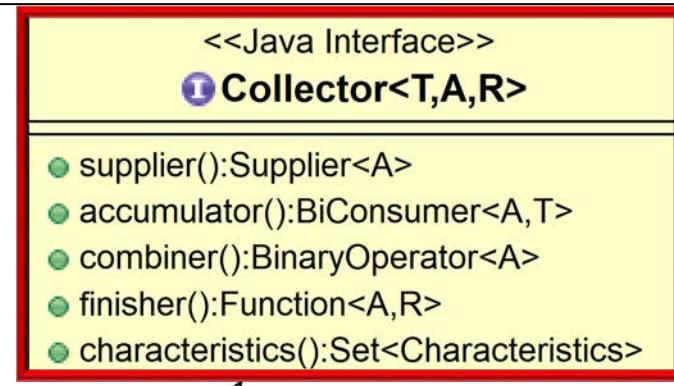
# Arbitrary-Arity Methods Process Futures in Bulk

- `FuturesCollector` provides a wrapper for `allOf()`
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete



# Arbitrary-Arity Methods Process Futures in Bulk

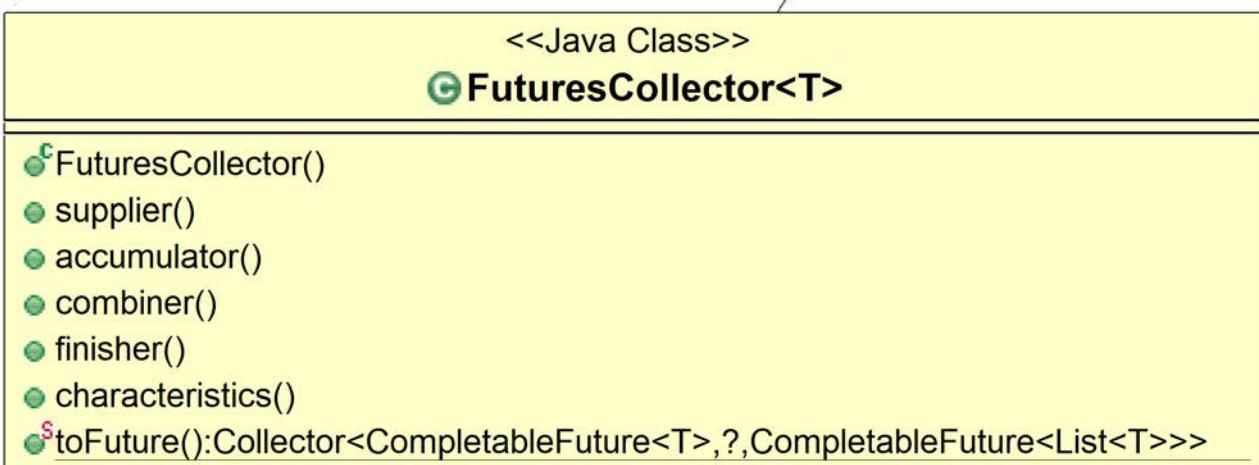
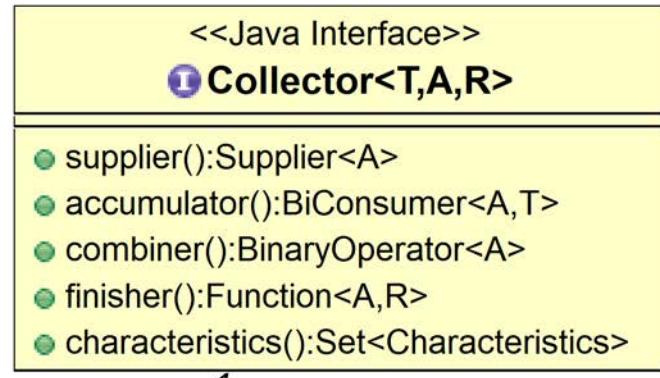
- `FuturesCollector` provides a wrapper for `allOf()`
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete
  - Implements the `Collector` interface that accumulates input elements into a mutable result container



See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()



FuturesCollector provides a powerful wrapper for some complex code!!!

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
    implements Collector<CompletableFuture<T>,
               List<CompletableFuture<T>>,
               CompletableFuture<List<T>>> {
```

...

*Implements a  
custom collector*

# Arbitrary-Arity Methods Process Futures in Bulk

---

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>  
    implements Collector<CompletableFuture<T>,  
              List<CompletableFuture<T>>,  
              CompletableFuture<List<T>>> {
```

...

*The type of input elements  
to the accumulator() method*

# Arbitrary-Arity Methods Process Futures in Bulk

---

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>  
    implements Collector<CompletableFuture<T>,  
              List<CompletableFuture<T>>,  
              CompletableFuture<List<T>>> {  
    ...
```

*The mutable accumulation type  
of the accumulator() method*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>  
    implements Collector<CompletableFuture<T>,  
              List<CompletableFuture<T>>,  
              CompletableFuture<List<T>> {  
    ...
```

*The result type of the  
finisher() method, i.e., the  
final output of the collector*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
    implements Collector<CompletableFuture<T>,
               List<CompletableFuture<T>>,
               CompletableFuture<List<T>>> {
    public Supplier<List<CompletableFuture<T>>> supplier() {
        return ArrayList::new;
    }
    public BiConsumer<List<CompletableFuture<T>>,
                     CompletableFuture<T>> accumulator() {
        return List::add;
    }
    ...
}
```

*Factory method that creates & returns  
a new mutable array list container*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
    implements Collector<CompletableFuture<T>,
               List<CompletableFuture<T>>,
               CompletableFuture<List<T>>> {
    public Supplier<List<CompletableFuture<T>>> supplier() {
        return ArrayList::new;
    }
    public BiConsumer<List<CompletableFuture<T>>,
                     CompletableFuture<T>> accumulator() {
        return List::add;
    }
    ...
}
```

*Folds a new completable future into  
the mutable array list container*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>  
{  
    ...  
    public BinaryOperator<List<CompletableFuture<T>>> combiner() {  
        return (List<CompletableFuture<T>> one,  
                List<CompletableFuture<T>> another) -> {  
            one.addAll(another);  
            return one;  
        };  
    }  
    ...  
}
```

*Accepts two partial array list results  
& merges them into a single array list*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
{
    ...
    public Function<List<CompletableFuture<T>>,
                    CompletableFuture<List<T>>> finisher() {
        return futures -> CompletableFuture
            .allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(toList()));
    }
    ...
}
```

*Perform final transformation from the intermediate array list accumulation type to the final completable future result type*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
{
    ...
    public Function<List<CompletableFuture<T>>,
                    CompletableFuture<List<T>>> finisher() {
        return futures -> CompletableFuture
            .allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(toList())));
    }
    ...
}
```



*Convert list of futures to array of futures & pass to allOf() to obtain a future that will complete when all futures complete*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
{
    ...
    public Function<List<CompletableFuture<T>>,
                    CompletableFuture<List<T>>> finisher() {
        return futures -> CompletableFuture
            .allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(toList()));
    }
    ...
}
```



*When all futures have completed get a single future to a list of joined elements of type T*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
{
    ...
    public Function<List<CompletableFuture<T>>,
                    CompletableFuture<List<T>>> finisher() {
        return futures -> CompletableFuture
            .allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(toList())));
    }
    ...
}
```

*This call to join() will never block!*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>  
{  
    ...  
    public Function<List<CompletableFuture<T>>,  
                      CompletableFuture<List<T>>> finisher() {  
        return futures -> CompletableFuture  
            .allOf(futures.toArray(new CompletableFuture[0]))  
  
        Return a future to a list of elements of T  
    }  
    ...  
    .thenApply(v -> futures.stream()  
        .map(CompletableFuture::join)  
        .collect(toList()));  
}
```

# Arbitrary-Arity Methods Process Futures in Bulk

---

- FuturesCollector is used to return a completable future to a list of big fractions that are being reduced and multiplied asynchronously

```
static void testFractionMultiplications() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(printSortedList);  
}
```

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>
{
    ...
    public Set characteristics() {
        return Collections.singleton(Characteristics.UNORDERED);
    }
}
```

*Returns a set indicating the characteristics of FutureCollector*

```
public static <T> Collector<CompletableFuture<T>, ?,  
                    CompletableFuture<List<T>>>  
toFuture() {  
    return new FuturesCollector<>();  
}
```

FuturesCollector is thus a *non-concurrent* collector

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()

```
public class FuturesCollector<T>  
{  
    ...  
    public Set<Characteristics> characteristics() {  
        return Collections.singleton(Characteristics.UNORDERED);  
    }  
  
    public static <T> Collector<CompletableFuture<T>, ?,  
        CompletableFuture<List<T>>>  
        toFuture() {  
        return new FuturesCollector<>();  
    }  
}
```

*Static factory method creates  
a new FuturesCollector*

---

# End of Overview of Advanced Java 8 CompletableFuture Features (Part 4)

# Overview of Advanced Java 8

# CompletableFuture Features (Part 5)

Douglas C. Schmidt

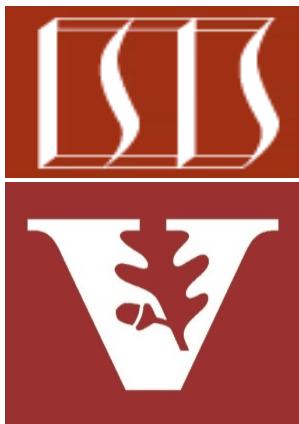
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

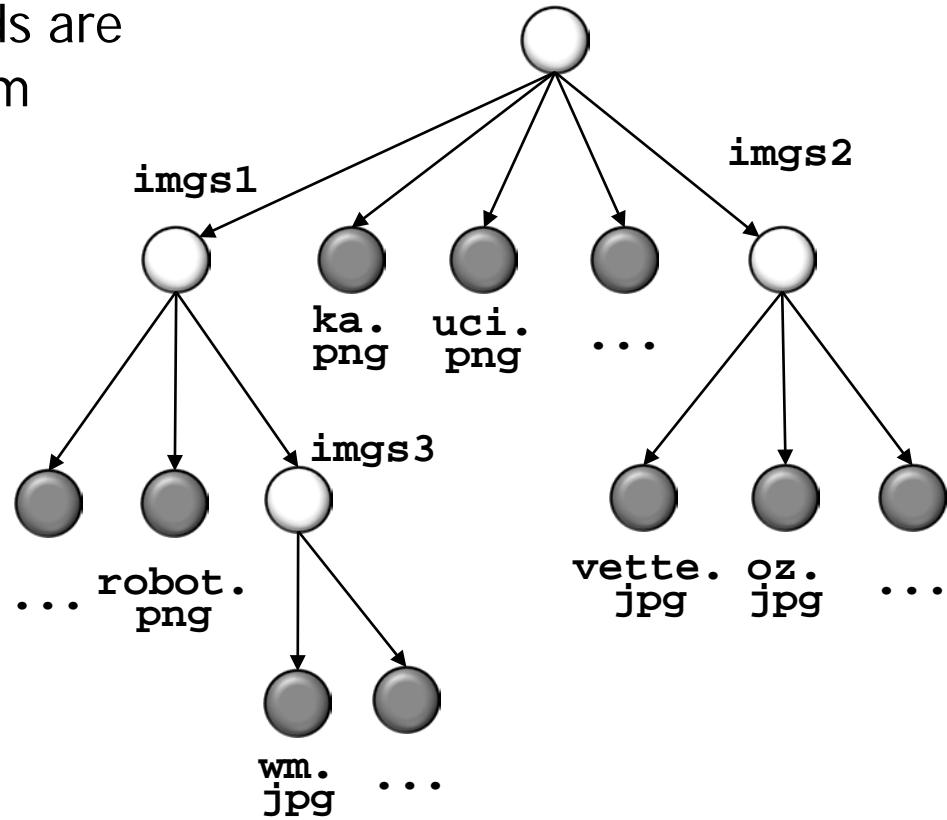
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



## Learning Objectives in this Part of the Lesson

- Know how completion stage methods are applied in the image crawler program



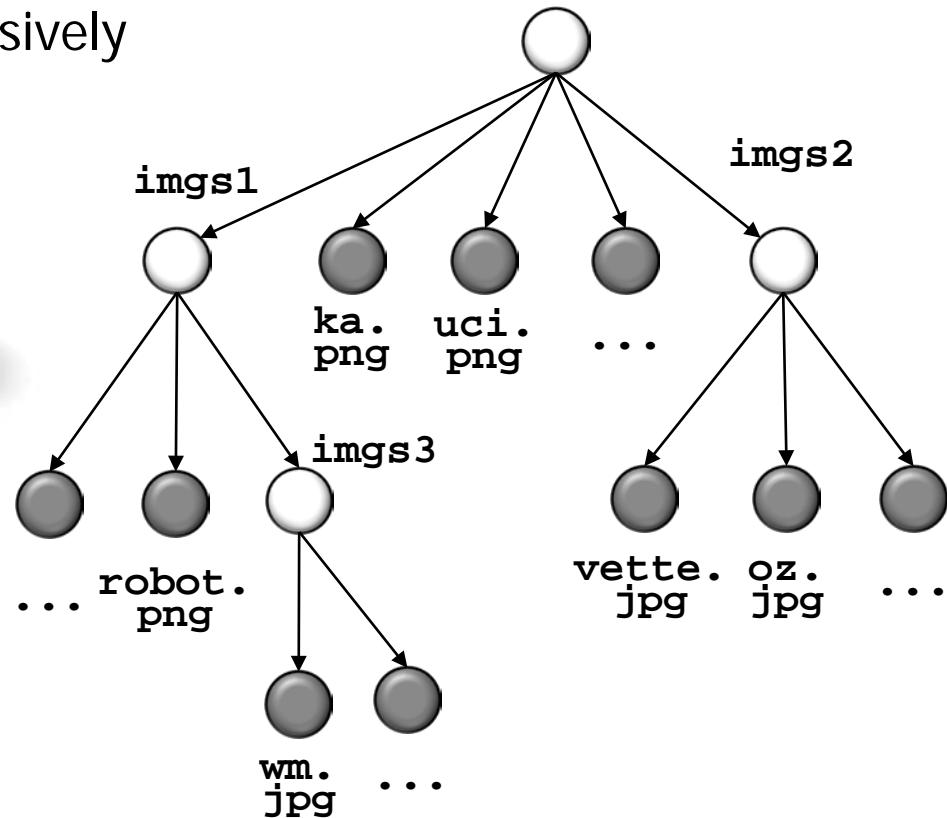
See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex19)

---

# Applying Completion Stage Methods to Image Crawler

# Applying Completion Stage Methods to Image Crawler

- We show how to apply key completable stage methods in the context of a program that crawls web pages recursively



# Applying Completion Stage Methods to Image Crawler

- We show how to apply key completable stage methods in the context of a program that crawls web pages recursively
  - This program counts the # of images on each page

```
ImageCounter[Depth2]: Already  
processed imgs1/index.html
```

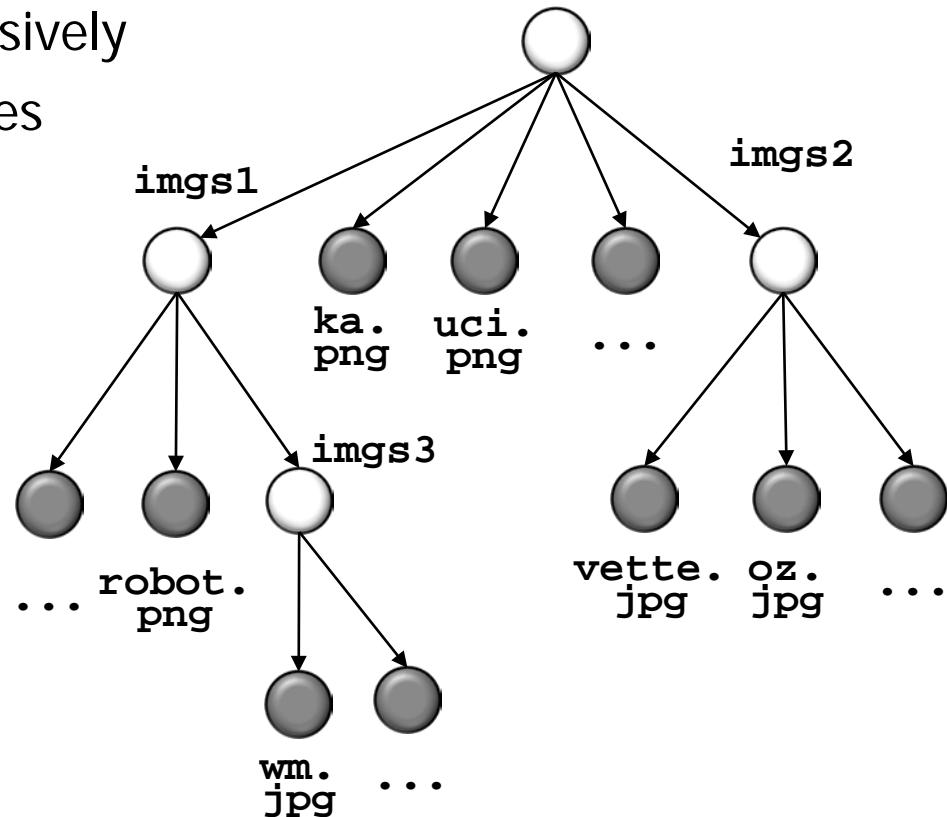
```
ImageCounter[Depth3]: Exceeded max  
depth of 2
```

```
ImageCounter[Depth2]: found 7 images  
for imgs1/index.html in thread 12
```

```
ImageCounter[Depth2]: found 7 images  
for imgs2/index.html in thread 13
```

```
ImageCounter[Depth1]: found 21 images  
for index.html in thread 13
```

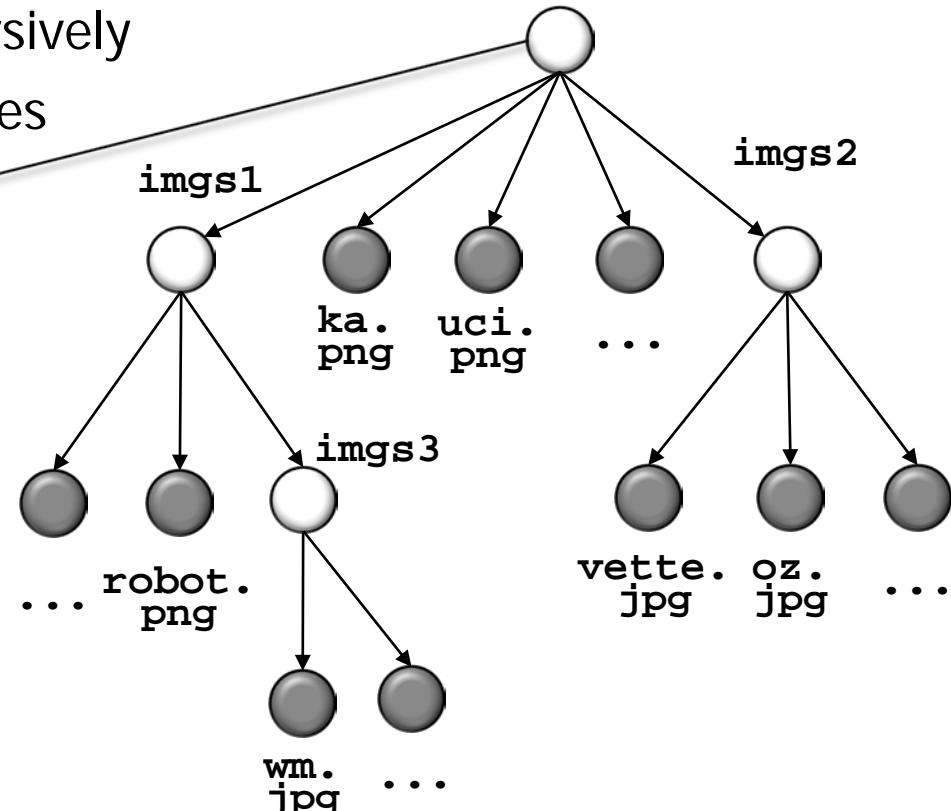
```
ImageCounter: 21 total image(s) are  
reachable from index.html
```



# Applying Completion Stage Methods to Image Crawler

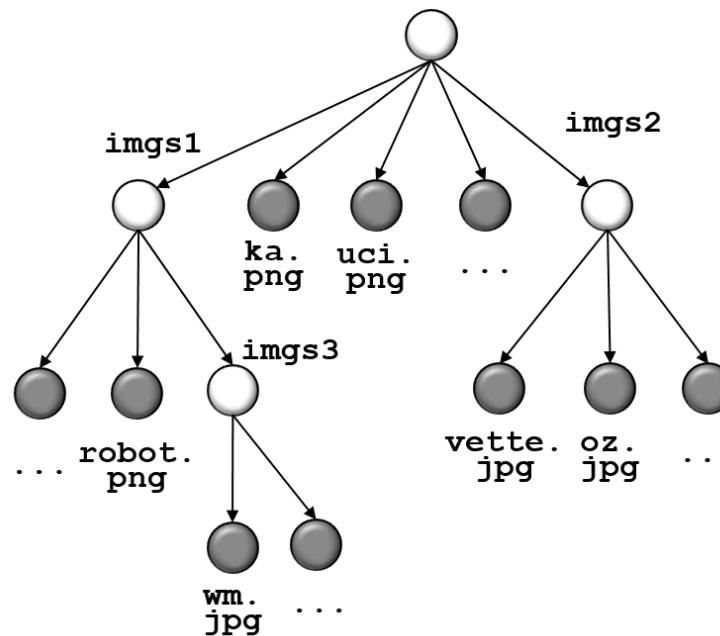
- We show how to apply key completable stage methods in the context of a program that crawls web pages recursively
  - This program counts the # of images on each page

*The root folder can either reside locally (filesystem-based) or be accessed remotely (web-based)*



# Applying Completion Stage Methods to Image Crawler

- The ImageCounter class is heart of this program



<<Java Class>>

**ImageCounter**

- ImageCounter()
- countImages(String,int):CompletableFuture<Integer>
- countImagesAsync(String,int):CompletableFuture<Integer>
- countImagesMapReduce(String,OtherParams):CompletableFuture<Integer>
- getStartPage(String):CompletableFuture<Document>
- getImagesOnPage(Document):Elements
- crawlLinksInPage(Document,int):CompletableFuture<List<Integer>>
- print(String):void

# Applying Completion Stage Methods to Image Crawler

---

- ImageCounter class & fields

```
class ImageCounter {
```

*Counts # of images in a recursively-defined folder structure using many features of completable future*

```
private final ConcurrentHashSet<String> mUniqueUris =  
    new ConcurrentHashMap<>();
```

```
private CompletableFuture<Integer> mZero =  
    CompletableFuture.completedFuture(0);
```

...

# Applying Completion Stage Methods to Image Crawler

- ImageCounter class & fields

```
class ImageCounter {
```

*A cache of unique URIs that have already been processed*

```
private final ConcurrentHashMap<String> mUniqueUris =  
    new ConcurrentHashMap<>();
```

```
private CompletableFuture<Integer> mZero =  
    CompletableFuture.completedFuture(0);
```

...

See [Java8/ex19/src/main/java/utils/ConcurrentHashSet.java](#)

# Applying Completion Stage Methods to Image Crawler

---

- ImageCounter class & fields

```
class ImageCounter {
```

```
    private final ConcurrentHashSet<String> mUniqueUris =  
        new ConcurrentHashMap<>();
```

```
    private CompletableFuture<Integer> mZero =  
        CompletableFuture.completedFuture(0);
```

```
    ...
```

*Stores a completed future with value of 0*

# Applying Completion Stage Methods to Image Crawler

---

- ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1)  
  
.exceptionally(ex -> 0)  
  
.thenAccept(total ->  
            print(TAG + ":"  
                + total  
                + " total image(s) are reachable from "  
                + rootUri))  
  
.join();  
}
```

# Applying Completion Stage Methods to Image Crawler

- ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1) // Start at root Uri of the web page  
  
    .exceptionally(ex -> 0)  
  
    .thenAccept(total ->  
        print(TAG + ":" + total  
              + " total image(s) are reachable from "  
              + rootUri))  
  
    .join();  
}
```

# Applying Completion Stage Methods to Image Crawler

- ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1) Perform image counting & return future to Long  
  
.exceptionally(ex -> 0)  
  
.thenAccept(total ->  
            print(TAG + ":"  
                  + total  
                  + " total image(s) are reachable from "  
                  + rootUri))  
  
.join();  
}
```

# Applying Completion Stage Methods to Image Crawler

- ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1)  
        .exceptionally(ex -> 0)  
            .thenAccept(total ->  
                print(TAG + ":"  
                    + total  
                    + " total image(s) are reachable from "  
                    + rootUri))  
        .join();  
}
```

*Return 0 if an exception occurs*

# Applying Completion Stage Methods to Image Crawler

- ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1)  
  
.exceptionally(ex -> 0)           When future completes print total # of images  
  
.thenAccept(total ->  
    print(TAG + ":"  
        + total  
        + " total image(s) are reachable from "  
        + rootUri))  
  
.join();  
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenAccept](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#thenAccept)

# Applying Completion Stage Methods to Image Crawler

- ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1)  
  
.exceptionally(ex -> 0)  
  
.thenAccept(total ->  
            print(TAG + ":"  
                + total  
                + " total image(s) are reachable from "  
                + rootUri))  
  
.join();  
}
```

*Block until all futures complete*

# Applying Completion Stage Methods to Image Crawler

- An alternative ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1)  
        .handle((total, ex) -> {  
            if (total == null) total = 0;  
            print(TAG + ": " + total  
                  + " image(s) are reachable from " + rootUri);  
            return 0;  
        })  
        .join();  
}
```

*An alternative means  
of handling exceptions*

# Applying Completion Stage Methods to Image Crawler

- An alternative ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1) The non-exception path  
        .handle((total, ex) -> {  
            if (total == null) total = 0;  
            print(TAG + ": " + total  
                  + " image(s) are reachable from " + rootUri);  
            return 0;  
        })  
        .join();  
}
```

# Applying Completion Stage Methods to Image Crawler

- An alternative ImageCounter constructor implementation

```
public ImageCounter() {  
    String rootUri = Options.instance().getRootUri();  
  
    countImages(rootUri, 1)  
        .handle((total, ex) -> {  
            if (total == null) total = 0;  
            print(TAG + ": " + total  
                  + " image(s) are reachable from " + rootUri);  
            return 0;  
        })  
        .join();  
}
```

*The exception path*

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found " + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...  
}
```

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    Return if recursion depth exceeded  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found " + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...
```

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found " + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...
```

*Process each page once..*

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found " + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...
```

*Helper method*

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found " + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...
```

*Handle  
outcome  
of stage*

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found " + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...
```

*Non-exception path*

# Applying Completion Stage Methods to Image Crawler

- Perform image counting asynchronously

```
CompletableFuture<Integer> countImages(String pageUri,  
                                         int depth) {  
  
    if (depth > Options.instance().maxDepth()) return mZero;  
  
    else if (!mUniqueUris.putIfAbsent(pageUri)) return mZero;  
  
    else return countImagesAsync(pageUri, depth)  
        .whenComplete((total, ex) -> {  
            if (total != null)  
                print(TAG + "[Depth" + depth + "]: found" + total  
                     + " images for " + pageUri);  
            else print(TAG + ": exception " + ex.getMessage());  
        }); ...  
}
```

*Exception path*

# Applying Completion Stage Methods to Image Crawler

---

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPageF = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApplyList::size;  
  
    CompletableFuture<Integer> imagesInLinksF =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
        .thenCompose(Function.identity());  
  
    return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPageF = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApplyList::size);  
  
    CompletableFuture<Integer> imagesInLinksF =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
        .thenCompose(Function.identity());  
  
    return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

*Async get the page  
at the root URI*

# Applying Completion Stage Methods to Image Crawler

- Returns a future to the page at the root URI

```
CompletableFuture<Document> getStartPage(String pageUri) {  
    return CompletableFuture  
        .supplyAsync(() -> Options  
            .instance()  
            .getJSuper()  
            .getPage(pageUri));  
}
```

*Uses supplyAsync() to return a future that completes when the requested page has finished downloading*

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPageF = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApply(List::size);  
  
    CompletableFuture<Integer> imagesInLinksF =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
        .thenCompose(Function.identity())  
  
    return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

*Async count # of  
images on page &  
return future to count*

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPage = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApply(List::size);  
  
    CompletableFuture<Integer> imagesInLinks =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
        .thenCompose(Function.identity())  
  
    return combineImageCounts(imagesInPage, imagesInLinks); ...
```

*Sync return a collection of  
IMG SRC URLs in this page*

# Applying Completion Stage Methods to Image Crawler

- Synchronously returns a collection of IMG SRC URLs in this page

```
Elements getImagesOnPage(Document page) {  
    return page.select("img");  
}
```

*This implementation uses the jsoup HTML parsing library, which operates synchronously – hence the need to call this method via supplyAsync()*

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPageF = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApply(List::size);  
  
    CompletableFuture<Integer> imagesInLinksF =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
            .thenCompose(Function.identity())
```

*Async count # of images in links on this page & return a future to count*

```
return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPageF = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApply(List::size);  
  
    CompletableFuture<Integer> imagesInLinksF =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
            .thenCompose(Function.identity())
```

*Sync return a list of the # of IMG SRC URLs accessible via links in this page*

```
return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

# Applying Completion Stage Methods to Image Crawler

- Synchronously return # of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,  
                                              int depth) {  
  
    return page  
        .select("a[href]").stream()  
        .map(hyperLink ->  
            countImages(Options.instance().getJSuper()  
                .getHyperLink(hyperLink), depth+1))  
  
    .collect(FuturesCollector.toFuture())  
  
    .thenApply(list -> list.stream().reduce(0, Integer::sum));  
}
```

*This jsoup call operates synchronously,  
so call this method via supplyAsync()*

See [jsoup.org/apidocs/org/jsoup/select/Selector.html](https://jsoup.org/apidocs/org/jsoup/select/Selector.html)

# Applying Completion Stage Methods to Image Crawler

- Synchronously return # of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,  
                                              int depth) {  
  
    return page  
        .select("a[href]").stream()  
        .map(hyperLink ->  
            countImages(Options.instance().getJSuper()  
                .getHyperLink(hyperLink), depth+1))  
  
        .collect(FuturesCollector.toFuture())  
  
        .thenApply(list -> list.stream().reduce(0, Integer::sum));  
}
```

*Convert list of hyperlinks into a stream*

# Applying Completion Stage Methods to Image Crawler

- Synchronously return # of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,  
                                              int depth) {  
  
    return page  
        .select("a[href]").stream()  
        .map(hyperLink ->  
            countImages(Options.instance().getJSuper()  
                .getHyperLink(hyperLink), depth+1))  
  
    .collect(FuturesCollector.toFuture())  
  
    .thenApply(list -> list.stream().reduce(0, Integer::sum));  
}
```

*Recursively visit all hyperlinks on this page*

# Applying Completion Stage Methods to Image Crawler

- Synchronously return # of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,  
                                              int depth) {  
  
    return page  
        .select("a[href]").stream()  
        .map(hyperLink ->  
            countImages(Options.instance().getJSuper()  
                .getHyperLink(hyperLink), depth+1))
```

*Custom collector converts a stream of futures into a single future that's triggered when all futures in stream complete*

```
.collect(FuturesCollector.toFuture())  
  
.thenApply(list -> list.stream().reduce(0, Integer::sum));
```

```
}
```

See coverage of the `FuturesCollector` class in Part 4 of this lesson

# Applying Completion Stage Methods to Image Crawler

- Synchronously return # of IMG SRC URLs accessible via links in this page

```
CompletableFuture<Integer> crawlLinksInPage(Document page,  
                                              int depth) {  
  
    return page  
        .select("a[href]").stream()  
        .map(hyperLink ->  
            countImages(Options.instance().getJSuper()  
                .getHyperLink(hyperLink), depth+1))  
  
    After all futures have completed then create a new future  
    that will trigger after all image counts are summed together  
  
    .collect(FuturesCollector.toFuture())  
  
    .thenApply(list -> list.stream().reduce(0, Integer::sum));  
}
```

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
    CompletableFuture<Document> pageF = getStartPage(pageUri);
```

```
CompletableFuture<Integer> imagesInPageF = pageF  
    .thenApplyAsync(this::getImagesOnPage)  
    .thenApply(List::size);
```



```
CompletableFuture<Integer> imagesInLinksF =  
    pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
    .thenCompose(Function.identity());
```

*These methods run concurrently after pageF completes*



```
return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

# Applying Completion Stage Methods to Image Crawler

- Helper method creates several lambdas variables to simplify implementation

```
CompletableFuture<Integer> countImagesAsync(String pageUri,  
                                              int depth) {  
    CompletableFuture<Document> pageF = getStartPage(pageUri);  
  
    CompletableFuture<Integer> imagesInPageF = pageF  
        .thenApplyAsync(this::getImagesOnPage)  
        .thenApply(List::size);  
  
    CompletableFuture<Integer> imagesInLinksF =  
        pageF.thenApplyAsync(page -> crawlLinksInPage(page, depth))  
            .thenCompose(Function.identity())  
  
    Async count # of images on this page & accessible via links on this page  
    return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

# Applying Completion Stage Methods to Image Crawler

- Helper method that combines image counts asynchronously

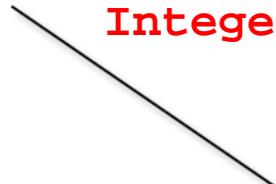
```
CompletableFuture<Integer> combineImageCounts  
(CompletableFuture<Integer> imagesInPageF,  
 CompletableFuture<Integer> imagesInLinksF){  
 imagesInPageF  
 .thenCombine(imagesInLinksF,  
 Integer::sum);
```

*Asynchronously count the # of images on this page  
plus # of images on hyperlinks accessible via the page*

# Applying Completion Stage Methods to Image Crawler

- Helper method that combines image counts asynchronously

```
CompletableFuture<Integer> combineImageCounts  
(CompletableFuture<Integer> imagesInPageF,  
 CompletableFuture<Integer> imagesInLinksF) {  
     imagesInPageF  
         .thenCombine(imagesInLinksF,  
                     Integer::sum);
```



*When both futures complete return a new future to a computation that combines & adds their results*

---

# End of Overview of Advanced Java 8 CompletableFuture Features (Part 5)

# Java 8 CompletableFuture

## ImageStreamGang Example (Part 1)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

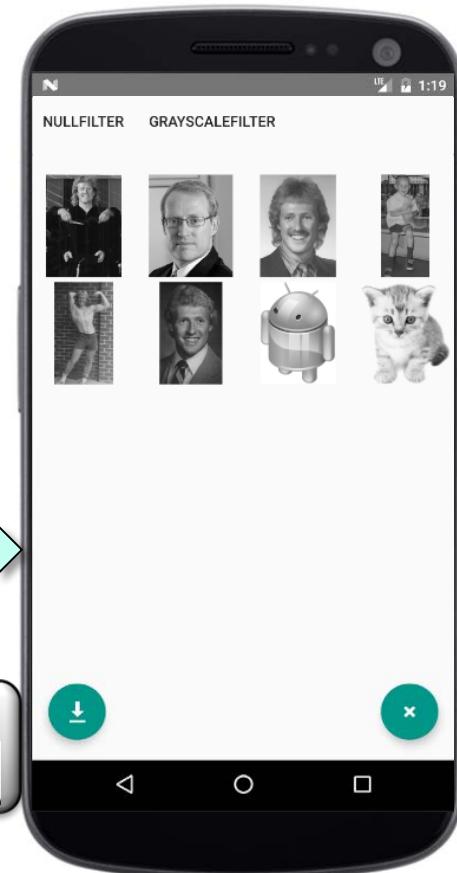
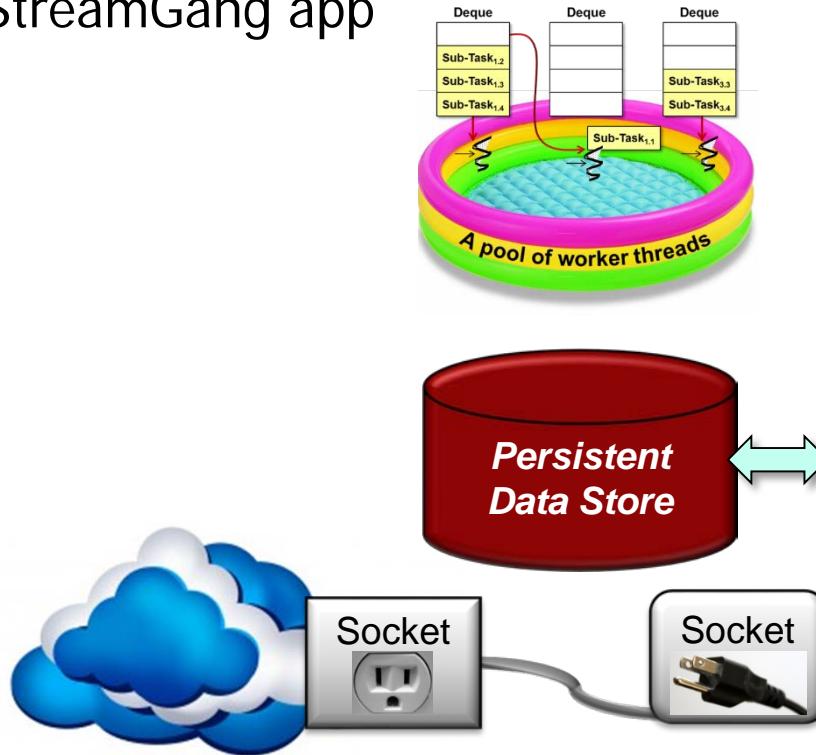
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Understand the design of the Java 8 completable future version of the ImageStreamGang app

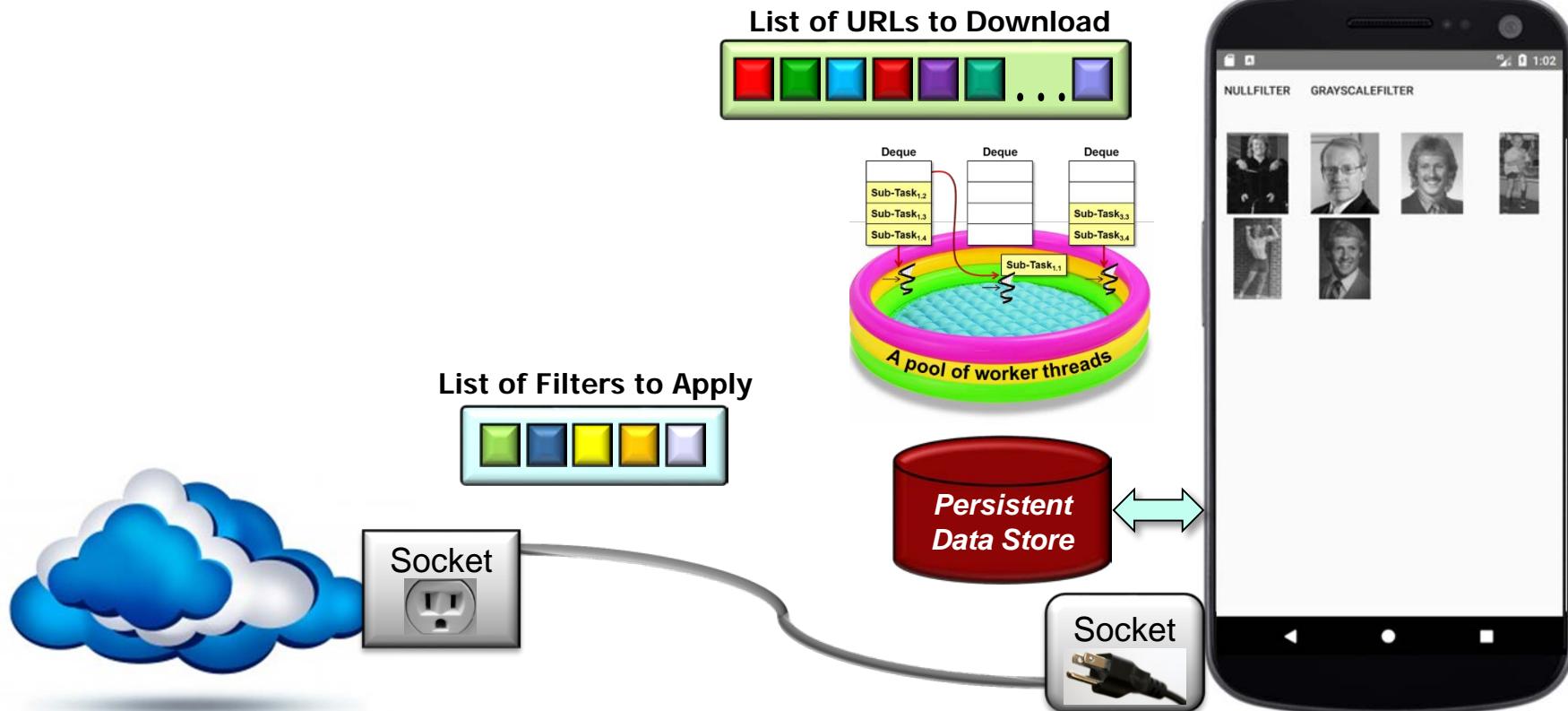


---

# Overview of the Completable Futures ImageStreamGang

# Overview of Completable Futures ImageStreamGang

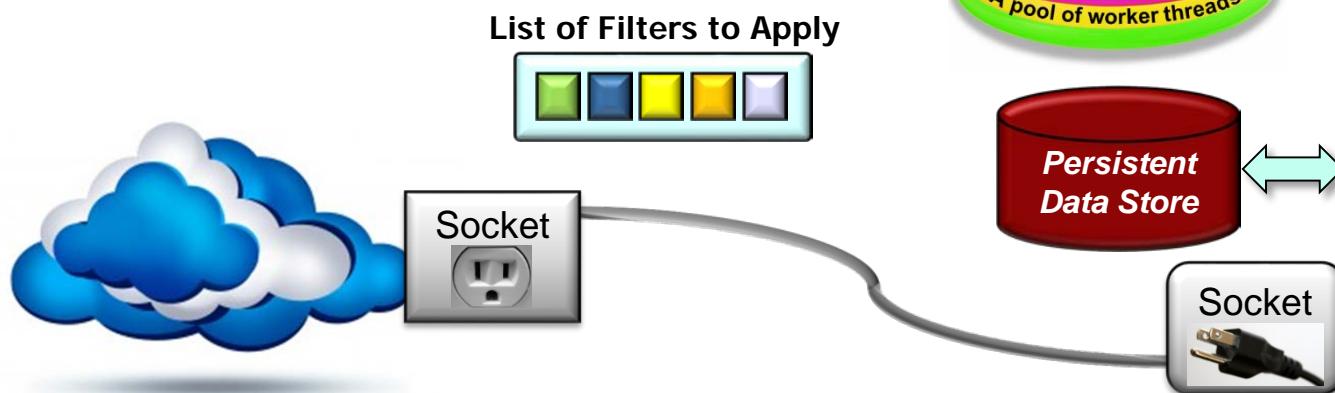
- ImageStreamGang shows advanced completable future features



See [github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang](https://github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang)

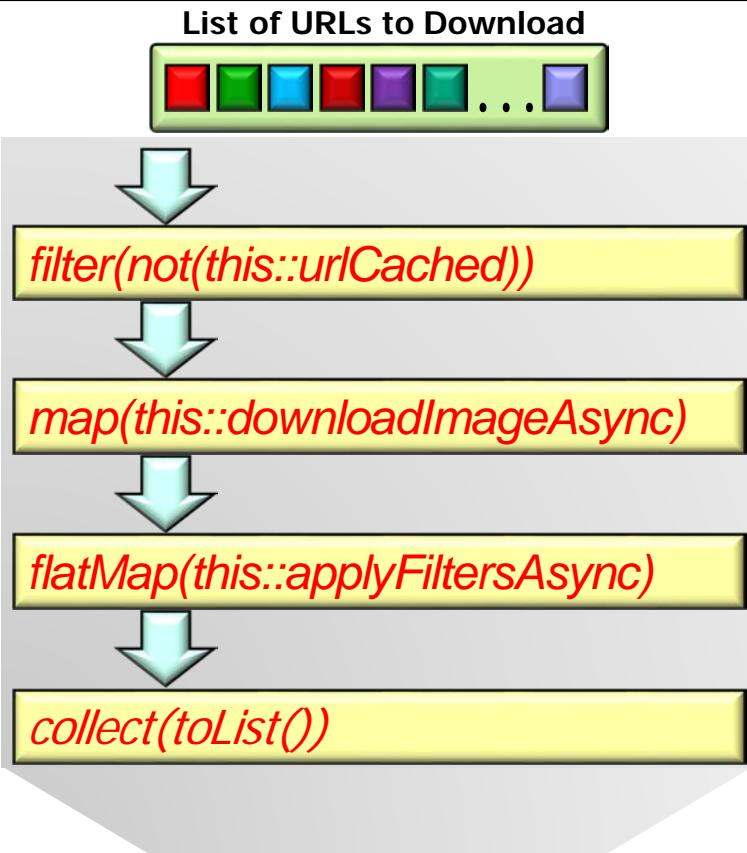
# Overview of Completable Futures ImageStreamGang

- ImageStreamGang shows advanced completable future features, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user



# Overview of Completable Futures ImageStreamGang

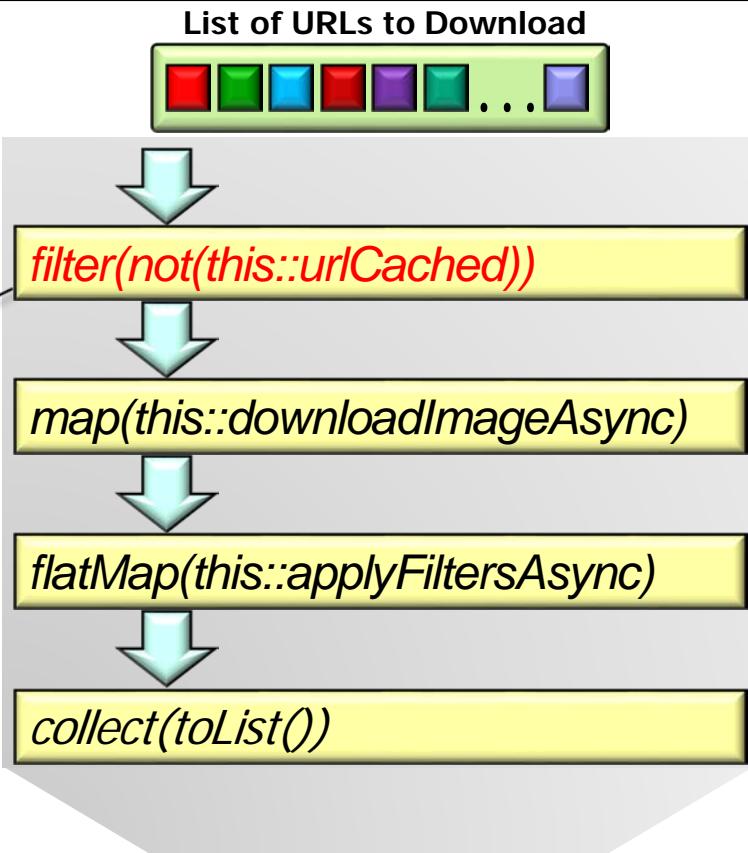
- The behaviors in this pipeline differ from the parallel streams variant



# Overview of Completable Futures ImageStreamGang

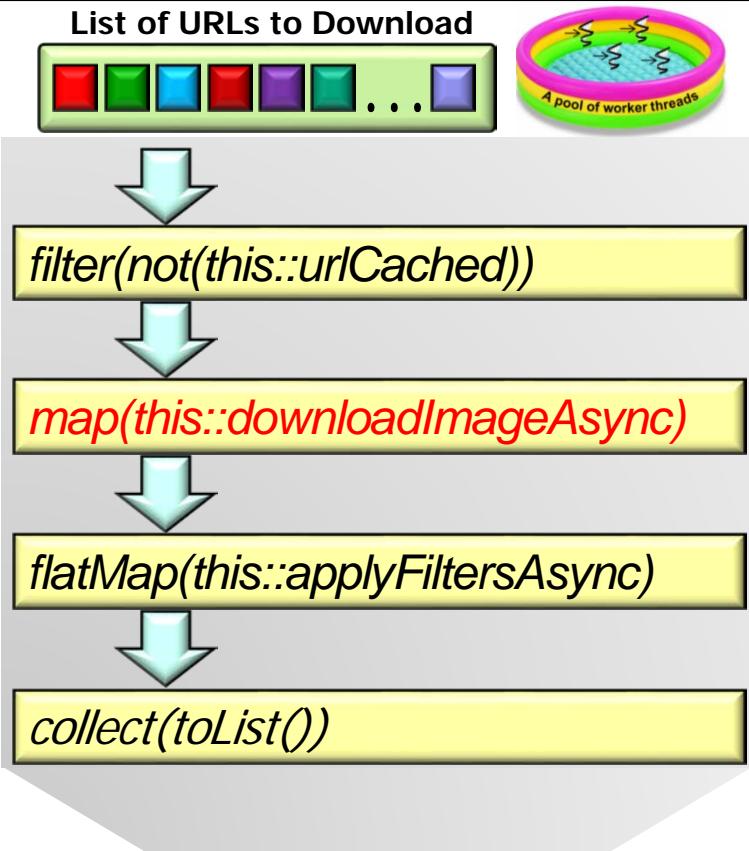
- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images

*Same as the parallel streams variant*



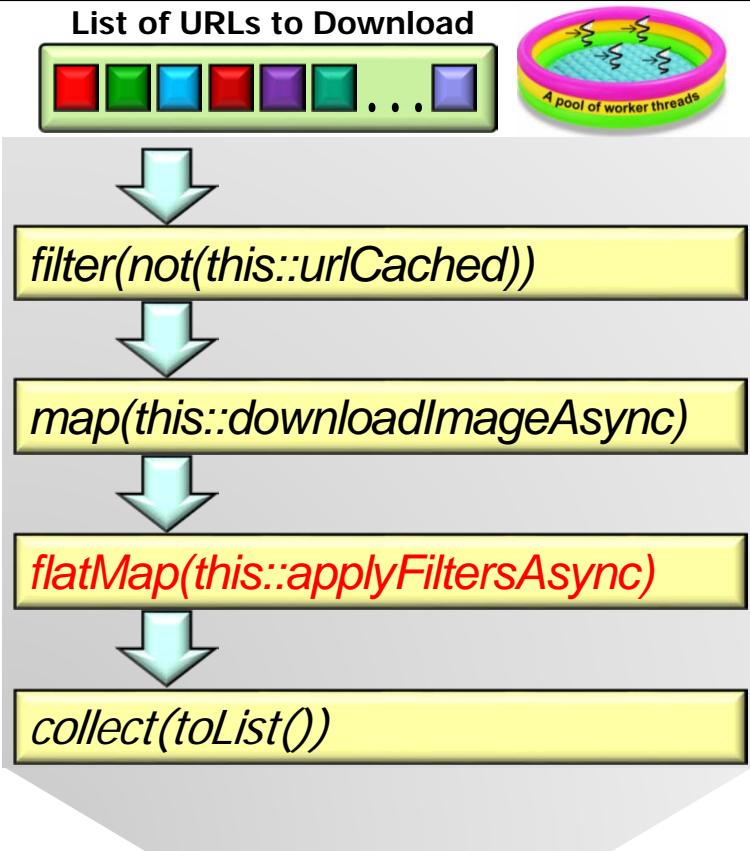
# Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images
  - Download non-cached images asynchronously



# Overview of Completable Futures ImageStreamGang

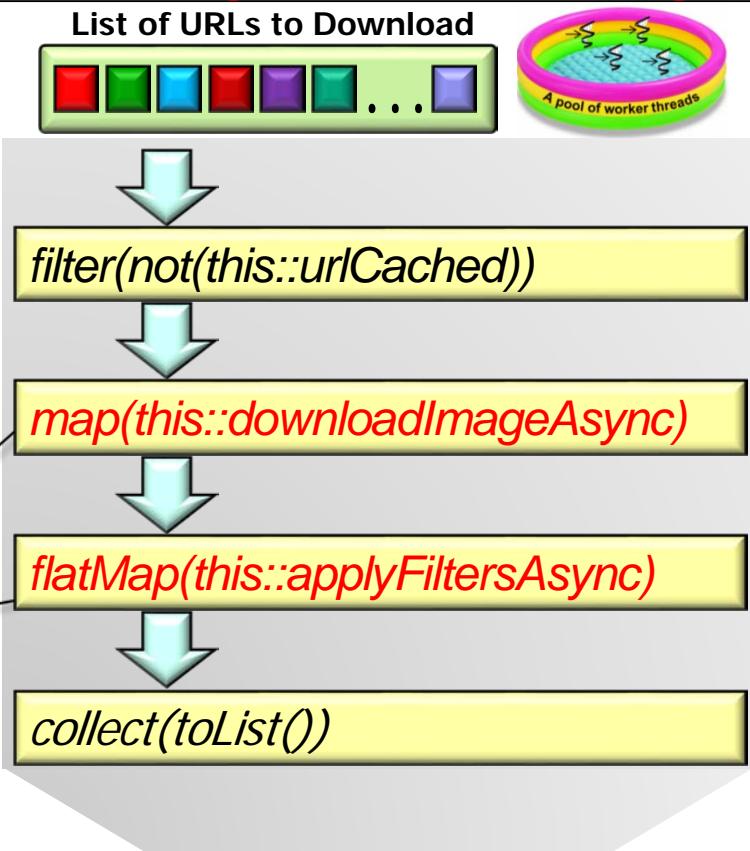
- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images
  - Download non-cached images asynchronously
  - As downloads complete asynchronously apply a list of filters & store filtered images in file system



# Overview of Completable Futures ImageStreamGang

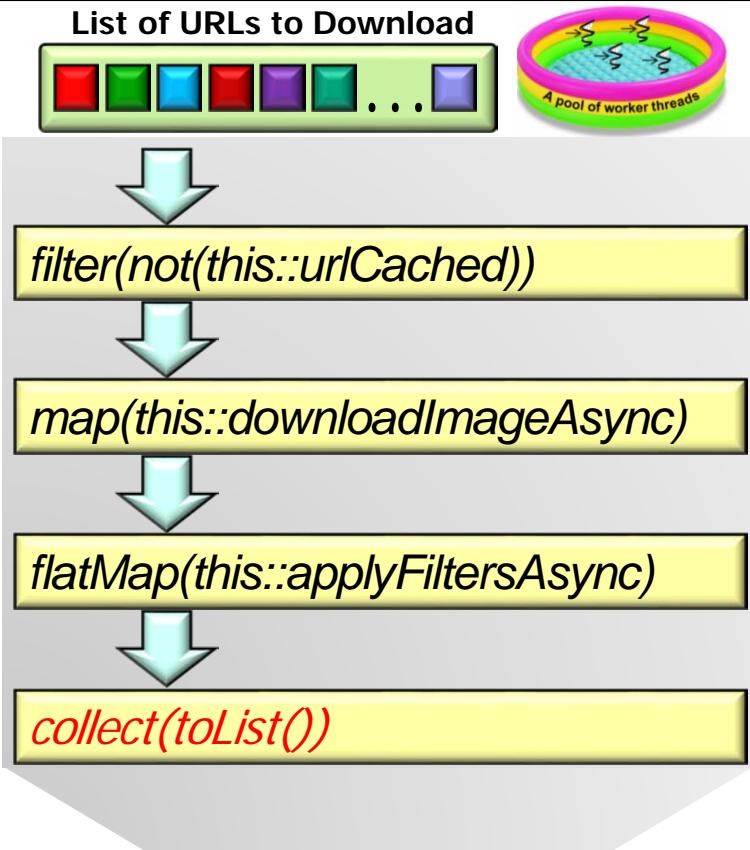
- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images
  - Download non-cached images asynchronously
  - As downloads complete asynchronously apply a list of filters & store filtered images in file system

*Different from the parallel streams variant*



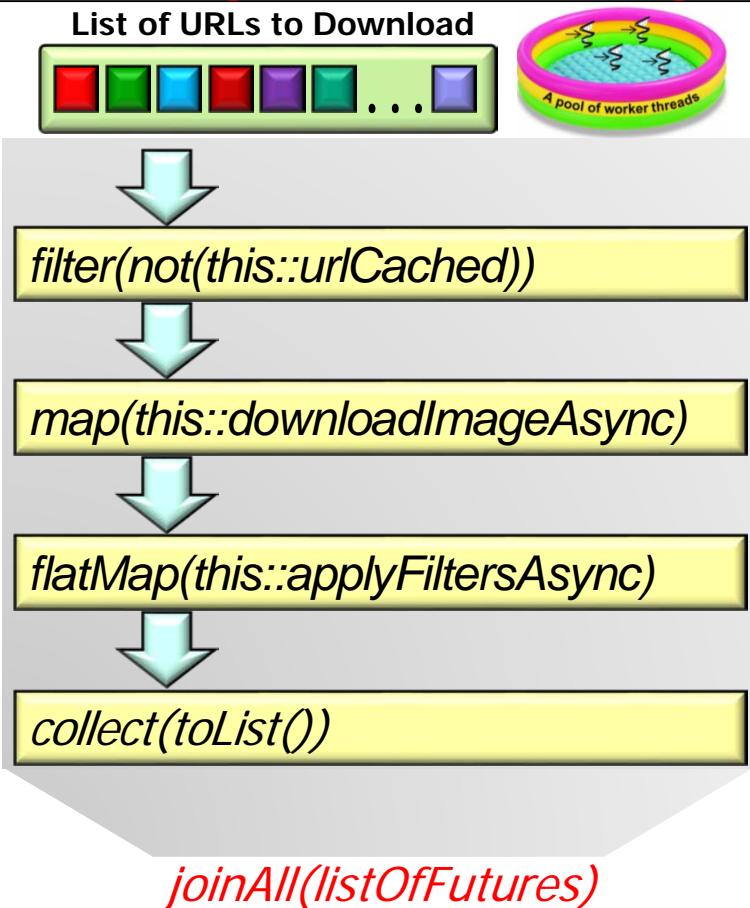
# Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images
  - Download non-cached images asynchronously
  - As downloads complete asynchronously apply a list of filters & store filtered images in file system
  - Trigger stream processing



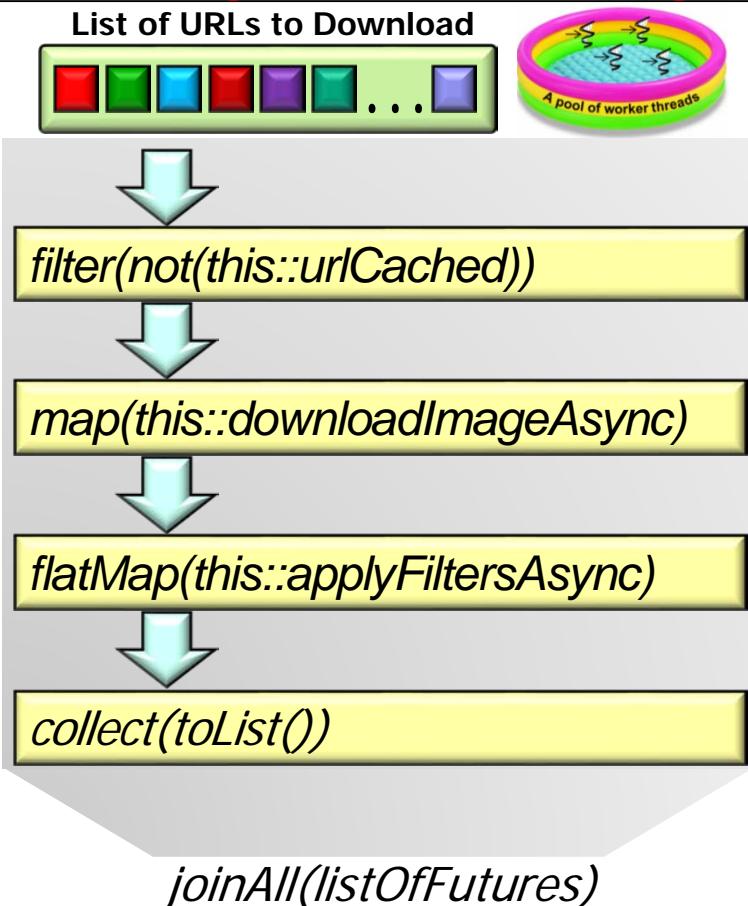
# Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images
  - Download non-cached images asynchronously
  - As downloads complete asynchronously apply a list of filters & store filtered images in file system
  - Trigger stream processing
  - Get results of asynchronous functions



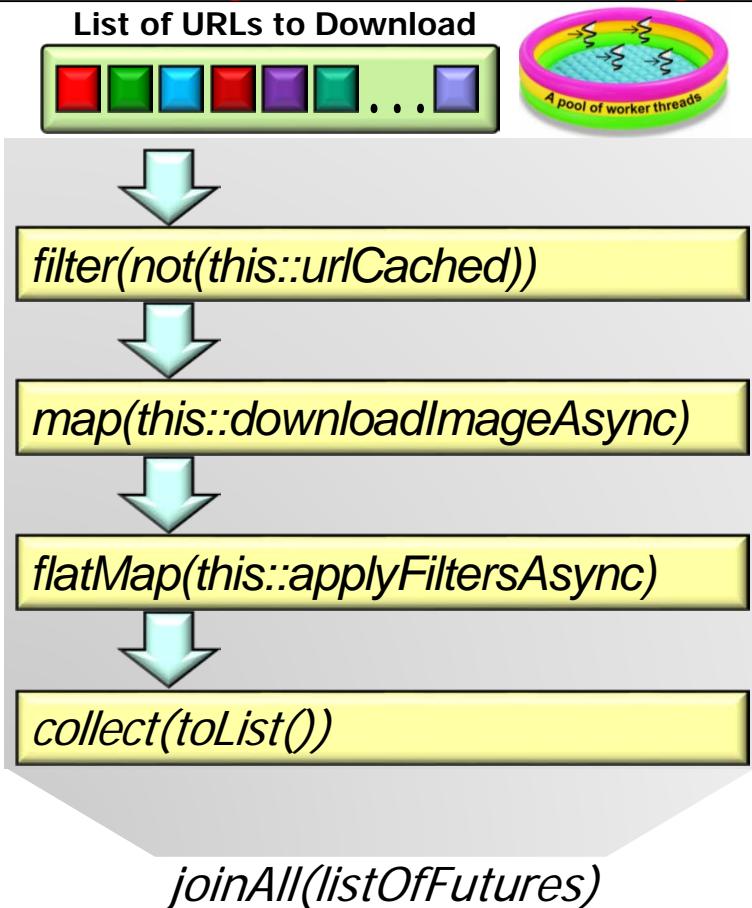
# Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the parallel streams variant, e.g.,
  - Ignore cached images
  - Download non-cached images asynchronously
  - As downloads complete asynchronously apply a list of filters & store filtered images in file system
  - Trigger stream processing
  - Get results of asynchronous functions
    - Ultimately display images to user



# Overview of Completable Futures ImageStreamGang

- The behaviors in this pipeline differ from the parallel streams variant



Combining completable futures & streams closes gap between design & implementation

---

# End of Java 8 CompletableFuture Futures ImageStreamGang Example (Part 1)

# Java 8 CompletableFuture

## ImageStreamGang Example (Part 2)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA

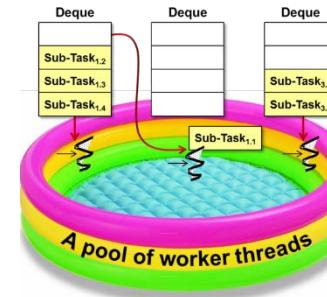
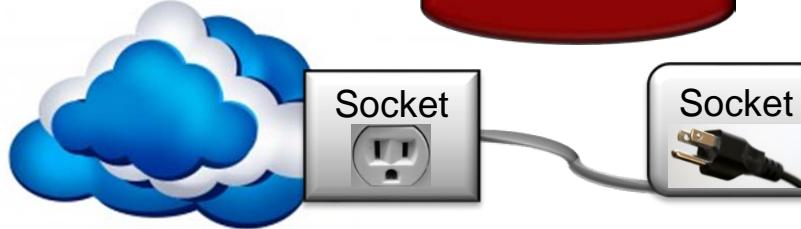


# Learning Objectives in this Part of the Lesson

- Understand the design of the Java 8 completable future version of ImageStreamGang
- Know how completable futures are applied to ImageStreamGang



List of Filters to Apply



# Learning Objectives in this Part of the Lesson

- Understand the design of the Java 8 completable future version of ImageStreamGang
- Know how completable futures are applied to ImageStreamGang
  - Factory methods & completion stage methods

<<Java Class>>

**CompletableFuture<T>**

<code>CompletableFuture()</code>
<code>cancel(boolean):boolean</code>
<code>isCancelled():boolean</code>
<code>isDone():boolean</code>
<code>get()</code>
<code>get(long,TimeUnit)</code>
<code>join()</code>
<code>complete(T):boolean</code>
<code>supplyAsync(Supplier&lt;U&gt;):CompletableFuture&lt;U&gt;</code>
<code>supplyAsync(Supplier&lt;U&gt;,Executor):CompletableFuture&lt;U&gt;</code>
<code>runAsync(Runnable):CompletableFuture&lt;Void&gt;</code>
<code>runAsync(Runnable,Executor):CompletableFuture&lt;Void&gt;</code>
<code>completedFuture(U):CompletableFuture&lt;U&gt;</code>
<code>thenApply(Function&lt;?&gt;):CompletableFuture&lt;U&gt;</code>
<code>thenAccept(Consumer&lt;? super T&gt;):CompletableFuture&lt;Void&gt;</code>
<code>thenCombine(CompletionStage&lt;? extends U&gt;,BiFunction&lt;?&gt;):CompletableFuture&lt;V&gt;</code>
<code>thenCompose(Function&lt;?&gt;):CompletableFuture&lt;U&gt;</code>
<code>whenComplete(BiConsumer&lt;?&gt;):CompletableFuture&lt;T&gt;</code>
<code>allOf(CompletableFuture[]&lt;?&gt;):CompletableFuture&lt;Void&gt;</code>
<code>anyOf(CompletableFuture[]&lt;?&gt;):CompletableFuture&lt;Object&gt;</code>

---

# Applying Completable Futures to ImageStreamGang

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

See [imagestreamgangstreamsImageStreamCompletableFuture1.java](#)

# Applying Completable Futures to ImageStreamGang

---

- Focus on processStream()

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

*Factory method  
creates a stream*

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This intermediate operation is identical to parallel streams

*Ignore images that are already in the cache*

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

# Applying Completable Futures to ImageStreamGang

- Focus on `processStream()`
  - This intermediate operation is identical to parallel streams

```
boolean urlCached(URL url) {  
    return mFilters  
        .stream()  
        .filter(filter ->  
            urlCached(url,  
                      filter.getName()))  
        .count() > 0;  
}
```

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
    ...  
}
```

See [imagestreamgang/streams/ImageStreamGang.java](#)

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This intermediate operation is identical to parallel streams

```
boolean urlCached(URL url,
                  String filterName) {
    File file =
        new File(getPath(),
                  filterName);

    File imageFile =
        new File(file,
                  getNameForUrl(url));

    return imageFile.exists();
}
```

```
void processStream() {
    List<CompletableFuture<List<Image>>>
    listOfFutures = getInput()
        .stream()
        .filter(not(this::urlCached))
        .map(this::downloadImageAsync)
        .flatMap(this::applyFiltersAsync)
        .collect(toList());
    ...
}
```

There are clearly more sophisticated ways of implementing an image cache!

# Applying Completable Futures to ImageStreamGang

---

- Focus on `processStream()`
  - This intermediate operation is identical to parallel streams
  - Other intermediate operations differ since they input/output stream of completable futures

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOffutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

---

# Applying Factory Methods in ImageStreamGang

# Applying Factory Methods in ImageStreamGang

- Context: `downloadImageAsync()` asynchronously downloads each URL in the input stream
  - It returns a completable future for each downloaded image in the output stream

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

# Applying Factory Methods in ImageStreamGang

- `downloadImageAsync()` uses the `supplyAsync()` factory method internally

```
CompletableFuture<Image>
downloadImageAsync(URL url) {
    return CompletableFuture
        .supplyAsync(() ->
            downloadImage(url),
            getExecutor());
}
```

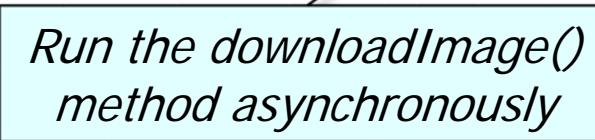
*Asynchronously  
download image  
at the given URL*

# Applying Factory Methods in ImageStreamGang

- `downloadImageAsync()` uses the `supplyAsync()` factory method internally

```
CompletableFuture<Image>
downloadImageAsync(URL url) {
    return CompletableFuture
        .supplyAsync(() ->
            downloadImage(url),
            getExecutor());
}
```

}



*Run the `downloadImage()` method asynchronously*

# Applying Factory Methods in ImageStreamGang

- `downloadImageAsync()` uses the `supplyAsync()` factory method internally

```
CompletableFuture<Image>
downloadImageAsync(URL url) {
    return CompletableFuture
        .supplyAsync(() ->
            downloadImage(url),
            getExecutor());
}
```

*Specify a fixed-size  
thread pool executor*

You could also simply use the default thread pool (common fork-join pool)

# Applying Factory Methods in ImageStreamGang

- `downloadImageAsync()` uses the `supplyAsync()` factory method internally

`CompletableFuture<Image>`

```
downloadImageAsync(URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            downloadImage(url),  
            getExecutor());  
}
```

*Returns a completable future to an image that triggers when image downloading is finished*

---

# Applying Completion Stage Methods in ImageStreamGang

# Applying Completion Stage Methods in ImageStreamGang

- Context: `applyFiltersAsync()` asynchronously filters & stores each image that's downloaded in the input stream
  - It returns a completable future for each filtered image in the output stream

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    listOfFutures = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Asynchronous filter images  
& store them into files*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                                getExecutor())));
}
```

See [imagestreamgangstreams/ImageStreamCompletableFuture1.java](#)

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*This completable future  
is what's returned from  
`downloadImageAsync()`*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture){
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Two completion  
stage methods*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Convert this list of filters into a stream*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                                getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Create a completable future to a FilterDecoratorWithImage object for each filter/image*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*`thenApply()` defines a computation  
that's not executed immediately,  
but is remembered & executed  
when `imFuture` completes*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
makeFilterDecoratorWithImage
                (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Returns a new completion stage that's executed with this stage's result as the argument to the supplied lambda expression*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                                getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Asynchronously filter  
the image & store it  
in an output file*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*`thenCompose()` can be used to combine two completable future together without blocking or waiting for intermediate results*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
filterFuture.thenCompose
            (filter -> CompletableFuture
                .supplyAsync(filter::run,
                    getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*It also returns a new completion stage that is executed with this stage's result as the argument to the supplied lambda expression*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the completion stage methods `thenApply()` & `thenCompose()` internally

*Returns a stream of completable futures to filtered/store images*

```
Stream<CompletableFuture<Image>>
applyFiltersAsync
(CompletableFuture<Image> imFuture) {
    return mFilters.stream()
        .map(filter -> imFuture.thenApply
            (image ->
                makeFilterDecoratorWithImage
                    (filter, image)))
        .map(filterFuture ->
            filterFuture.thenCompose
                (filter -> CompletableFuture
                    .supplyAsync(filter::run,
                        getExecutor())));
}
```

The `flatMap()` operation processes this stream return value, as we'll show shortly

---

# End of Java 8 CompletableFuture Future ImageStreamGang Example (Part 2)

# Java 8 CompletableFuture

## ImageStreamGang Example (Part 3)

Douglas C. Schmidt

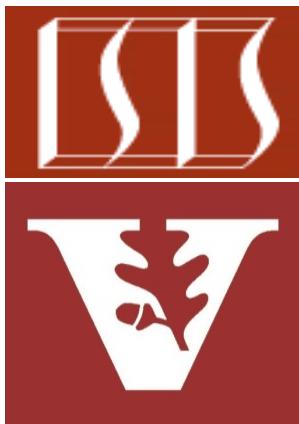
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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 the design of the Java 8 completable future version of ImageStreamGang
- Know how completable futures are applied to ImageStreamGang
  - Factory methods & completion stage methods
  - Arbitrary-arity methods

<<Java Class>>

**CompletableFuture<T>**

<code>CompletableFuture()</code>
<code>cancel(boolean):boolean</code>
<code>isCancelled():boolean</code>
<code>isDone():boolean</code>
<code>get()</code>
<code>get(long,TimeUnit)</code>
<code>join()</code>
<code>complete(T):boolean</code>
<code>supplyAsync(Supplier&lt;U&gt;):CompletableFuture&lt;U&gt;</code>
<code>supplyAsync(Supplier&lt;U&gt;,Executor):CompletableFuture&lt;U&gt;</code>
<code>runAsync(Runnable):CompletableFuture&lt;Void&gt;</code>
<code>runAsync(Runnable,Executor):CompletableFuture&lt;Void&gt;</code>
<code>completedFuture(U):CompletableFuture&lt;U&gt;</code>
<code>thenApply(Function&lt;?&gt;):CompletableFuture&lt;U&gt;</code>
<code>thenAccept(Consumer&lt;? super T&gt;):CompletableFuture&lt;Void&gt;</code>
<code>thenCombine(CompletionStage&lt;? extends U&gt;,BiFunction&lt;?&gt;):CompletableFuture&lt;V&gt;</code>
<code>thenCompose(Function&lt;?&gt;):CompletableFuture&lt;U&gt;</code>
<code>whenComplete(BiConsumer&lt;?&gt;):CompletableFuture&lt;T&gt;</code>
<code>allOf(CompletableFuture[]&lt;?&gt;):CompletableFuture&lt;Void&gt;</code>
<code>anyOf(CompletableFuture[]&lt;?&gt;):CompletableFuture&lt;Object&gt;</code>

---

# Applying Arbitrary-Arity Methods in ImageStreamGang

# Applying Arbitrary-Arity Methods in ImageStreamGang

---

- Context: collect() creates a list of futures to images being filtered & stored

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    futureList = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

---

- Context: collect() creates a list of futures to images being filtered & stored
  - These images must be displayed after their processing completes

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    futureList = getInput()  
        .stream()  
        .filter(not(this::urlCached))  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toList());  
  
    ...  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

- Context: collect() creates a list of futures to images being filtered & stored

- These images must be displayed after their processing completes
- joinAll() uses the “arbitrary-arity” method allOf()

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    futureList = ...  
  
    CompletableFuture<List  
        <List<Image>>> allImagesDone =  
    StreamsUtils.joinAll(futureList);  
  
    int imagesProcessed = allImagesDone  
        .join()  
        .stream()  
        .collect(summingInt(List::size));
```

*Return a completable future that's used to know when all asynchronous functions have completed*

# Applying Arbitrary-Arity Methods in ImageStreamGang

- Context: collect() creates a list of futures to images being filtered & stored
  - These images must be displayed after their processing completes
  - joinAll() uses the “arbitrary-arity” method allOf()

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    futureList = ...  
  
    CompletableFuture<List  
        <List<Image>>> allImagesDone =  
    StreamsUtils.joinAll(futureList);  
  
    int imagesProcessed = allImagesDone  
        .join()  
        .stream()  
        .collect(summingInt(List::size));
```

*Wait for all the futures to complete*

# Applying Arbitrary-Arity Methods in ImageStreamGang

- Context: collect() creates a list of futures to images being filtered & stored
  - These images must be displayed after their processing completes
  - joinAll() uses the “arbitrary-arity” method allOf()

```
void processStream() {  
    List<CompletableFuture<List<Image>>>  
    futureList = ...  
  
    CompletableFuture<List  
        <List<Image>>> allImagesDone =  
    StreamsUtils.joinAll(futureList);  
  
    int imagesProcessed = allImagesDone  
        .join()  
        .stream()  
        .collect(summingInt(List::size));
```

*Sum up the total count of images after they are downloaded, filtered, & stored asynchronously*

---

# Implementing StreamUtils.joinAll() in ImageStreamGang

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

See [AndroidGUI/app/src/main/java/livelessons/utils/StreamUtils.java](#)

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

*The return value converts a list of completed futures into a list of joined results*

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
       fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

*The parameter is a list of completable futures to some generic type T*

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

*Returns a completable future  
that completes when all  
completable futures complete*

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

*Create an array that stores the list of completable futures*

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

*Creates a completable future to a list of joined results when all completable futures complete*

# Implementing StreamUtils.joinAll() in ImageStreamGang

- The StreamUtils.joinAll() method is a wrapper that encapsulates allOf()

```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf
        (fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

*Return a completable future  
to the list of joined results*

# Implementing StreamUtils.joinAll() in ImageStreamGang

- joinAll() provides a very powerful wrapper for some complex code!!!



```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf(
        fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

# Implementing StreamUtils.joinAll() in ImageStreamGang

- joinAll() provides a very powerful wrapper for some complex code!!!



```
static <T> CompletableFuture<List<T>>
joinAll(List<CompletableFuture<T>>
        fList) {
    CompletableFuture<Void>
    dFuture = CompletableFuture.allOf(
        fList.toArray(new
            CompletableFuture[fList.size()]));

    CompletableFuture<List<T>> dList =
        dFuture.thenApply(v -> fList
            .stream()
            .map(CompletableFuture::join)
            .collect(toList()));
    return dList;
}
```

Also see [www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html](http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html)

---

# End of Java 8 CompletableFuture Future ImageStreamGang Example (Part 3)

# Pros & Cons of Java 8 CompletableFuture

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://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 the pros & cons of using the completable futures framework

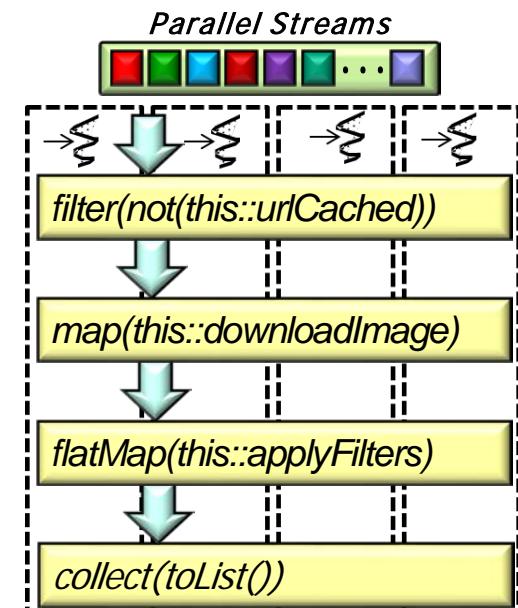
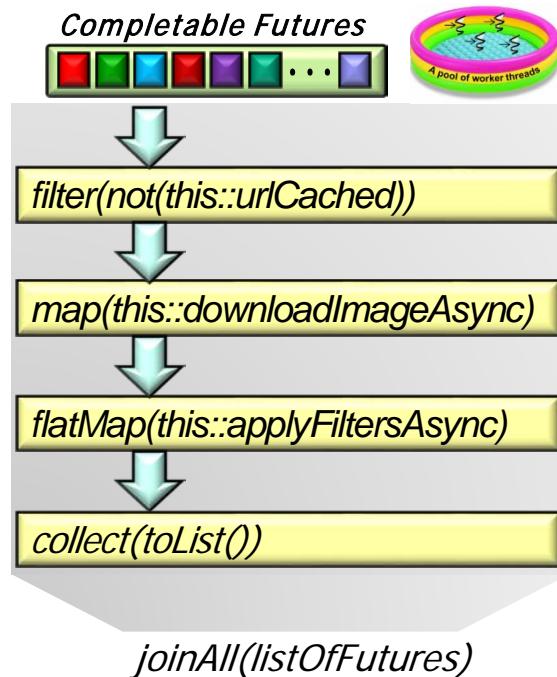


---

# Pros & Cons of Java 8 CompletableFuture

# Pros & Cons of Java 8 Completable Futures

- We'll evaluate the Java 8 completable futures framework compared with the parallel streams framework



# Pros & Cons of Java 8 Completable Futures

Completable Futures



```
filter(not(this::urlCached))
```

```
map(this::downloadImageAsync)
```

```
flatMap(this::applyFiltersAsync)
```

```
collect(toList())
```

```
joinAll(listOfFutures)
```



No explicit synchronization is required in this implementation

# Pros & Cons of Java 8 Completable Futures

## Completable Futures



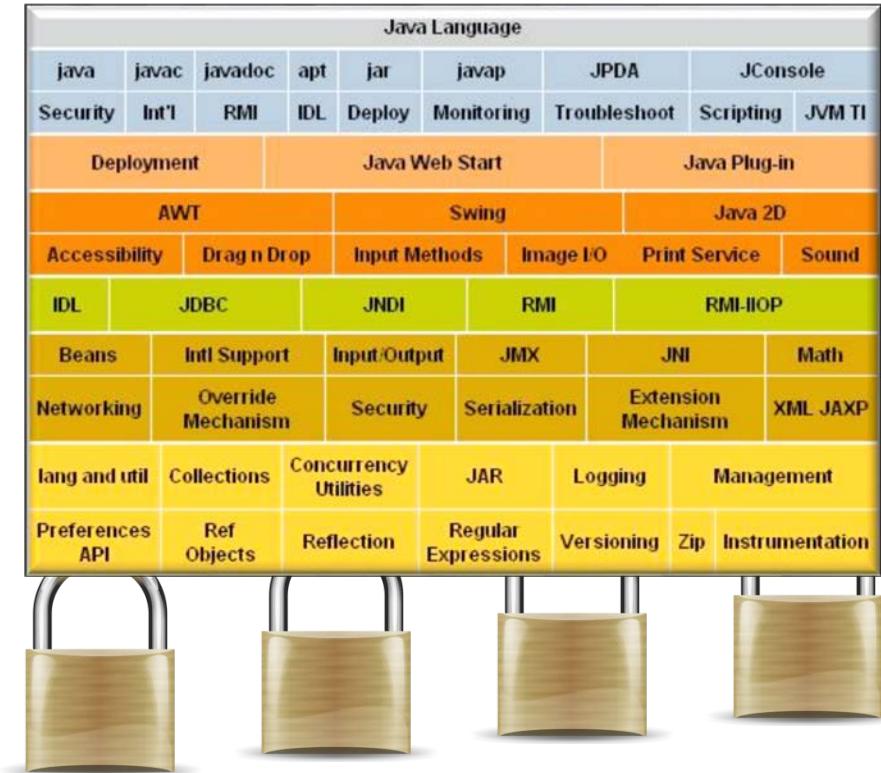
```
filter(not(this::urlCached))
```

```
map(this::downloadImageAsync)
```

```
flatMap(this::applyFiltersAsync)
```

```
collect(toList())
```

```
joinAll(listOfFutures)
```



Java libraries handle any locking needed to read/write to files & connections

# Pros & Cons of Java 8 Completable Futures

---

- Java 8 completable futures framework is much more complex to program

```
List<CompletableFuture<List<Image>>>  
listOfFutures = getInput()  
    .stream()  
    .filter(not(this::urlCached))  
    .map(this::downloadImageAsync)  
    .flatMap(this::applyFiltersAsync)  
    .collect(toList());
```

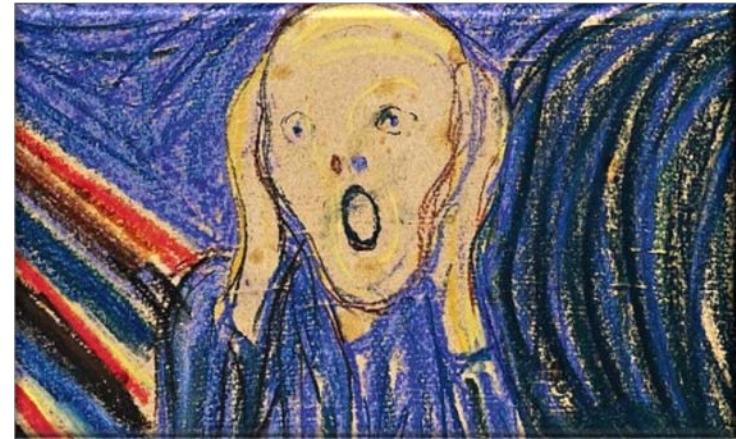


```
CompletableFuture<List<List<Image>>>  
allImagesDone = StreamsUtils.joinAll(listOfFutures);  
  
int imagesProcessed = allImagesDone.join().stream()  
    .collect(summingInt(List::size));
```

# Pros & Cons of Java 8 Completable Futures

- Java 8 completable futures framework is much more complex to program

```
List<CompletableFuture<List<Image>>>  
listOfFutures = getInput()  
    .stream()  
    .filter(not(this::urlCached))  
    .map(this::downloadImageAsync)  
    .flatMap(this::applyFiltersAsync)  
    .collect(toList());
```

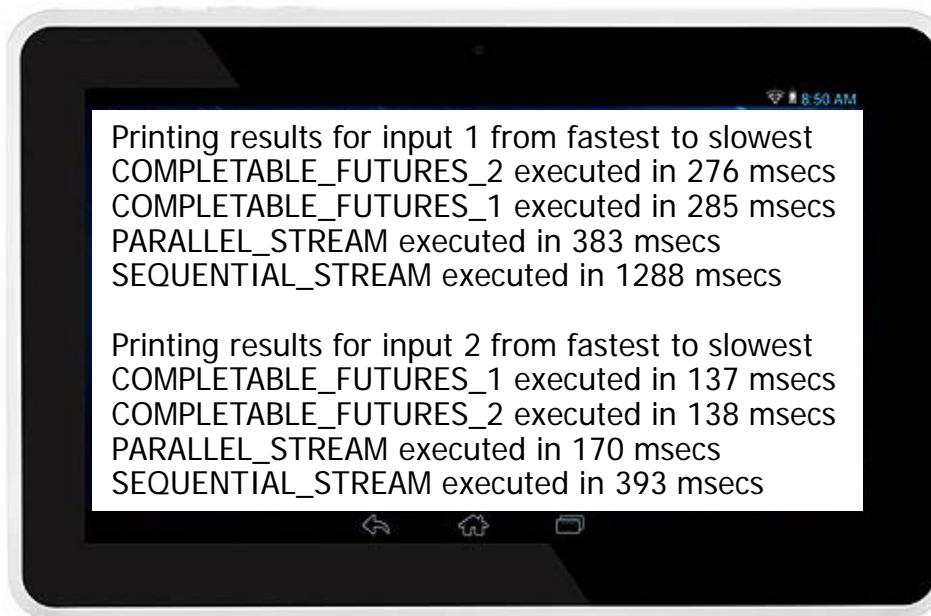


```
CompletableFuture<List<List<Image>>>  
allImagesDone = StreamsUtils.joinAll(listOfFutures);  
  
int imagesProcessed = allImagesDone.join().stream()  
    .collect(summingInt(List::size));
```

In general, asynchrony patterns aren't as well understood by many developers

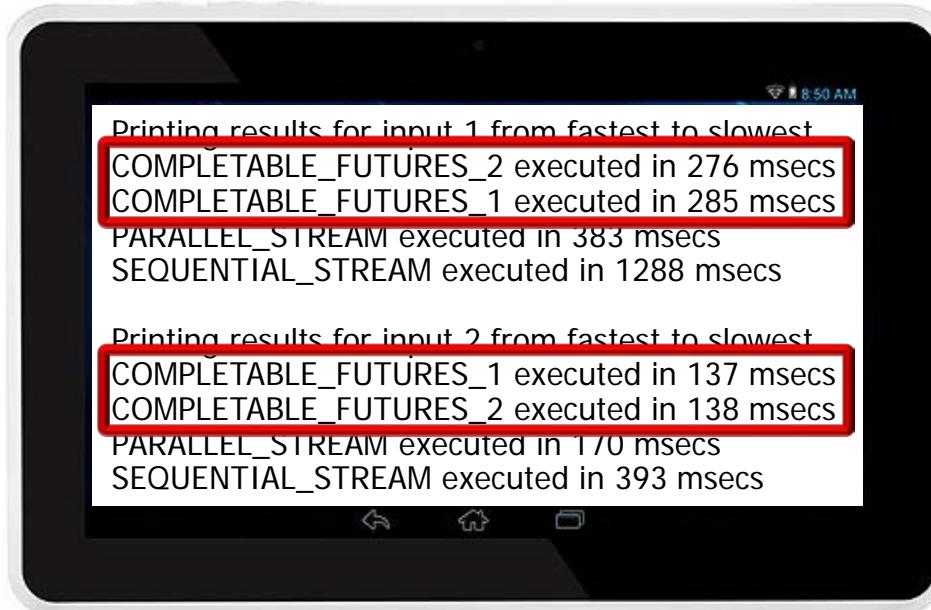
# Pros & Cons of Java 8 Completable Futures

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks



# Pros & Cons of Java 8 Completable Futures

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program



# Pros & Cons of Java 8 Completable Futures

---

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program
  - Parallel streams are often easier to program, but are less efficient & scalable



# Pros & Cons of Java 8 Completable Futures

---

- There's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks, e.g.
  - Completable futures are more efficient & scalable, but are harder to program
  - Parallel streams are often easier to program, but are less efficient & scalable



# Pros & Cons of Java 8 Completable Futures

- Java 9 fixes some completable future limitations

```
CompletableFuture
```

```
.supplyAsync(  
    () -> findBestPrice("LDN - NYC"),  
    executorService)  
.thenCombine(CompletableFuture  
    .supplyAsync  
        (() -> queryExchangeRateFor("GBP")),  
        this::convert)  
.orTimeout(1, TimeUnit.SECONDS)  
.whenComplete((amount, error) -> {  
    if (error == null) { System.out.println("The price is: "  
        + amount + "GBP"); }  
    else { System.out.println("Sorry, no result"); } });
```



---

# End of Pros & Cons of Java 8 CompletableFuture