

# **Overview of Reactive Programming with Java 8 Completable Futures**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Lesson

---

- Know what topics we'll cover



## *CompletableFuture*

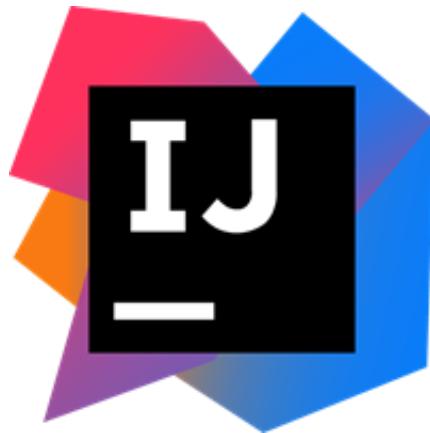
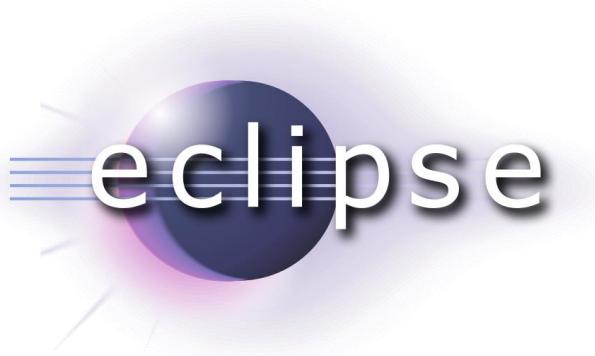


A pool of worker threads

# 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 LIKE I  
DO



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

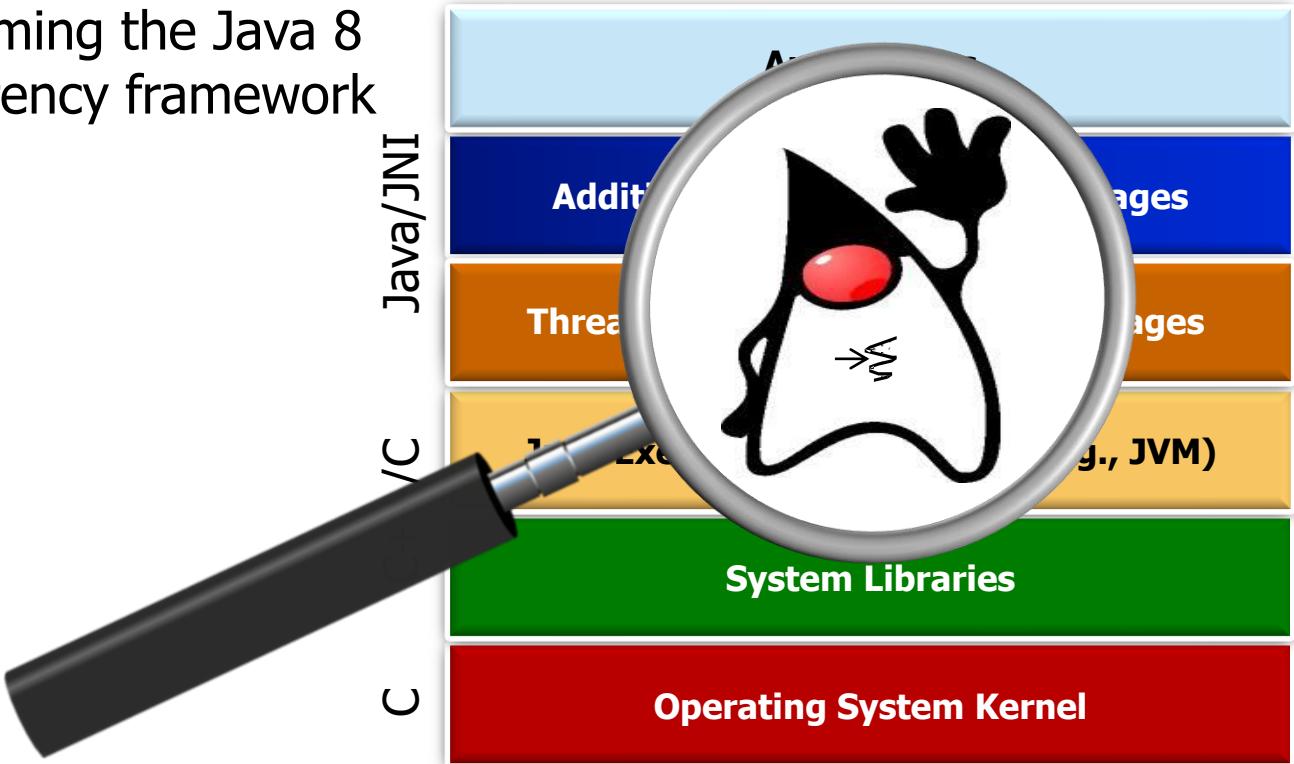
*We'll review many of  
these examples, so  
feel free to clone or  
download this repo!*

---

# Overview of this Course

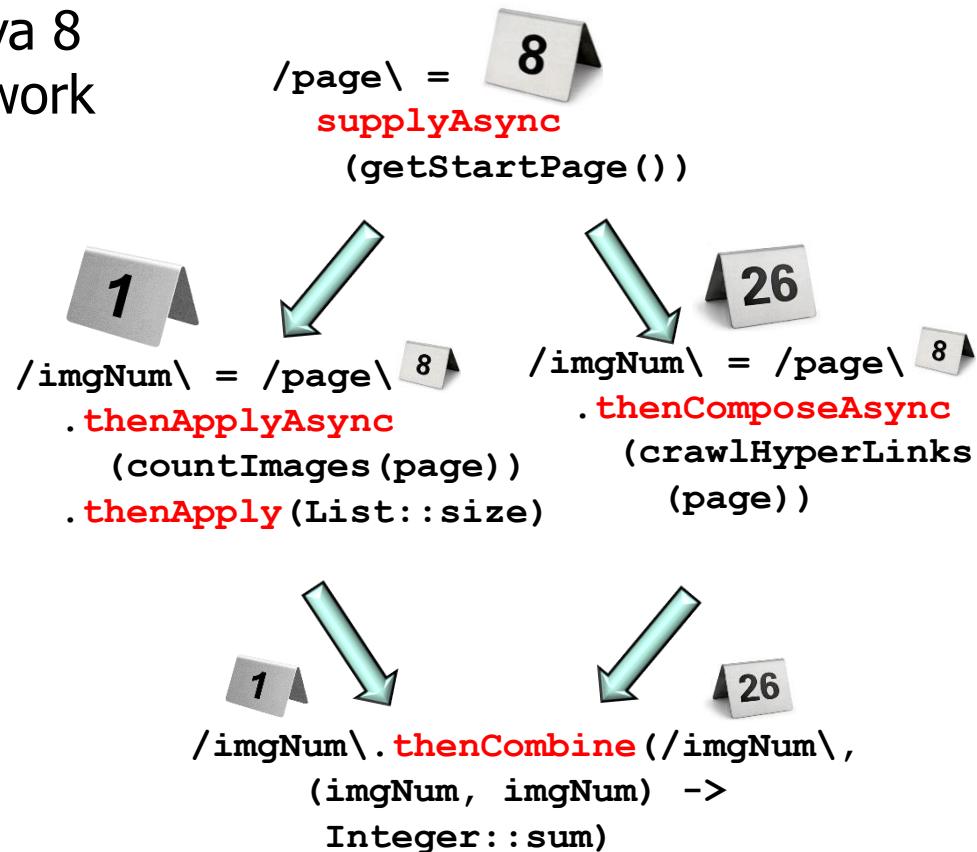
# Overview of this Course

- We focus on programming the Java 8 asynchronous concurrency framework



# Overview of this Course

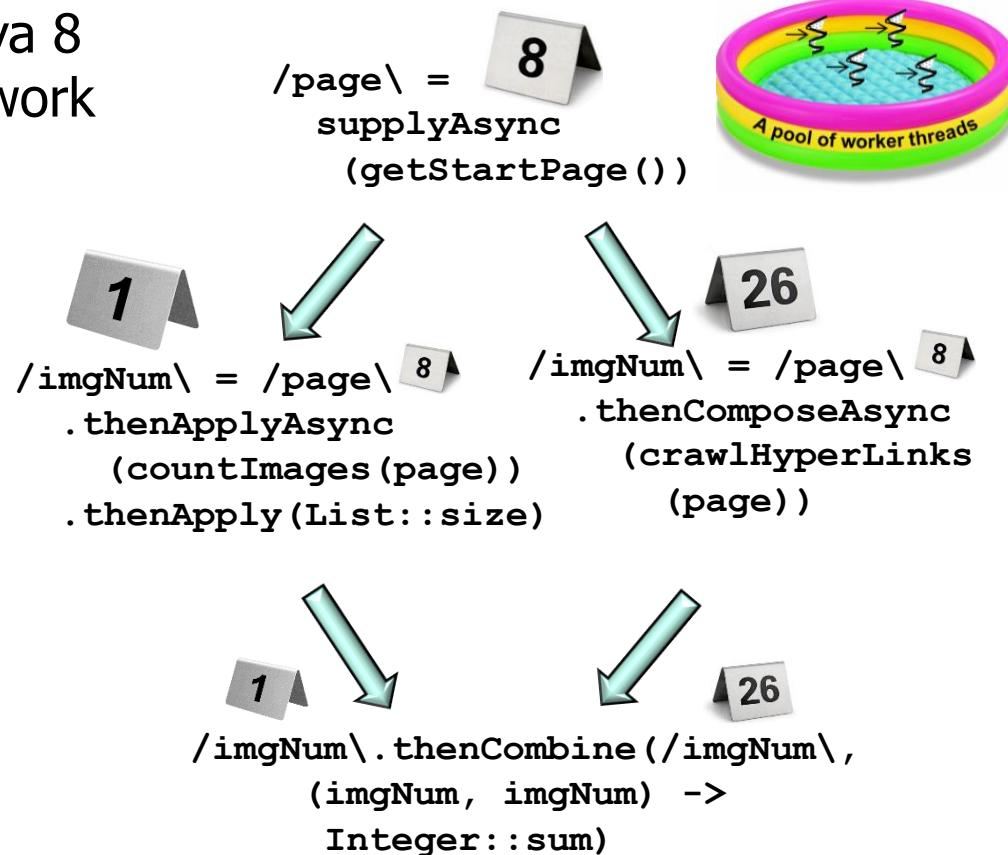
- We focus on programming the Java 8 asynchronous concurrency framework
  - **Competable futures**



# Overview of this Course

- We focus on programming the Java 8 asynchronous concurrency framework
  - **Competable futures**

- Support dependent actions that trigger upon completion of asynchronous operations
- Can be used 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

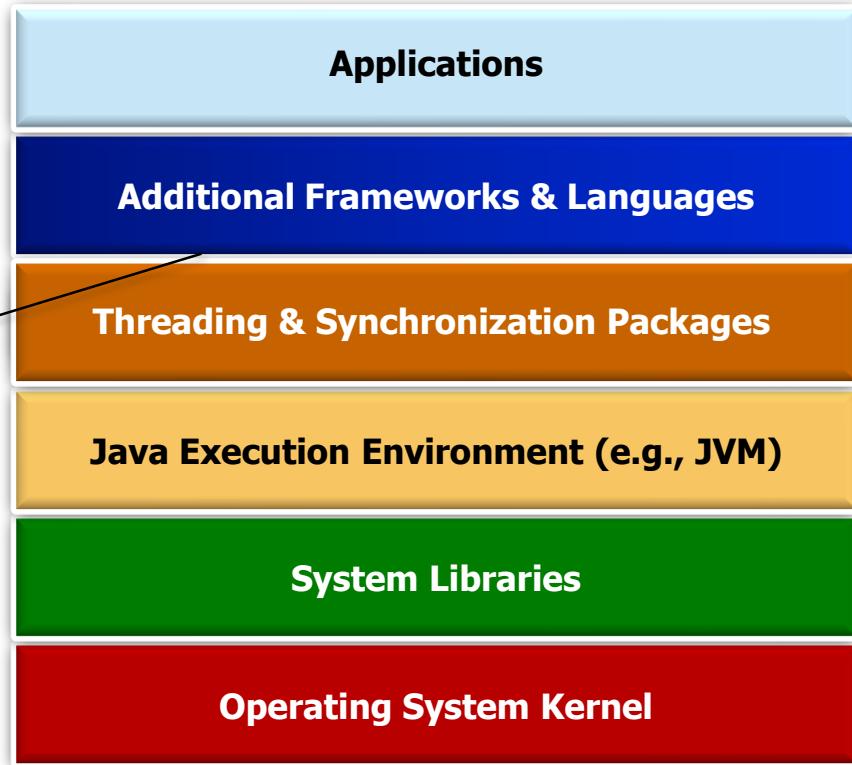


These features are the foundation for Java 8's concurrency/parallelism frameworks

# Overview of this Course

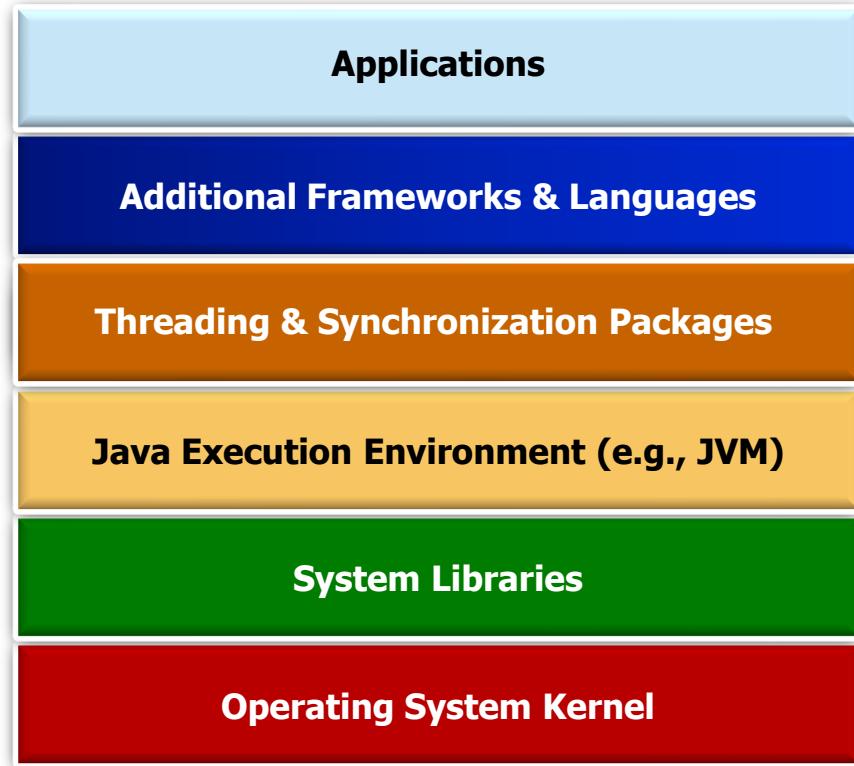
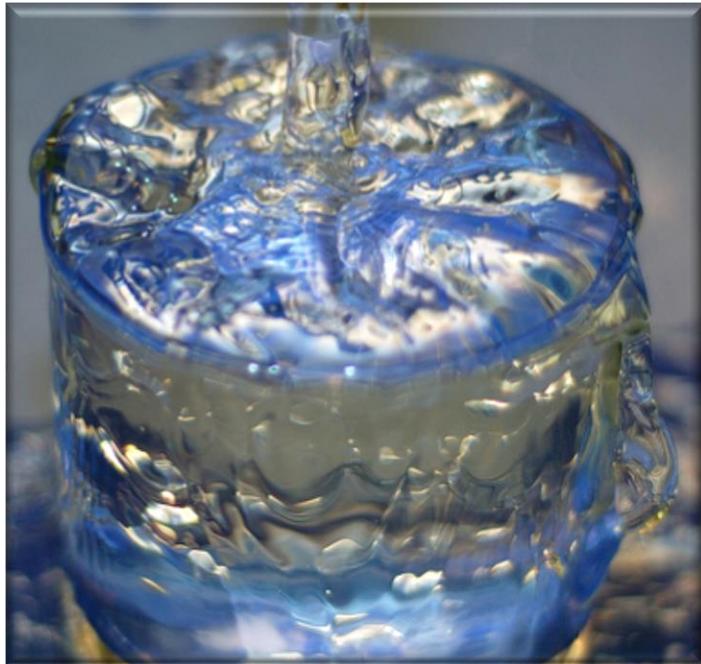
- There are also other Java-based & JVM-related frameworks & languages for concurrency

*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](#)

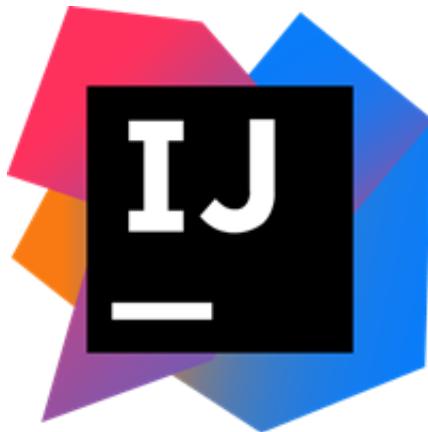
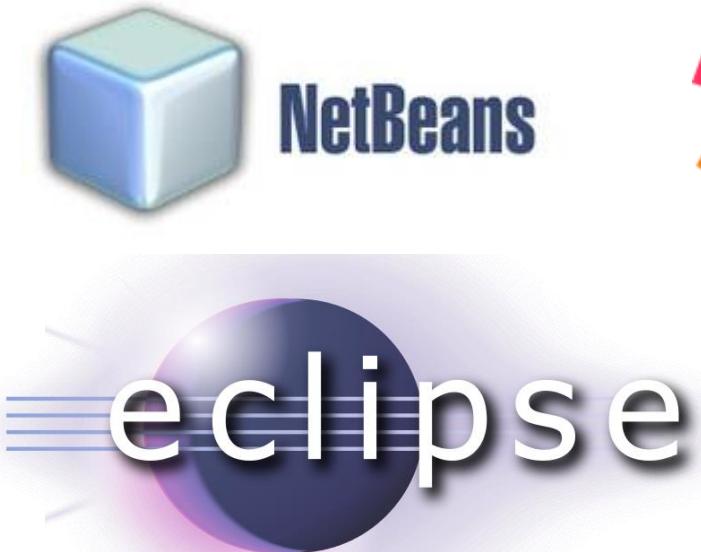
- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
  - JDK ReadMe
  - JRE ReadMe

JDK DOWNLOAD  
Server JRE DOWNLOAD  
JRE DOWNLOAD



# Accessing Java 8 Features & Functionality

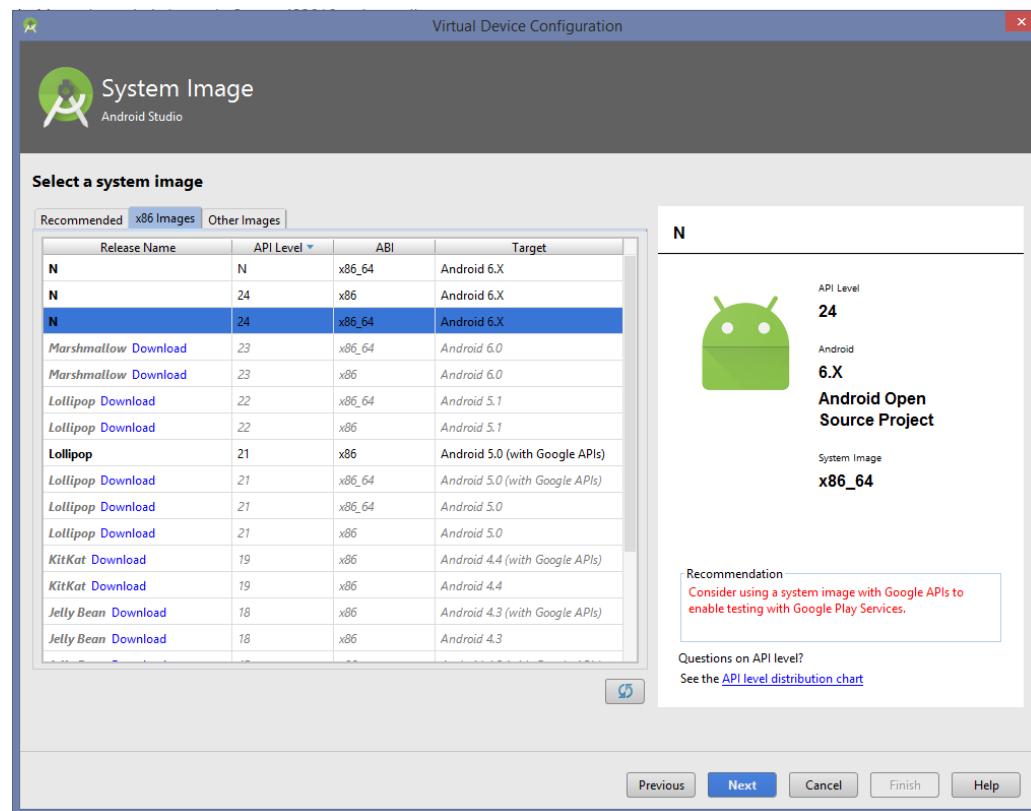
- The Java 8 (& beyond) runtime environment (JRE) supports Java 8 features
  - IntelliJ & Eclipse are popular Java 8 IDEs



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

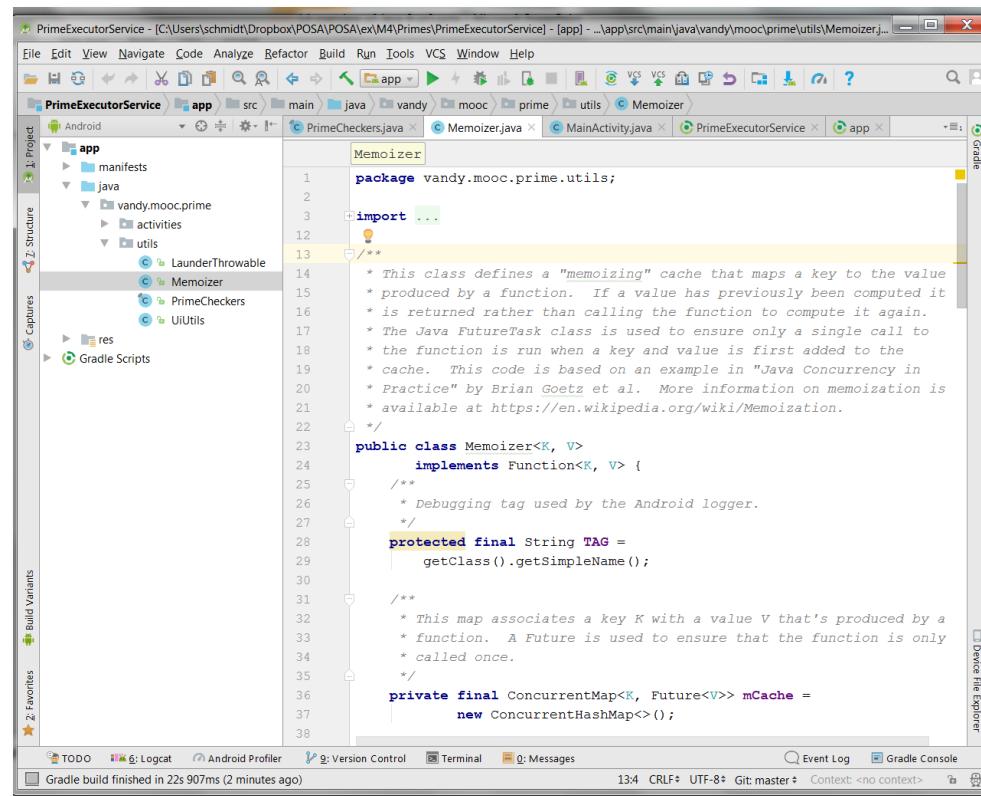
# Accessing Java 8 Features & Functionality

- 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)



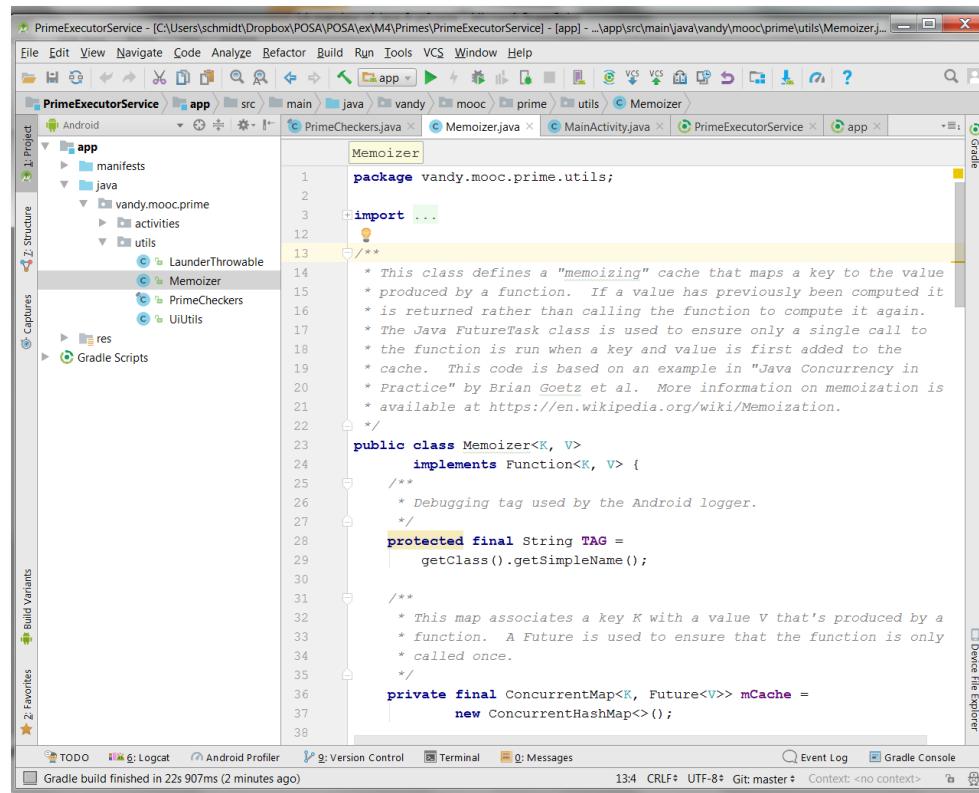
# Accessing Java 8 Features & Functionality

- 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/openjdk/8-b132/java](http://grepcode.com/java/root/jdk/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 following content:

- Java.net** logo and tagline "The Source for Java Technology Collaboration".
- Login | Register | Help** links.
- JDK 8** sidebar menu with links: Downloads, Feedback Forum, OpenJDK, and Planet JDK.
- JDK 8 Project** heading: "Building the next generation of the JDK platform".
- JDK 8 snapshot builds** section:
  - Download 8u40 early access snapshot builds
  - Source code (instructions)
  - Official Java SE 8 Reference Implementations
  - Early Access Build Test Results (instructions)
- We Want Contributions!** section: "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."
- Feedback** section: "Please use the [Project Feedback](#) forum if you have suggestions for or encounter issues using JDK 8." and "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."

---

# Other Digital Learning Resources

# Other Digital Learning Resources

- There are several other related Live Training courses

Programming with Java 8 Lambdas and Streams



DOUGLAS SCHMIDT

August 30<sup>th</sup>, 2018  
9:00am – 1:00pm CST

[SEE PRICING OPTIONS](#)

Scalable Programming with Java 8 Parallel Streams



DOUGLAS SCHMIDT

August 20<sup>th</sup>, 2018  
10:00am – 2:00pm CST

[SEE PRICING OPTIONS](#)

Design Patterns in Java



DOUGLAS SCHMIDT

September 18<sup>th</sup> & 19<sup>th</sup> 2018  
10:00am – 2:00pm CST

[SEE PRICING OPTIONS](#)

Scalable Concurrency with the Java Executor Framework



DOUGLAS SCHMIDT

October 29, 2018  
11:00am – 2:00pm CDT

[SEE PRICING OPTIONS](#)

194 spots available  
Registration closes October 28, 2018 5:00 PM

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

# Other Digital Learning Resources

- Examples not covered in these courses 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, it says "Introduction". In the center, there is a large play button icon. Below the play button, the title "Java Concurrency LiveLessons 2nd Edition" is written in large, bold, white font. Below the title, the author's name "Doug Schmidt" is also in white. At the bottom left, there is a Addison-Wesley logo. On the right side of the cover, there are three screenshots of video lessons. One shows a man in a red shirt holding a laptop, another shows a code editor with some Java code, and the third shows a terminal window with command-line output. On the far left, there is a sidebar with a blue header "douglasraigschmidt updates" and a list of links:

- BarrierTaskGang
- BuggyQueue
- BusySynchronizedQueue
- DeadlockQueue
- ExpressionTree
- Factorials
- ImageStreamGang
- ImageTaskGangApplication
- Java8
- PalantiriManagerApplication
- PingPongApplication
- PingPongWrong
- SearchStreamForkJoin

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

# Other Digital Learning Resources

- 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 with the Facebook logo, a search bar, and user navigation links for 'Douglas', 'Home', and other icons. The main content area has a large orange background image with the text 'Java Concurrency LiveLessons' in white. On the left, a sidebar lists group features like 'Concurrent Programming in Java', 'Public Group', 'Discussion', 'Members', 'Events', 'Photos', 'Group Insights', and 'Manage Group'. Below the sidebar is a search bar and a 'Write Post' button. The main feed area shows a post by a user named 'Douglas' with a profile picture of a man holding a laptop. The bottom right corner of the feed shows a '1,647 Members' count with several small profile pictures of group members.

See [www.facebook.com/groups/1532487797024074](http://www.facebook.com/groups/1532487797024074)

# Other Digital Learning Resources

- See my website for many more videos & screencasts related to programming with Java 8 & associated topics



## 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
  - [January 9th, 2018, 9:00am-12:00pm central time](#)
  - [February 1st, 2018, 9:00am-12:00pm central time](#)
  - March 1st, 2018, 9:00am-12:00pm central time
- Scalable Programming with Java 8 Parallel Streams
  - [January 10th, 2018, 11:00am-3:00pm central time](#)
  - February 6th, 2018, 11:00am-3:00pm central time
  - March 6th, 2018, 11:00am-3:00pm central time
- Reactive Programming with Java 8 Completable Futures
  - [January 12th, 2018, 10:00am-1:00pm central time](#)
  - [February 13th, 2018, 10:00am-2:00pm central time](#)
  - March 13th, 2018, 10:00am-2:00pm central time

### Pearson LiveLessons Courses

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

### Coursera MOOCs

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

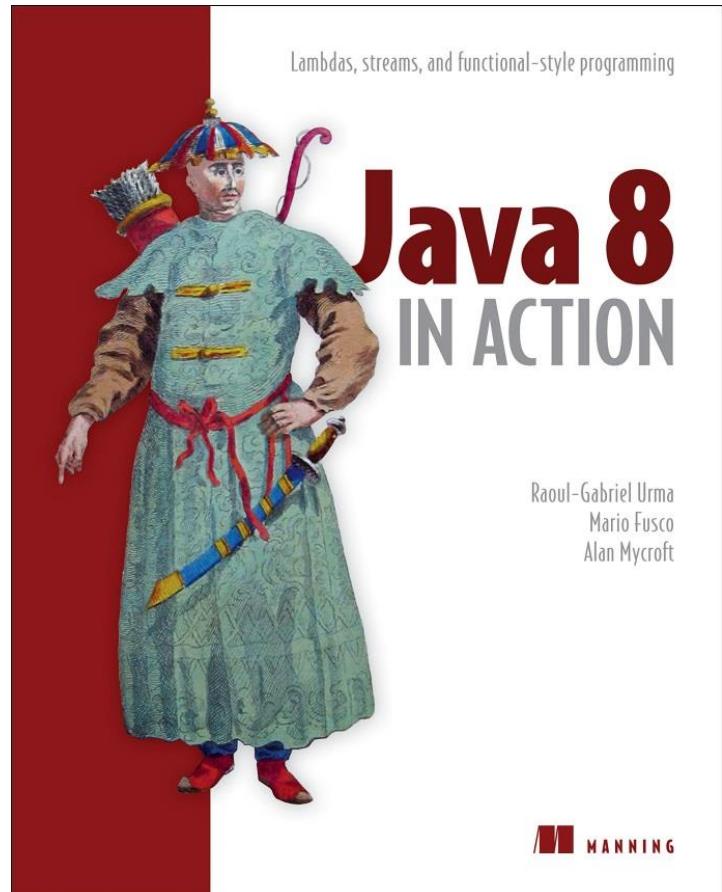
### Vanderbilt University Courses

- [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)

# Other Digital Learning Resources

- Another excellent source of material to consult is the book *Java 8 in Action*



See [www.manning.com/books/java-8-in-action](http://www.manning.com/books/java-8-in-action)

# Other Digital Learning Resources

- There are good online articles on the Java 8 completable future framework

## Java 8: Definitive guide to CompletableFuture

May 09, 2013

Java 8 is coming so it's time to study new features. While Java 7 and Java 6 were rather minor releases, version 8 will be a big step forward. Maybe even too big? Today I will give you a thorough explanation of new abstraction in JDK 8 - `CompletableFuture<T>`. As you all know Java 8 will hopefully be released in less than a year, therefore this article is based on JDK 8 build 88 with lambda support. `CompletableFuture<T>` extends `Future<T>` by providing functional, monadic (!) operations and promoting asynchronous, event-driven programming model, as opposed to blocking in older Java. If you opened [JavaDoc of `CompletableFuture<T>`](#) you are surely overwhelmed. About **fifty methods** (!), some of them being extremely cryptic and exotic, e.g.:

```
1  public <U,V> CompletableFuture<V> thenCombineAsync(      1
2      CompletableFuture<? extends U> other,           2
3      BiFunction<? super T,>? super U,>? extends V> fn,  3
4      Executor executor)                                4
```

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

# Other Digital Learning Resources

- There are good online articles on the Java 8 completable future framework

## Java 8: CompletableFuture in action

May 12, 2013

After thoroughly exploring `CompletableFuture` API in Java 8 we are prepared to write a simplistic web crawler. We solved similar problem already using `ExecutorCompletionService`, `Guava ListenableFuture` and `Scala/Akka`. I choose the same problem so that it's easy to compare approaches and implementation techniques.

First we shall define a simple, blocking method to download the contents of a single URL:

```
private String downloadSite(final String site) {
    try {
        log.debug("Downloading {}", site);
        final String res = IOUtils.toString(new URL("http://" + site), UTF_
        log.debug("Done {}", site);
        return res;
    } catch (IOException e) {
        throw Throwables.propagate(e);
    }
}
```

See [www.nurkiewicz.com/2013/05/java-8-completablefuture-in-action.html](http://www.nurkiewicz.com/2013/05/java-8-completablefuture-in-action.html)

---

# End of Course Overview

# Reactive Programming & 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

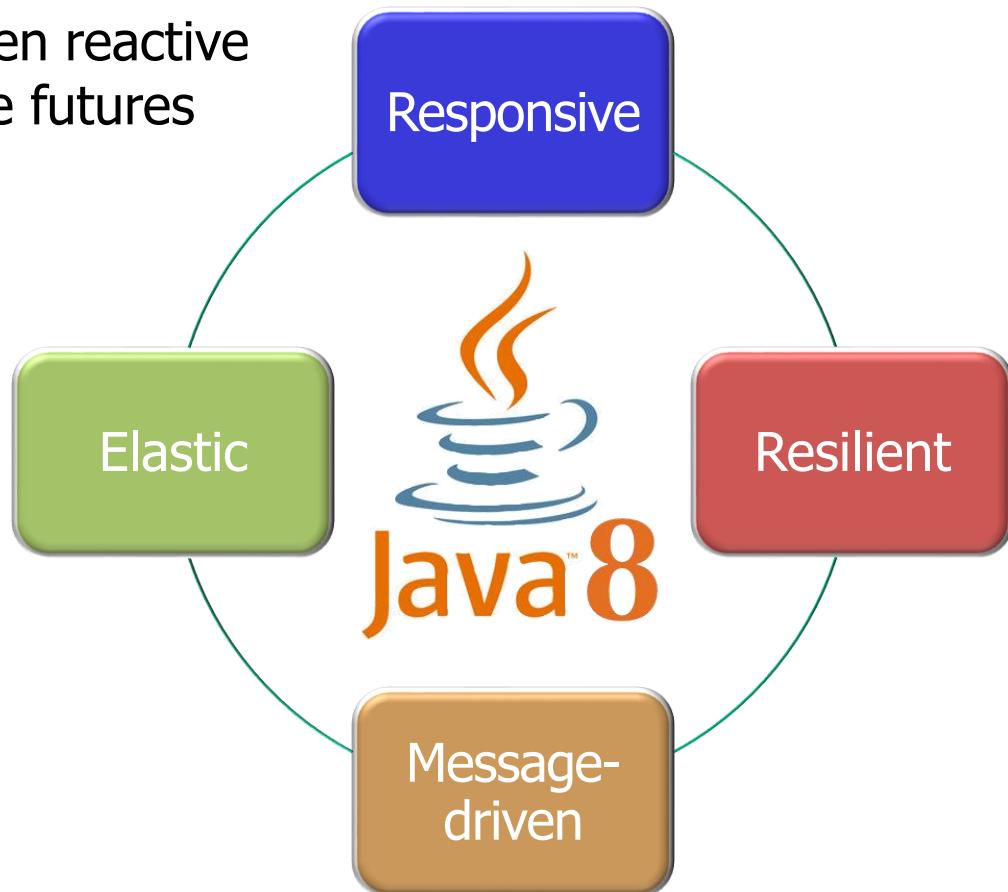
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Lesson

- Understand the relationship between reactive programming & Java 8 completable futures



---

# Reactive Programming & Java 8 CompletableFuture

# Reactive Programming & Java 8 Completable Futures

---

- Reactive programming is an asynchronous programming paradigm concerned with processing data streams & propagation of changes



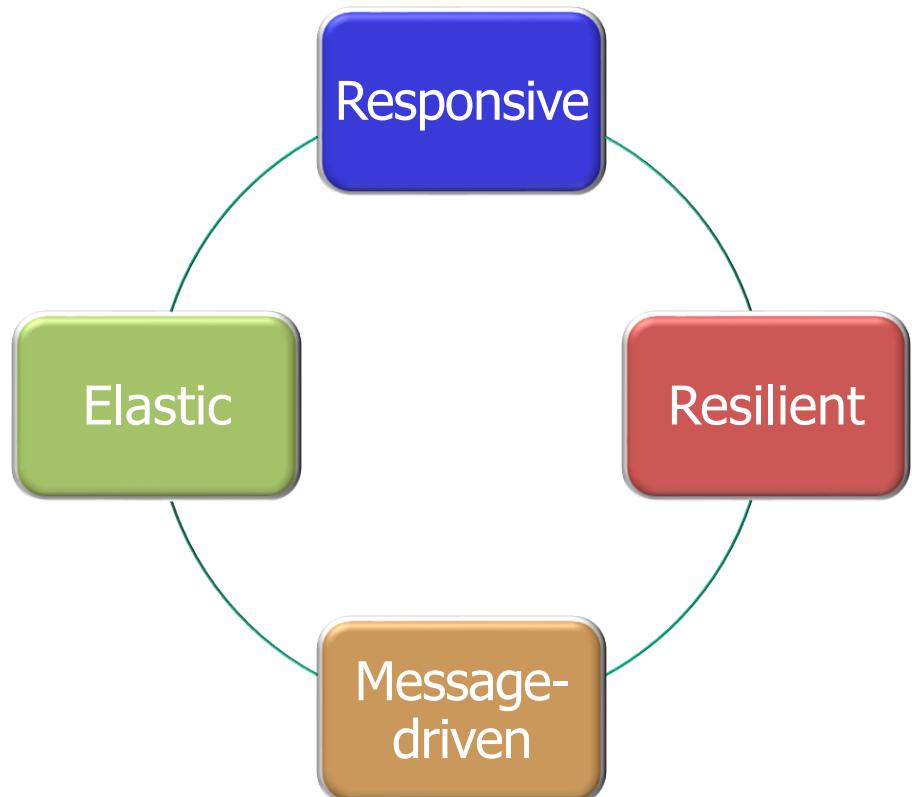
---

See [en.wikipedia.org/wiki/Reactive\\_programming](https://en.wikipedia.org/wiki/Reactive_programming)

# Reactive Programming & Java 8 Completable Futures

---

- Reactive programming is based on four key principles



# Reactive Programming & Java 8 Completable Futures

---

- Reactive programming is based on four key principles, e.g.

- **Responsive**

- Provide rapid & consistent response times



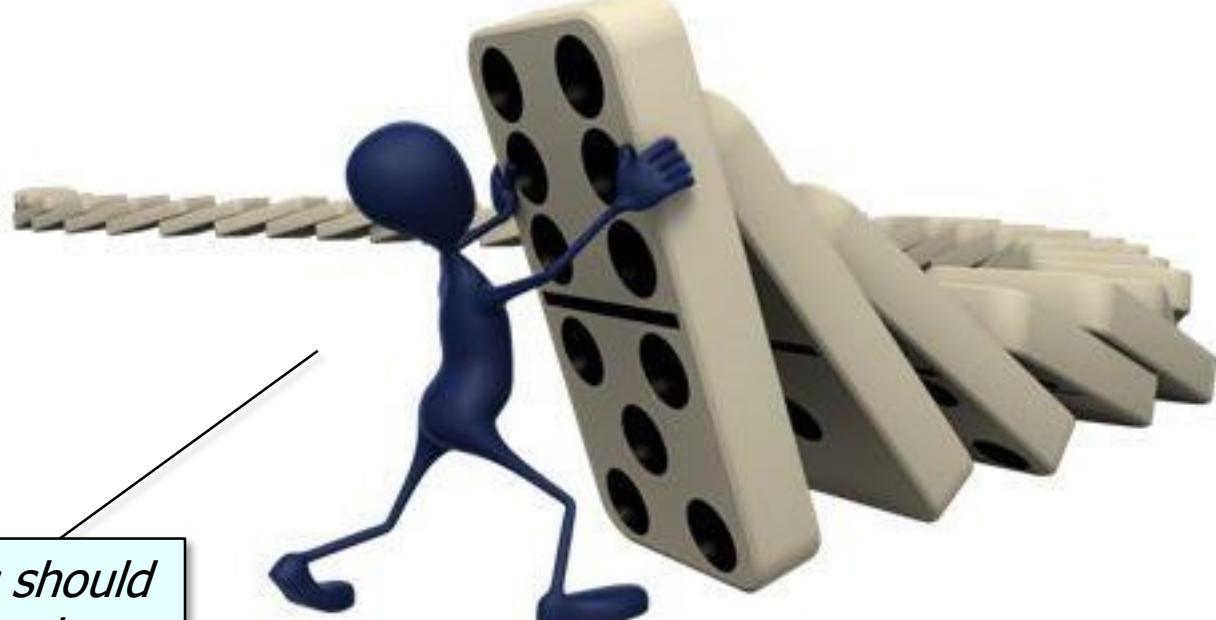
*Establish reliable upper bounds to deliver consistent quality of service & prevent delays*

---

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

# Reactive Programming & Java 8 Completable Futures

- Reactive programming is based on four key principles, e.g.
  - **Responsive**
  - **Resilient**
    - The system remains responsive, even in the face of failure



*Failure of some operations should not bring the entire system down*

# Reactive Programming & Java 8 Completable Futures

- Reactive programming is based on four key principles, e.g.

- **Responsive**
- **Resilient**
- **Elastic**

- A system should remain responsive, even under varying workload

*It should be possible to "auto-scale" performance*



# Reactive Programming & Java 8 Completable Futures

- Reactive programming is based on four key principles, e.g.

- **Responsive**

*This principle is an "implementation detail" wrt the others..*

- **Resilient**

- **Elastic**

- **Message-driven**

- Asynchronous message-passing ensures loose coupling, isolation, & location transparency between components



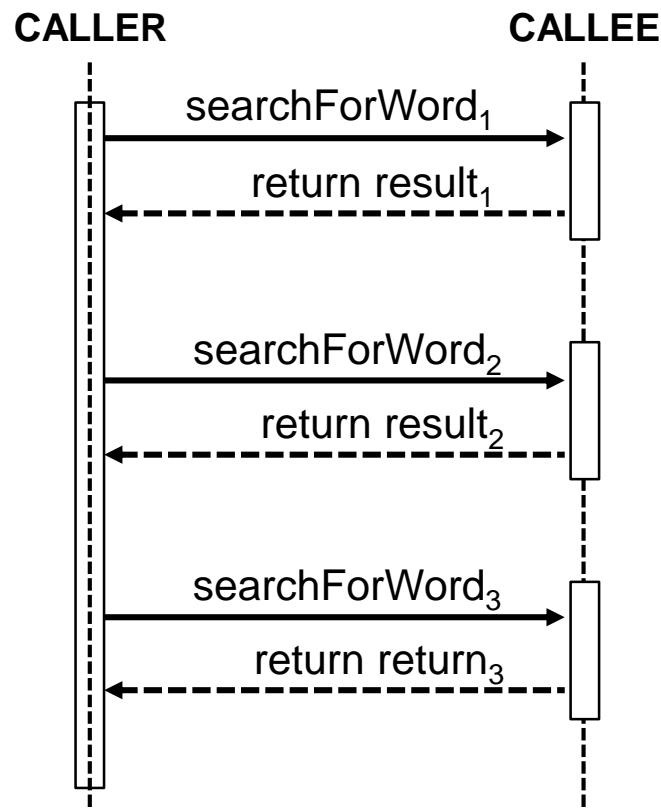
See [en.wikipedia.org/wiki/Message-oriented\\_middleware](https://en.wikipedia.org/wiki/Message-oriented_middleware)

# Reactive Programming & Java 8 Completable Futures

- Java 8 completable futures map onto key reactive programming principles, e.g.

- Responsive**

- Avoid blocking in user code
  - Blocking underutilizes cores, impedes inherent parallelism, & complicates program structure



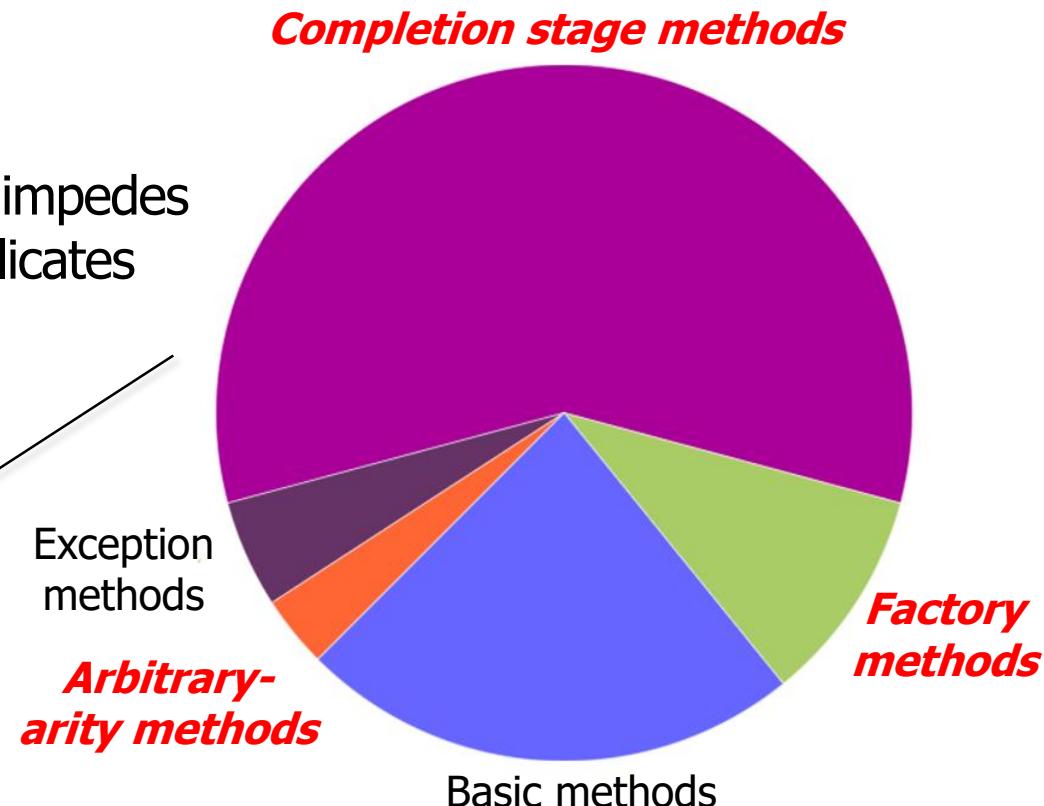
# Reactive Programming & Java 8 Completable Futures

- Java 8 completable futures map onto key reactive programming principles, e.g.

- Responsive**

- Avoid blocking in user code
  - Blocking underutilizes cores, impedes inherent parallelism, & complicates program structure

*Factory, completion stage, & arbitrary-arity methods avoid blocking threads*



# Reactive Programming & Java 8 Completable Futures

- Java 8 completable futures map onto key reactive programming principles, e.g.
  - Responsive**
    - Avoid blocking in user code
    - Avoid changing threads
      - Incurs excessive overhead wrt synchronization, context switching, & memory/cache management



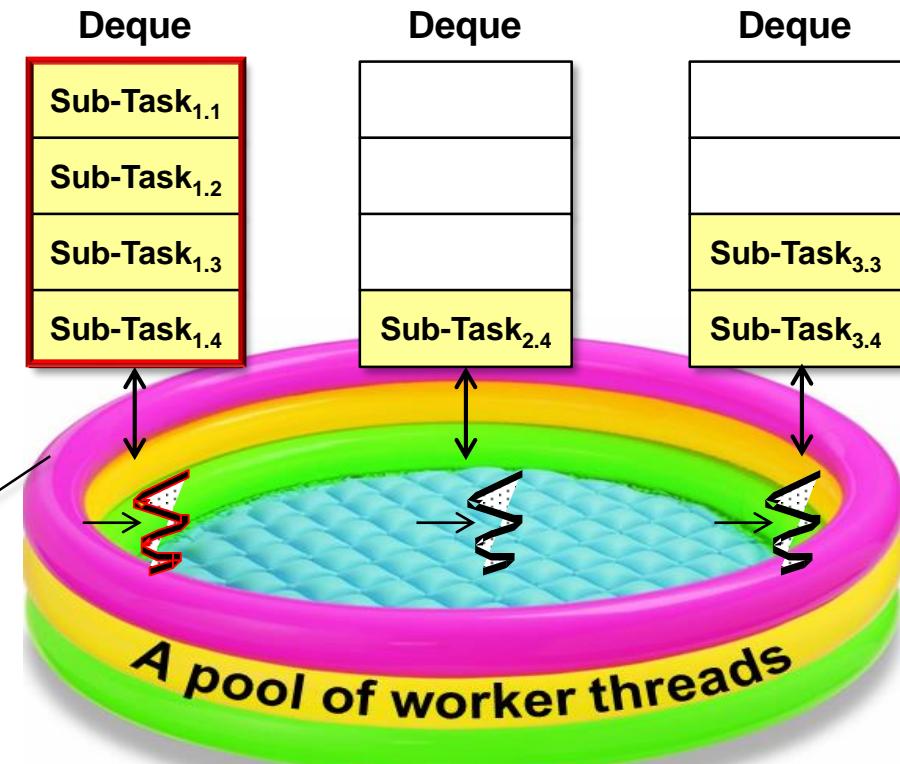
# Reactive Programming & Java 8 Completable Futures

- Java 8 completable futures map onto key reactive programming principles, e.g.

- Responsive**

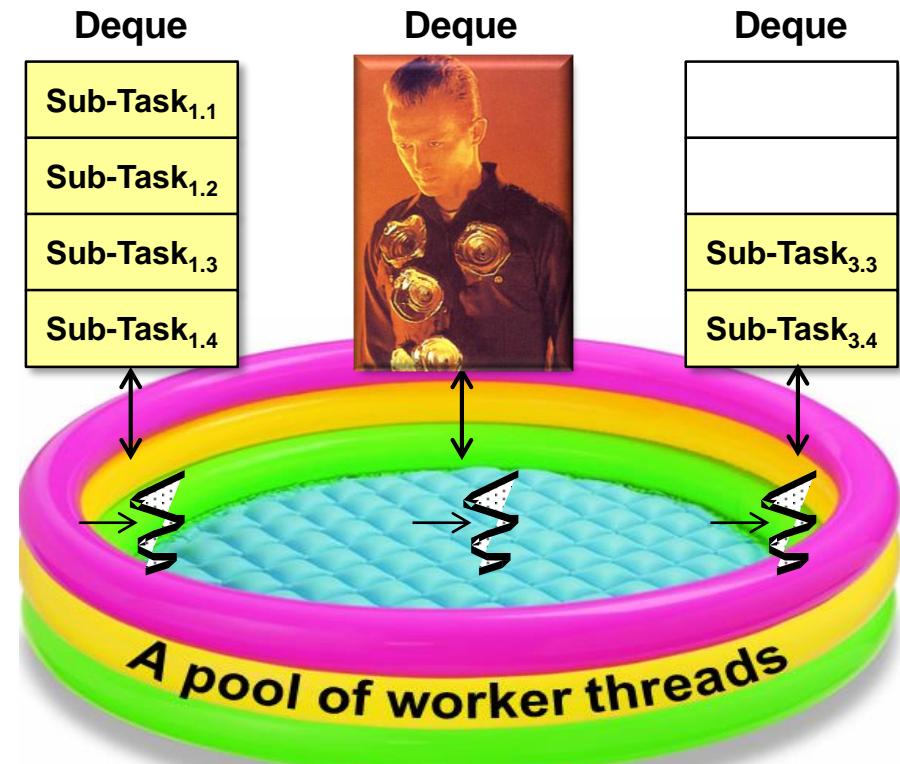
- Avoid blocking in user code
- Avoid changing threads
  - Incurs excessive overhead wrt synchronization, context switching, & memory/cache management

*The fork-join pool & non-\*Async() methods avoid changing threads*



# Reactive Programming & Java 8 Completable Futures

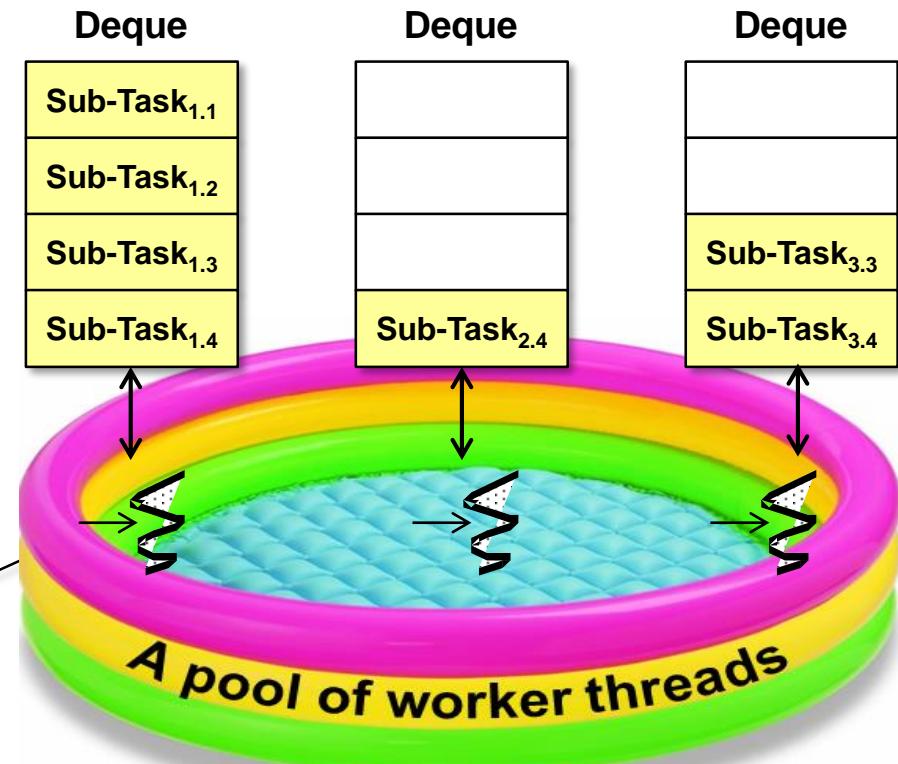
- Java 8 completable futures map onto key reactive programming principles, e.g.
  - Responsive**
  - Resilient**
    - Exception methods help systems be resilient to crippling failures



However, completable futures are localized to a single process, *not* a cluster!

# Reactive Programming & Java 8 Completable Futures

- Java 8 completable futures map onto key reactive programming principles, e.g.
  - Responsive**
  - Resilient**
  - Elastic**
    - Async computations can run scalably in a pool of threads

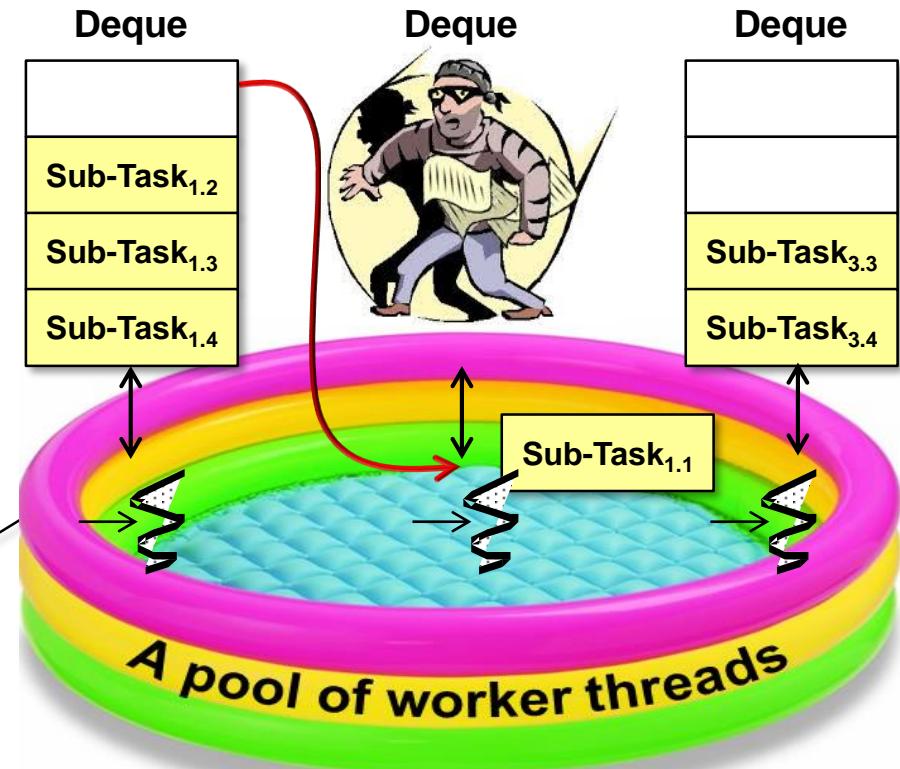


*Can be a fork-join pool  
or a custom thread pool*

# Reactive Programming & Java 8 Completable Futures

- Java 8 completable futures map onto key reactive programming principles, e.g.
  - Responsive**
  - Resilient**
  - Elastic**
  - Message-driven**
    - The Java fork-join pool passes messages between threads in the pool internally

*The Java fork-join pool implements "work-stealing" between dequeues*



See [en.wikipedia.org/wiki/Work\\_stealing](https://en.wikipedia.org/wiki/Work_stealing)

# Reactive Programming & Java 8 Completable Futures

- Java 9 support reactive programming via “Reactive Streams” & the Flow API

## Class Flow

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

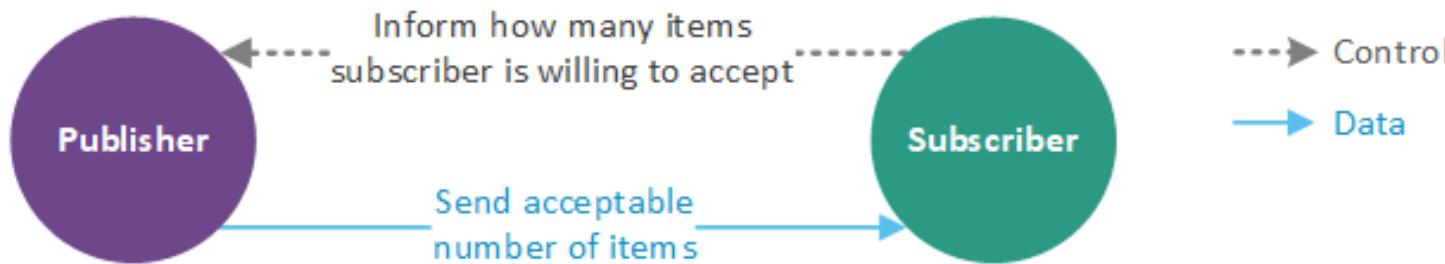
```
public final class Flow  
extends Object
```

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.

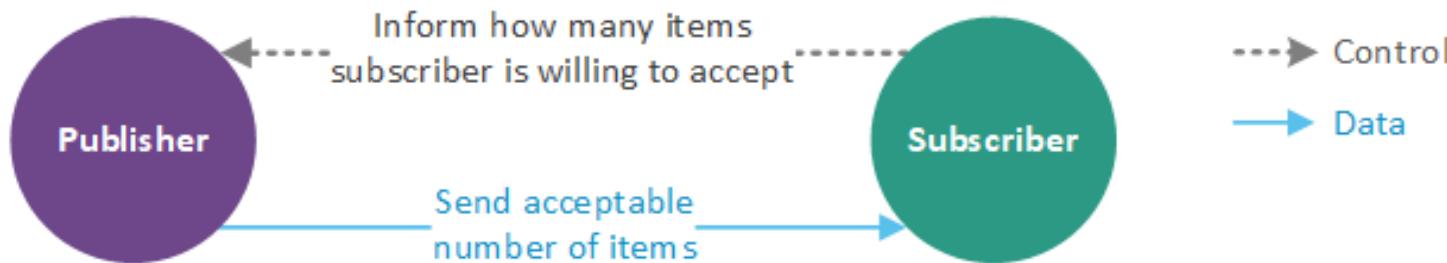
# Reactive Programming & Java 8 Completable Futures

- Java 9 support reactive programming via “Reactive Streams” & the Flow API
  - Adds support for stream-oriented pub/sub patterns



# Reactive Programming & Java 8 Completable Futures

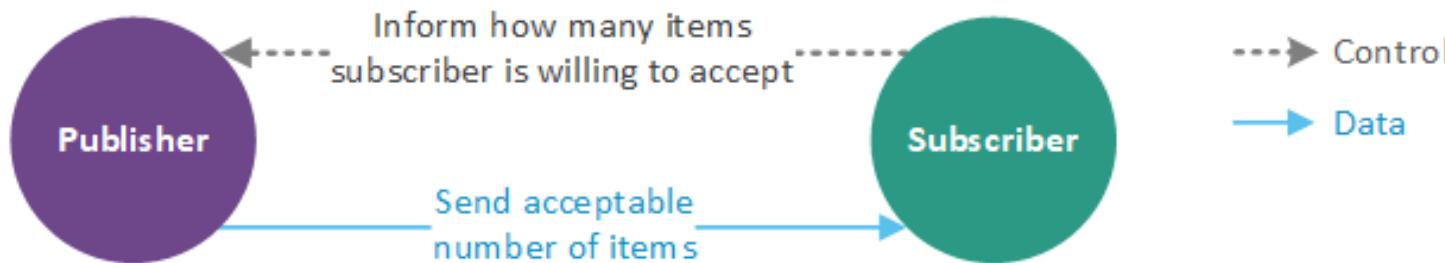
- Java 9 support reactive programming via “Reactive Streams” & the Flow API
  - Adds support for stream-oriented pub/sub patterns



- Combines two patterns
  - Iterator*, which applies a pull model where apps pulls items from a source
  - Observer*, which applies a push model that reacts when item is pushed from a source to a subscriber

# Reactive Programming & Java 8 Completable Futures

- Java 9 support reactive programming via “Reactive Streams” & the Flow API
  - Adds support for stream-oriented pub/sub patterns

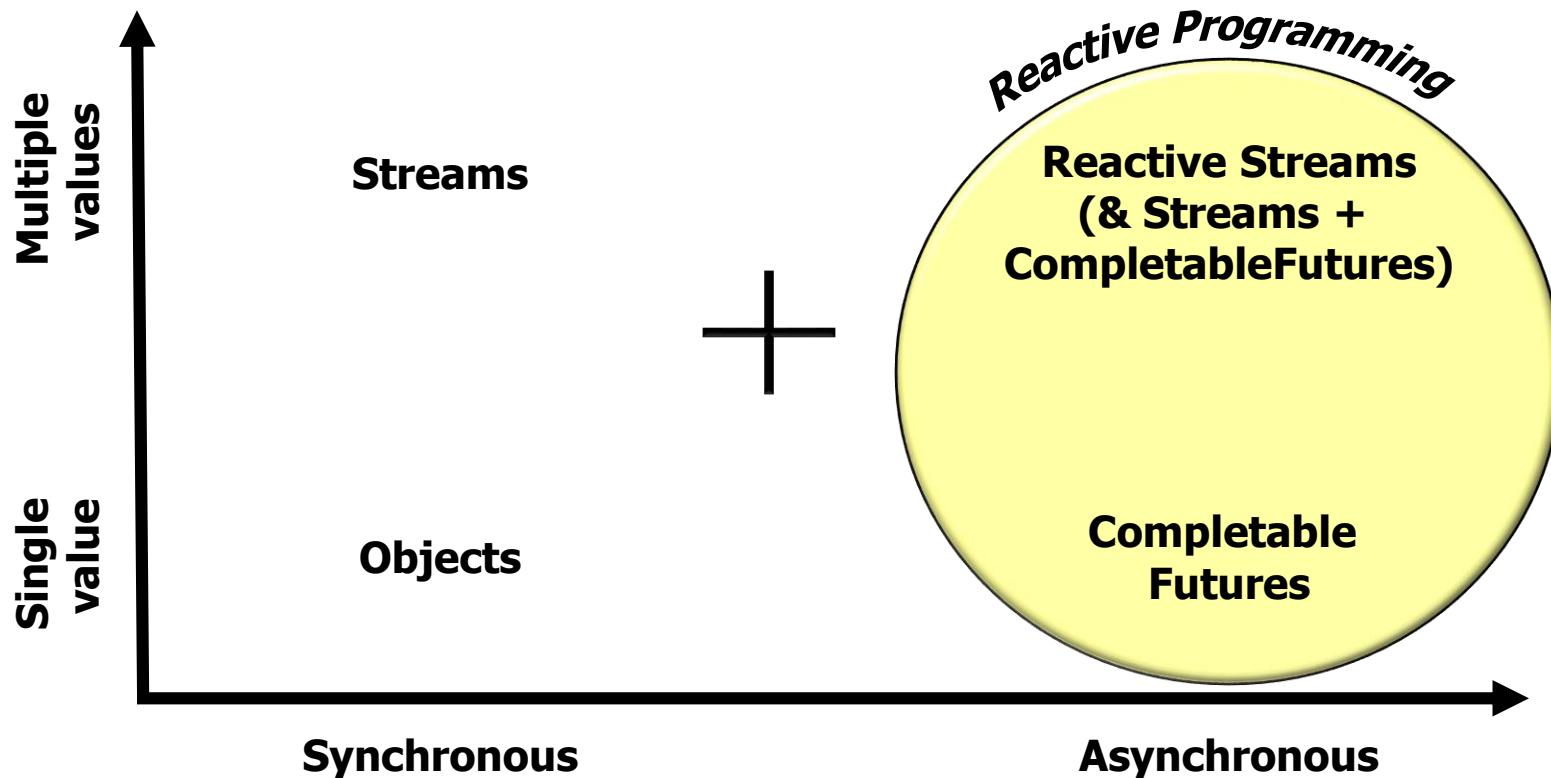


- Combines two patterns
- Intended as an interoperable foundation for other reactive programming frameworks



# Reactive Programming & Java 8 Completable Futures

- Comparing reactive programming with other Java programming paradigms



---

# End of Reactive Programming & Java 8 CompletableFuture

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

Douglas C. Schmidt

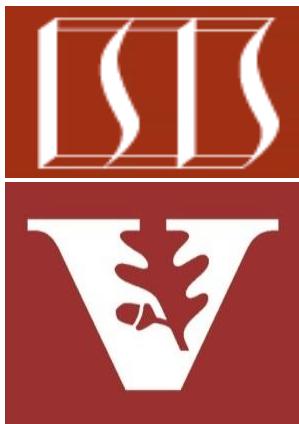
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[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 by understanding the pros & cons of synchrony & asynchrony

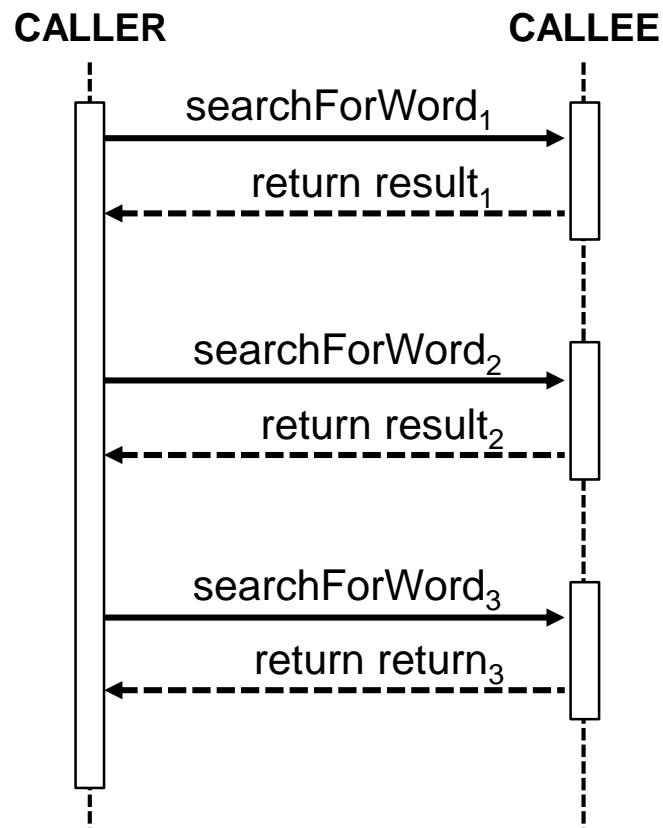


---

# Motivating Need for Futures: Pros & Cons of Synchrony

# Motivating Need for Futures: Pros & Cons of Synchrony

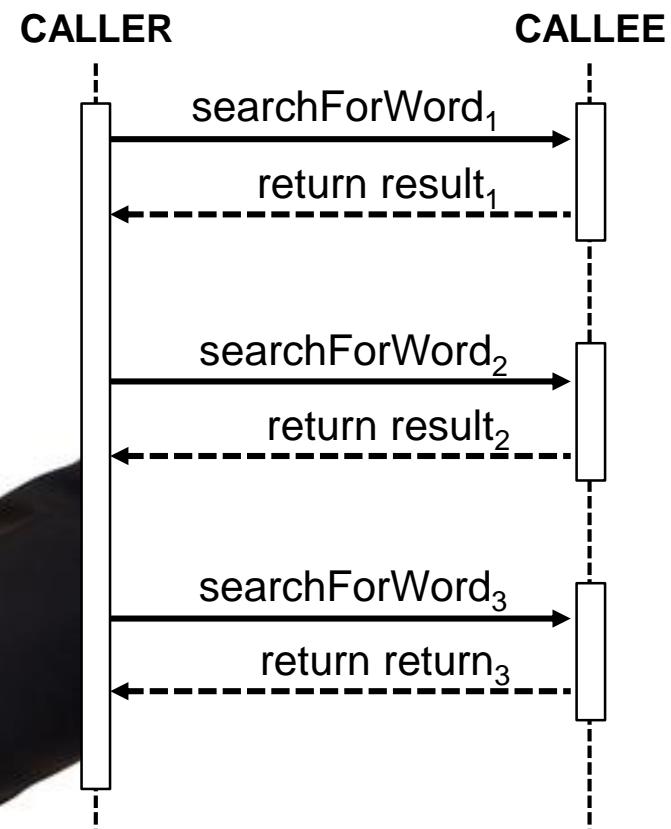
- Method calls in typical Java programs are largely *synchronous*



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

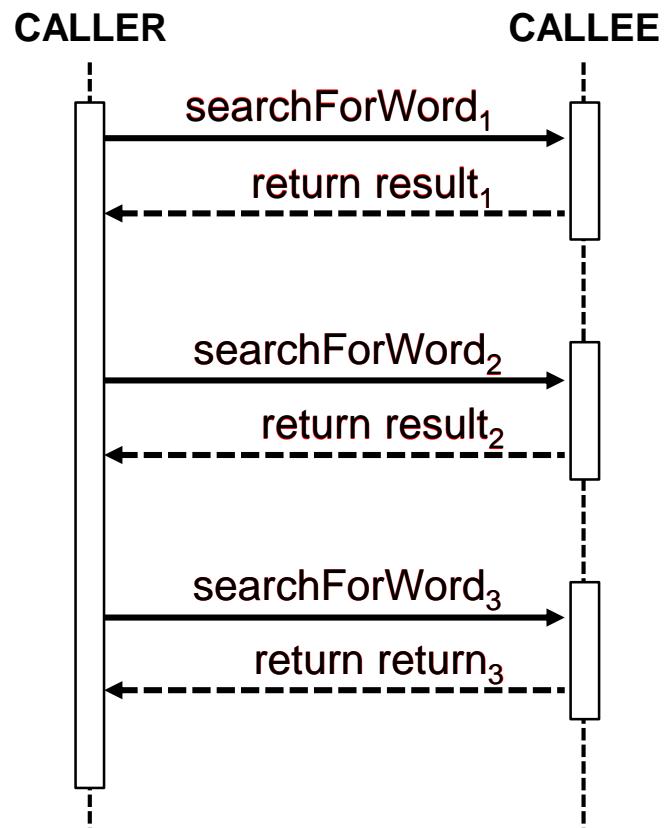
# Motivating Need for Futures: Pros & Cons of Synchrony

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



# Motivating Need for Futures: Pros & Cons of Synchrony

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



# Motivating Need for Futures: Pros & Cons of Synchrony

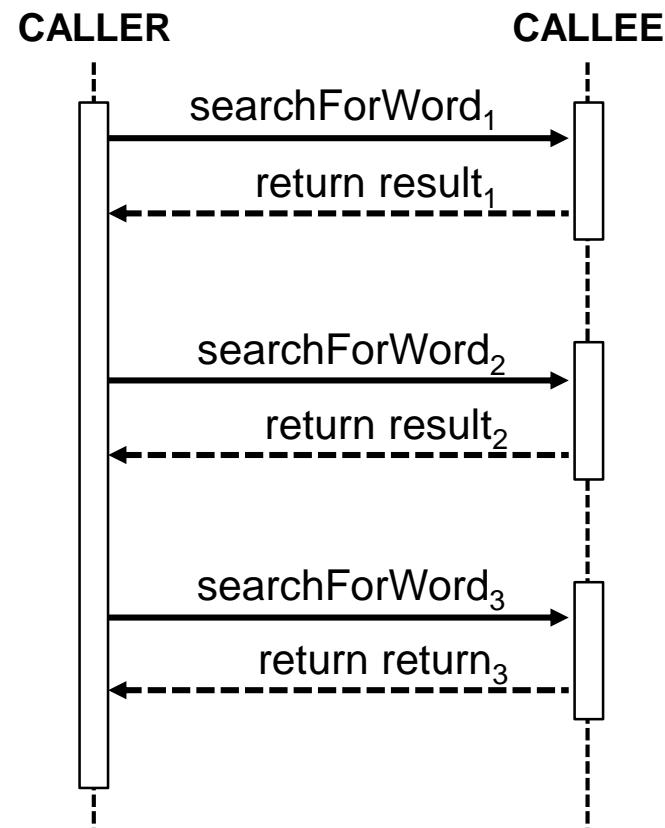
---

- Synchronous calls have pros & cons



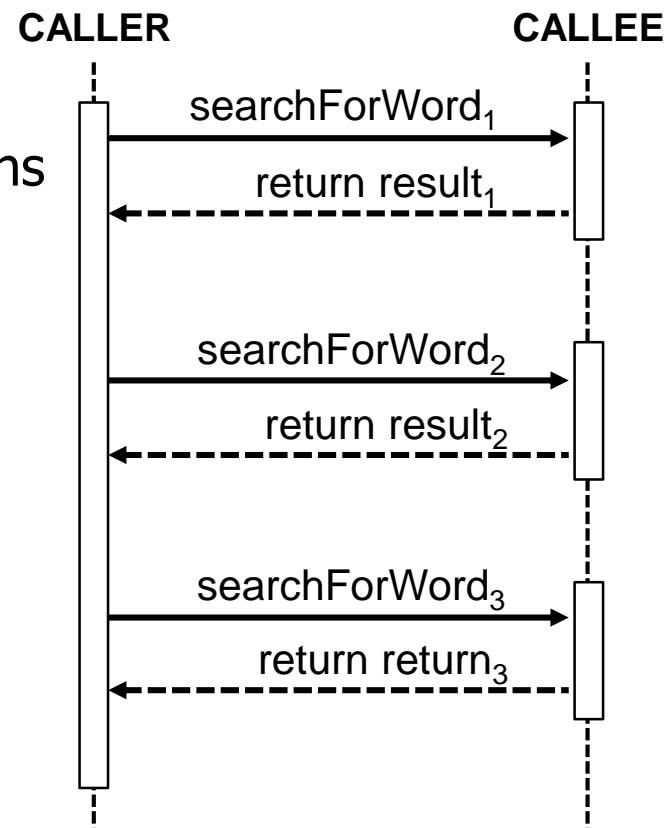
# Motivating Need for Futures: Pros & Cons of Synchrony

- Pros of synchronous calls:
  - “Intuitive” to program & debug



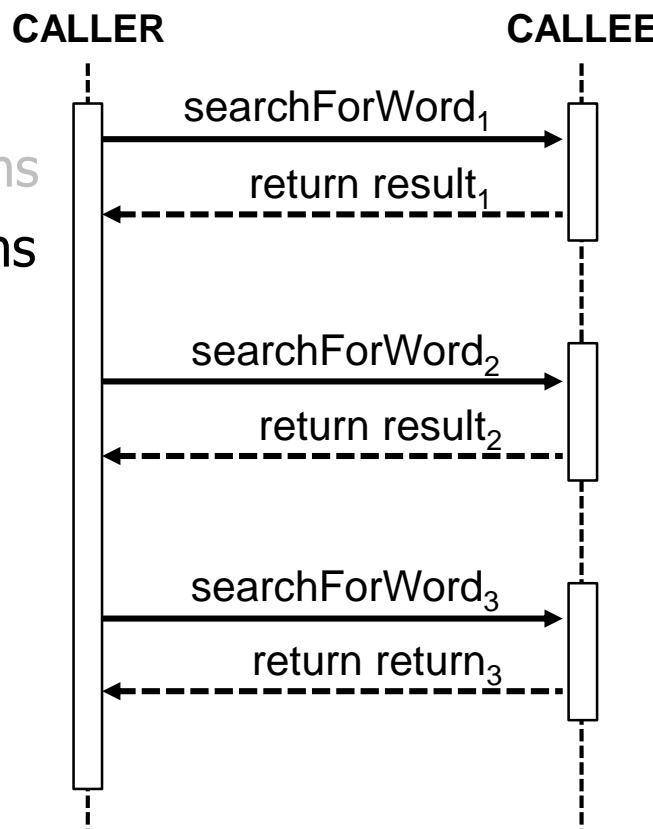
# Motivating Need for Futures: Pros & Cons of Synchrony

- Pros of synchronous calls:
  - “Intuitive” to program & debug, e.g.
  - Maps onto common two-way method patterns



## Motivating Need for Futures: Pros & Cons of Synchrony

- Pros of synchronous calls:
    - “Intuitive” to program & debug, e.g.
      - Maps onto common two-way method patterns
      - Local caller state retained when callee returns



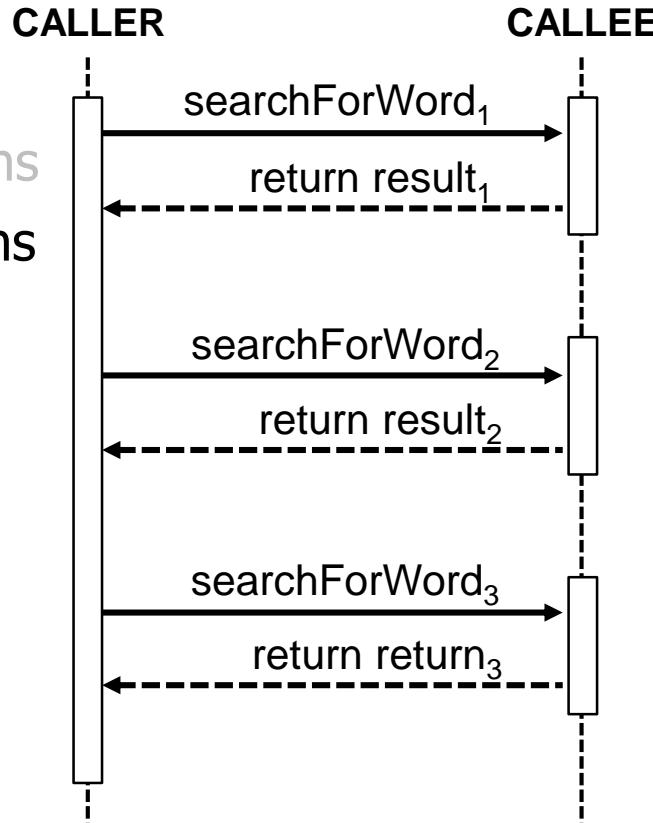
See [wiki.c2.com/?ActivationRecord](http://wiki.c2.com/?ActivationRecord)

# Motivating Need for Futures: Pros & Cons of Synchrony

- Pros of synchronous calls:

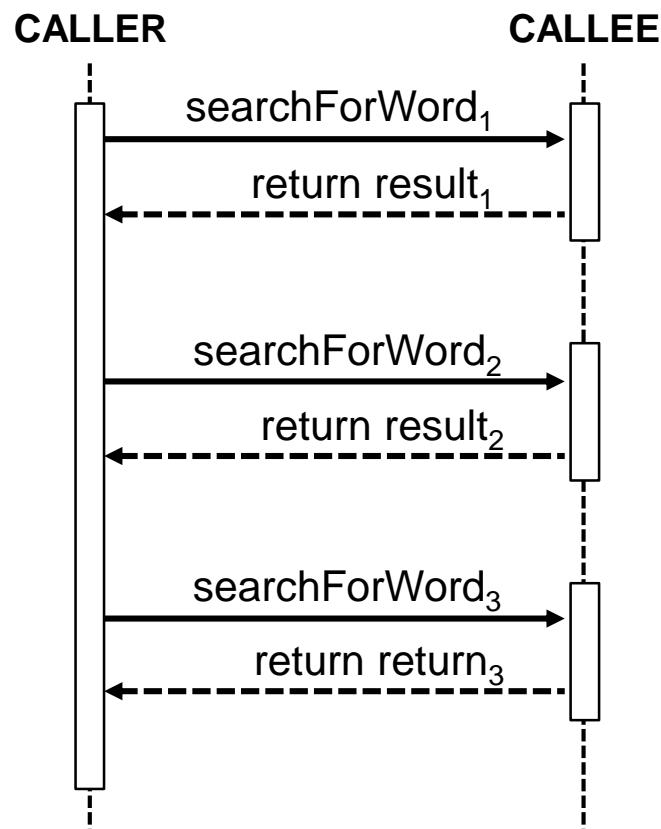
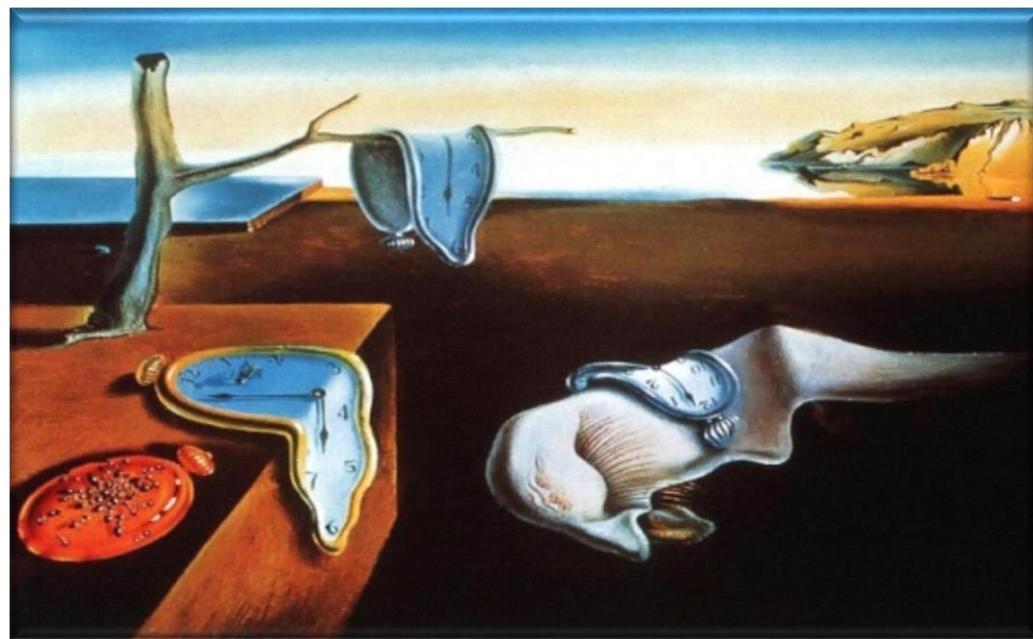
- “Intuitive” to program & debug, e.g.
  - Maps onto common two-way method patterns
  - Local caller state retained when callee returns

```
byte[] downloadContent(URL url) {  
    byte[] buf = new byte[BUFSIZ];  
    ByteArrayOutputStream os =  
        new ByteArrayOutputStream();  
    InputStream is = url.openStream();  
  
    for (int bytes;  
         (bytes = is.read(buf)) > 0;)  
        os.write(buf, 0, bytes); ...  
}
```



# Motivating Need for Futures: Pros & Cons of Synchrony

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



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

# Motivating Need for Futures: Pros & Cons of Synchrony

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

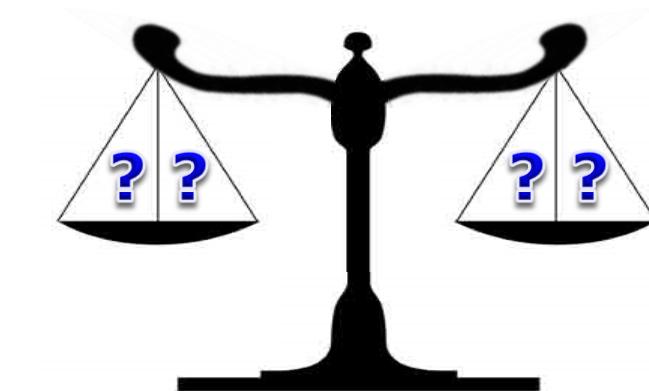


See [www.ibm.com/developerworks/library/j-jvmc3](http://www.ibm.com/developerworks/library/j-jvmc3)

# Motivating Need for Futures: Pros & Cons of Synchrony

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

```
List<Image> filteredImages = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```



*Efficient Performance*

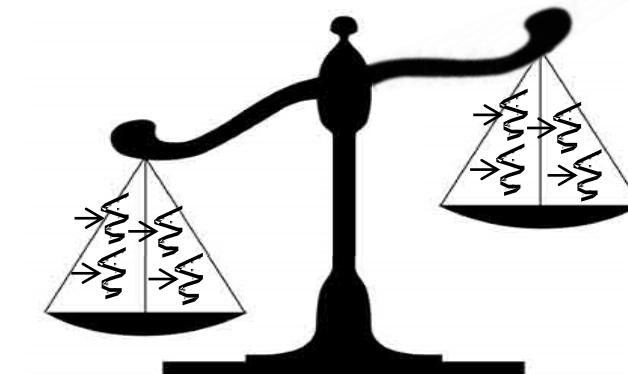
*Efficient Resource Utilization*

```
Image downloadImage(URL url) {
    return new Image(url,
        downloadContent(url));
}
```

# Motivating Need for Futures: Pros & Cons of Synchrony

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

```
List<Image> filteredImages = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```



*Efficient  
Performance*

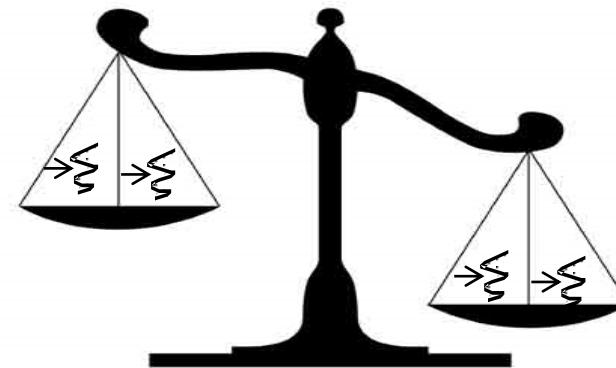
*Efficient  
Resource  
Utilization*

*A large # of threads may  
improve performance at the  
cost of wasted resources*

# Motivating Need for Futures: Pros & Cons of Synchrony

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

```
List<Image> filteredImages = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```



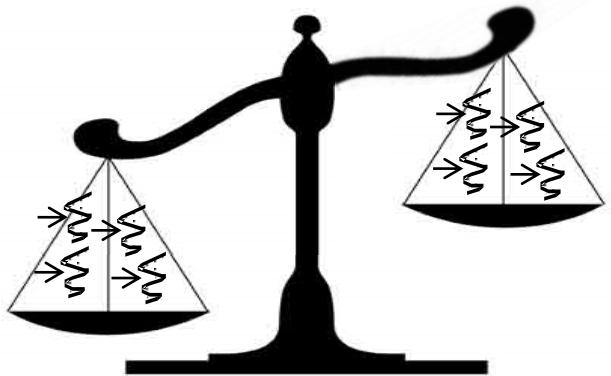
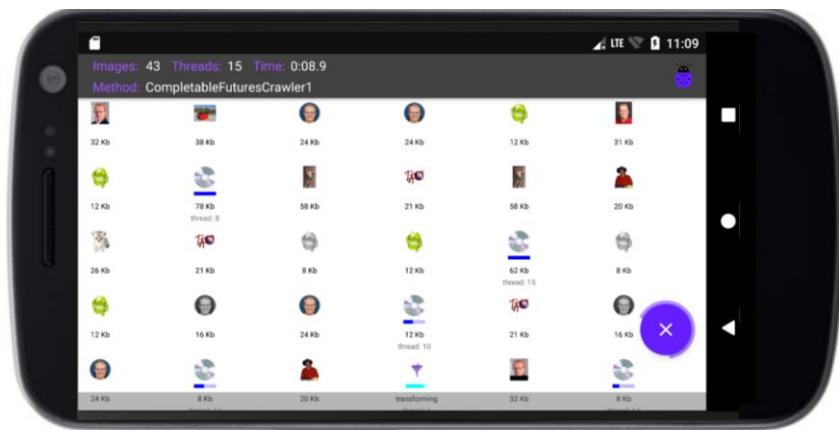
*Efficient Performance*

*Efficient Resource Utilization*

*A small # of threads may conserve resources at the cost of performance*

# Motivating Need for Futures: Pros & Cons of Synchrony

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



*Efficient  
Performance*

*Efficient  
Resource  
Utilization*

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

# Motivating Need for Futures: Pros & Cons of Synchrony

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



See [dzone.com/articles/think-twice-using-java-8](https://dzone.com/articles/think-twice-using-java-8)

---

# Motivating Need for Futures: Pros & Cons of Asynchrony

# Motivating Need for Futures: Pros & Cons of Asynchrony

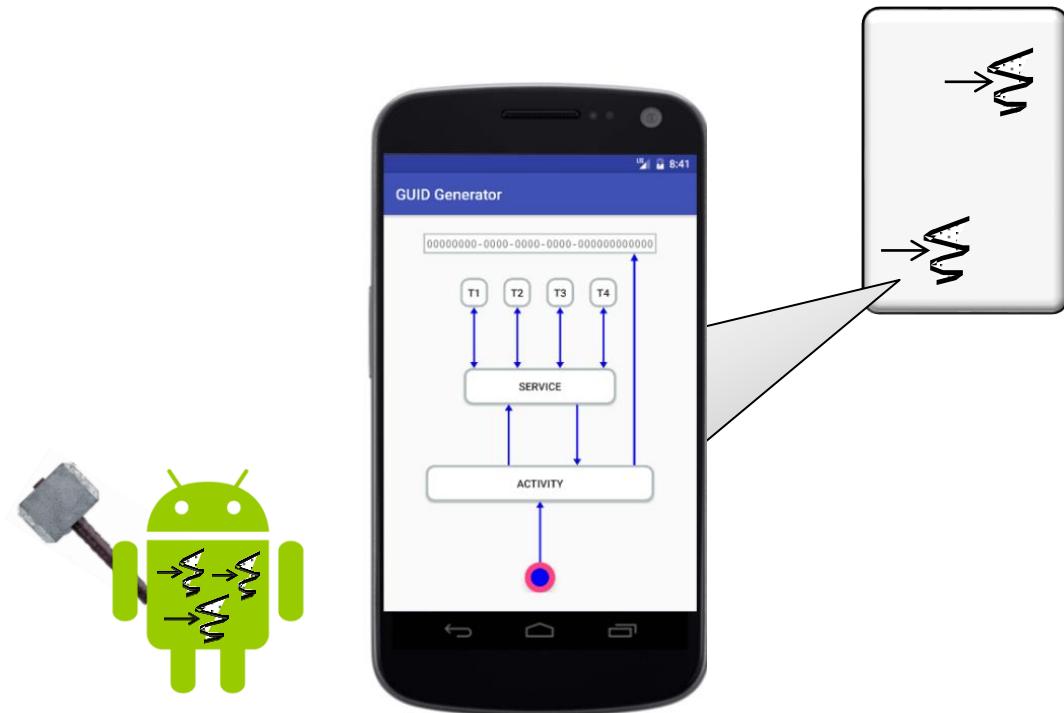
- Asynchronous (async) calls can alleviate the limitations with synchronous calls



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

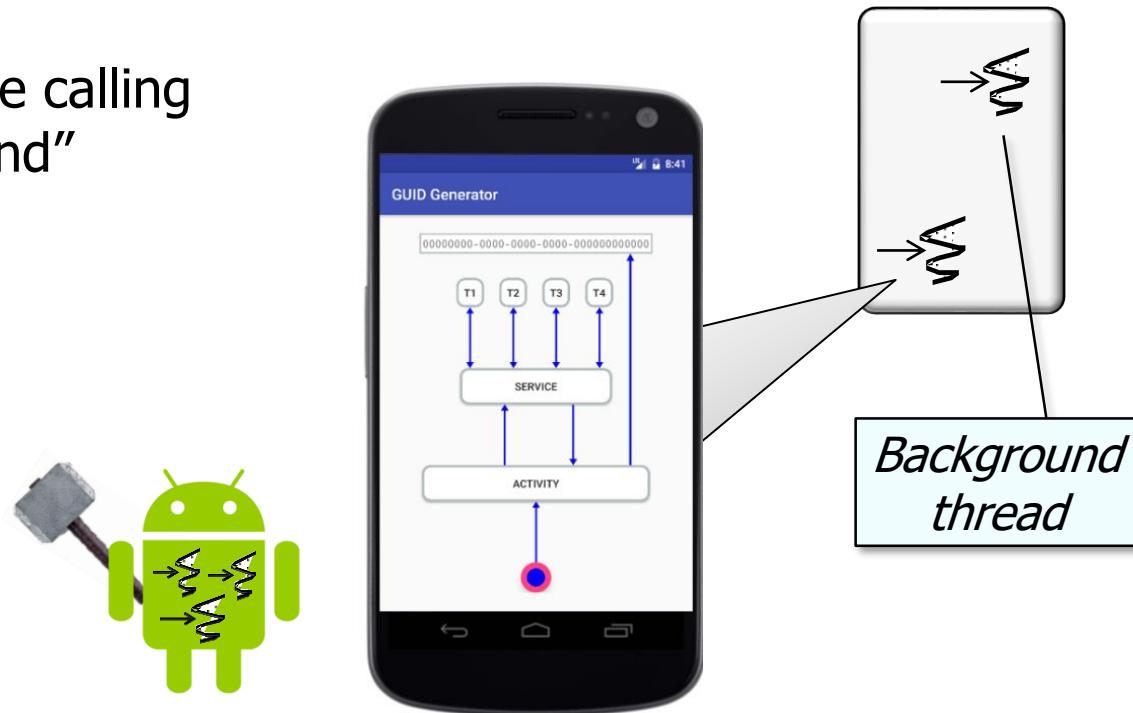
# Motivating Need for Futures: Pros & Cons of Asynchrony

- Asynchronous (async) calls can alleviate the limitations with synchronous calls
  - Asynchrony is a means of concurrent programming where a unit of work has certain properties



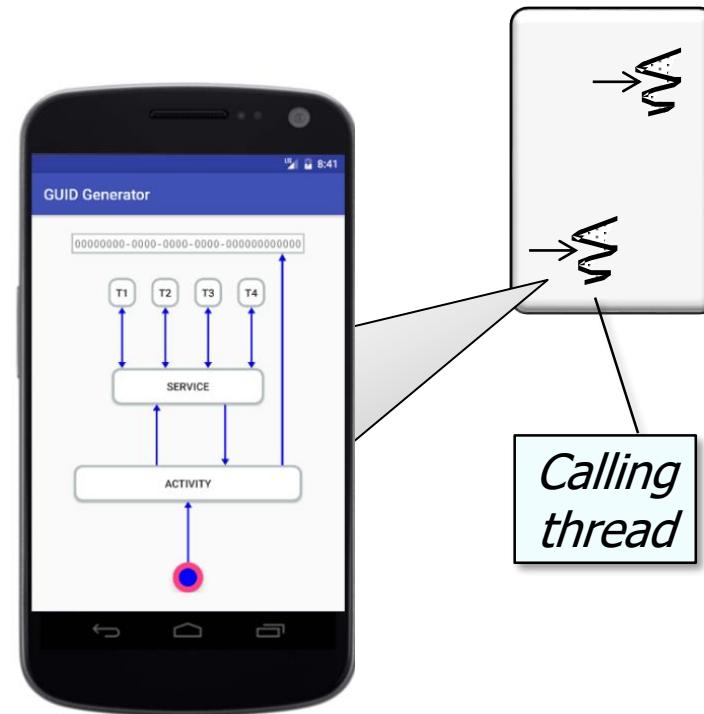
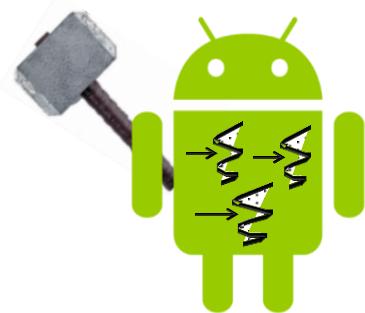
# Motivating Need for Futures: Pros & Cons of Asynchrony

- Asynchronous (async) calls can alleviate the limitations with synchronous calls
  - Asynchrony is a means of concurrent programming where a unit of work has certain properties
    - Runs separately from the calling thread “in the background”



# Motivating Need for Futures: Pros & Cons of Asynchrony

- Asynchronous (async) calls can alleviate the limitations with synchronous calls
  - Asynchrony is a means of concurrent programming where a unit of work has certain properties
  - Runs separately from the calling thread “in the background”
  - Notifies the calling thread of its completion, failure, or progress



# Motivating Need for Futures: Pros & Cons of Asynchrony

- Asynchronous calls have pros & cons



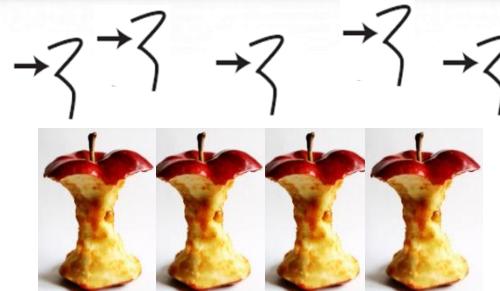
# Motivating Need for Futures: Pros & Cons of Asynchrony

- Pros of asynchronous calls
  - Responsiveness
    - A thread does not block waiting for other requests to complete



# Motivating Need for Futures: Pros & Cons of Asynchrony

- Pros of asynchronous calls
  - Responsiveness
  - Elasticity
    - Multiple requests can run scalably & concurrently in multiple cores



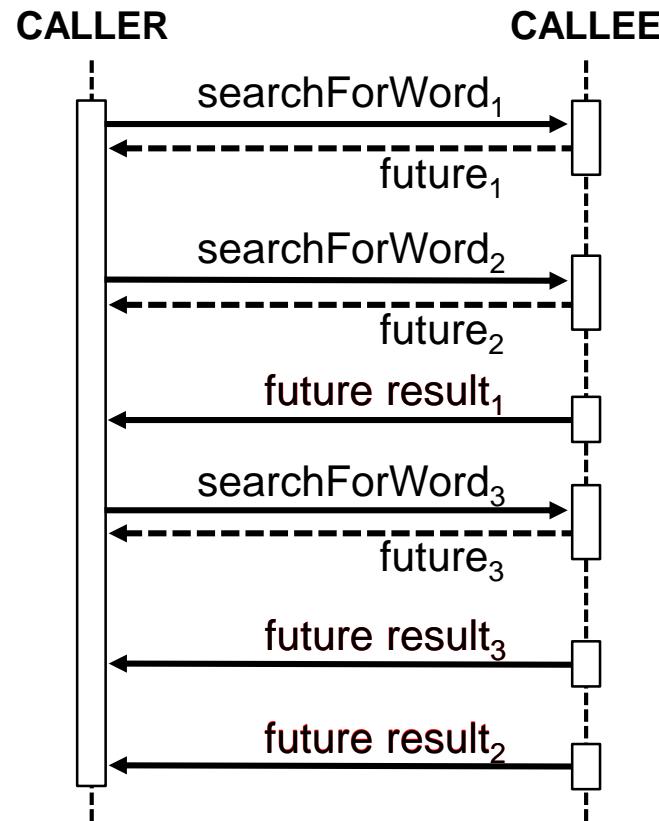
# Motivating Need for Futures: Pros & Cons of Asynchrony

- Cons of asynchronous calls
  - Unpredictability
  - Response times may not be predictable



# Motivating Need for Futures: Pros & Cons of Asynchrony

- Cons of asynchronous calls
  - Unpredictability
    - Response times may not be predictable
    - Results can occur in a different order than the original calls were made



# Motivating Need for Futures: Pros & Cons of Asynchrony

- Cons of asynchronous calls
  - Unpredictability
  - Complicated debugging
    - Errors can be hard to track due to unpredictability



See [www.jetbrains.com/help/idea/tutorial-java-debugging-deep-dive.html](http://www.jetbrains.com/help/idea/tutorial-java-debugging-deep-dive.html)

---

# End of Motivating the Need for Java 8 Completable 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

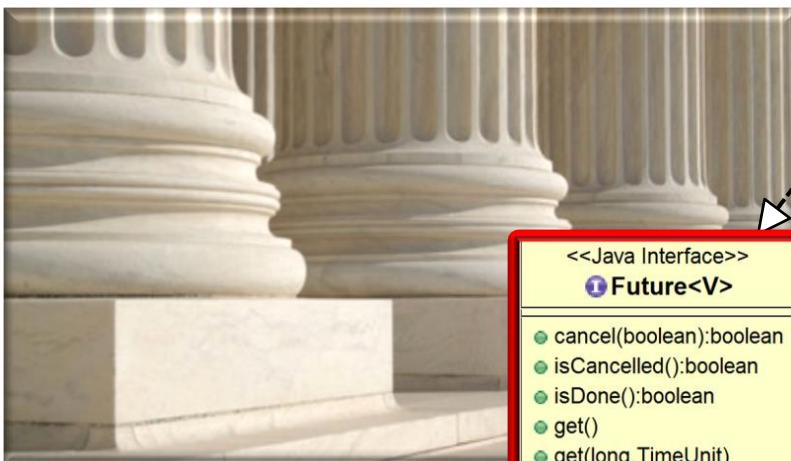
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Motivate the need for Java futures by understanding the pros & cons of synchrony & asynchrony
- Know how Java futures provide the foundation for completable futures in Java 8



## Future<V>

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

<<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>`

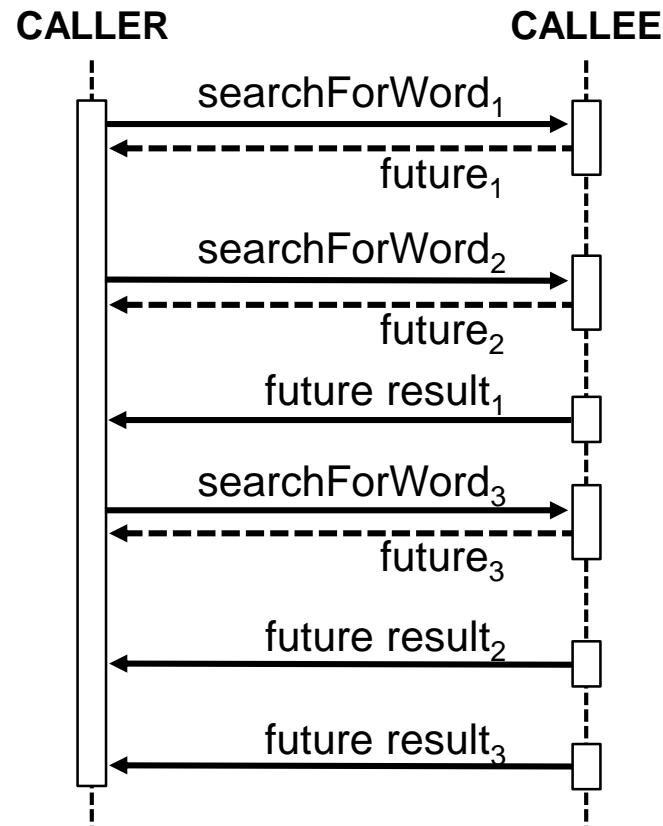
`anyOf(CompletableFuture[]<?>):CompletableFuture<Object>`

---

# Overview of Java Futures

# Overview of Java Futures

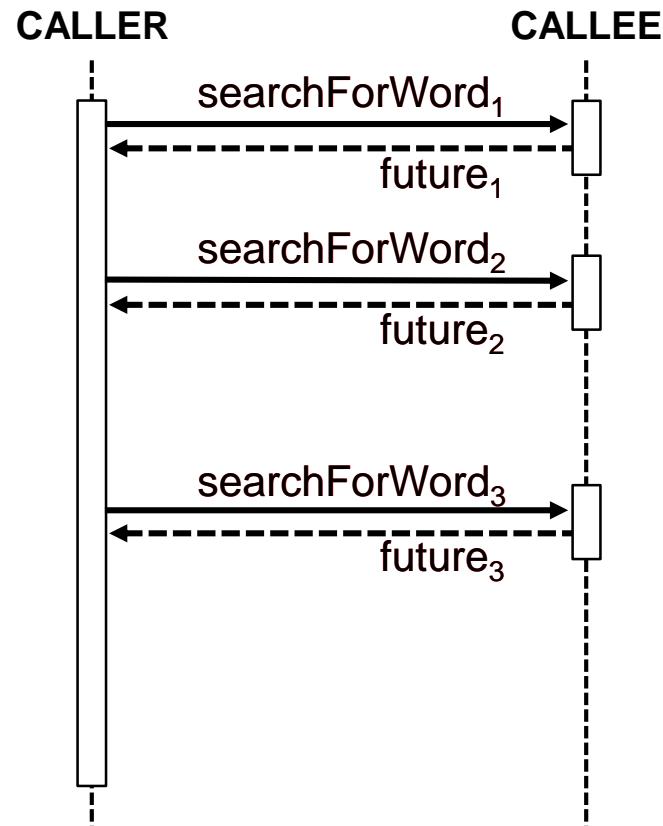
- Java 1.5 (JDK 5) added support for async calls via Java futures



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)

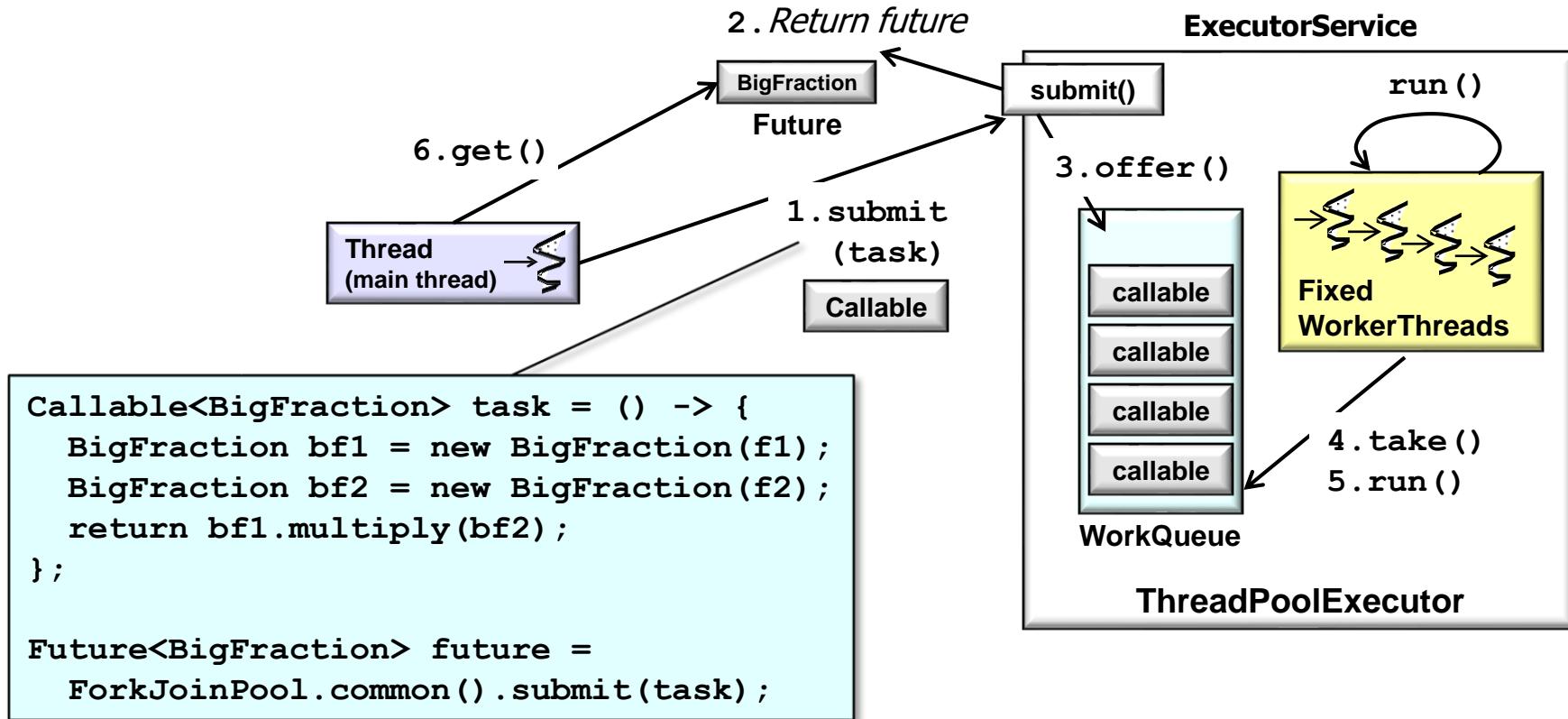
# Overview of Java Futures

- Java 1.5 (JDK 5) added support for async calls via Java futures
  - Async calls return a future & continue running the computation in the background



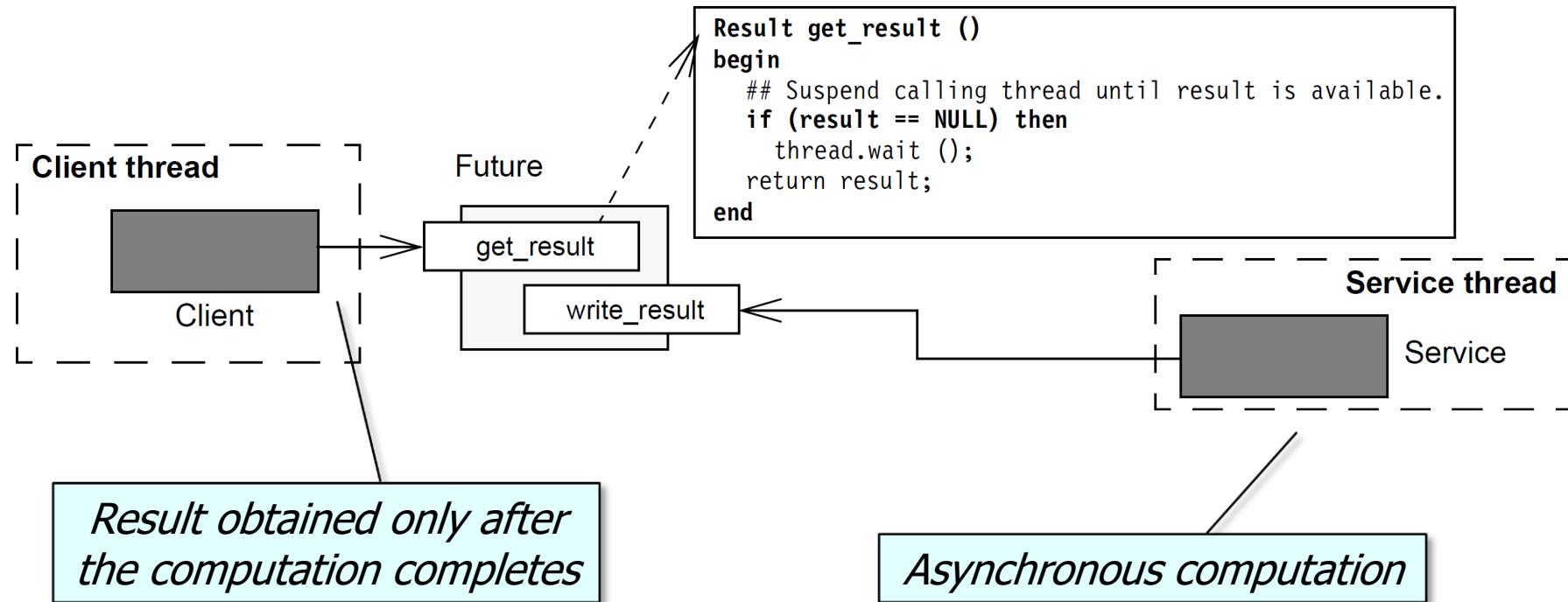
# Overview of Java Futures

- ExecutorService.submit() is an example of an async call in Java



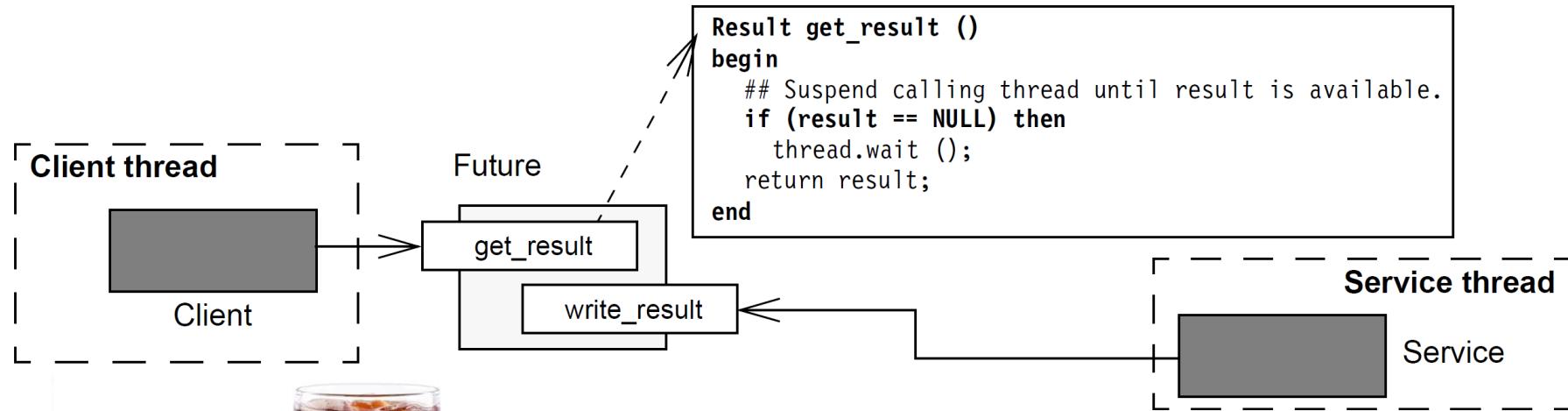
# Overview of Java Futures

- A future is a proxy that represents the result(s) of an async computation



# Overview of Java Futures

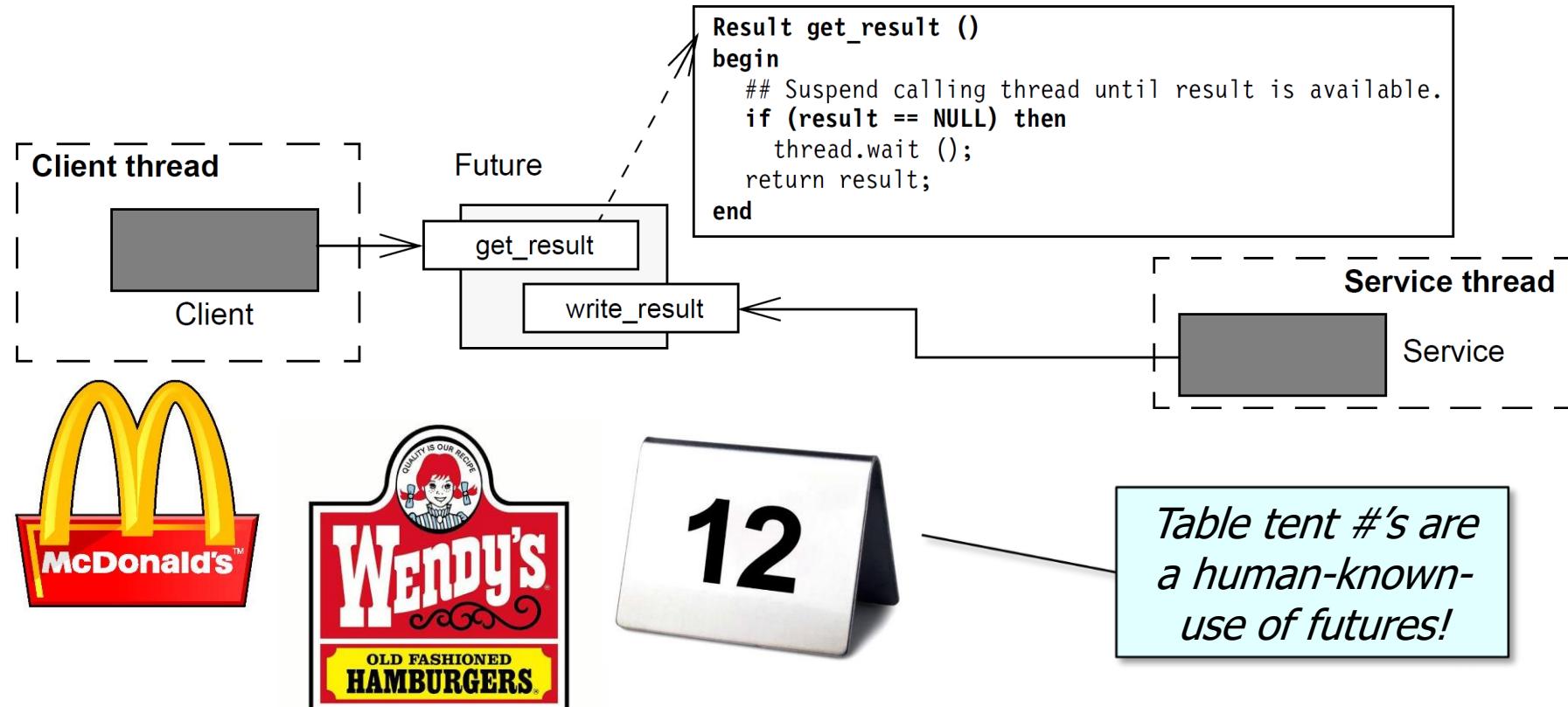
- A future is a proxy that represents the result(s) of an async computation



*Table tent #'s are a human-known-use of futures!*

# Overview of Java Futures

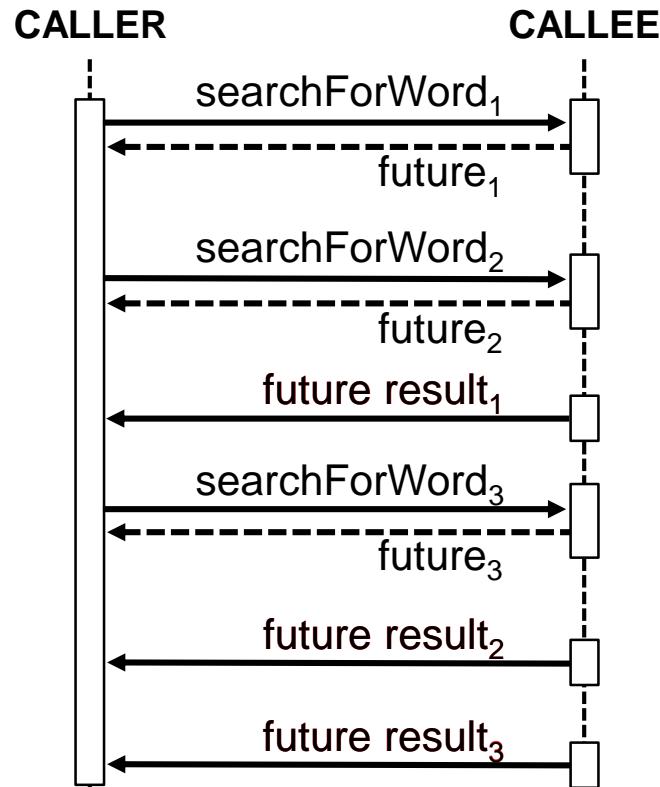
- A future is a proxy that represents the result(s) of an async computation



e.g., McDonald's vs Wendy's model of preparing fast foot

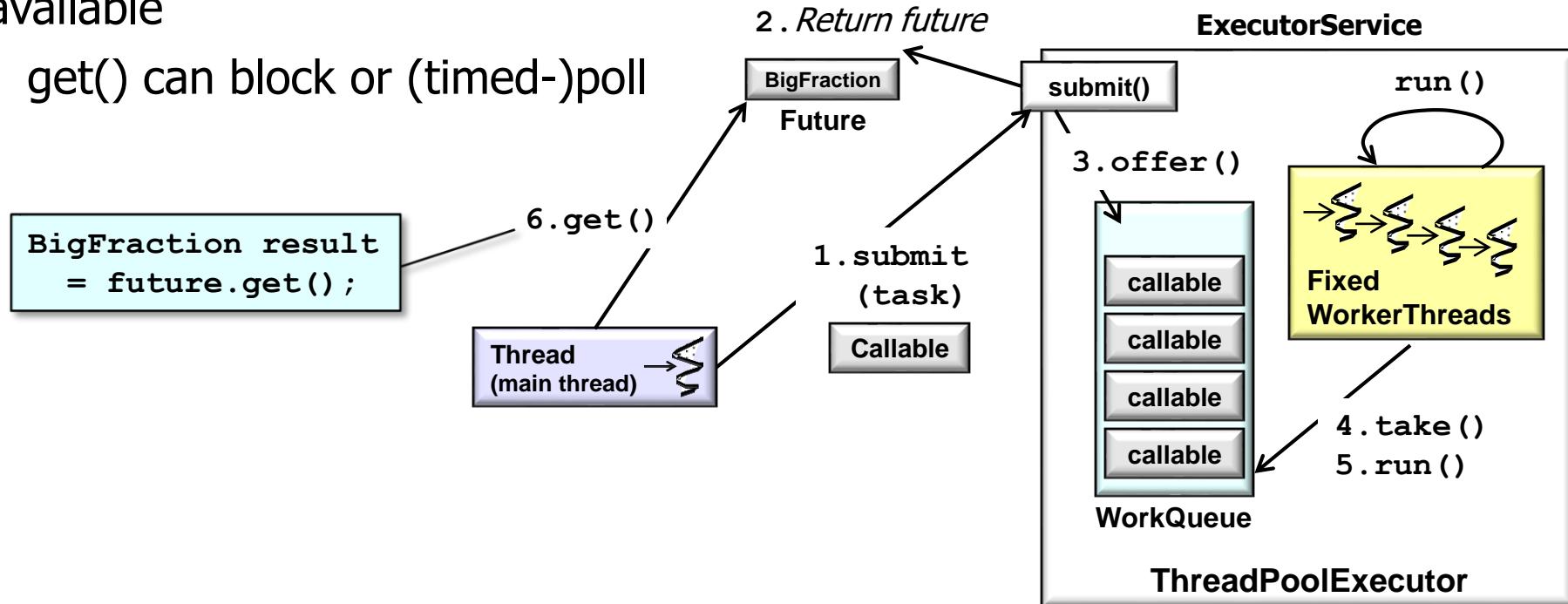
# Overview of Java Futures

- When the async computation completes the future is triggered & the result is available



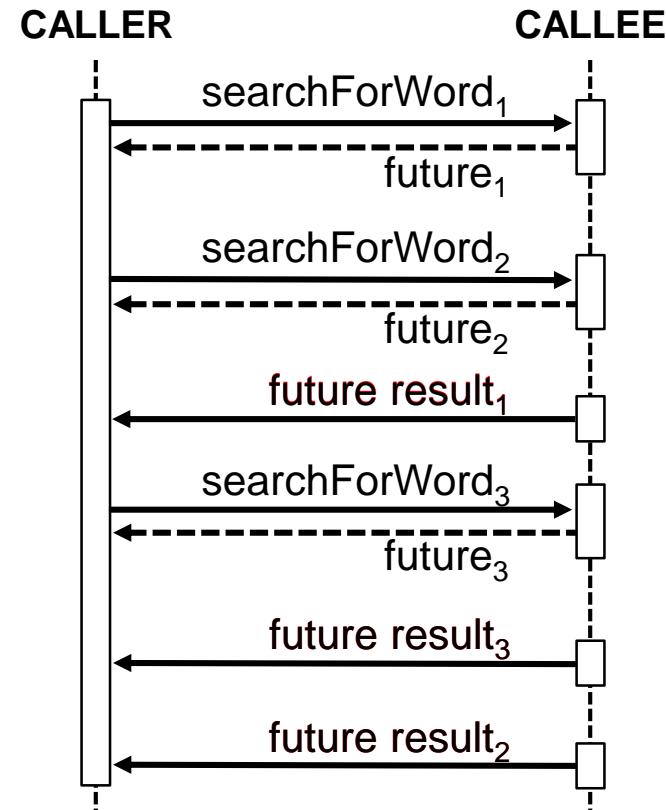
# Overview of Java Futures

- When the async computation completes the future is triggered & the result is available
  - get() can block or (timed-)poll



# Overview of Java Futures

- When the async computation completes the future is triggered & the result is available
  - get() can block or (timed-)poll
  - 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 two-way 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); };
```
- A rectangular sign with a black border. Inside, there are two white arrows pointing in opposite directions (left and right). In the center, the words "TWO WAY" are written in a bold, sans-serif font.
- ```
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
- ```
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();
```
- Can pass values to a callable via effectively final variables*

# 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

```
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();
```

*Other code can run concurrently here*

# 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();
```

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

# 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(n, SECONDS);
```

*get() can also perform polling & timed-blocks*

---

# End of Motivating the Need for Java 8 CompletableFuture Futures (Part 2)

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

Douglas C. Schmidt

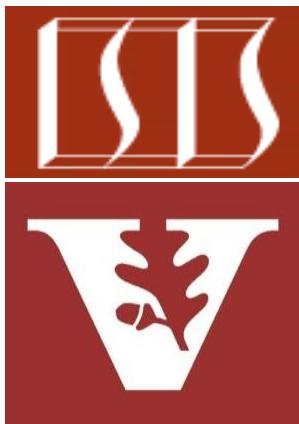
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[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

---

- Motivate the need for Java futures by understanding the pros & cons of synchrony & asynchrony
- Know how Java futures provide the foundation for completable futures in Java 8
- Motivate the need for Java 8 completable futures by recognizing the pros & cons with Java futures

<<Java Interface>>

 **Future<V>**

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



---

# Motivating the Need for Completable Futures

# Motivating the Need for Completable Futures

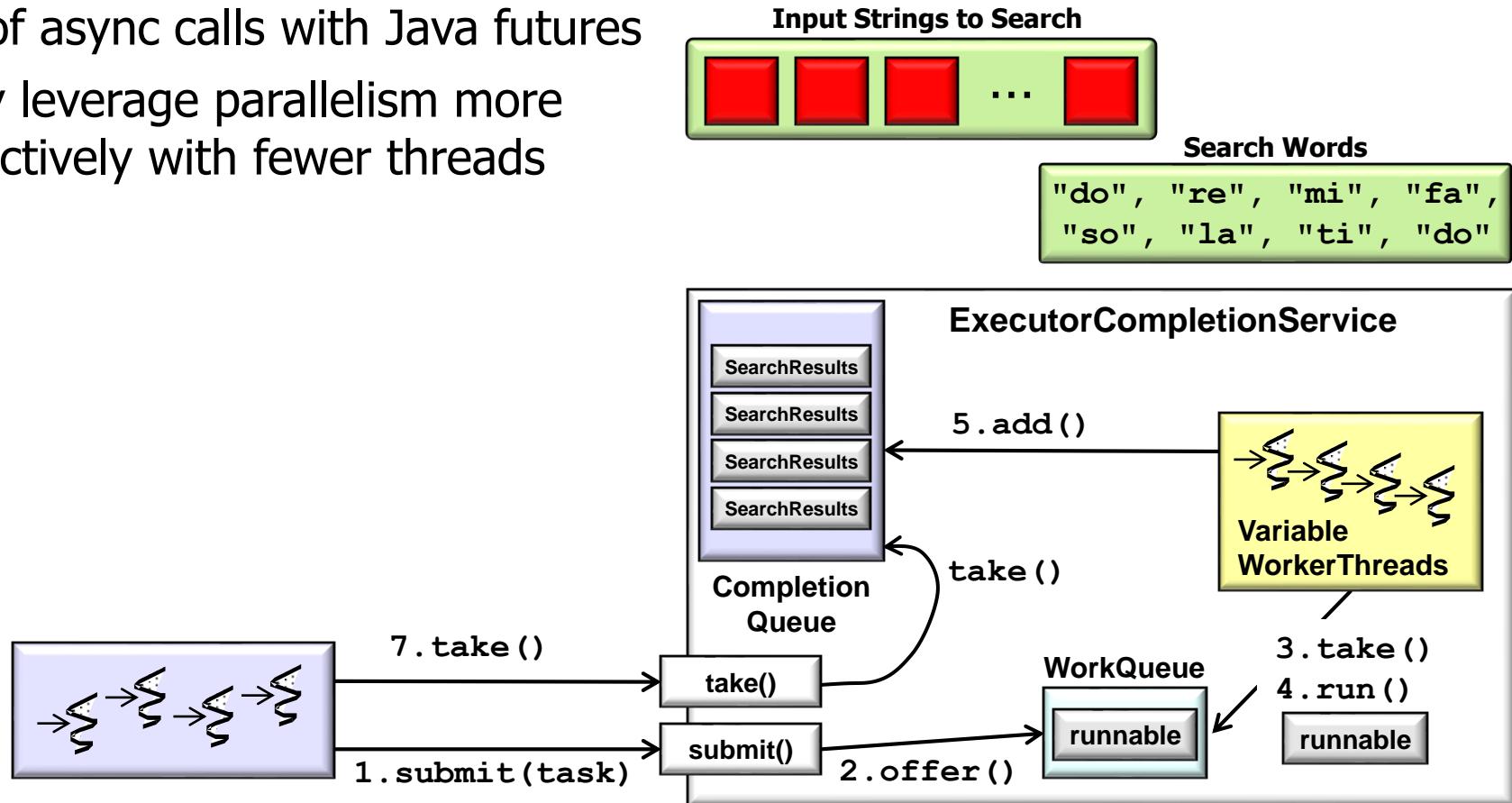
---

- 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



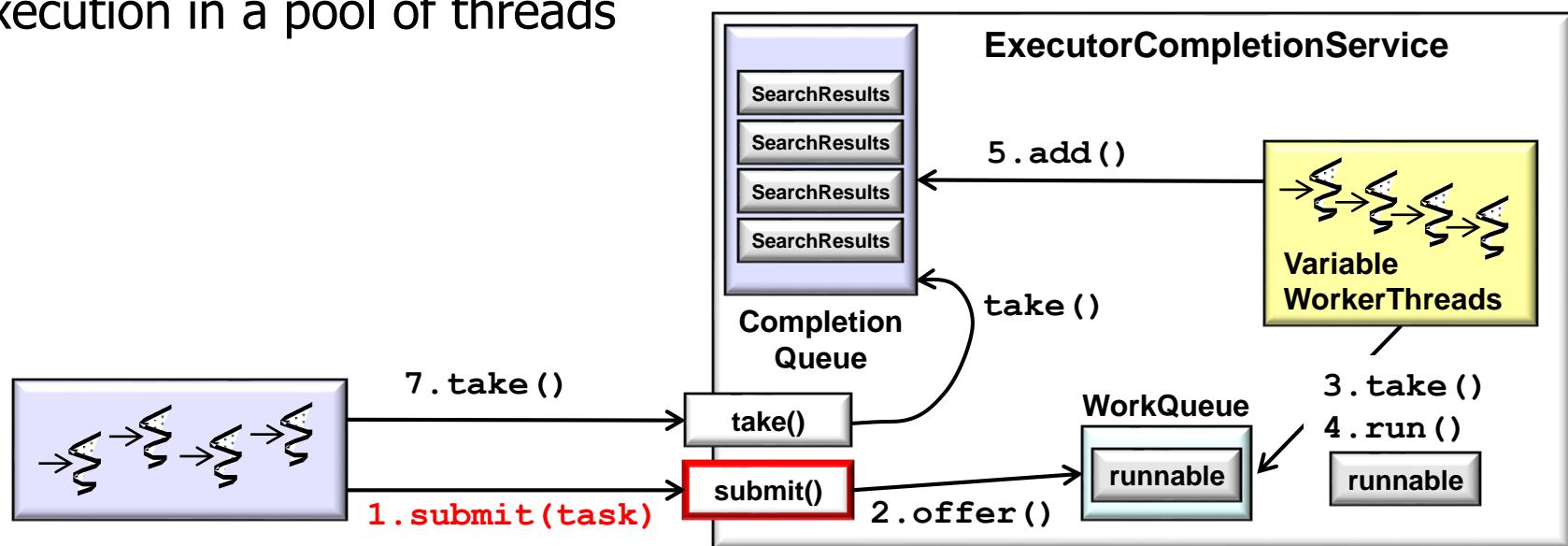
# Motivating the Need for Completable Futures

- 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));

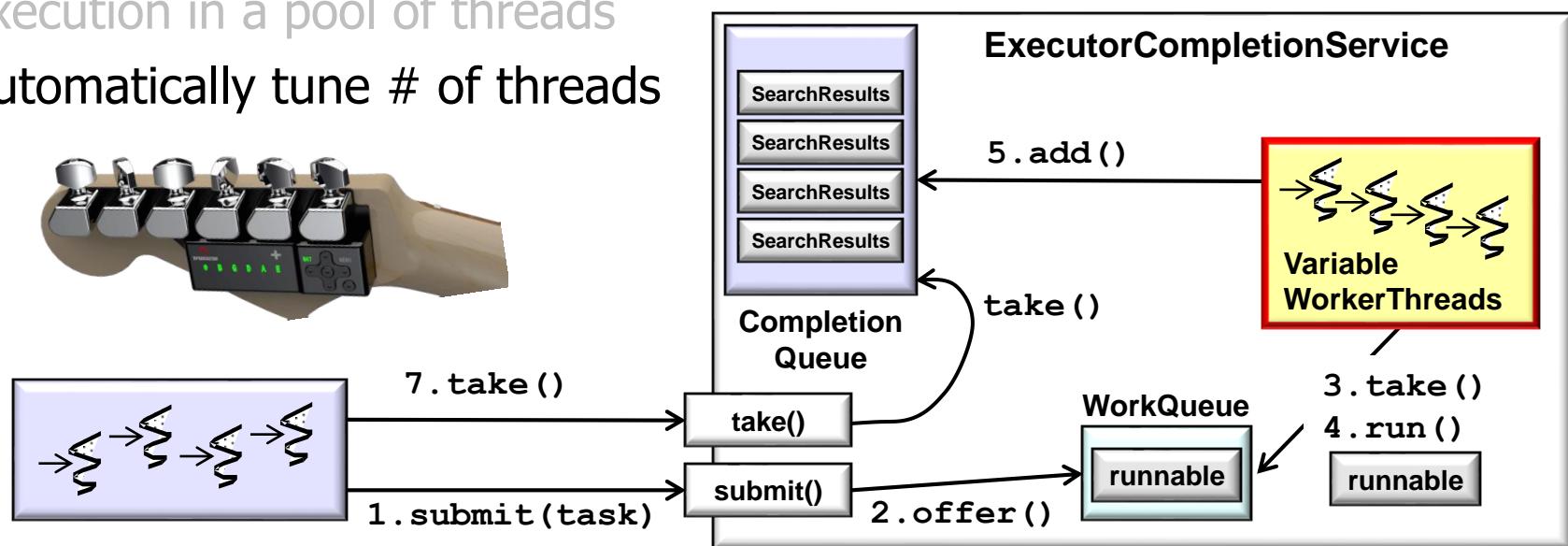


# Motivating the Need for Completable Futures

- 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



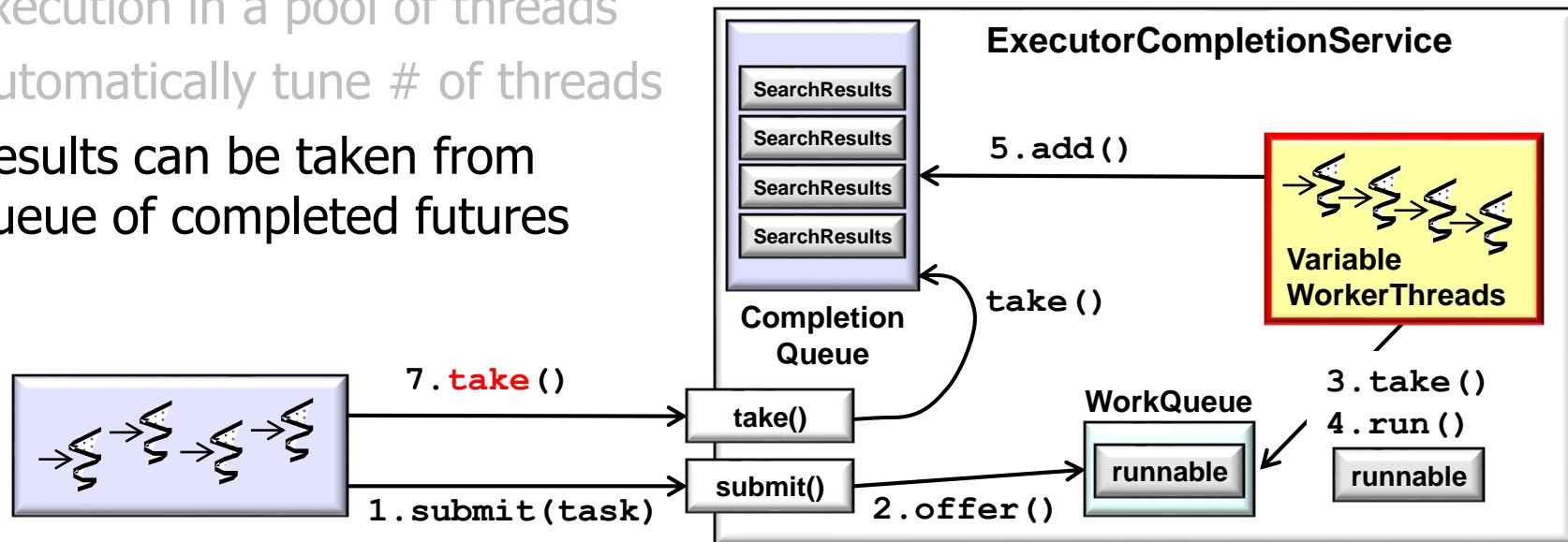
```
mCompletionService  
.submit(() ->  
    searchForWord(word,  
    input));
```



# Motivating the Need for Completable Futures

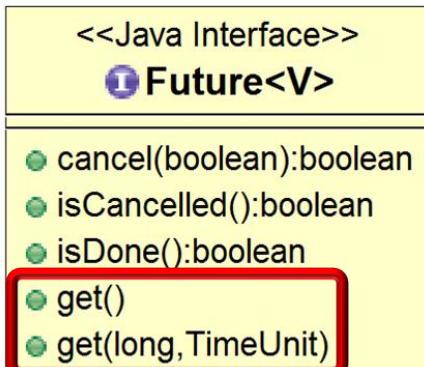
- Pros of async calls with Java futures
  - May leverage parallelism more effectively with fewer threads, e.g.,
    - Queue async computations for execution in a pool of threads
    - Automatically tune # of threads
    - Results can be taken from queue of completed futures

```
Future<SearchResults> resultF =  
    mCompletionService.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

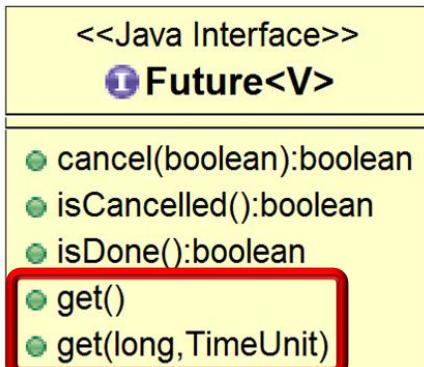


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

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 poll or time-block

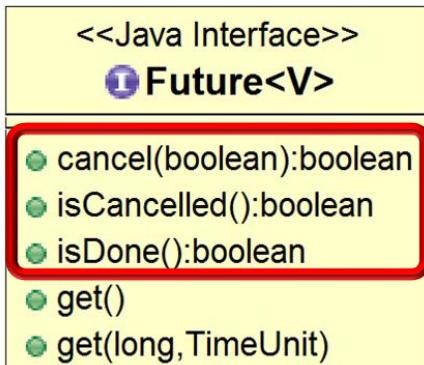


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

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

# Motivating the Need for Completable Futures

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



```
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

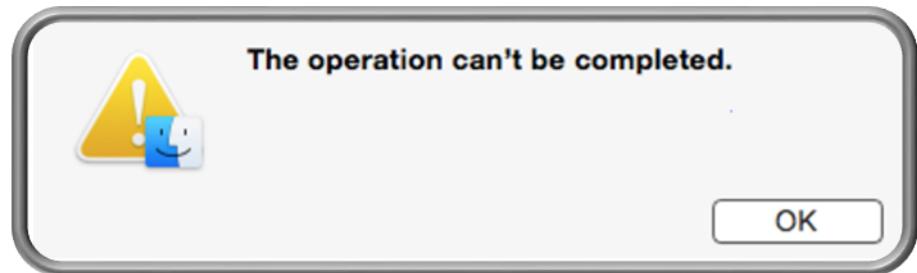
- Cons of async calls with Java futures
  - Limited feature set

<<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



See [en.wikipedia.org/wiki/Fluent\\_interface](https://en.wikipedia.org/wiki/Fluent_interface)

# Motivating the Need for Completable Futures

---

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



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

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

# Motivating the Need for Completable Futures

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



"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

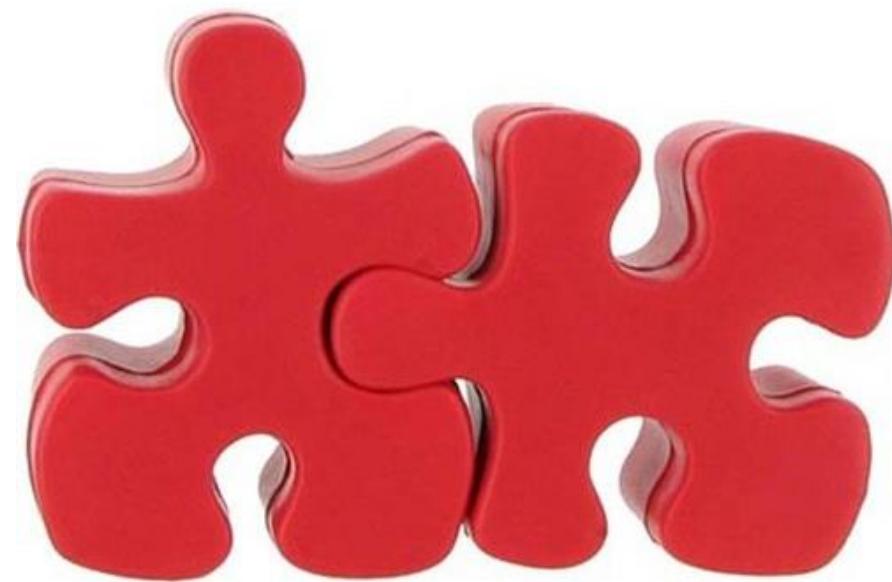
- Cons of async calls with Java futures
    - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently to handle async results
    - *Cannot* be triggered reactively
    - *Cannot* be treated efficiently as a *collection* of futures
- ```
Future<BigFraction> future1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

*Can't wait efficiently for the completion of whichever async computation finishes first*

# Motivating the Need for Completable Futures

---

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



---

In general, it's awkward & inefficient to “compose” multiple futures

---

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

# 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

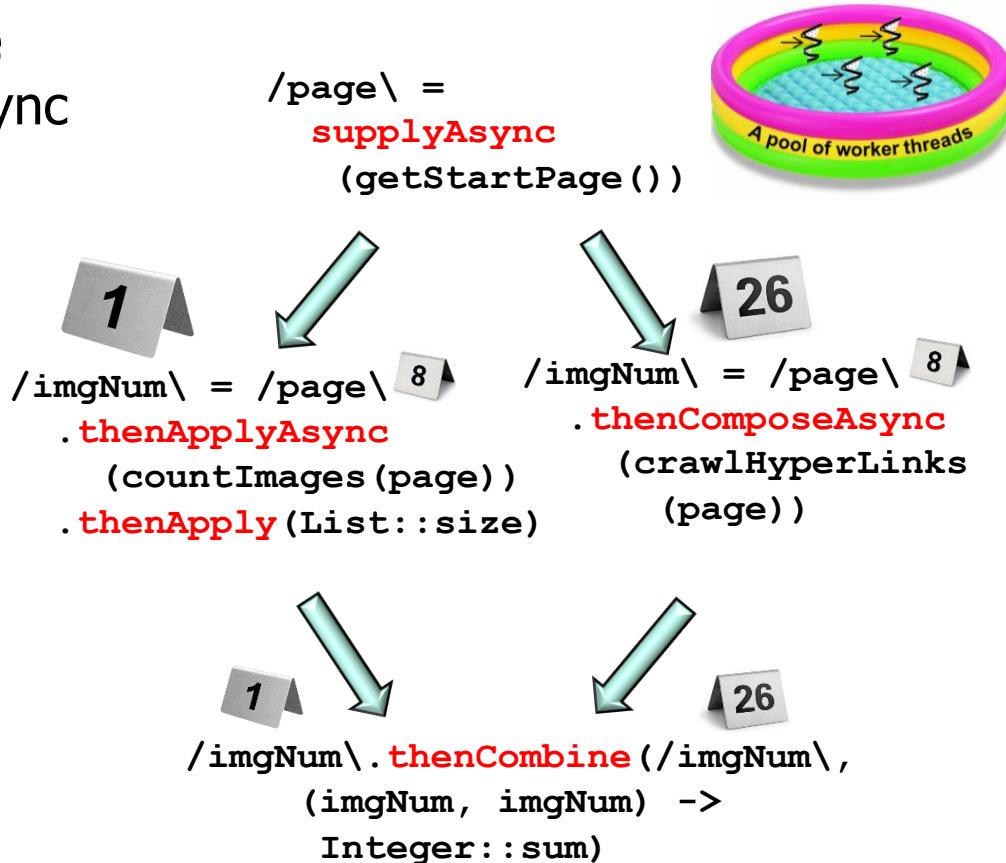
Institute for Software  
Integrated Systems

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 5 futures

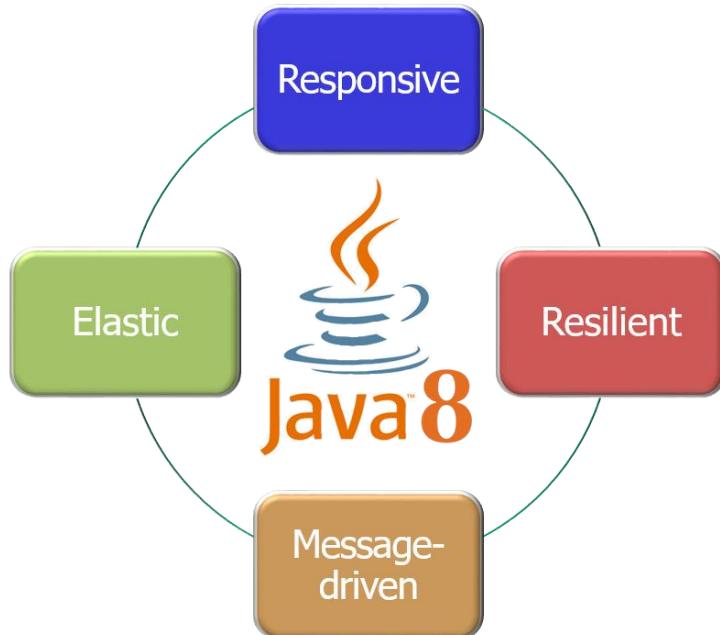


---

# Overview of Completable Futures

# Overview of Completable Futures

- Java's 8 completable future framework provides an asynchronous & reactive 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>
```

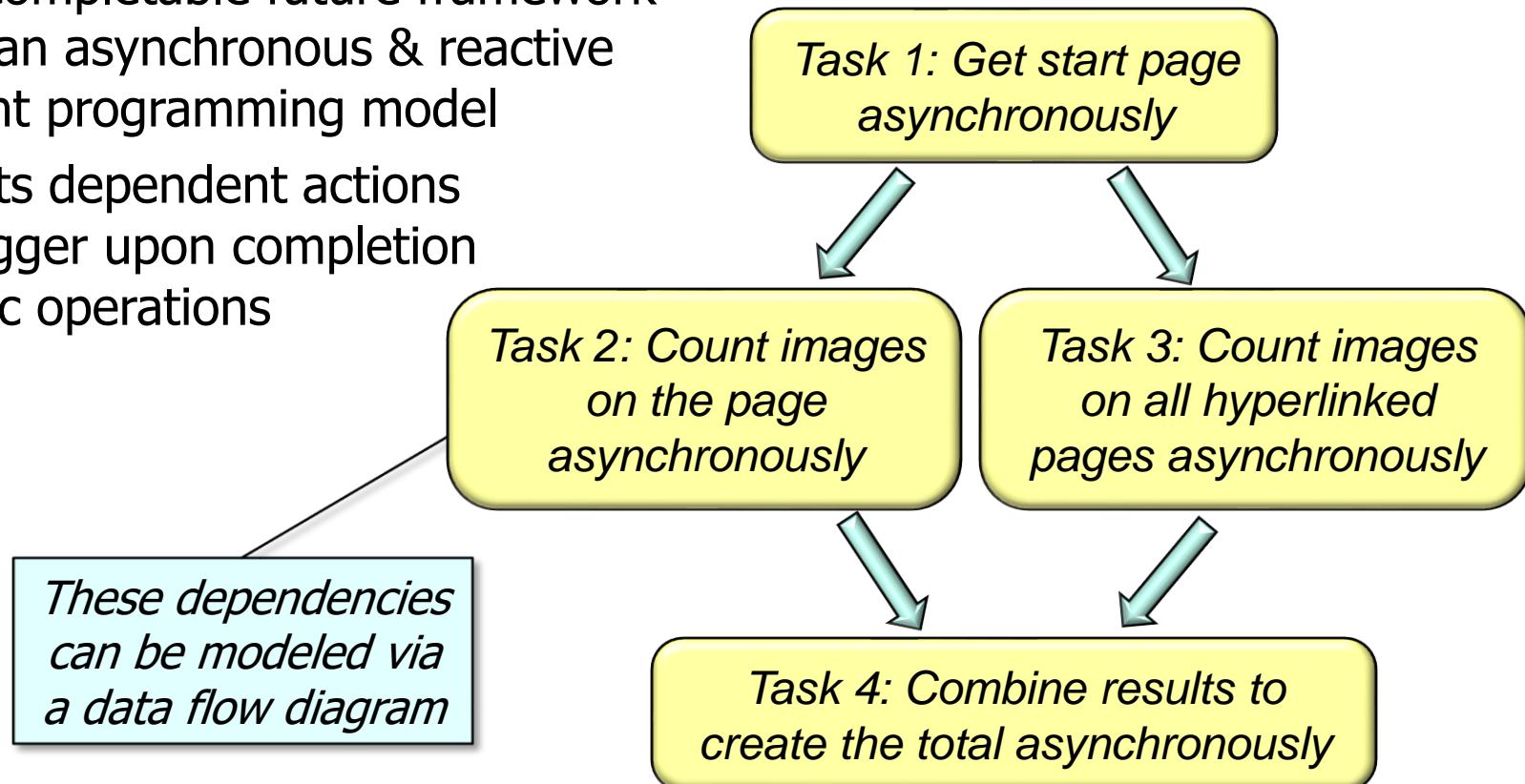
A `Future` that may be explicitly completed (setting its value and status), and may be used as a `CompletionStage`, supporting dependent functions and actions that trigger upon its completion.

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:

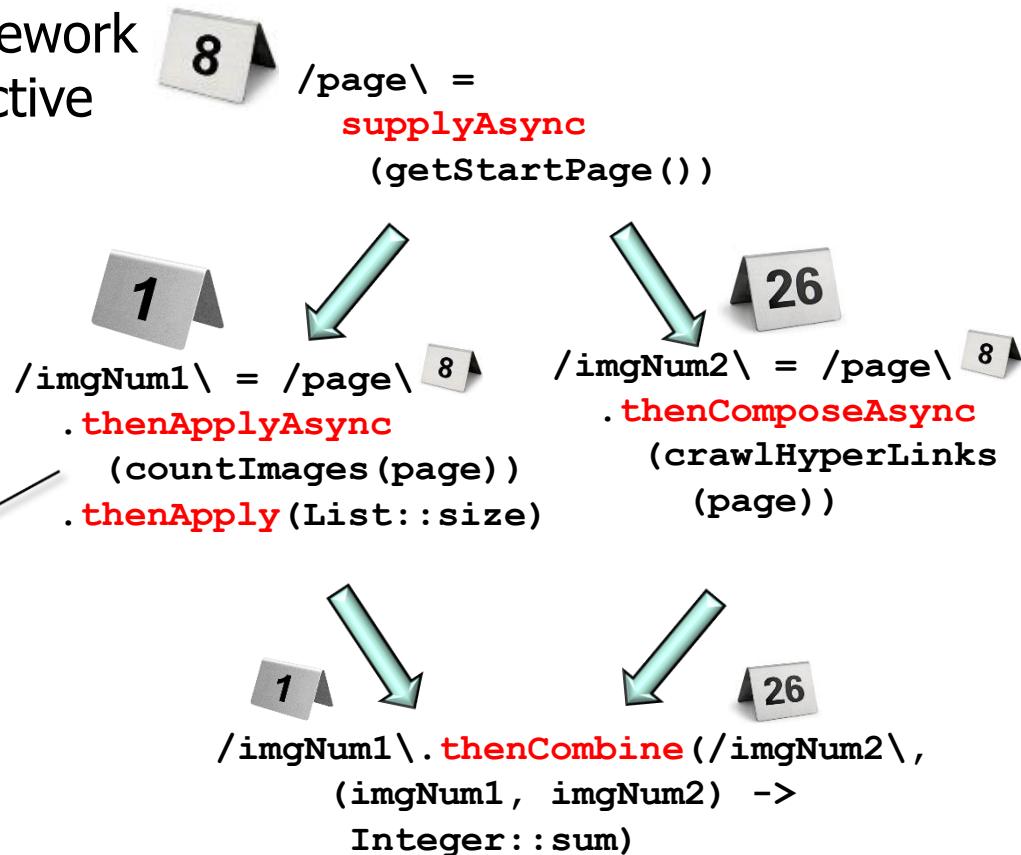
# Overview of Completable Futures

- Java's 8 completable future framework provides an asynchronous & reactive concurrent programming model
  - Supports dependent actions that trigger upon completion of async operations



# Overview of Completable Futures

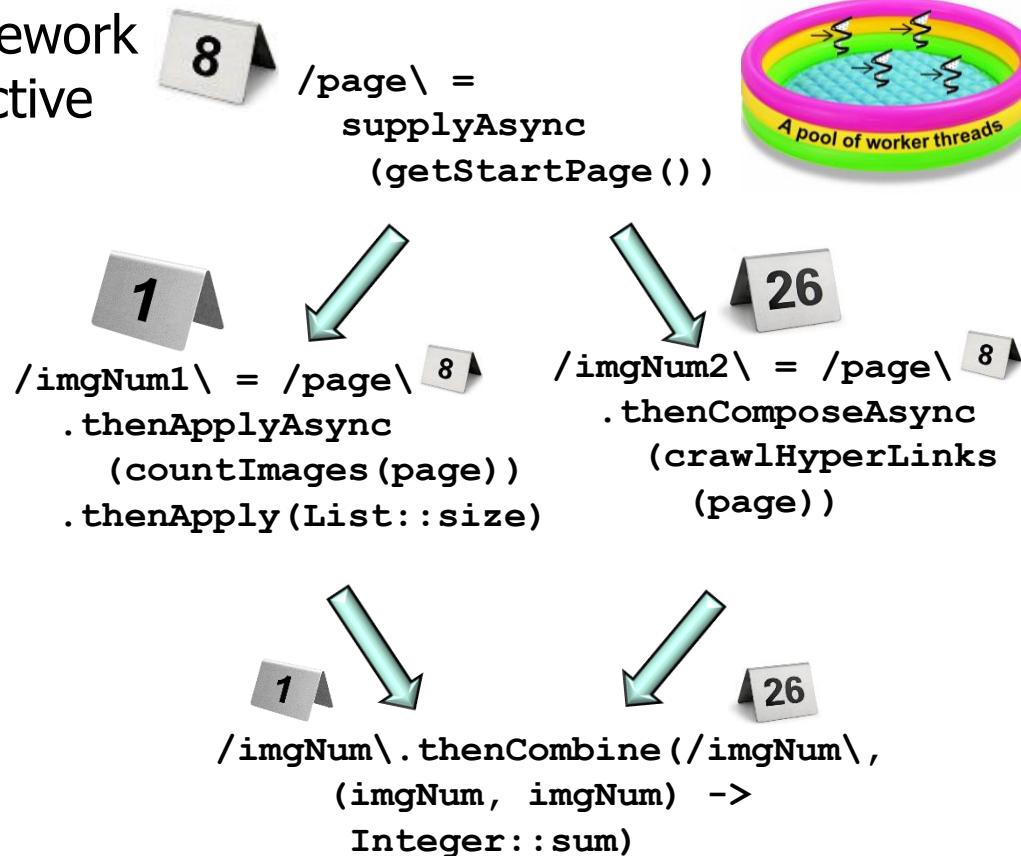
- Java's 8 completable future framework provides an asynchronous & reactive concurrent programming model
  - Supports dependent actions that trigger upon completion of async operations



*Async operations can be forked, chained, & joined*

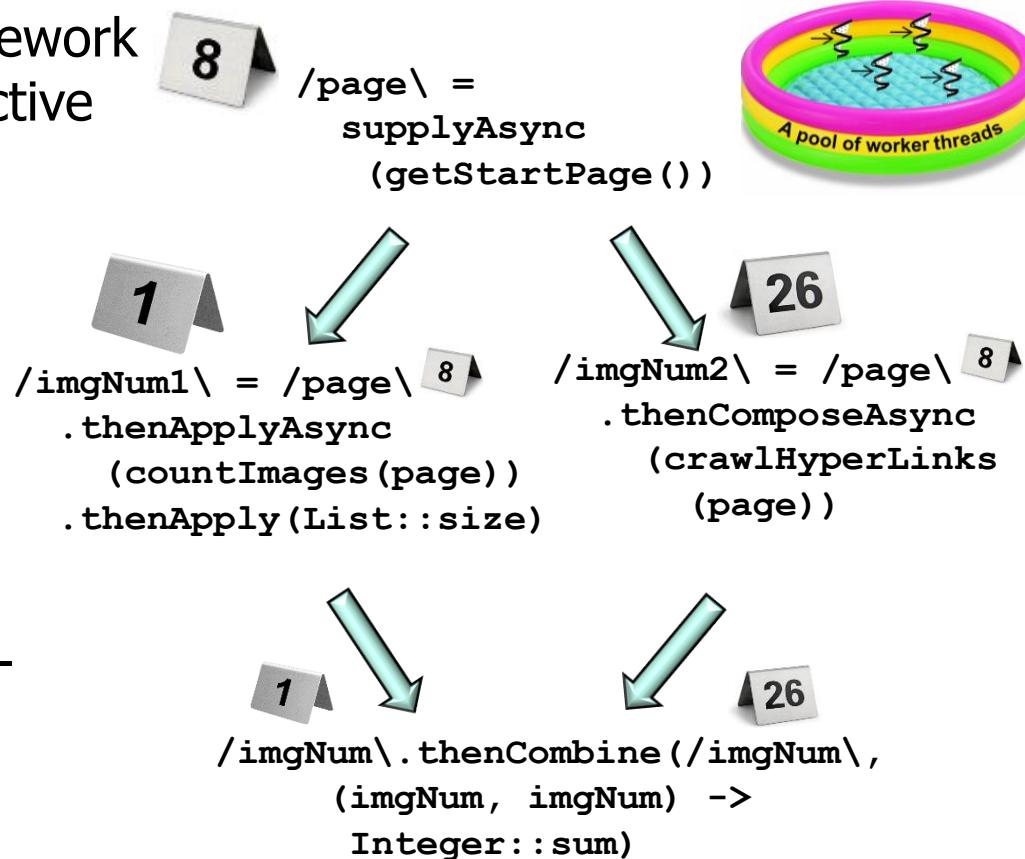
# Overview of Completable Futures

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



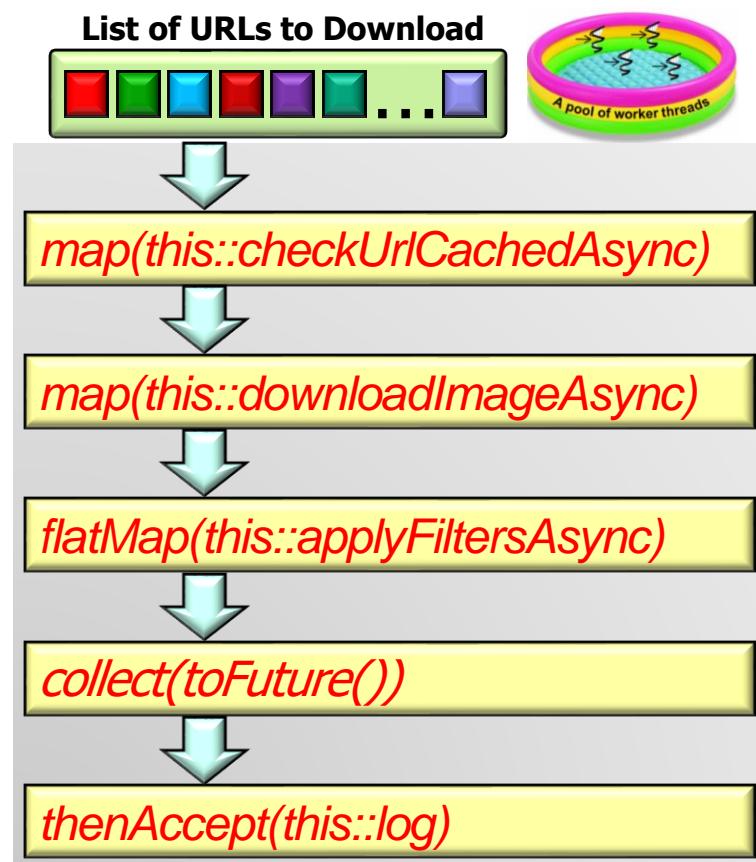
# Overview of Completable Futures

- Java's 8 completable future framework provides an asynchronous & reactive 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 various types of user-designed thread pools



# Overview of Completable Futures

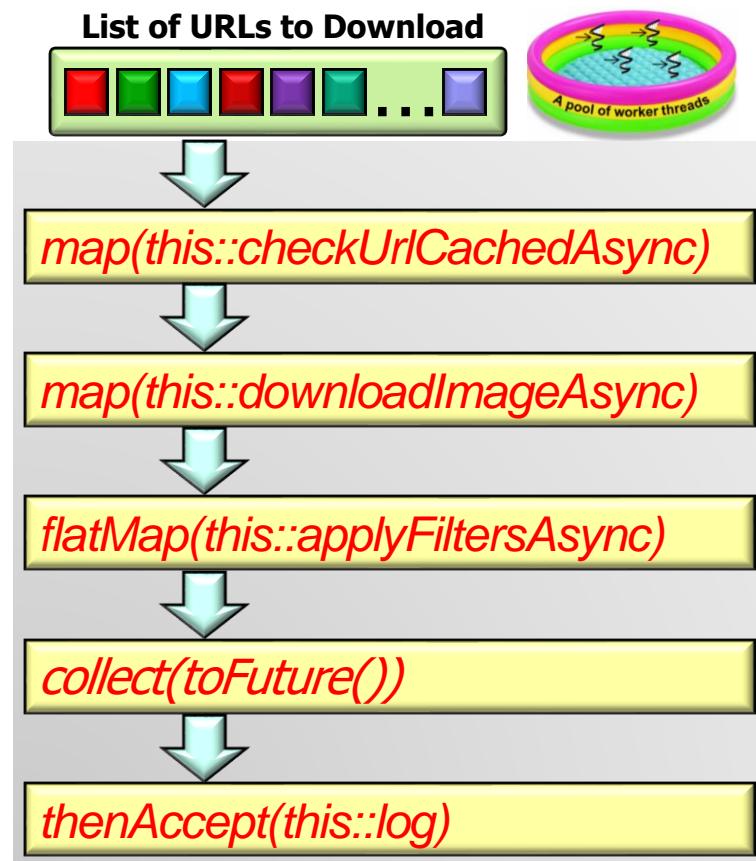
- Java 8 completable futures, streams, & functional programming features can be combined to good effects!!



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

# Overview of Completable Futures

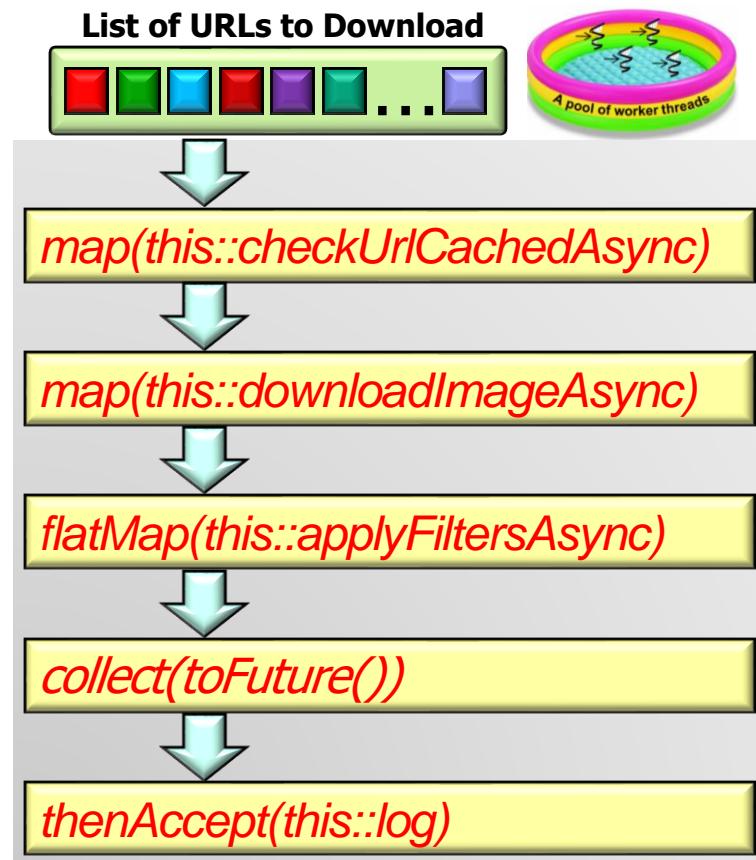
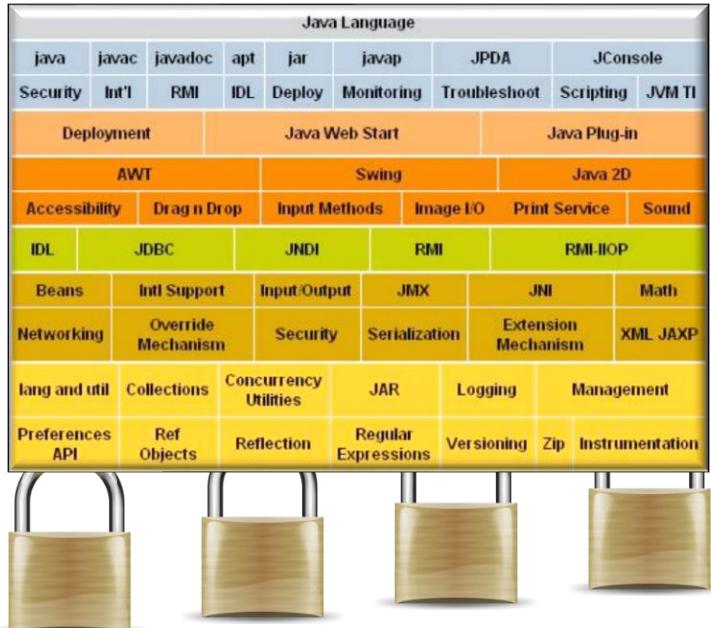
- Java 8 completable futures often need no explicit synchronization or threading when developing concurrent apps!



Alleviates many accidental & inherent complexities of concurrent programming

# Overview of Completable Futures

- Java 8 completable futures often need no explicit synchronization or threading when developing concurrent apps!



Java class libraries handle locking needed to protect shared mutable state

---

# 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 fluently to handle async results efficiently & cleanly



`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 fluently to handle async results efficiently & cleanly
  - 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);
```

Create a single future that will be triggered when a group of other futures all complete

# Overcoming Limitations with Java Futures

- The completable future framework overcomes Java future limitations
  - Can* be completed explicitly
  - Can* be chained fluently to handle async results efficiently & cleanly
  - 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 fluently to handle async results efficiently & cleanly
  - 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 Completable Futures (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

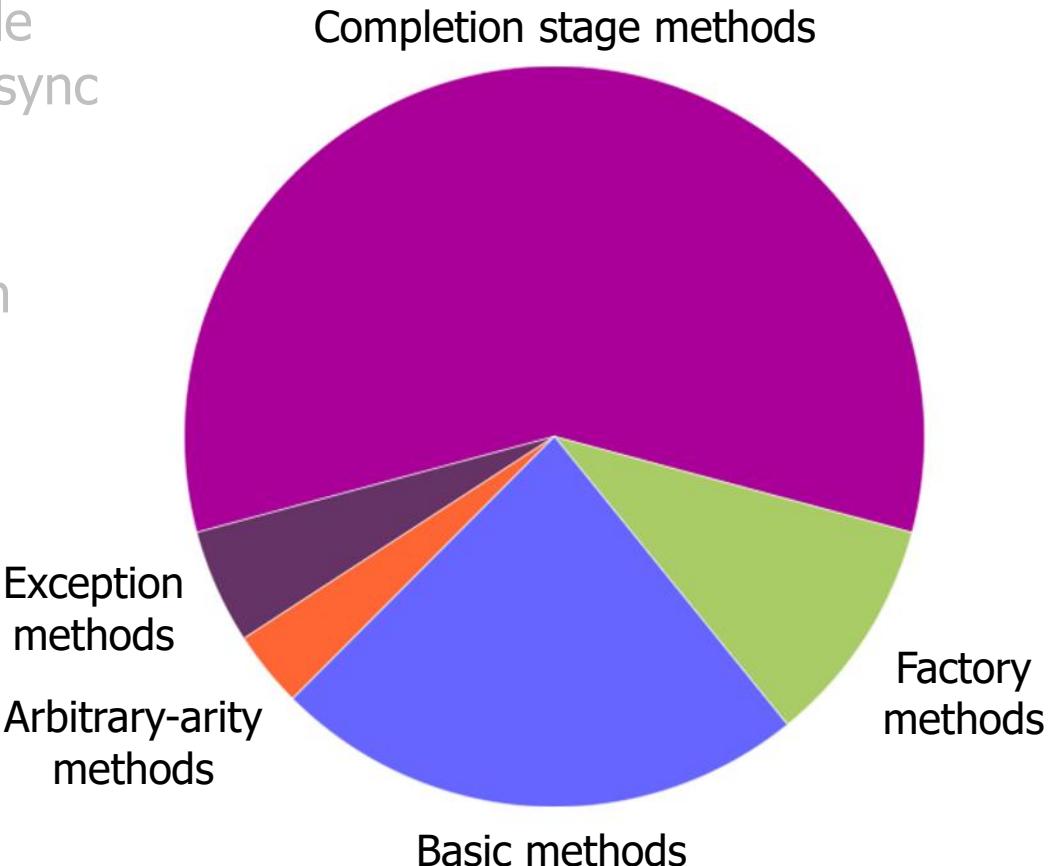
Vanderbilt University  
Nashville, Tennessee, USA



# 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

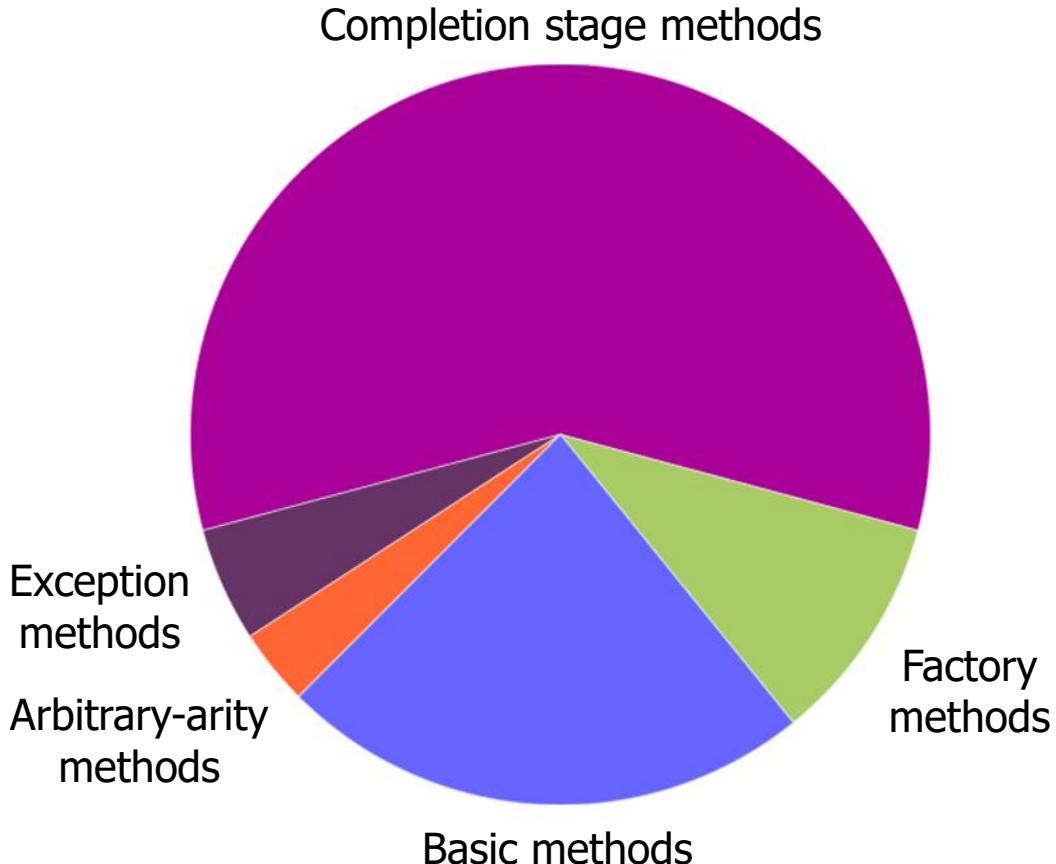
# Grouping the Java 8 CompletableFuture API

- The entire completable future framework resides in 1 public 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>                      |

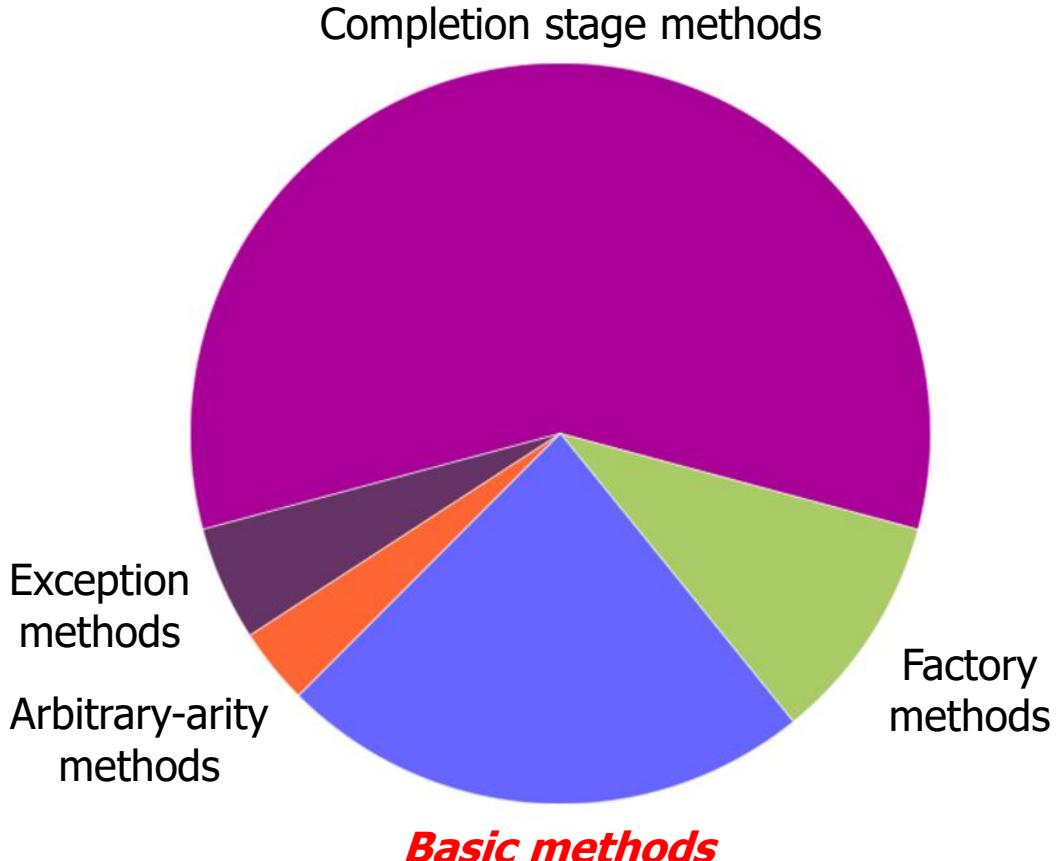
# Grouping the Java 8 CompletableFuture API

- The entire completable future framework resides in 1 public class with 60+ methods!!!
  - It therefore helps to have a “birds-eye” view of this class



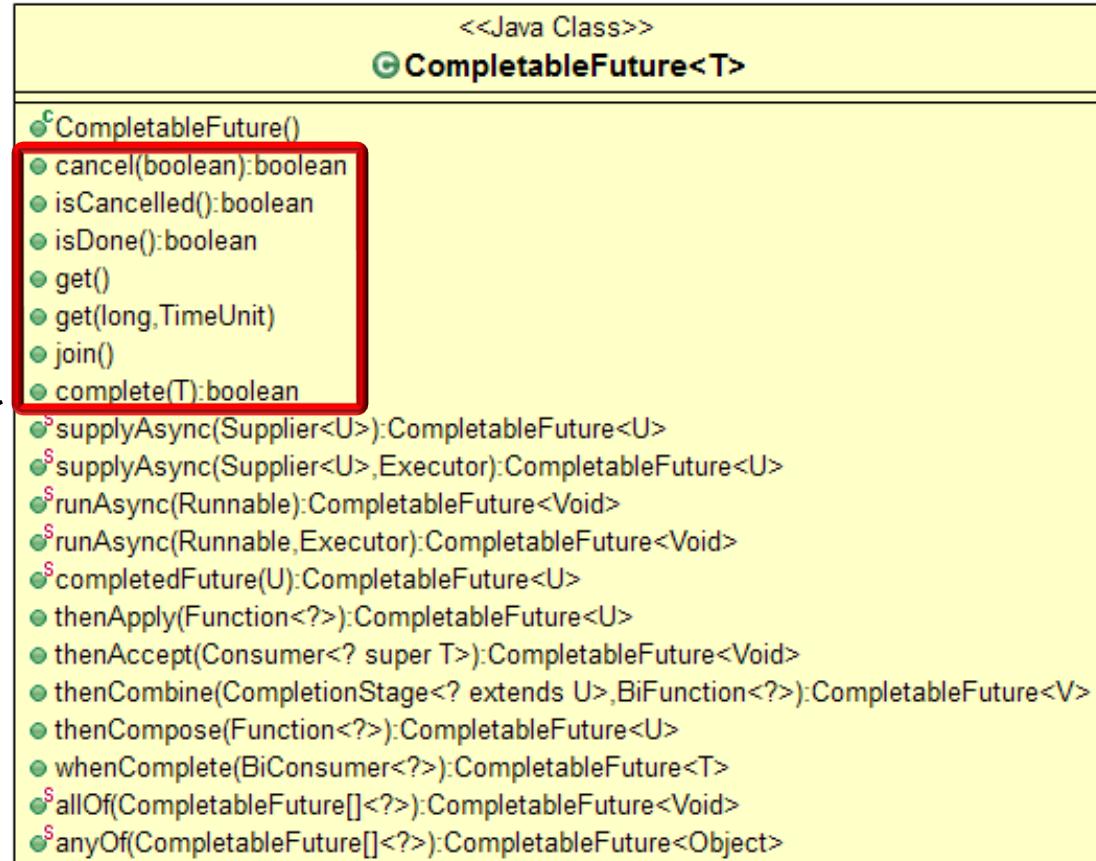
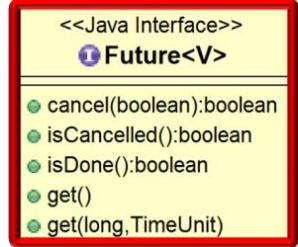
# Grouping the Java 8 CompletableFuture API

- 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



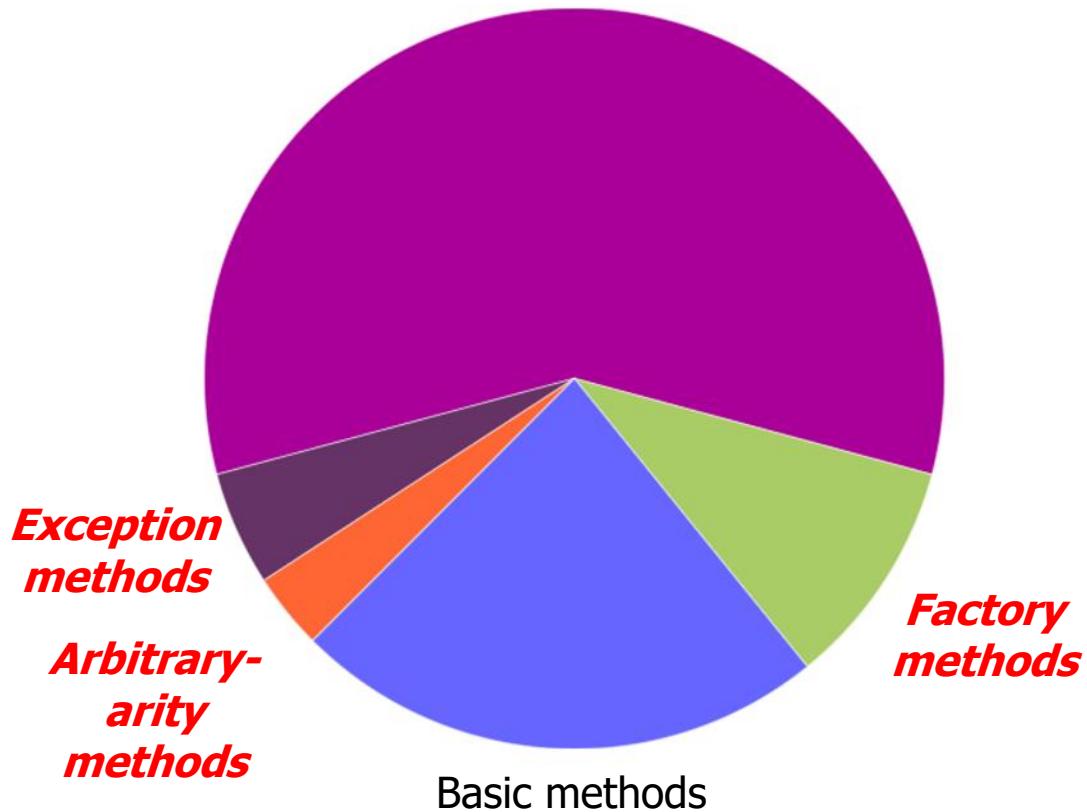
Only slightly better than the conventional Future interface

# Grouping the Java 8 CompletableFuture API

- Other completable future features are more advanced



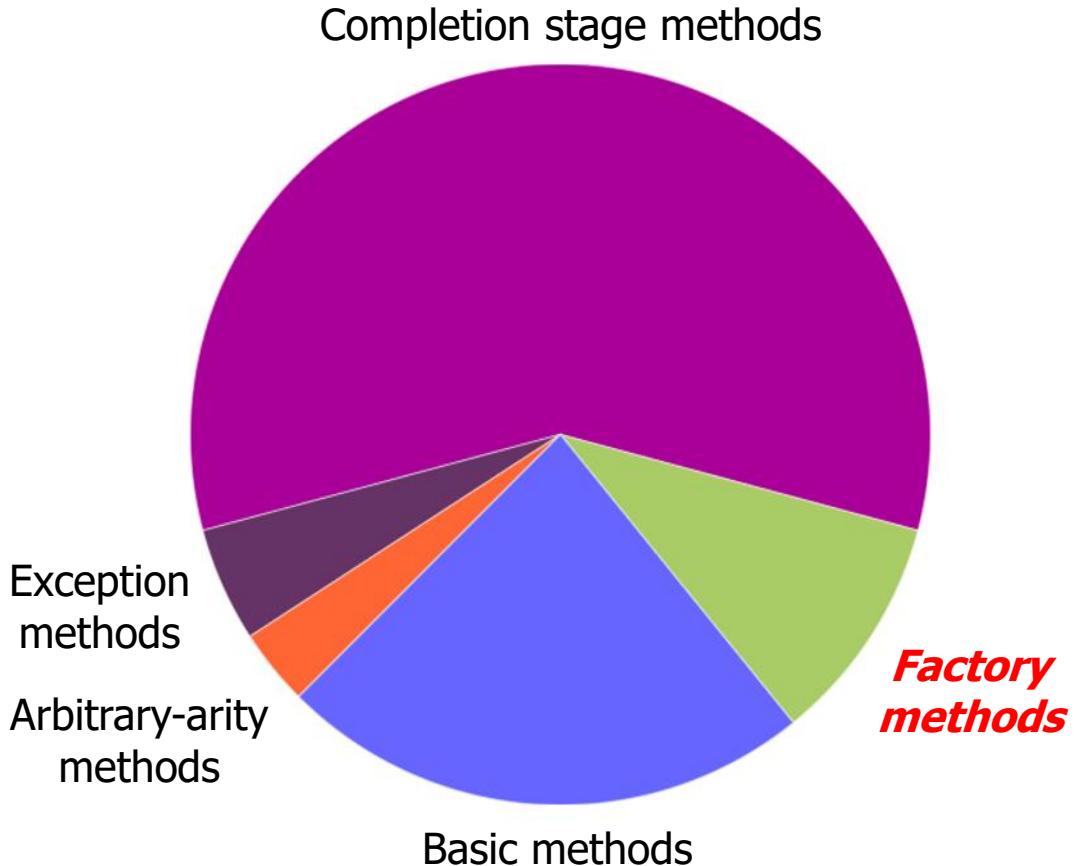
*Completion stage methods*



# 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 without using threads explicitly

<<Java Class>>

**CompletableFuture<T>**

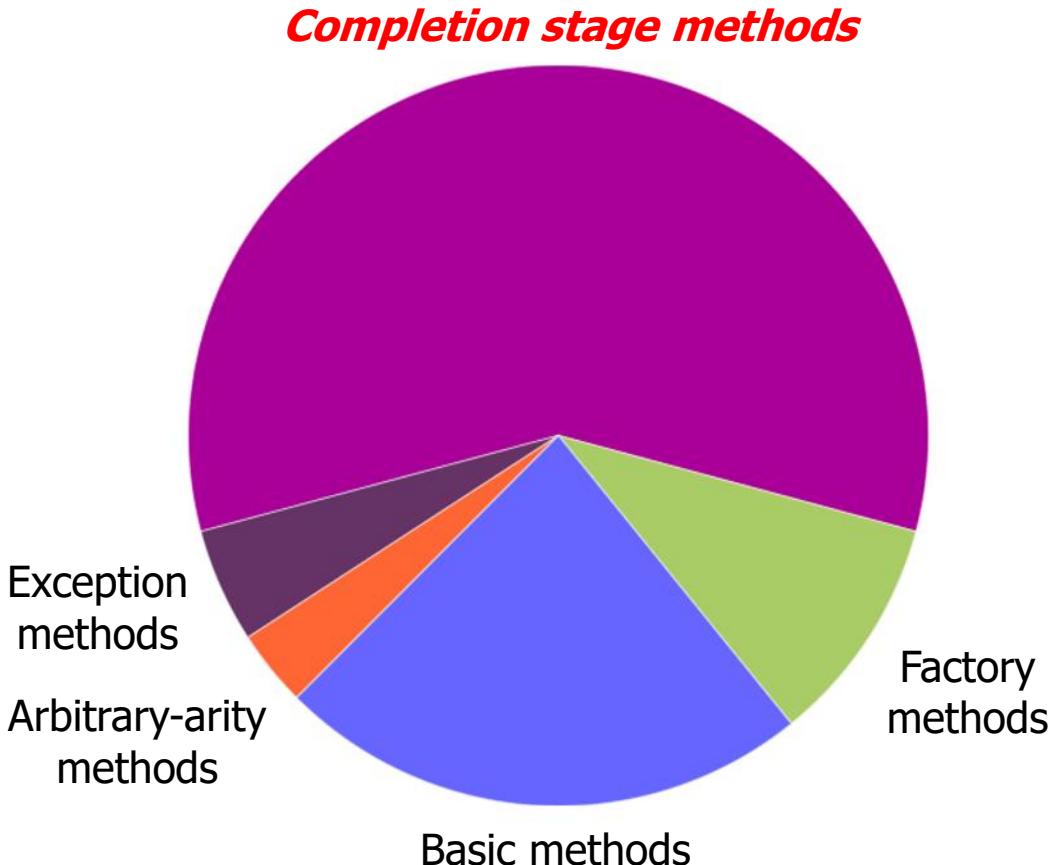
|                                                                                                             |
|-------------------------------------------------------------------------------------------------------------|
| <code>CompletableFuture()</code>                                                                            |
| <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>                            |

Help make programs more *elastic* by leveraging a pool of worker threads

# 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
    - Chain together actions that perform async result processing & composition



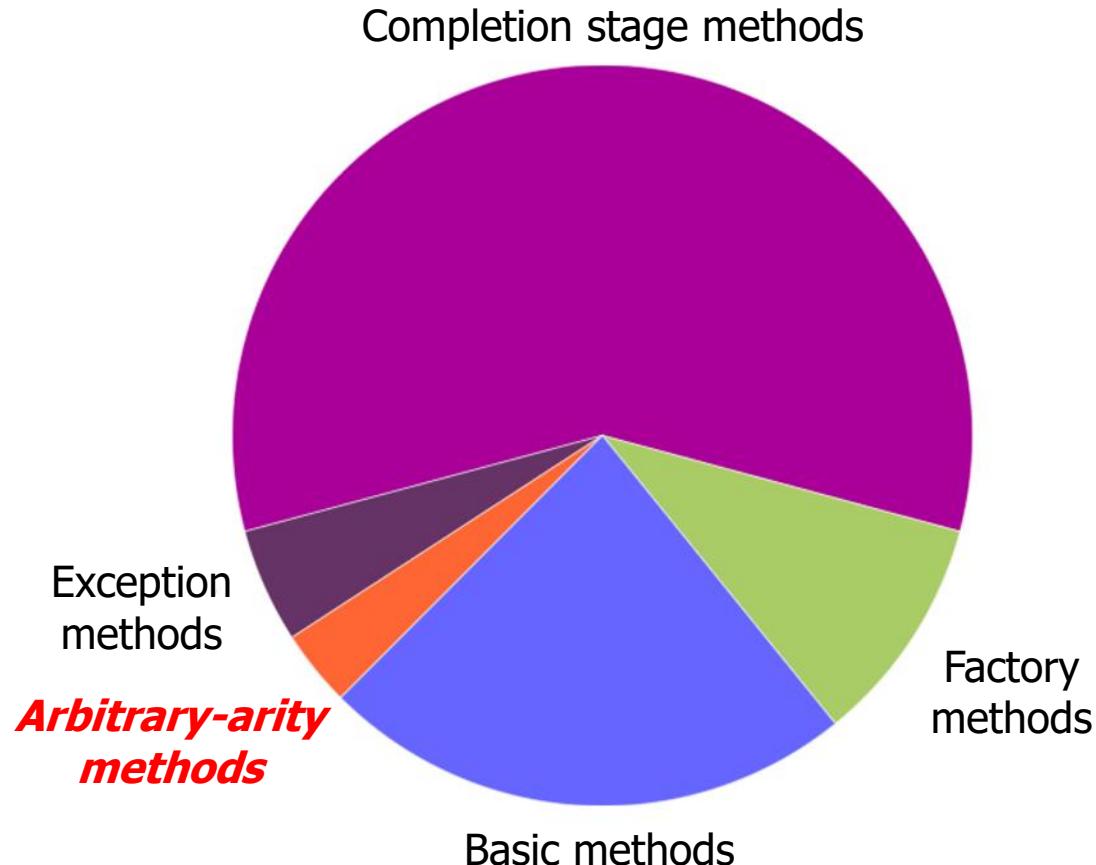
| <<Java Interface>>   |                                                                  |
|----------------------|------------------------------------------------------------------|
| I CompletionStage<T> |                                                                  |
| •                    | thenApply(Function<?>):CompletionStage<U>                        |
| •                    | thenAccept(Consumer<?>):CompletionStage<Void>                    |
| •                    | thenCombine(CompletionStage<?>,BiFunction<?>):CompletionStage<V> |
| •                    | thenCompose(Function<?>):CompletionStage<U>                      |
| •                    | whenComplete(BiConsumer<?>):CompletionStage<T>                   |

| <<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>                      |

Help make programs more *responsive* by not blocking user code

# 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



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

# 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>>

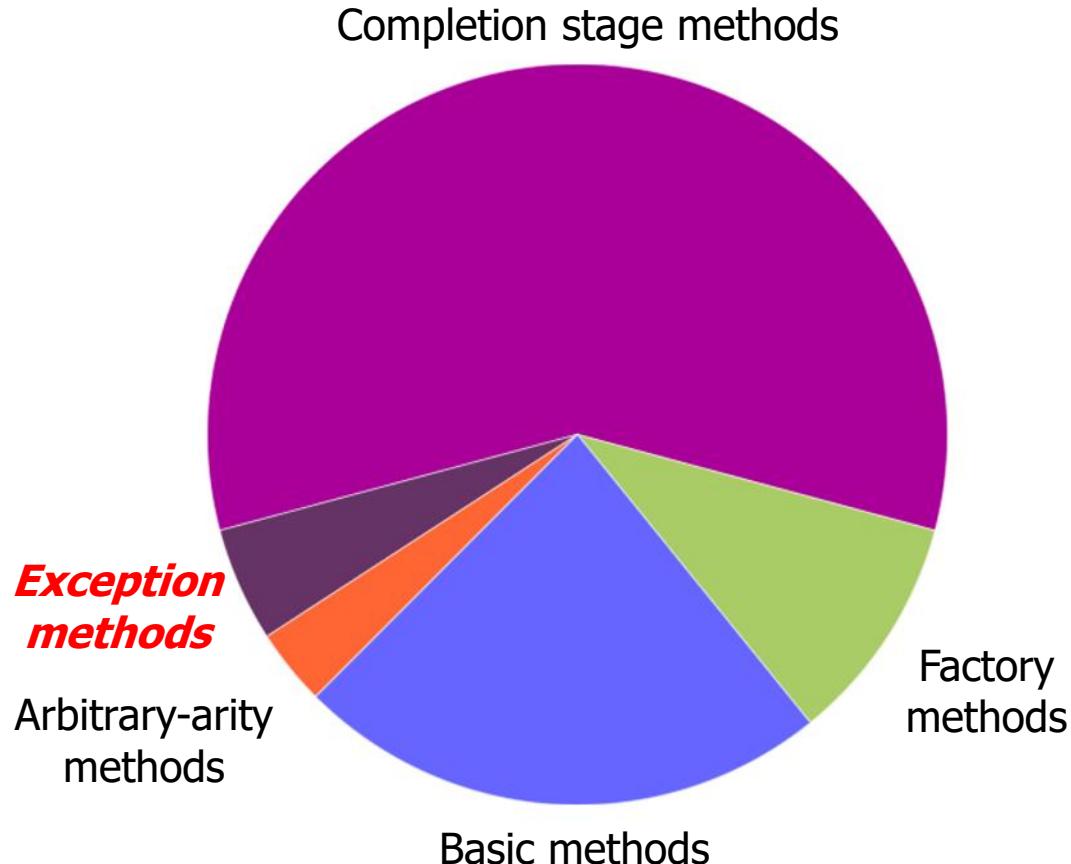
**CompletableFuture<T>**

|                                                                                                               |
|---------------------------------------------------------------------------------------------------------------|
| • <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>                            |

Help make programs more *responsive* by not blocking user code

# 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
  - Exception methods



# 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
  - Exception methods
    - Handle exceptional conditions at runtime

<<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>                      |

Help make programs more *resilient* by handling erroneous computations gracefully

---

# End of Overview of Java 8 Completable Futures (Part 2)

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

**Douglas C. Schmidt**

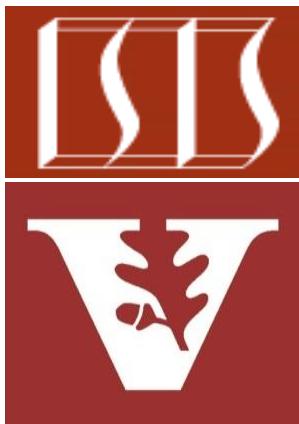
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- 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.

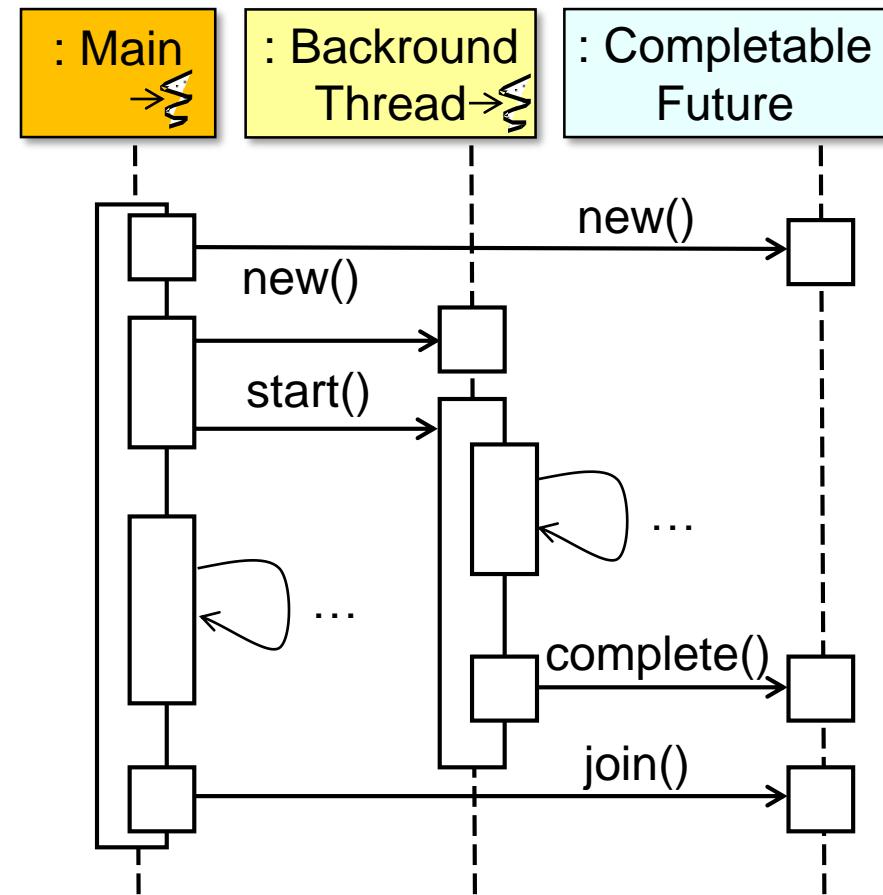
In addition to these and related methods for directly manipulating status and results, **CompletableFuture** implements interface **CompletionStage** with the following policies:

---

# Basic Completable Future Features

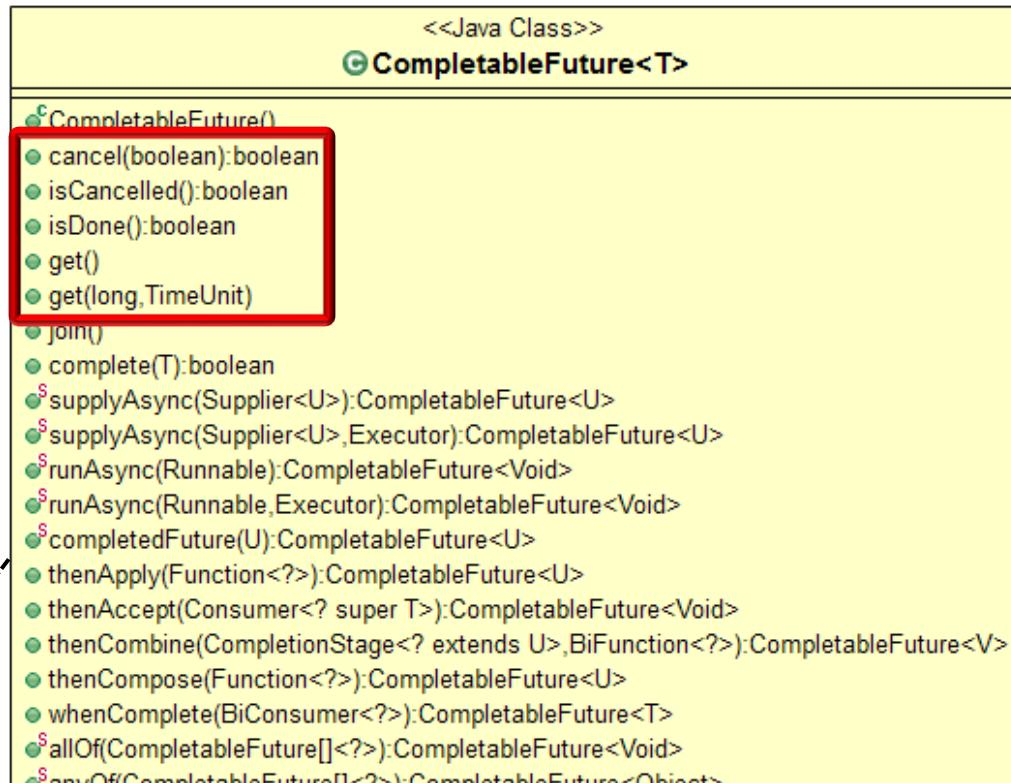
# Basic Completable Future Features

- 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";

ForkJoinTask<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

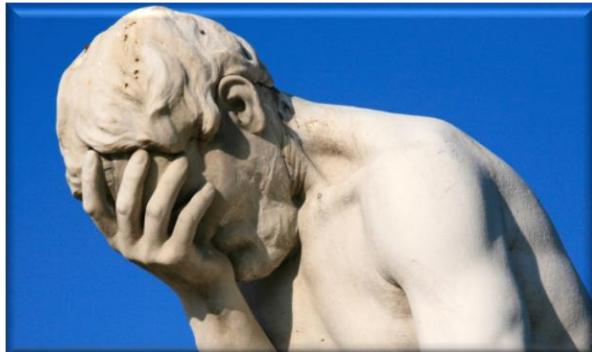
```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

ForkJoinTask<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..

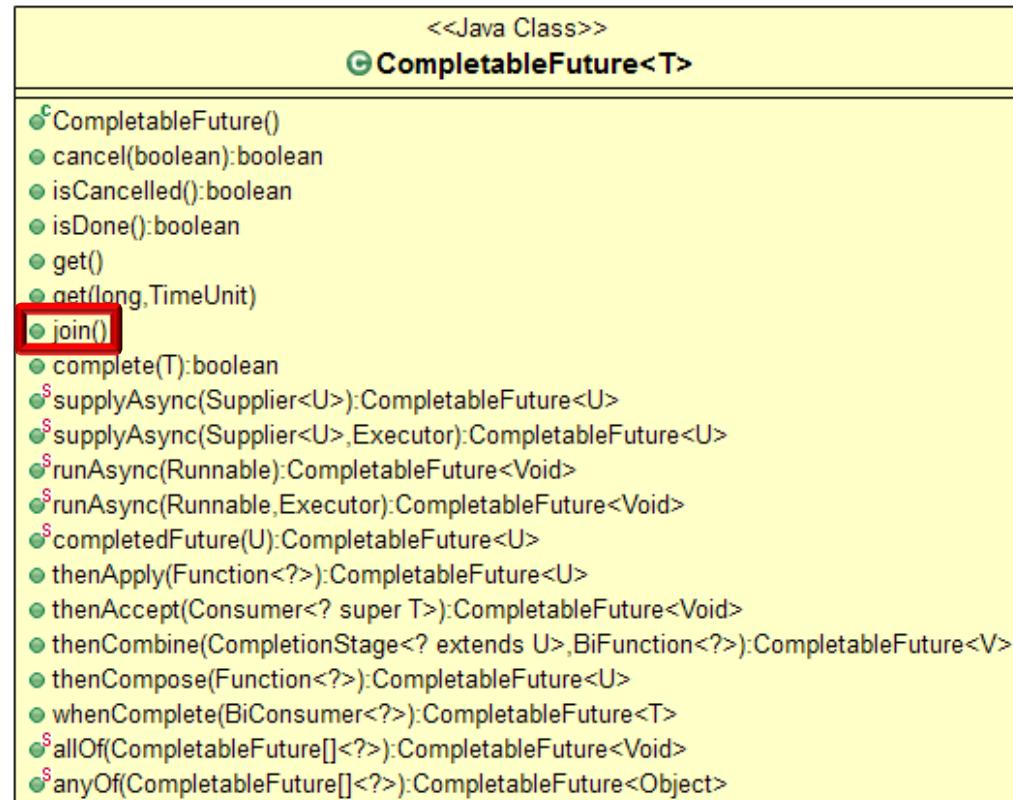


```
String f1 = "62675744/15668936";
String f2 = "609136/913704";

ForkJoinTask<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»

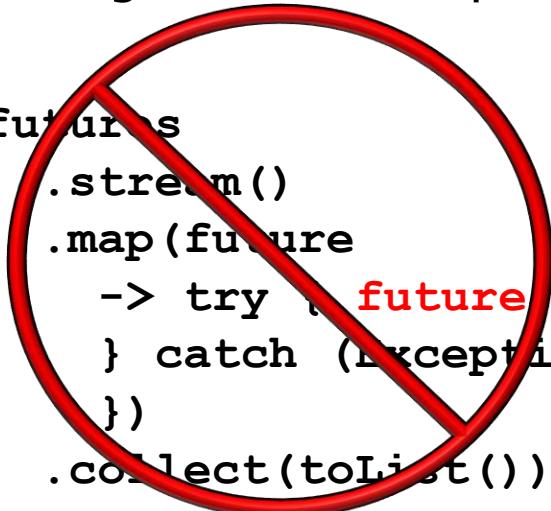
**CompletableFuture<T>**

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

Future::join can be used as a method reference in a Java 8 stream

# 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

<<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>                                                                                         |
| • <b>complete(T):boolean</b>                                                                                  |
| • <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>                            |

# 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<>();  
  
final 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**

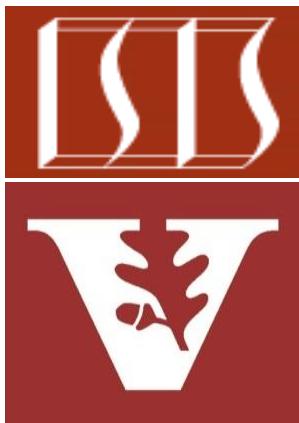
**d.schmidt@vanderbilt.edu**

**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 operate on big fractions

|                                                                                        |
|----------------------------------------------------------------------------------------|
| <<Java Class>>                                                                         |
| <b>BigFraction</b>                                                                     |
| <span style="color: red;">F</span> mNumerator: BigInteger                              |
| <span style="color: red;">F</span> mDenominator: BigInteger                            |
| <span style="color: green;">C</span> BigFraction()                                     |
| <span style="color: green;">S</span> <u>valueOf(Number):BigFraction</u>                |
| <span style="color: green;">S</span> <u>valueOf(Number,Number):BigFraction</u>         |
| <span style="color: green;">S</span> <u>valueOf(String):BigFraction</u>                |
| <span style="color: green;">S</span> <u>valueOf(Number,Number,boolean):BigFraction</u> |
| <span style="color: green;">S</span> <u>reduce(BigFraction):BigFraction</u>            |
| <span style="color: green;">S</span> <u>getNumerator():BigInteger</u>                  |
| <span style="color: green;">S</span> <u>getDenominator():BigInteger</u>                |
| <span style="color: green;">S</span> <u>add(Number):BigFraction</u>                    |
| <span style="color: green;">S</span> <u>subtract(Number):BigFraction</u>               |
| <span style="color: green;">S</span> <u>multiply(Number):BigFraction</u>               |
| <span style="color: green;">S</span> <u>divide(Number):BigFraction</u>                 |
| <span style="color: green;">S</span> <u>gcd(Number):BigFraction</u>                    |
| <span style="color: green;">S</span> <u>toMixedString():String</u>                     |

# Learning Objectives in this Part of the Lesson

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

---

# 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>BigFraction</b>                                      |
|  <code>mNumerator: BigInteger</code>                     |
|  <code>mDenominator: BigInteger</code>                   |
|  <code>BigFraction()</code>                              |
|  <code>valueOf(Number):BigFraction</code>                |
|  <code>valueOf(Number,Number):BigFraction</code>         |
|  <code>valueOf(String):BigFraction</code>                |
|  <code>valueOf(Number,Number,boolean):BigFraction</code> |
|  <code>reduce(BigFraction):BigFraction</code>            |
|  <code>getNumerator():BigInteger</code>                  |
|  <code>getDenominator():BigInteger</code>                |
|  <code>add(Number):BigFraction</code>                    |
|  <code>subtract(Number):BigFraction</code>               |
|  <code>multiply(Number):BigFraction</code>               |
|  <code>divide(Number):BigFraction</code>                 |
|  <code>gcd(Number):BigFraction</code>                    |
|  <code>toMixedString():String</code>                    |

See [LiveLessons/blob/master/Java8/ex8/src/utils/BigFraction.java](#)

# 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**

■ mNumerator: BigInteger  
■ mDenominator: BigInteger

■ BigFraction()  
■ valueOf(Number):BigFraction  
■ valueOf(Number,Number):BigFraction  
■ valueOf(String):BigFraction  
■ valueOf(Number,Number,boolean):BigFraction  
■ reduce(BigFraction):BigFraction  
■ 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, e.g.
    - $44/55 \rightarrow 4/5$
    - $12/24 \rightarrow 1/2$
    - $144/216 \rightarrow 2/3$

| <<Java Class>>                                                                      |                                                                                                        |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
|                                                                                     |  <b>BigFraction</b> |
|  | mNumerator: BigInteger                                                                                 |
|  | mDenominator: BigInteger                                                                               |
|  | BigFraction()                                                                                          |
|  | valueOf(Number):BigFraction                                                                            |
|  | valueOf(Number,Number):BigFraction                                                                     |
|  | valueOf(String):BigFraction                                                                            |
|  | valueOf(Number,Number,boolean):BigFraction                                                             |
|  | reduce(BigFraction):BigFraction                                                                        |
|  | 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)
    - e.g.,  $12/24 \rightarrow 1/2$

| <<Java Class>>                                                                                                                                                                                                     |                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                    |  <b>BigFraction</b> |
|   mNumerator: BigInteger                     |                                                                                                        |
|   mDenominator: BigInteger                   |                                                                                                        |
|  BigFraction()                                                                                                                  |                                                                                                        |
|   valueOf(Number):BigFraction                |                                                                                                        |
|   valueOf(Number,Number):BigFraction         |                                                                                                        |
|   valueOf(String):BigFraction                |                                                                                                        |
|   valueOf(Number,Number,boolean):BigFraction |                                                                                                        |
|   reduce(BigFraction):BigFraction            |                                                                                                        |
|   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.,  $18/4 \times 2/3 = 3$

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

# 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>BigFraction</b> |
|  | mNumerator: BigInteger                                                                                 |
|  | mDenominator: BigInteger                                                                               |
|  | BigFraction()                                                                                          |
|  | valueOf(Number):BigFraction                                                                            |
|  | valueOf(Number,Number):BigFraction                                                                     |
|  | valueOf(String):BigFraction                                                                            |
|  | valueOf(Number,Number,boolean):BigFraction                                                             |
|  | reduce(BigFraction):BigFraction                                                                        |
|  | 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

- Multiplying big fractions w/a completable future

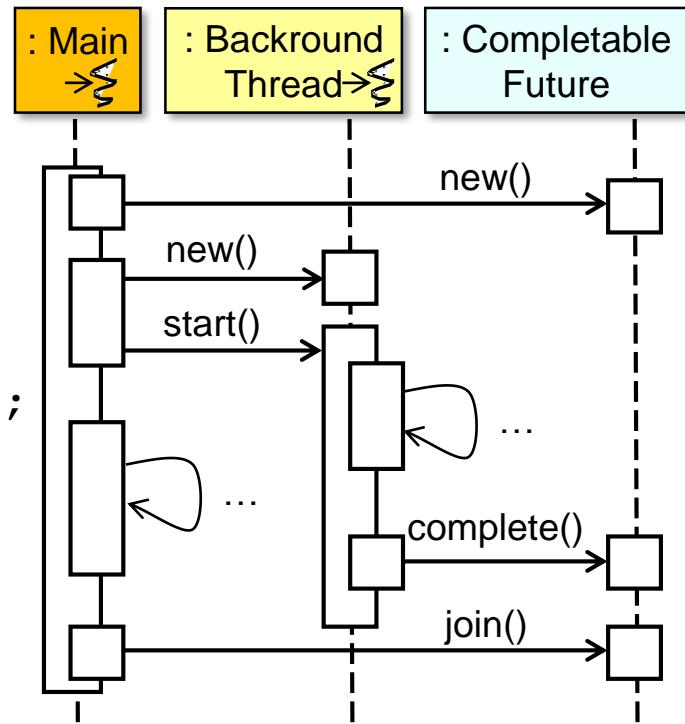
```
CompletableFuture<BigFraction> future  
    = new CompletableFuture<>();
```

```
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

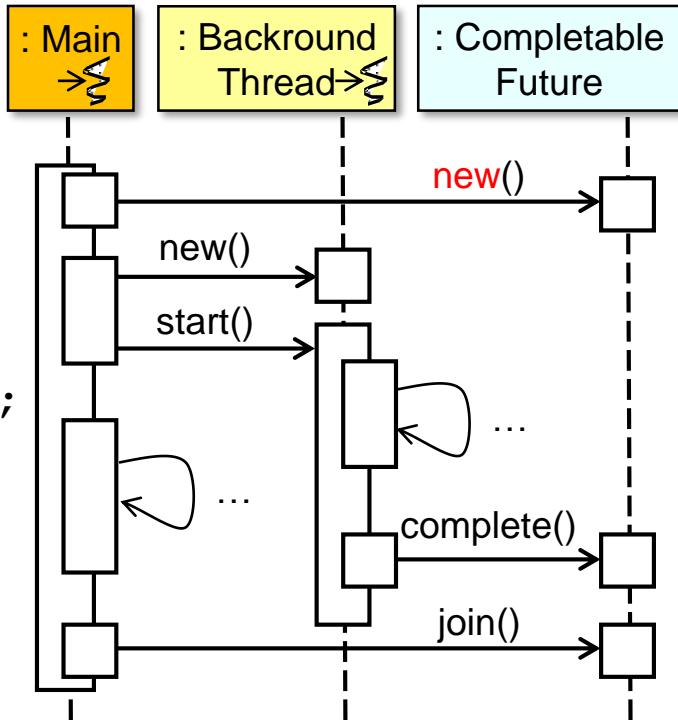
```
CompletableFuture<BigFraction> future  
= new CompletableFuture<>();
```

*Make "empty" future*

```
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());
```



# Applying Basic Completable Future Features

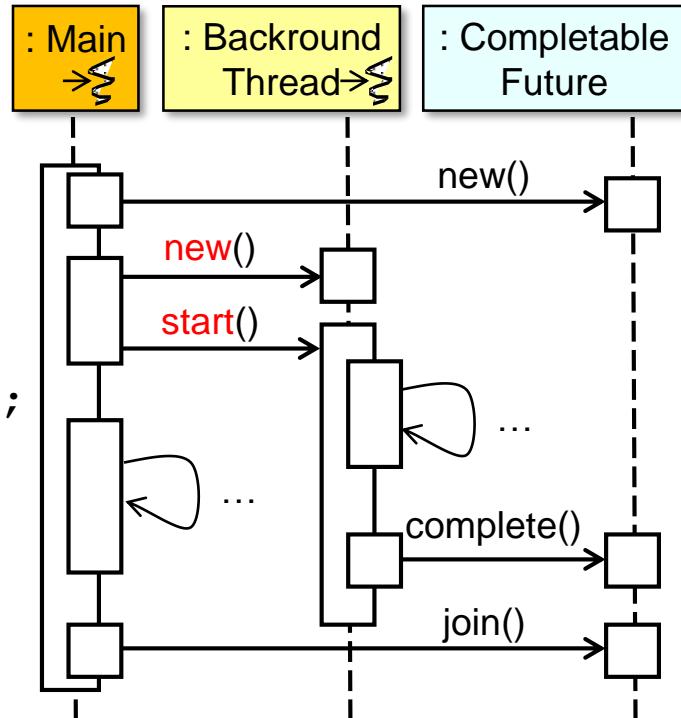
- Multiplying big fractions w/a completable future

```
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 CompletableFuture Features

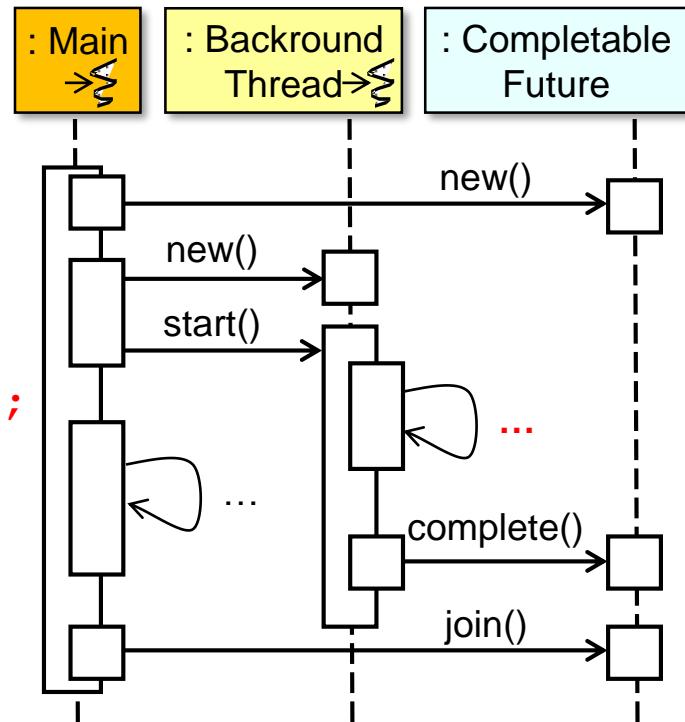
- Multiplying big fractions w/a completable future

```
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 BigFractions (via 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

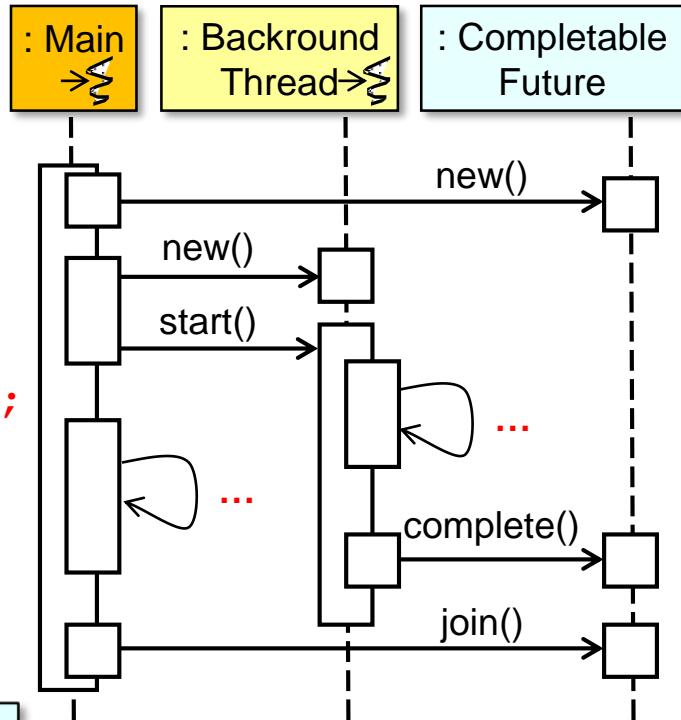
- Multiplying big fractions w/a completable future

```
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());
```



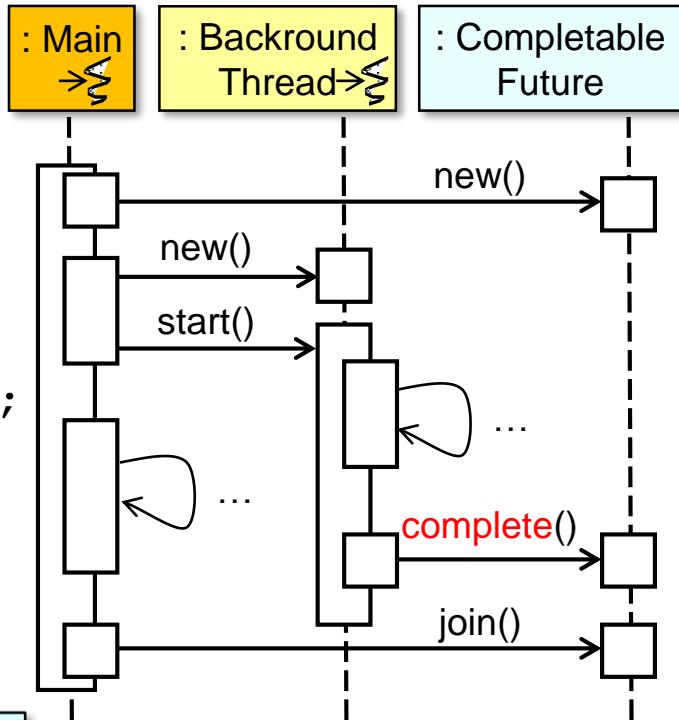
# Applying Basic Completable Future Features

- 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();
```

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



*Explicitly complete the future w/result*

# Applying Basic CompletableFuture Features

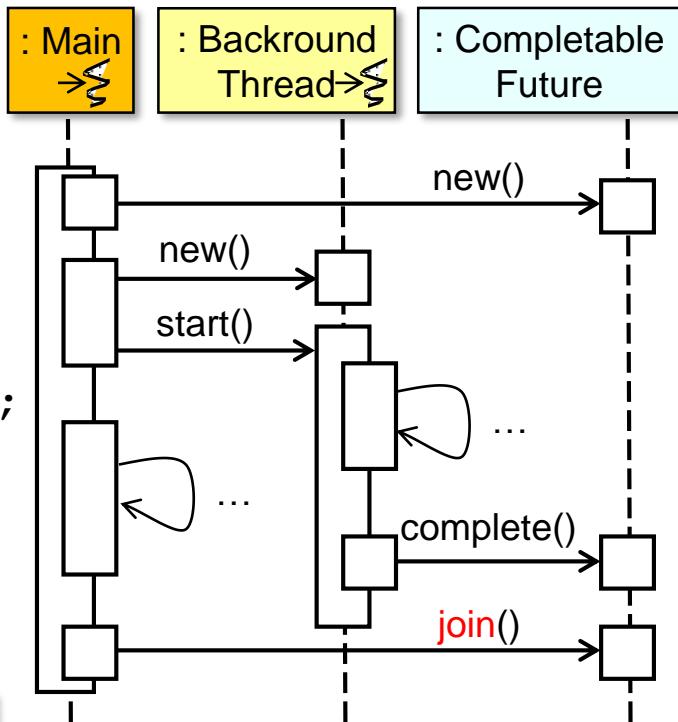
- Multiplying big fractions w/a completable future

```
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());
```



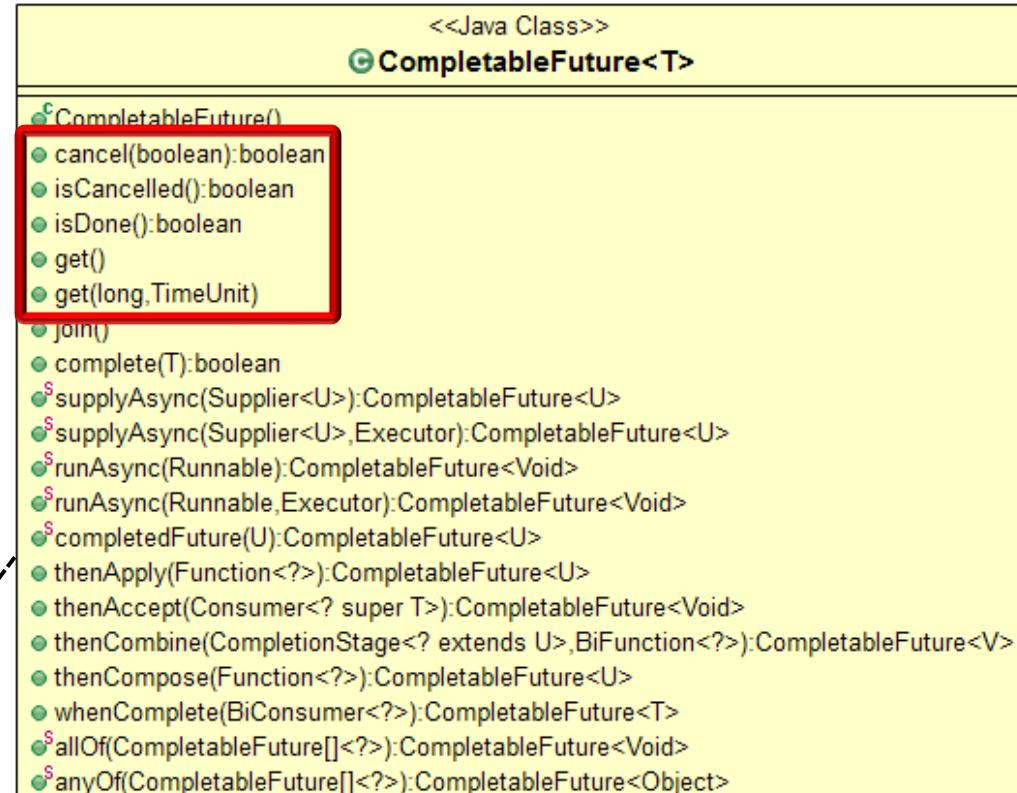
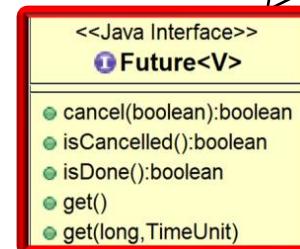
---

# 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

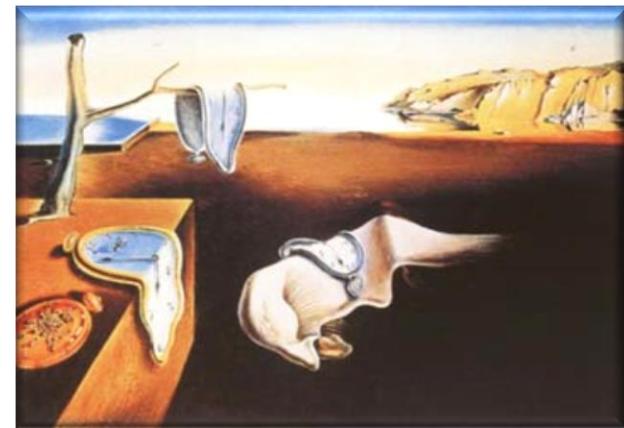
- e.g., `join()` blocks until the future is completed..

```
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());
```



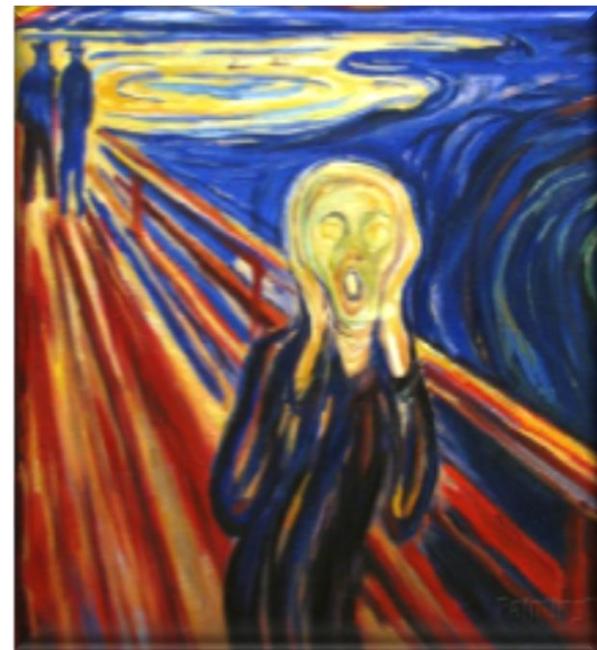
*This blocking call underutilizes cores & increases overhead*

# 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();
```



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

...

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

See [crondev.blog/2017/01/23/timeouts-with-java-8-completablefuture-youre-probably-doing-it-wrong](http://crondev.blog/2017/01/23/timeouts-with-java-8-completablefuture-youre-probably-doing-it-wrong)

# 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.

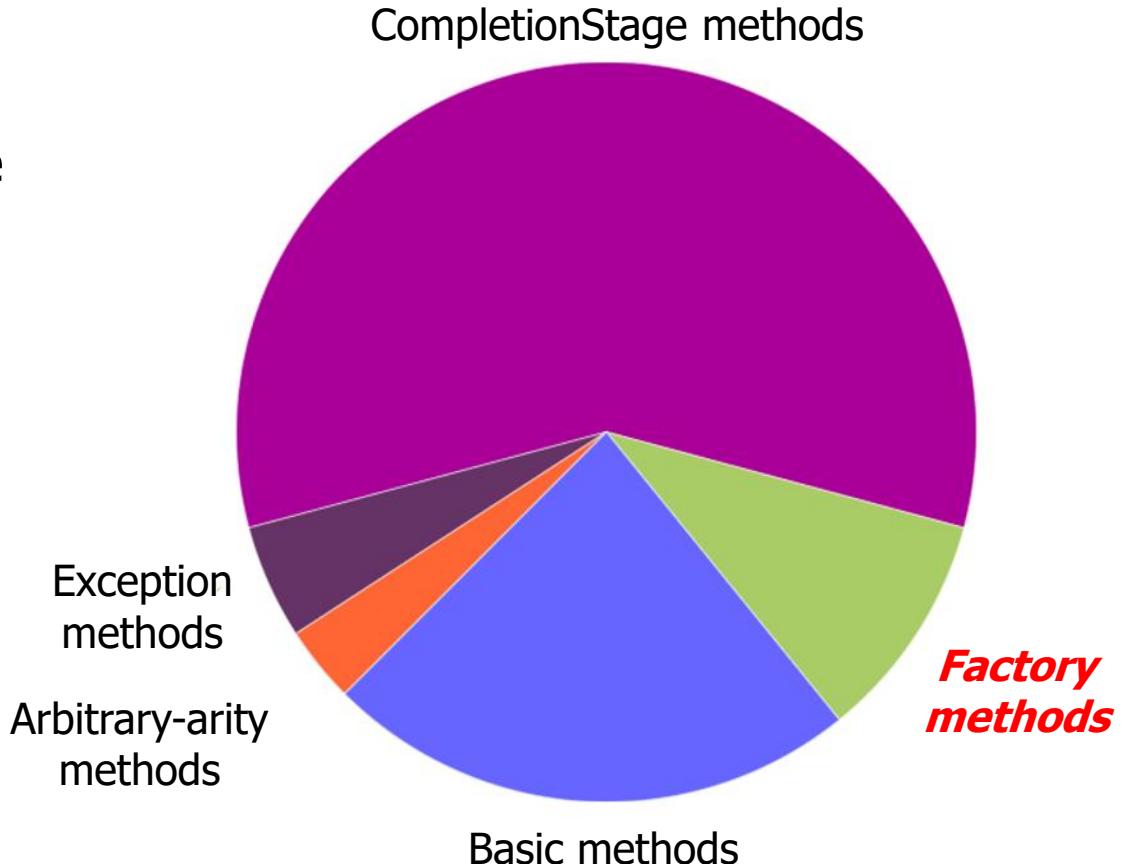
When two or more threads attempt to complete, completeExceptionally, or cancel a CompletableFuture, only one of them succeeds.

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>**

|                                                                                                             |
|-------------------------------------------------------------------------------------------------------------|
| <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>                            |

# 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;T&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
    - `supplyAsync()` allows two-way calls via a supplier
    - `runAsync()` enables one-way calls via a runnable
    - Can be passed params, but returns no values

*Any output must therefore come from "side-effects"*

```
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
  - 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 =
```

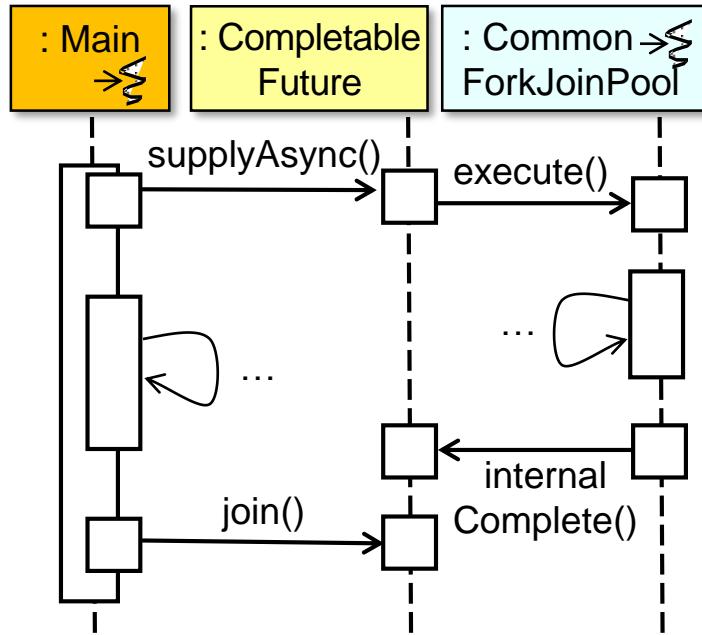
```
                new BigFraction(f2);
```

```
            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";
```

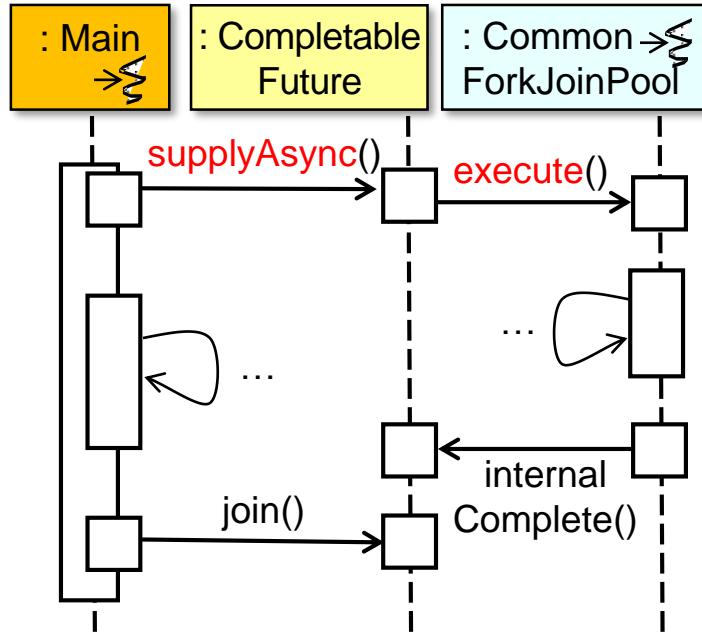
```
CompletableFuture<BigFraction> future =
```

```
    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 =
```

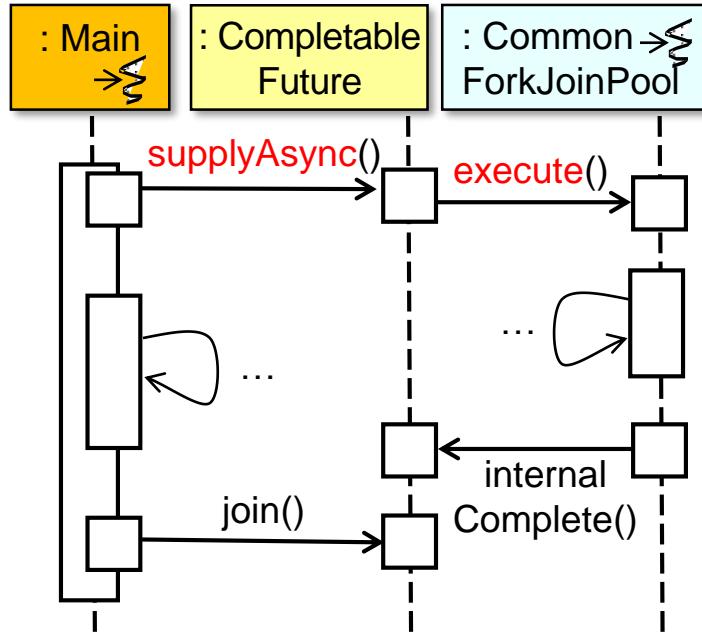
```
                new BigFraction(f2);
```

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

```
});
```

```
...
```

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



# 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 =
```

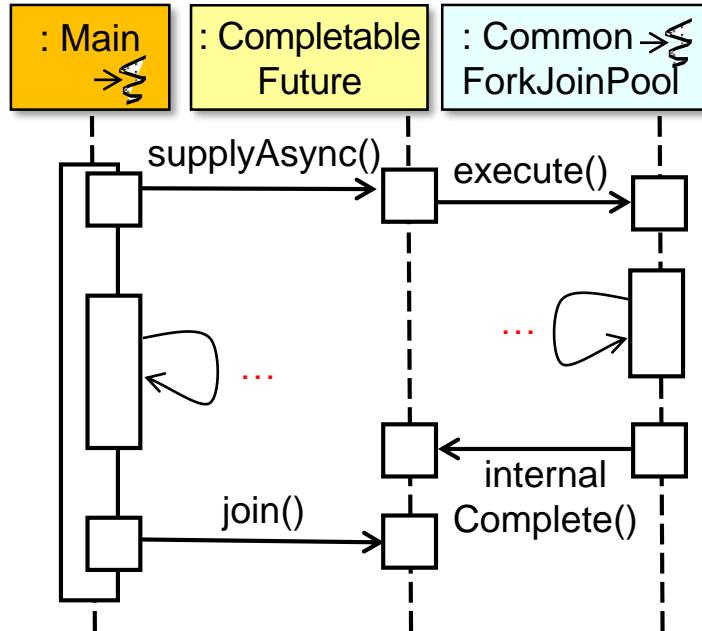
```
                new BigFraction(f2);
```

```
            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 =
```

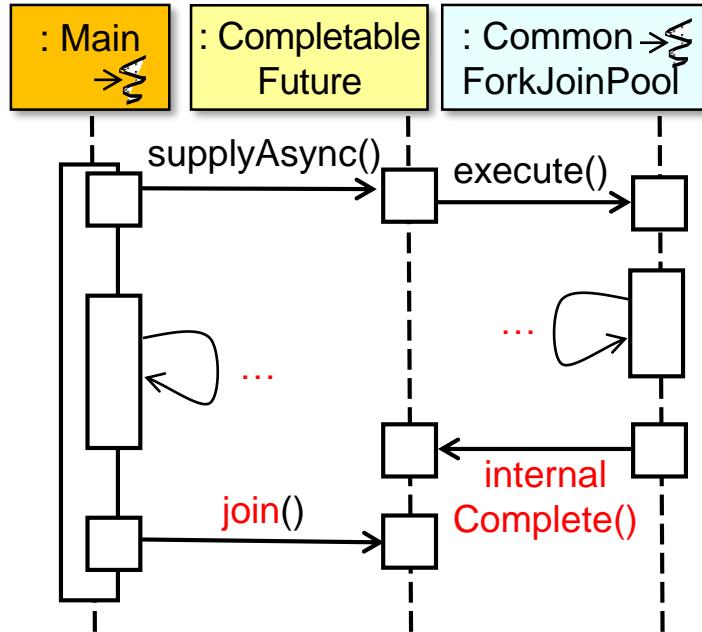
```
                new BigFraction(f2);
```

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

```
});
```

```
...
```

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



*join() blocks until result is complete*

# 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());
```



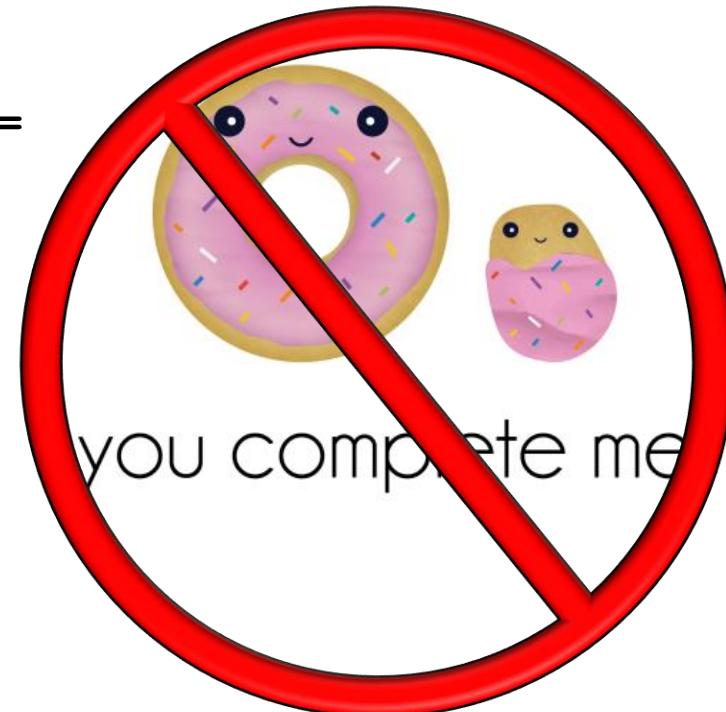
Calling CompletableFuture.supplyAsync() avoids the use of threads in this example!

# 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());
```



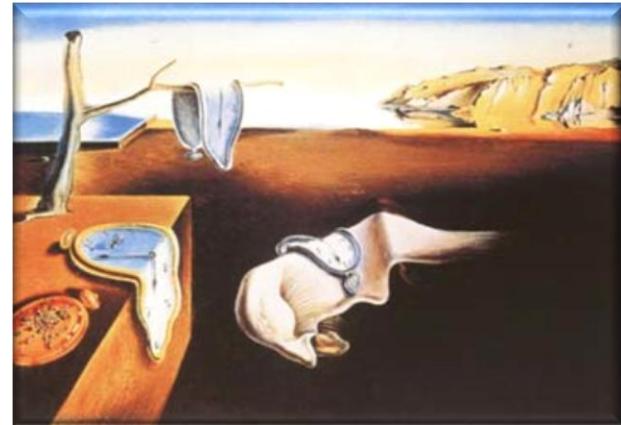
There's no need to explicitly complete the future since supplyAsync() returns one

# 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());
```



However, we still have the problem with having to call join() explicitly..

---

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

# **Overview of Advanced Java 8**

# **CompletableFuture Features (Part 2)**

**Douglas C. Schmidt**

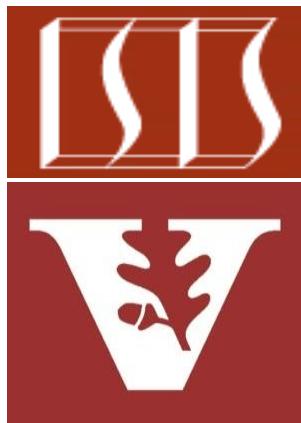
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**

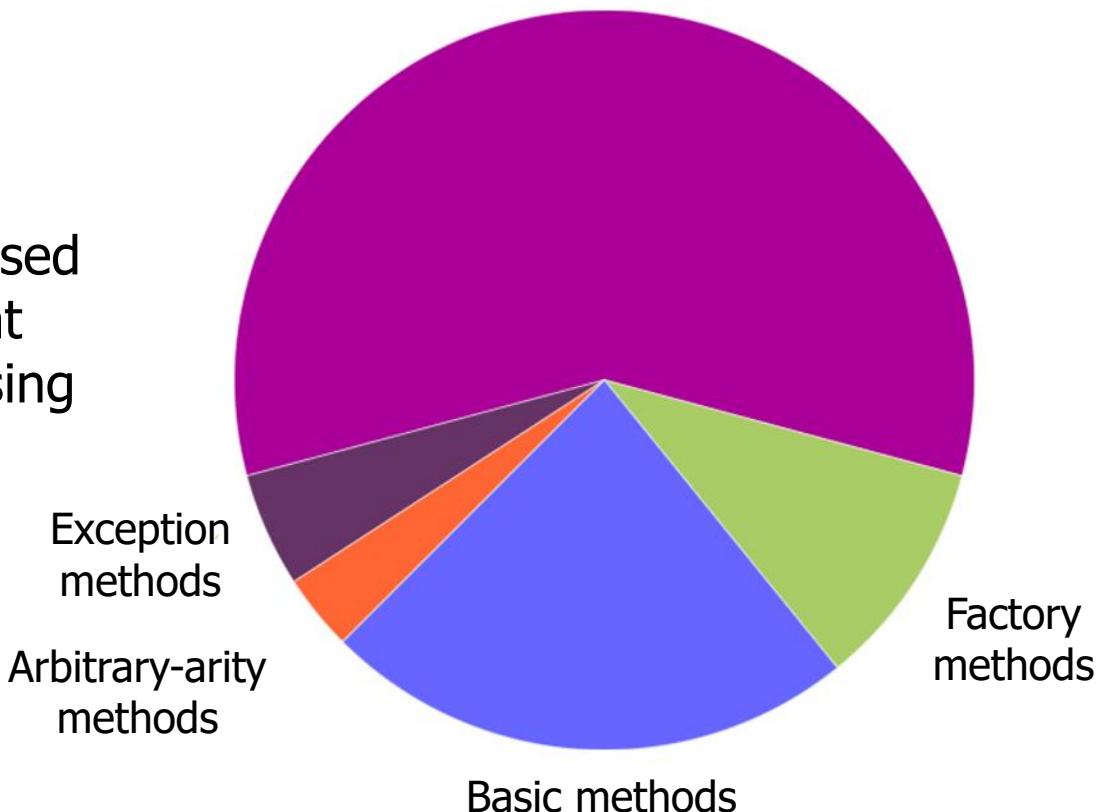


# Learning Objectives in this Part of the Lesson

---

- 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*



---

# 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);
```

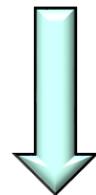
# Completion Stage Methods Chain Actions Together

- 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
    - A lambda action is called only after previous stage completes successfully

```
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 previous stage completes successfully

```
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);
```



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

# Completion Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained



# Completion Stage Methods Chain Actions Together

- Use completion stages to avoid blocking a thread until the result *must* be obtained, e.g.
  - Try not to call `join()` or `get()` unless absolutely necessary



Servers may avoid blocking completely, whereas clients may need `join()` sparingly

# Completion Stage Methods Chain Actions Together

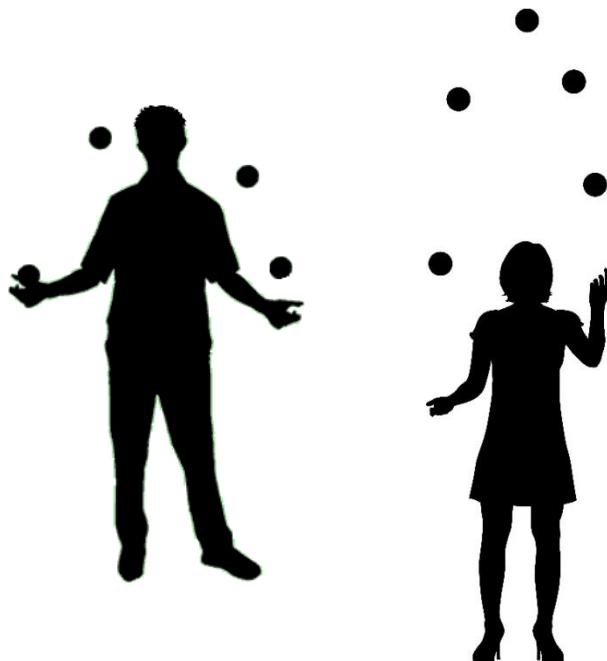
---

- Use completion stages to avoid blocking a thread until the result *must* be obtained, e.g.
  - Try not to call `join()` or `get()` unless absolutely necessary
  - This approach helps improve responsiveness



# 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>                       |

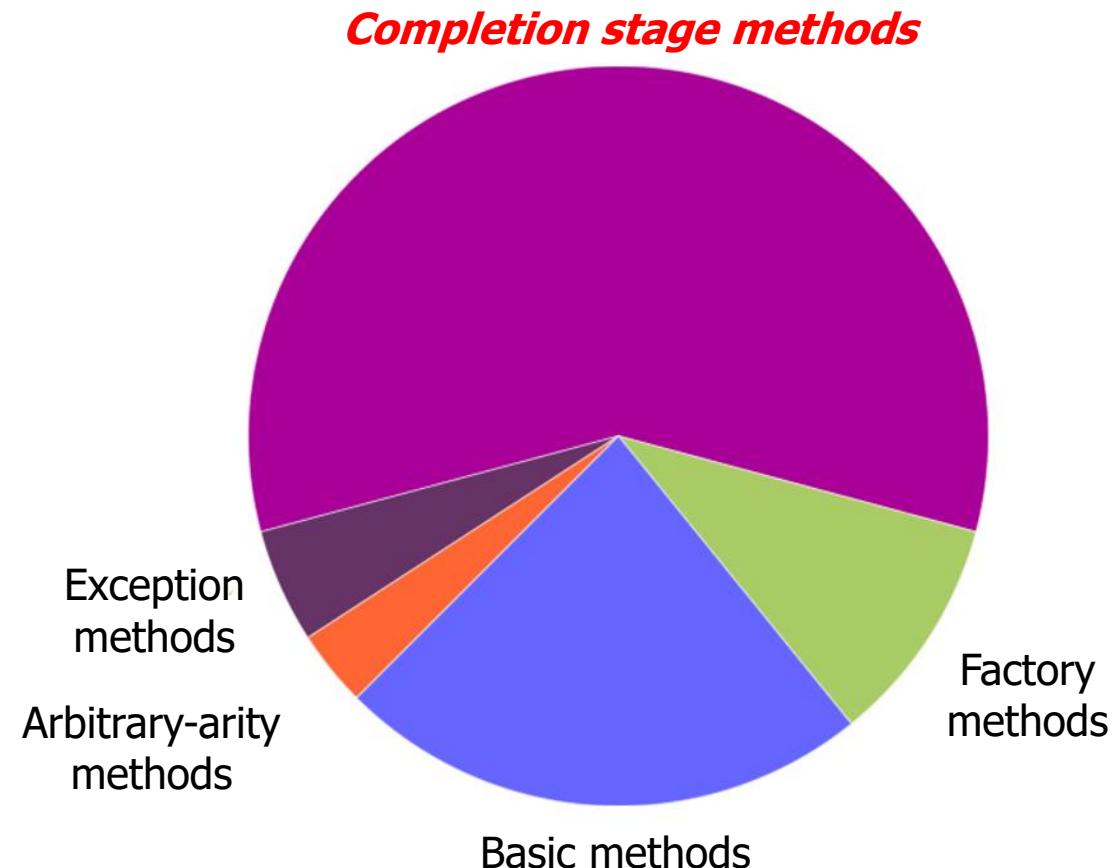
Juggling is a good analogy for completion stages!

---

# Grouping CompletableFuture Completion Stage Methods

# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by one or more previous stage(s)



# Grouping CompletableFuture Completion Stage Methods

- Completion stage methods are grouped based on how a stage is triggered by one or more previous stage(s)
  - 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&gt;</code>                                         | Apply function to result of the previous stage |
| <code>thenCompose<br/>(Async)</code> | <code>Function</code> | <code>CompletableFuture&lt;Function&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 one or more previous stage(s)
  - 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 |
| <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 one or more previous stage(s)
  - Completion of a single previous stage
  - Completion of both of 2 previous stages
    - i.e., an “and”

| Methods                                                                                 | Params                                   | Returns                                                                                                       | Behavior                                            |
|-----------------------------------------------------------------------------------------|------------------------------------------|---------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| <code>then</code><br><code>Combine</code><br>( <code>Async</code> )                     | <code>Bi</code><br><code>Function</code> | <code>CompletableFuture&lt;Future&lt;T&gt;&gt;</code><br><code>Future&lt;T&gt; with Bi Function result</code> | Apply bifunction to results of both previous stages |
| <code>then</code><br><code>Accept</code><br><code>Both</code><br>( <code>Async</code> ) | <code>Bi</code><br><code>Consumer</code> | <code>CompletableFuture&lt;Void&gt;</code>                                                                    | BiConsumer handles results of both previous stages  |
| <code>runAfter</code><br><code>Both</code><br>( <code>Async</code> )                    | <code>Runnable</code>                    | <code>CompletableFuture&lt;Void&gt;</code>                                                                    | 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 one or more previous stage(s)
  - 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

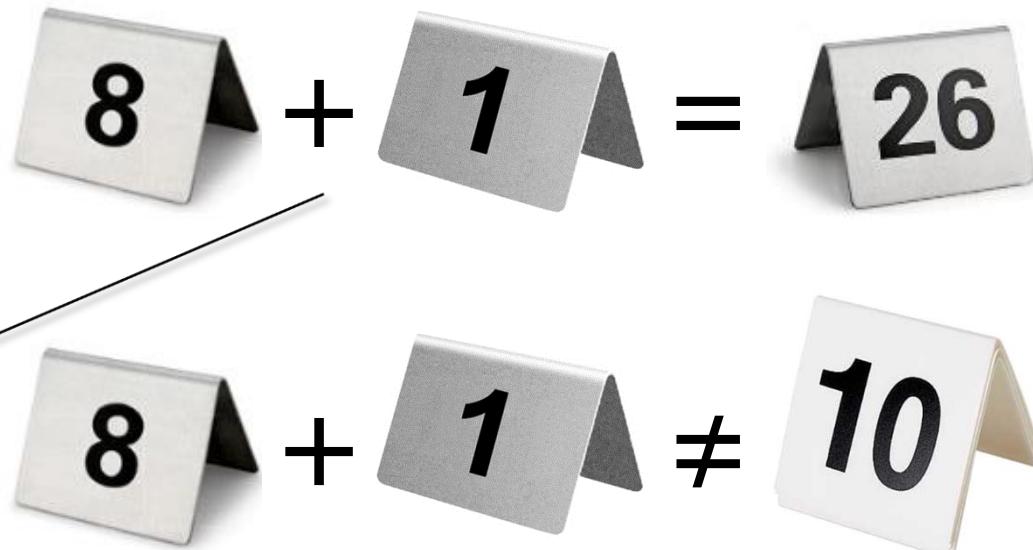
- Methods triggered by completion of a single previous stage

- `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

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

```
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
    - Used for an *async* action that returns a completable future

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

.thenCompose
(reduced -> CompletableFuture<BF>
.supplyAsync(() ->
reduced.multiply(...)));
...
```

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()`

Concise!

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

`thenApplyAsync()` can often replace `thenCompose(supplyAsync())` nestings

# 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()`
    - 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()))  
  
.thenComposeAsync  
(this::longBlockerReturnsCF)  
...
```

*Runs longBlockerReturnsCF() in a thread in the fork-join pool*

**thenComposeAsync()** can be used to avoid calling `supplyAsync()` again in a chain

# 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!"

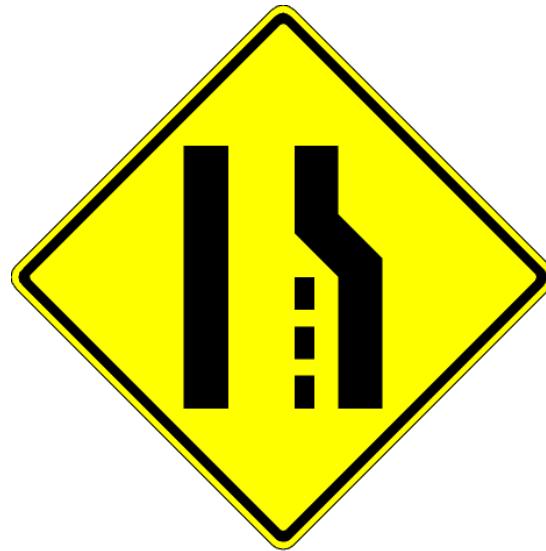


See [dzone.com/articles/callback-hell](https://dzone.com/articles/callback-hell)

# 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 asynchronous execution

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

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

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

```
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

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

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

*Create two completable futures that will contain the results of sorting the list using two different algorithms in two different threads*

# 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

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

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

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

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

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**

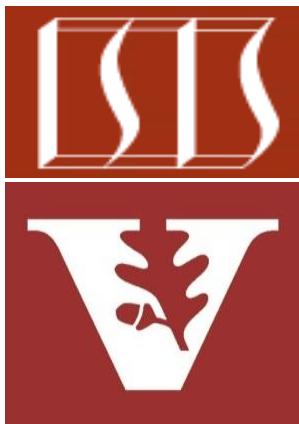
**d.schmidt@vanderbilt.edu**

**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

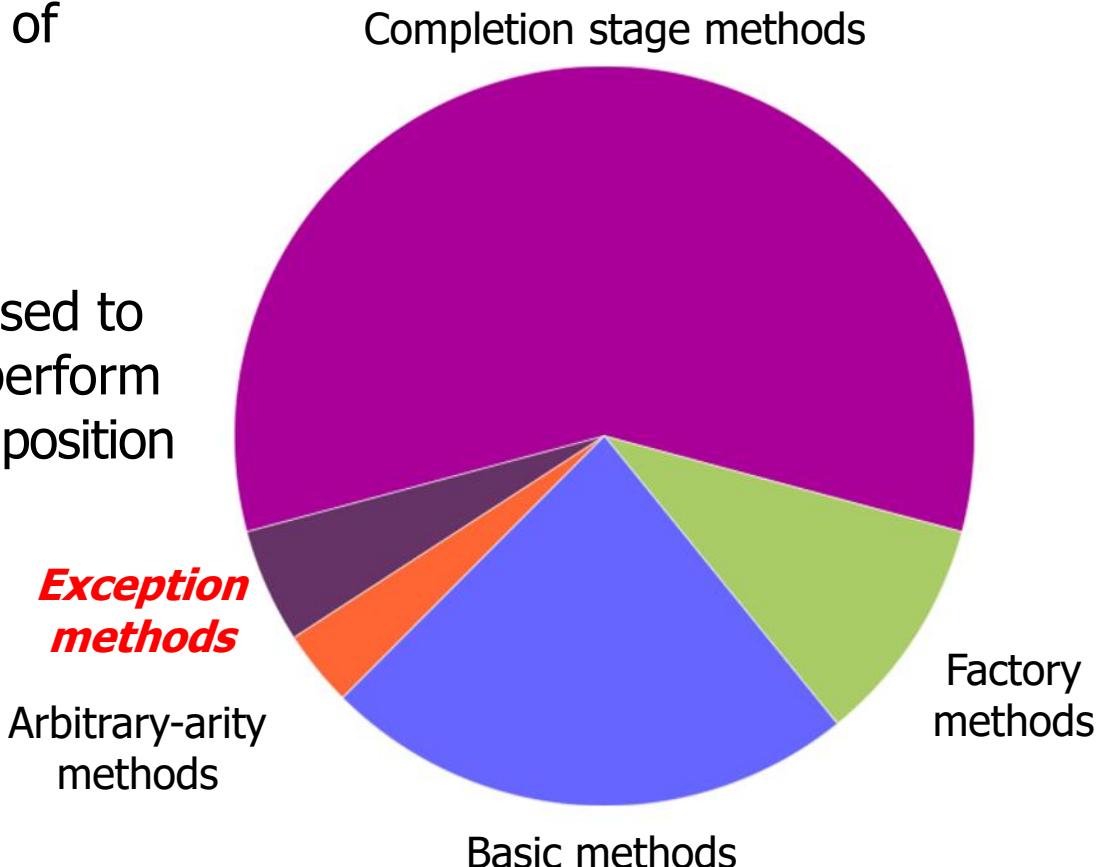
- Understand advanced features of completable futures, e.g.
  - Factory methods that initiate async functionality
  - Completion stage methods used to chain together actions that perform async result processing & composition
    - Apply completion stage methods to BigFractions

<<Java Class>>
<b>C BigFraction</b>
(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
↳ 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

# 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 testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFractions)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(ex8::sortAndPrintList);  
}
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(ex8::sortAndPrintList);  
}
```



*Generate a bounded # of  
large & random fractions*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() 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 big fraction*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() 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);  
}
```

*Make a random numerator uniformly distributed over range 0 to ( $2^{150000} - 1$ )*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() 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));  
  
    Make a denominator by dividing the  
    numerator by random # between 1 & 10  
  
    return BigFraction.valueOf(numerator,  
        denominator,  
        reduced);  
}
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

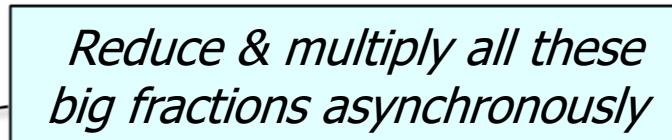
```
BigFraction makeBigFraction(Random random, boolean reduced) {  
    BigInteger numerator =  
        new BigInteger(150000, random);  
  
    BigInteger denominator =  
        numerator.divide(BigInteger  
            .valueOf(random.nextInt(10) + 1));  
  
    return BigFraction.valueOf(numerator,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

*Return a BigFraction w/the  
numerator & denominator*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(ex8::sortAndPrintList);  
}
```



*Reduce & multiply all these  
big fractions asynchronously*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

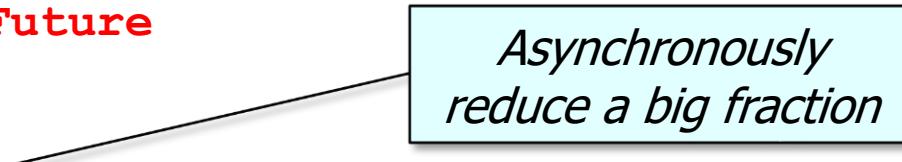
```
static void testFractionMultiplications1() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
                .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
                .thenCompose(reducedFrac -> 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 testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    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 testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
  
            .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
  
            .thenCompose(reducedFrac -> CompletableFuture  
                .supplyAsync(() -> reducedFrac  
                    .multiply(sBigFraction)));  
  
    Asynchronously  
    multiply big fractions
```

...

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture<
```

*thenCompose() acts like flatMap() to ensure one level of CompletableFuture nesting*

```
        .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
  
        .thenCompose(reducedFrac -> CompletableFuture  
            .supplyAsync(() -> reducedFrac  
                .multiply(sBigFraction)));
```

...

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications2() {  
    Function<BigFraction, CompletableFuture<BigFraction>>  
        reduceAndMultiplyFraction = unreducedFrac ->  
            CompletableFuture  
  
            .supplyAsync(() -> BigFraction.reduce(unreducedFrac))  
  
            .thenApplyAsync(reducedFrac ->  
                reducedFrac.multiply(sBigFraction)));  
}
```

*Asynchronously  
multiply big fractions*

...

thenApplyAsync() provides a way to avoid calling supplyAsync() again

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(ex8::sortAndPrintList);  
}
```

*Outputs a stream of completable futures  
to async operations on big fractions*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(ex8::sortAndPrintList);  
}
```

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

FuturesCollector is a non-concurrent collector covered in Part 4 of this lesson

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(ex8::sortAndPrintList);  
}
```

*Sort & print results when all async computations complete*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void sortAndPrintList(List<BigFraction> list) {
```

*Sorts & prints a list of reduced fractions*

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

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

```
    quickSortF.acceptEither(mergeSortF, sortedList ->  
        sortedList.forEach(frac -> display(frac.toMixedString())));  
    } ; ...
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

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

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

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

Select whichever result finishes first..

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

```
static void sortAndPrintList(List<BigFraction> list) {
```

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

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

*If future is already completed the action runs in the thread that registered the action*

```
    quickSortF.acceptEither(mergeSortF, sortedList ->  
        sortedList.forEach(frac -> display(frac.toMixedString())));  
    } ; ...
```

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

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

*Otherwise, the action runs in the thread in which the previous stage ran*

# Applying Completable Future Completion Stage Methods

- We show key completion stage methods via the testFractionMultiplications1() method that multiplies big fractions using a stream of CompletableFutures

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

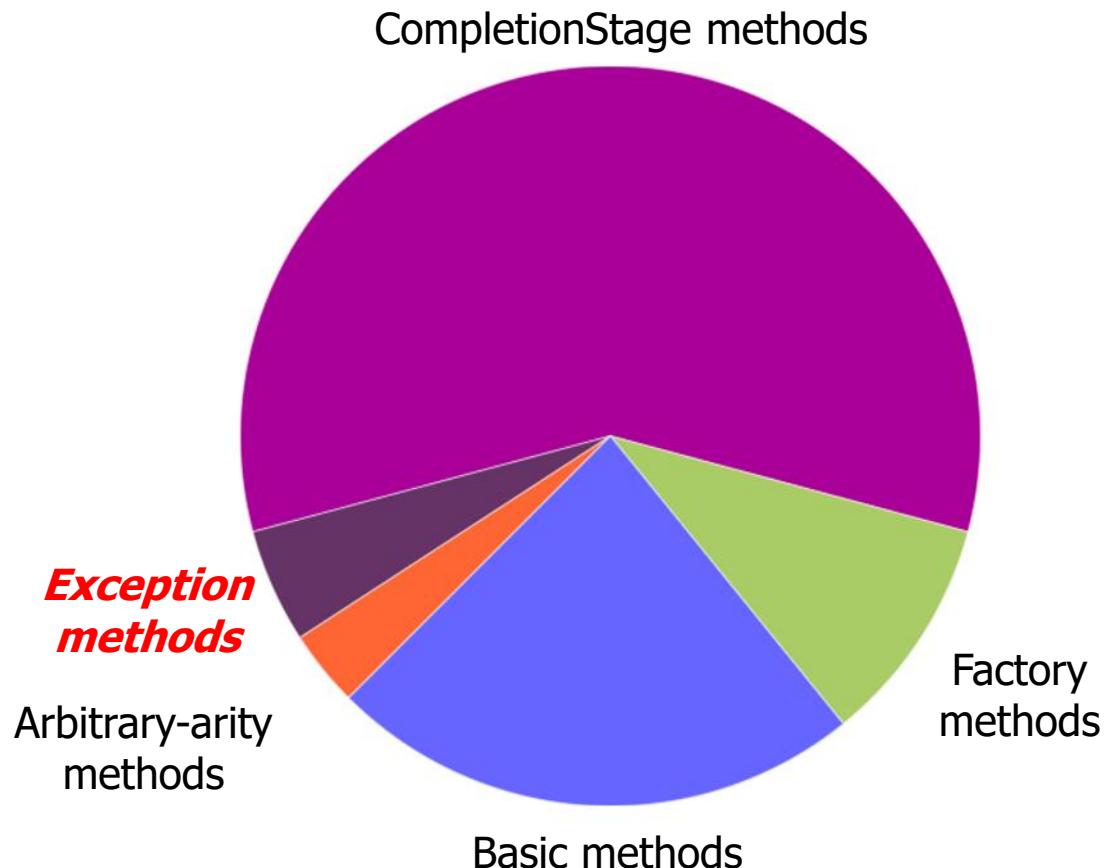
acceptEither() does *not* cancel the second future after the first one completes

---

# 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;, Throwable, R&gt;</code>	<code>CompletableFuture&lt;R&gt;</code>	Handle outcome of a stage & return new value
<code>exceptionally</code>	<code>Function&lt;Throwable, T&gt;</code>	<code>CompletableFuture&lt;T&gt;</code>	When exception occurs, replace exception with result value

# Handling Runtime Exceptions in Completion Stages

- This example shows three ways to handle exceptions w/completable futures

```
CompletableFuture
```

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

```
...
```

*An exception will occur if  
denominator param is 0!*

# 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 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() must return a value (& can thus change the return value)*

# 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()));
```

# 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*

exceptionally() is akin to a catch in a Java try/catch block, i.e., control xfers to it

# Handling Runtime Exceptions in Completion Stages

- Using the whenComplete() method to perform a exceptional or normal action

```
CompletableFuture
```

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

```
    .thenApply(fraction ->  
        fraction.multiply(sBigReducedFraction))
```

*Called under both normal & exception conditions*

```
.whenComplete((fraction, ex) -> {  
    if (fraction != null)  
        System.out.println(fraction.toMixedString());  
    else  
        System.out.println(ex.getMessage());  
});
```

# Handling Runtime Exceptions in Completion Stages

- Using the whenComplete() method to perform a exceptional or normal action

```
CompletableFuture
```

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

```
    .thenApply(fraction ->  
        fraction.multiply(sBigReducedFraction))
```

```
.whenComplete((fraction, ex) -> {  
    if (fraction != null)  
        System.out.println(fraction.toMixedString());  
    else  
        System.out.println(ex.getMessage());  
});
```

*Handle the normal case*

# Handling Runtime Exceptions in Completion Stages

- Using the whenComplete() method to perform a exceptional or normal action

```
CompletableFuture
```

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

```
.thenApply(fraction ->  
    fraction.multiply(sBigReducedFraction))
```

```
.whenComplete((fraction, ex) -> {  
    if (fraction != null)  
        System.out.println(fraction.toMixedString());  
    else // ex != null  
        System.out.println(ex.getMessage());  
});
```

*Handle the  
exceptional case*

# Handling Runtime Exceptions in Completion Stages

- Using the whenComplete() method to perform a exceptional or normal action

```
CompletableFuture
```

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

```
    .thenApply(fraction ->  
        fraction.multiply(sBigReducedFraction))
```

```
.whenComplete((fraction, ex) -> {  
    if (fraction != null)  
        System.out.println(fraction.toMixedString());  
    else // ex != null  
        System.out.println(ex.getMessage());  
});
```

*whenComplete() is like Java 8 streams peek(): it has a side-effect & doesn't change the return value*

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

---

# 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**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

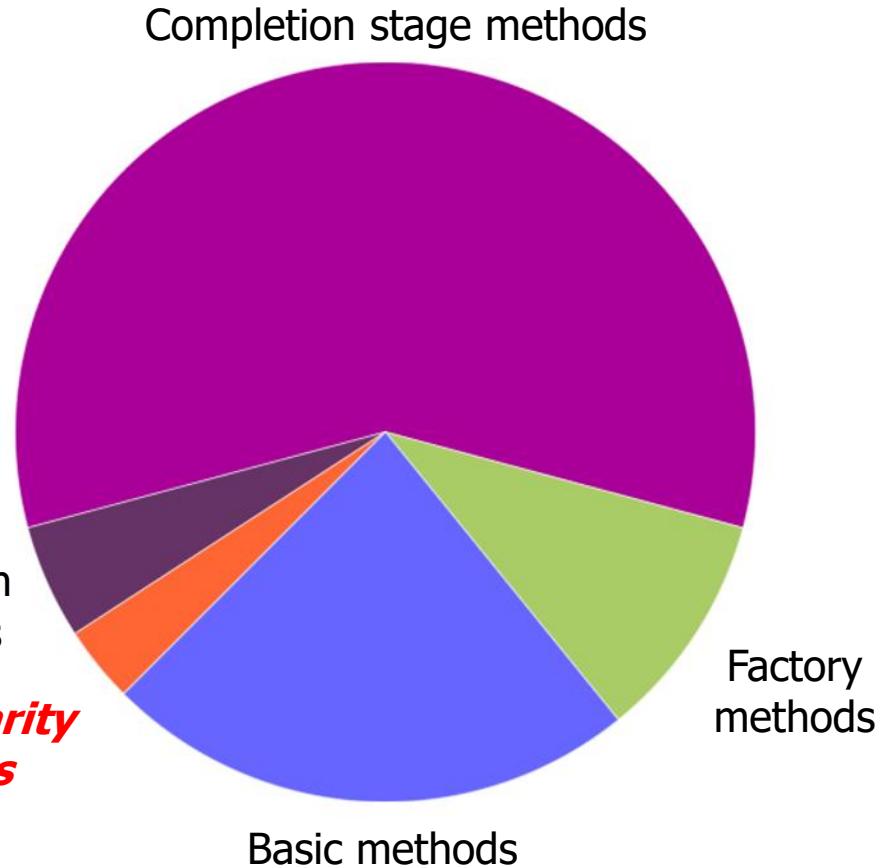
**Vanderbilt University  
Nashville, Tennessee, USA**



# 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 return futures that are triggered after completion of some/all 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

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

# Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods return futures that are triggered after completion of some/all futures
  - The returned future can be used to wait for any or all of  $N$  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 return futures that are triggered after completion of some/all futures
  - The returned future can be used to wait for any or all of  $N$  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>
```

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

# Arbitrary-Arity Methods Process Futures in Bulk

- Arbitrary-arity methods return futures that are triggered after completion of some/all futures
  - The returned future can be used to wait for any or all of  $N$  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 returns a completable future to a list of big fractions that are being reduced and multiplied asynchronously

```
static void testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFractions)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(this::sortAndPrintList);  
}
```

*collect() converts a stream of completable futures into a single completable future*

# Arbitrary-Arity Methods Process Futures in Bulk

- FuturesCollector provides a wrapper for allOf()



<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



<<Java Class>>

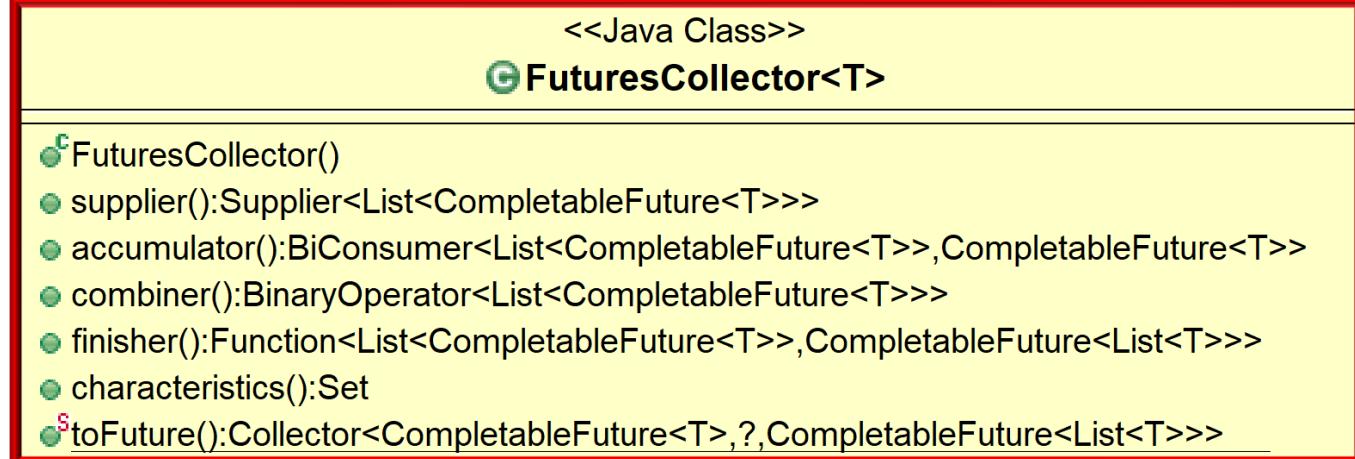
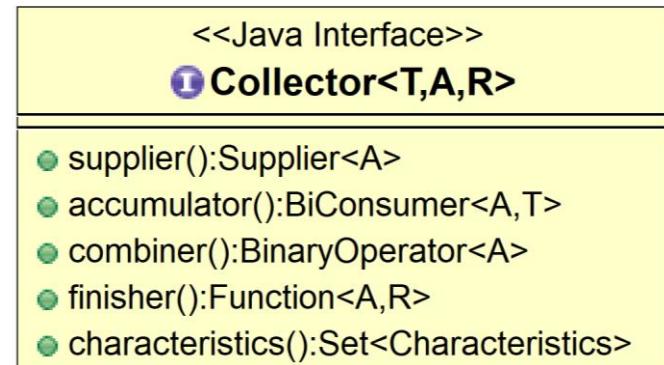
G FuturesCollector<T>

- FuturesCollector()
- supplier():Supplier<List<CompletableFuture<T>>>
- accumulator():BiConsumer<List<CompletableFuture<T>>,CompletableFuture<T>>
- combiner():BinaryOperator<List<CompletableFuture<T>>>
- finisher():Function<List<CompletableFuture<T>>,CompletableFuture<List<T>>>
- characteristics():Set
- toFuture():Collector<CompletableFuture<T>,?,CompletableFuture<List<T>>>

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

# Arbitrary-Arity Methods Process Futures in Bulk

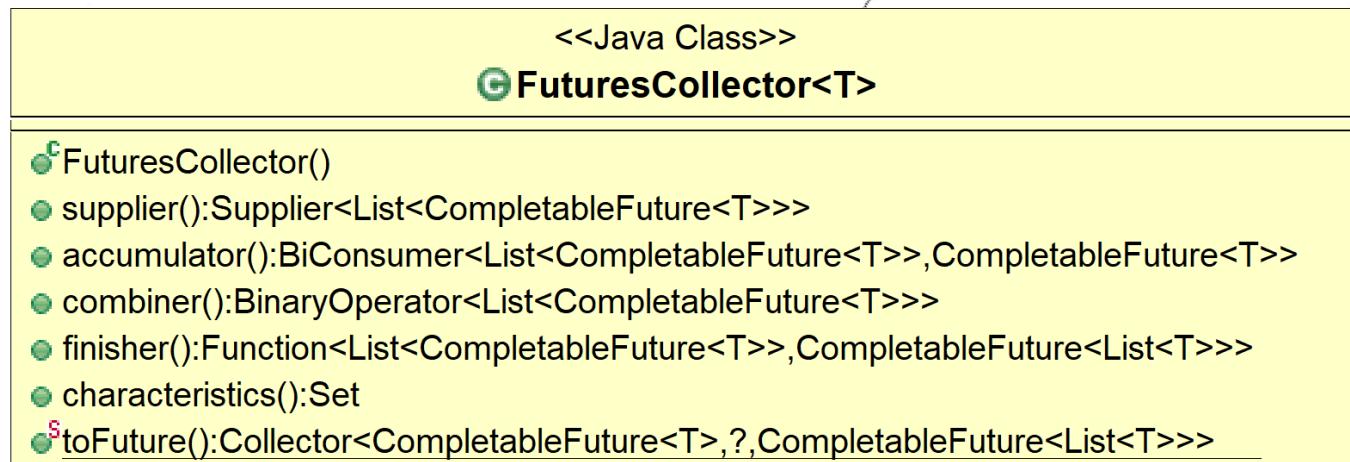
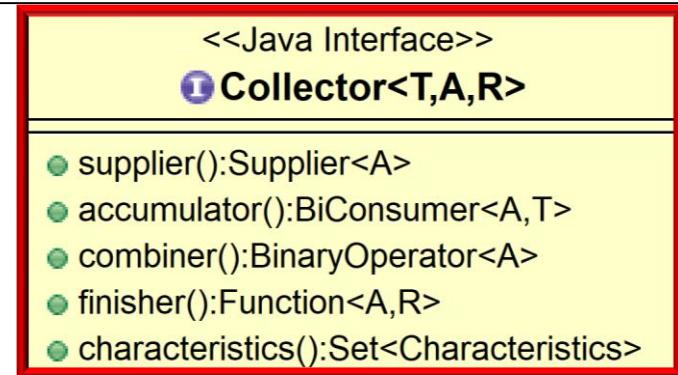
- `FuturesCollector` provides a wrapper for `allOf()`
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete



`FuturesCollector` is a non-concurrent collector (supports parallel & sequential streams)

# Arbitrary-Arity Methods Process Futures in Bulk

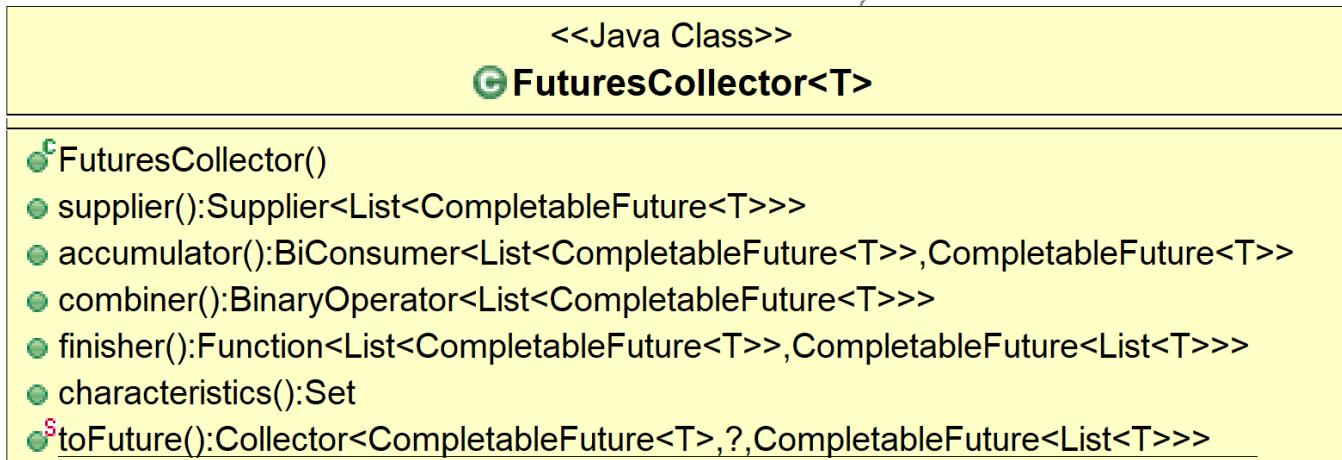
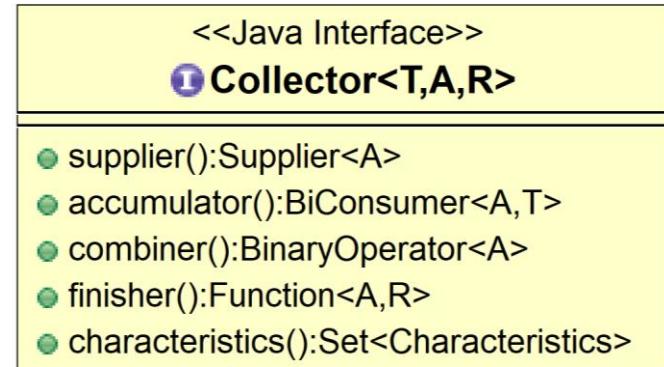
- `FuturesCollector` provides a wrapper for `allOf()`
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete
  - Implements the `Collector` interface that accumulates input elements into a mutable result container



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;
    }
```

*This factory method returns a supplier used by the Java 8 streams collector framework to create a new mutable array list container*

```
public BiConsumer<List<CompletableFuture<T>>,
                  CompletableFuture<T>> accumulator()
{ return List::add; }
...
```

# 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;
    }
```

*This factory method returns a bi-consumer used by the Java 8 streams collector framework to add a new completable future into the mutable array list container*

```
public BiConsumer<List<CompletableFuture<T>>,
                  CompletableFuture<T>> accumulator()
{ return List::add; }
...
```

This method is only ever called in a single thread (so no locks are needed)

# 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;
        };
    }
    ...
}
```

*This factory method returns a binary operator that merges two partial array list results into a single array list (only relevant for parallel streams)*

This method is only ever called in a single thread (so no locks are needed)

# 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]))  
    }  
}
```

*This factory method returns a function used by the Java 8 streams collector framework to transform the array list accumulation type to the completable future result type*

```
.thenApply(v -> futures.stream()  
                    .map(CompletableFuture::join)  
                    .collect(toList()));  
}  
...
```

# 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()));  

    }
    ...
}
```

*Convert the array list of futures into a stream of futures*

# 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 testFractionMultiplications1() {  
    ...  
    Stream.generate(() -> makeBigFraction(new Random(), false))  
        .limit(sMAX_FRACTIONS)  
        .map(reduceAndMultiplyFraction)  
        .collect(FuturesCollector.toFuture())  
        .thenAccept(this::sortAndPrintList);  
}
```

*thenAccept() is called only when the future returned from collect() completes*

# 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);  
    }  
}
```

*This static factory method creates a new FuturesCollector*

```
public static <T> Collector<CompletableFuture<T>, ?,  
                    CompletableFuture<List<T>>>  
toFuture() {  
    return 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**

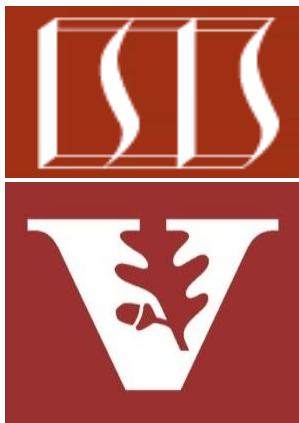
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

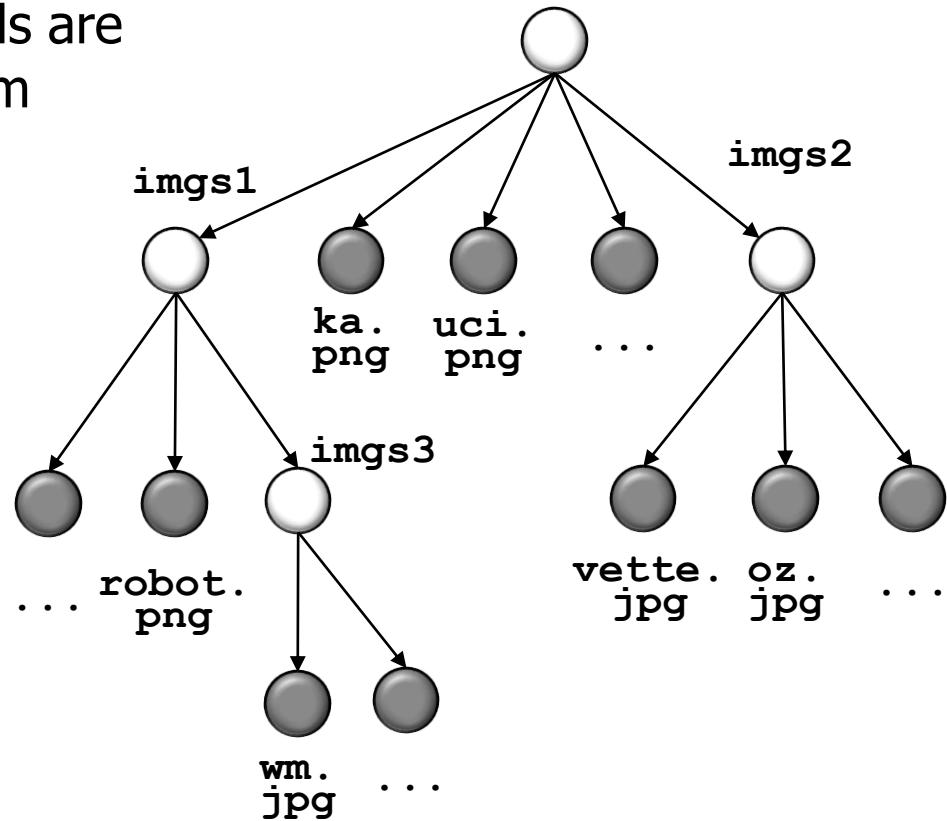
**Institute for Software  
Integrated Systems**

**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

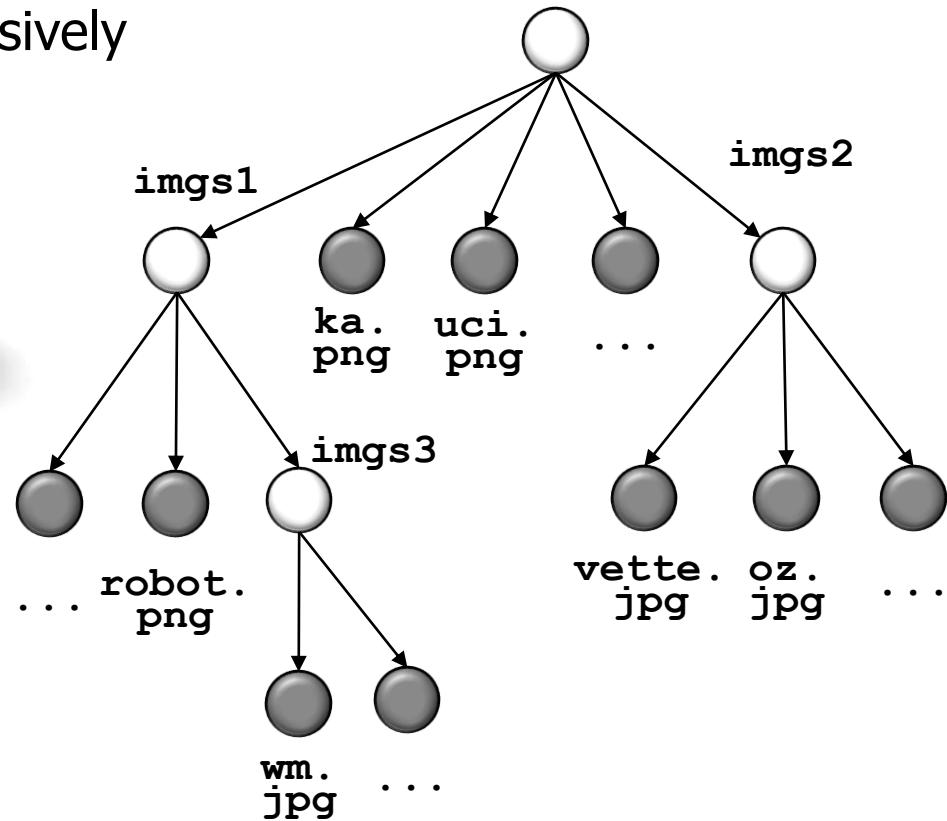


---

# 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

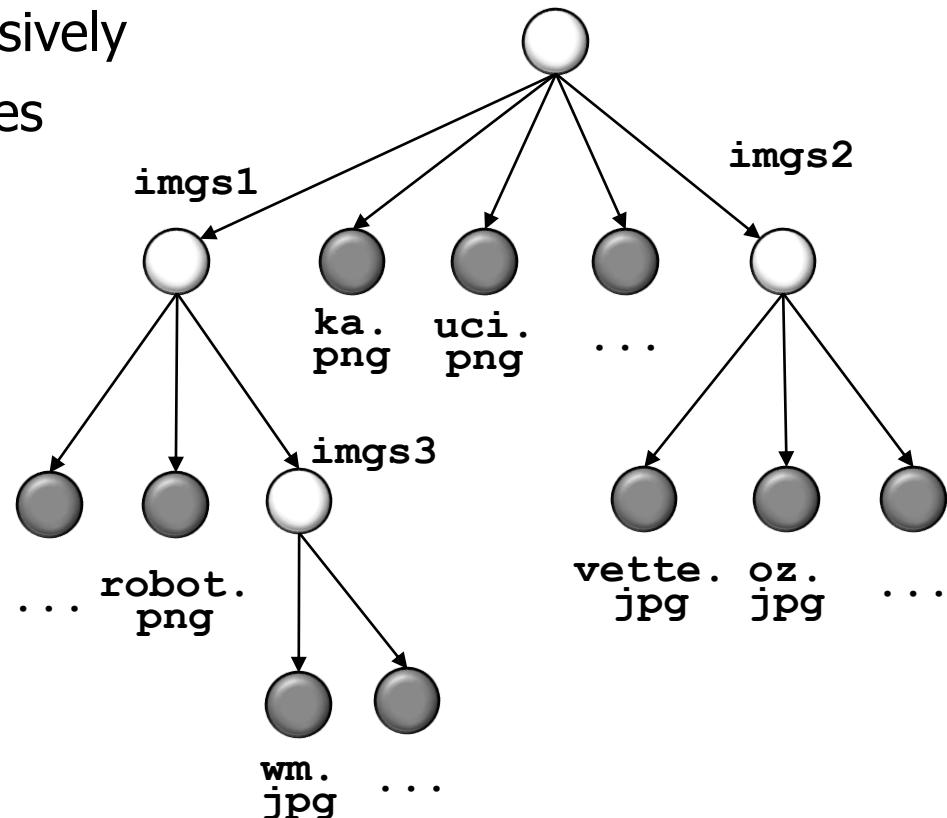
ImageCounter [Depth3]: Exceeded max depth of 2

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

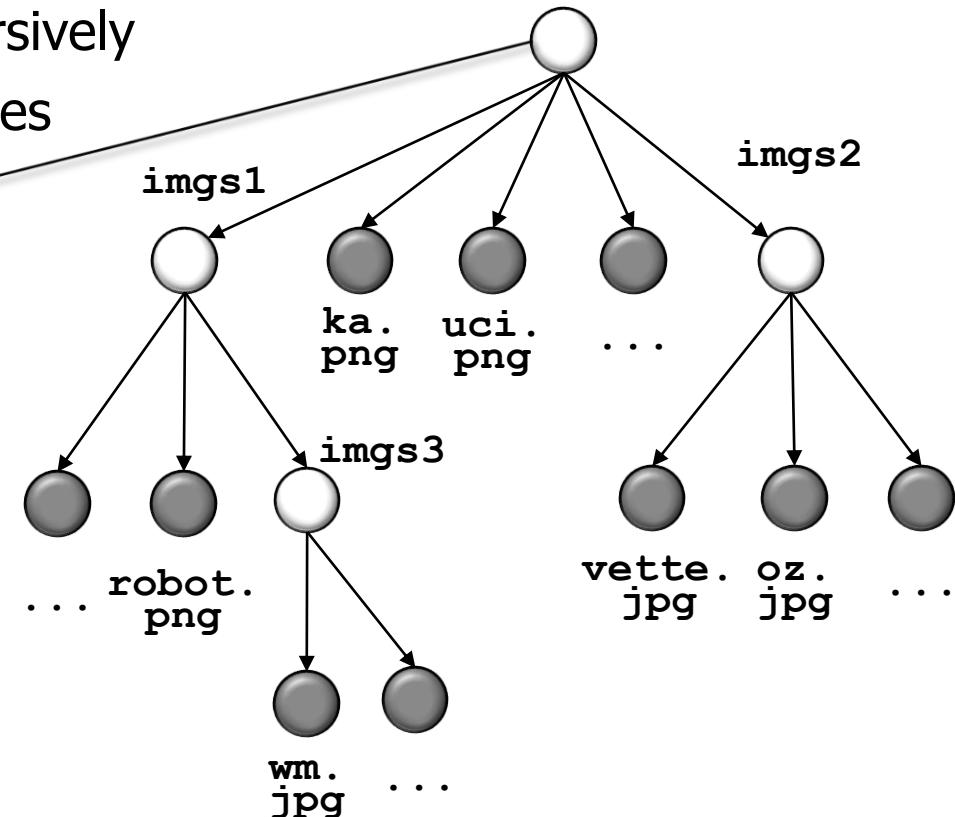
ImageCounter: 21 total image(s) are reachable from index.html



# Applying Completion Stage Methods to Image Crawler

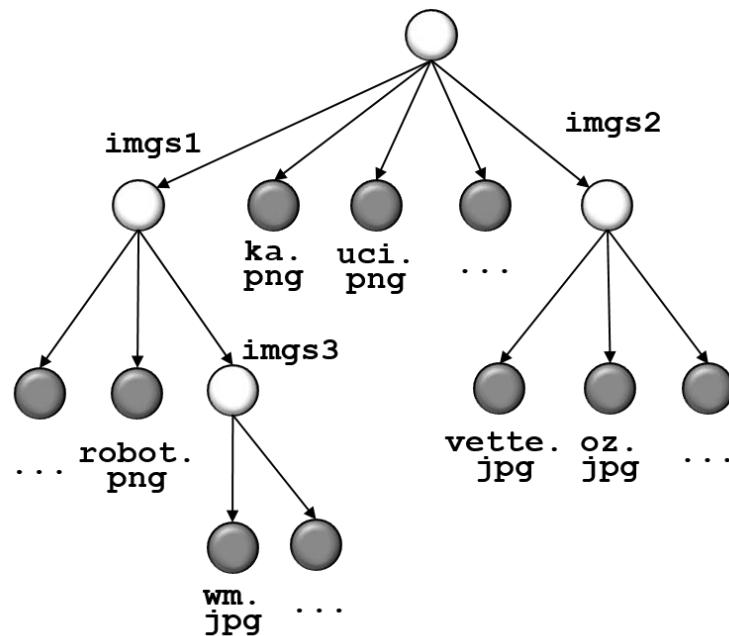
- We show how to apply key completable stage methods in the context of a program that crawls web pages recursively
  - This program counts the # of images on each page

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



# Applying Completion Stage Methods to Image Counter

- 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 final 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 final 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 final 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)  
        .exceptionally(ex -> 0)  
        .thenAccept(total ->  
            print(TAG + ":"  
                  + total  
                  + " total image(s) are reachable from "  
                  + rootUri))  
        .join();  
}
```



*Perform image counting &  
return future to Integer*

# 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();  
}
```

# 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::getImagesInPage)  
        .thenApplyList::size;  
  
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))  
  
    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) {
```

*The control flow can  
be modeled via a  
data flow diagram*

Task1: Get start page asynchronously

Task 2: Count images on  
the page asynchronously

Task 3: Count images on all  
hyperlinked pages asynchronously

Task 4: Combine results to create the total asynchronously

```
CompletableFuture<Integer> ImagesInPageF = pageF  
.thenApplyAsync(this :: getImagesInPage)  
.thenApplyAsync(this :: crawlLinksInPage)  
.thenComposeAsync(page -> crawlLinksInPage(page, depth))  
  
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::getImagesInPage)  
        .thenApplyList::size);
```

*Asynchronously get the  
page at the root URI*

```
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

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

# 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

- Returns a future to the page at the root URI

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

*This code provides a thin wrapper around the jsoup Java HTML parser that works both with web content & local (file) content*

# 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::getImagesInPage)  
        .thenApply(List::size);
```



*Both of these two methods run concurrently after pageF completes*

```
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```



```
    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::getImagesInPage)  
        .thenApply(List::size);
```

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

```
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

```
    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::getImagesInPage)  
        .thenApply(List::size);  
  
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))  
  
    return combineImageCounts(imagesInPageF, imagesInLinksF); ...
```

*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 getImagesInPage(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 thenApplyAsync()*

# 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::getImagesInPage)  
    .thenApply(List::size);
```

*Sync return the # of IMG SRC URLs in this page*

```
CompletableFuture<Integer> imagesInLinksF = pageF  
    .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

```
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::getImagesInPage)  
        .thenApply(List::size);
```

```
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

*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::getImagesInPage)  
        .thenApply(List::size);
```

```
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

*Sync return future to 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));  
}
```

*Sync return future to the # of IMG SRC URLs accessible via links in 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))  
  
        .collect(FuturesCollector.toFuture())  
  
        .thenApply(list -> list.stream().reduce(0, Integer::sum));  
}
```

*This jsoup call operates synchronously, so the thenComposeAsync() method is used*

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

*After all futures have completed then create a new future  
that will trigger after all image counts are summed together*

# 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::getImagesInPage)  
        .thenApply(List::size);
```

```
    CompletableFuture<Integer> imagesInLinksF = pageF  
        .thenComposeAsync(page -> crawlLinksInPage(page, depth))
```

*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

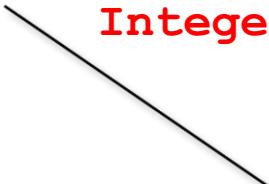
```
CompletableFuture<Integer> combineImageCounts  
    (CompletableFuture<Integer> imagesInPageF,  
     CompletableFuture<Integer> imagesInLinksF) {  
    imagesInPageF  
        .thenCombine(imagesInLinksF,  
                    Integer::sum);
```

*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

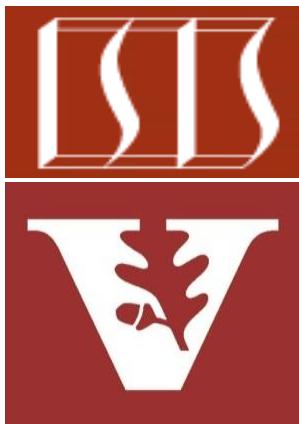
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

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

Professor of Computer Science

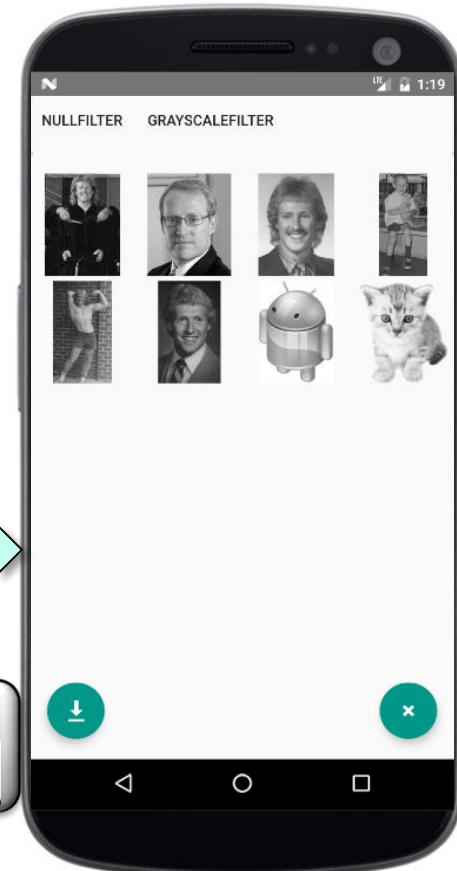
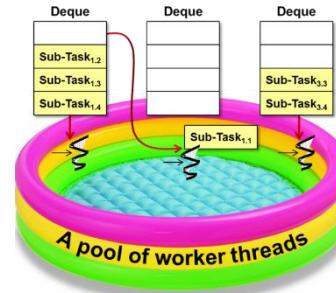
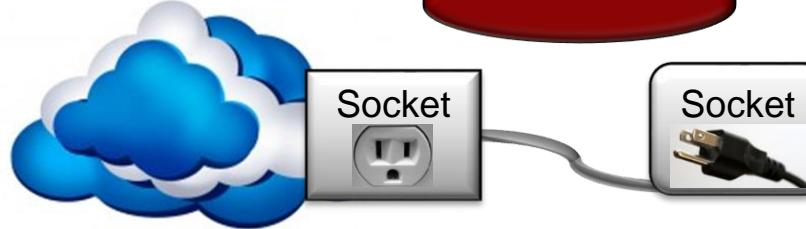
Institute for Software  
Integrated Systems

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



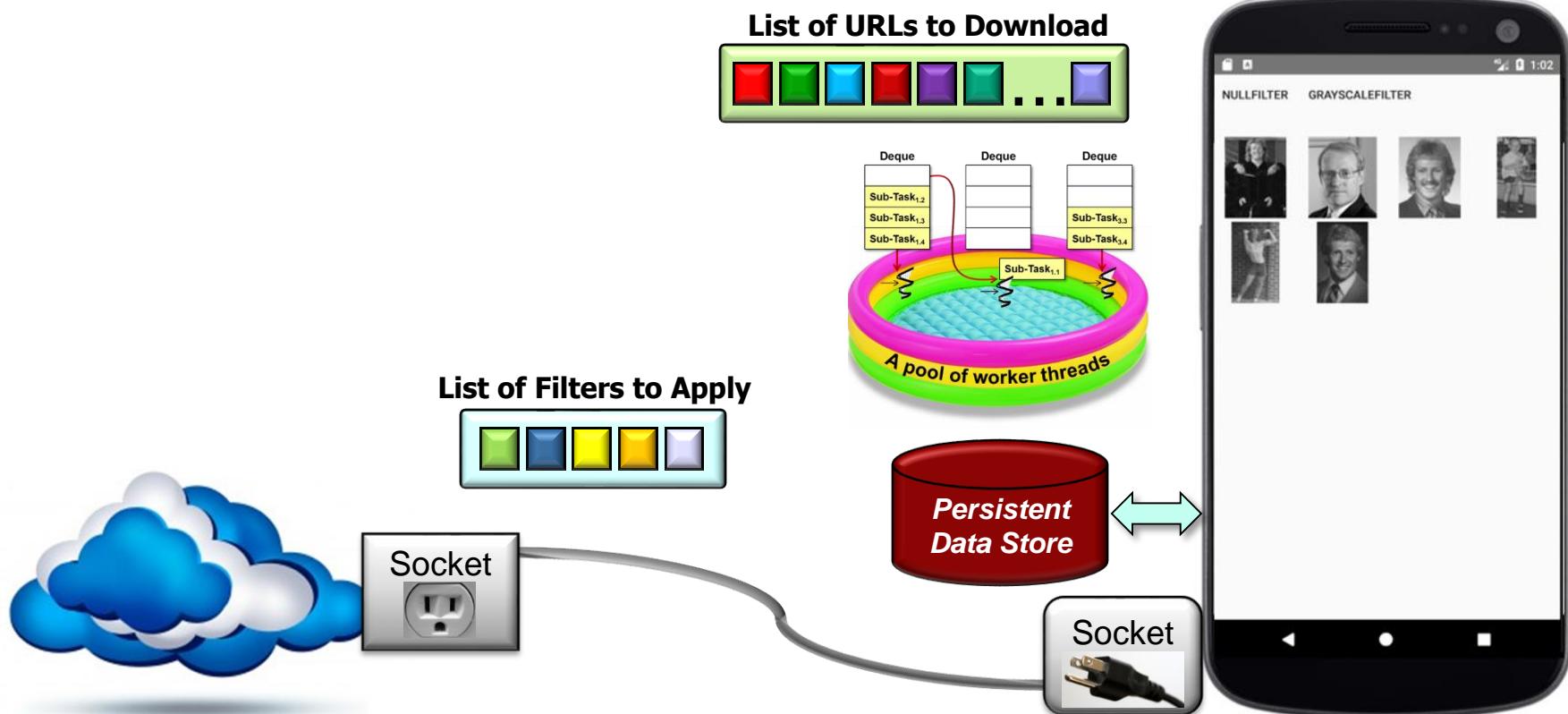
See [github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang](https://github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang)

---

# Overview of the Completable Futures ImageStreamGang

# Overview of Completable Futures ImageStreamGang

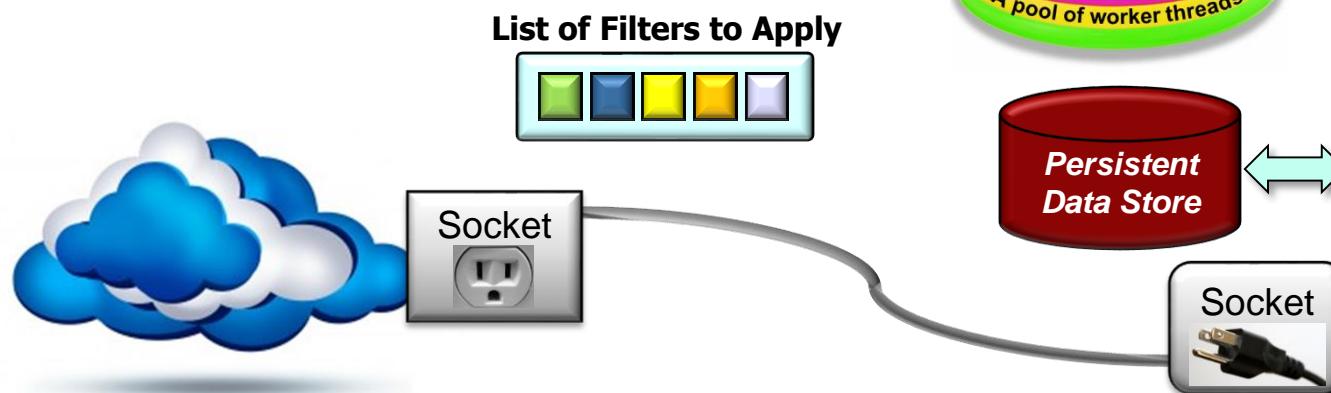
- ImageStreamGang applies completable future to optimize performance



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

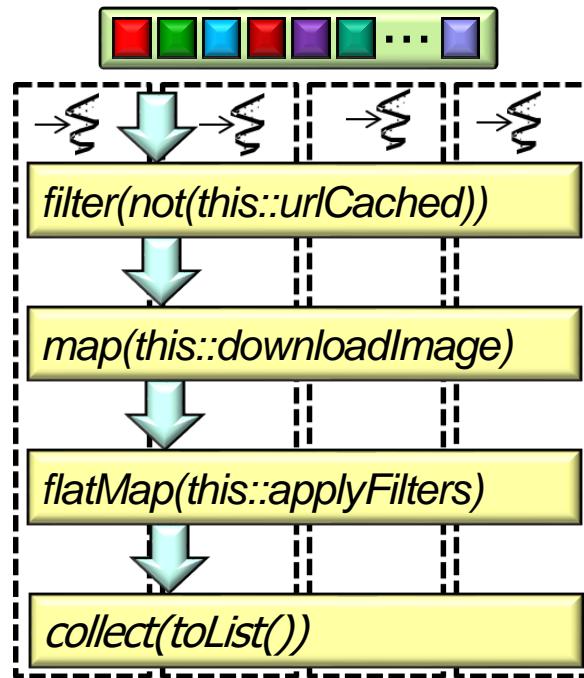
# Overview of Completable Futures ImageStreamGang

- ImageStreamGang applies completable future to optimize performance
  - 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

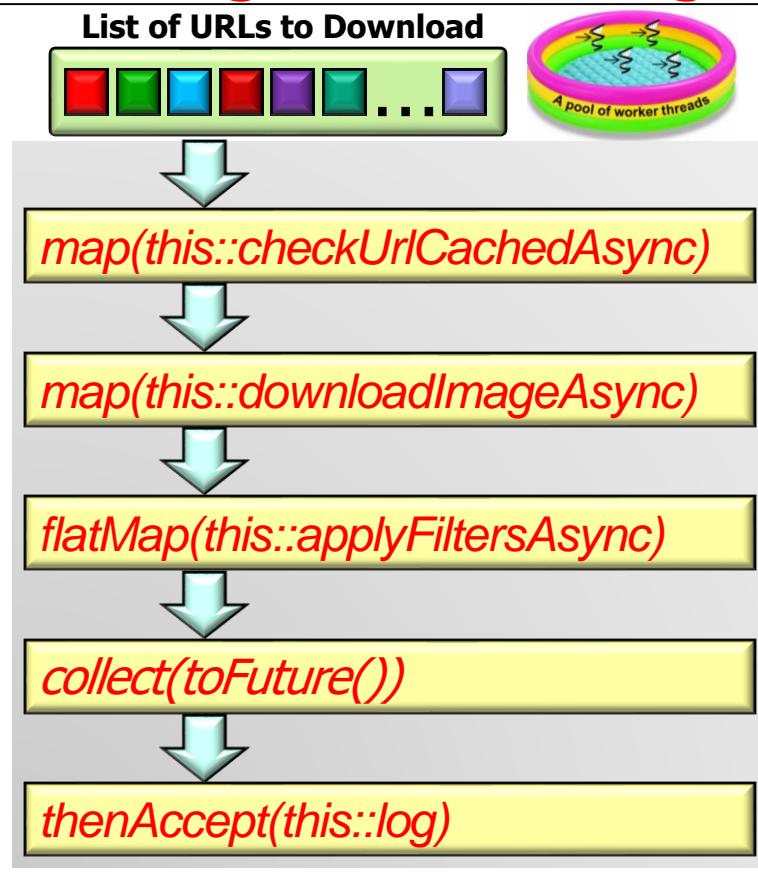


# Overview of Completable Futures ImageStreamGang

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



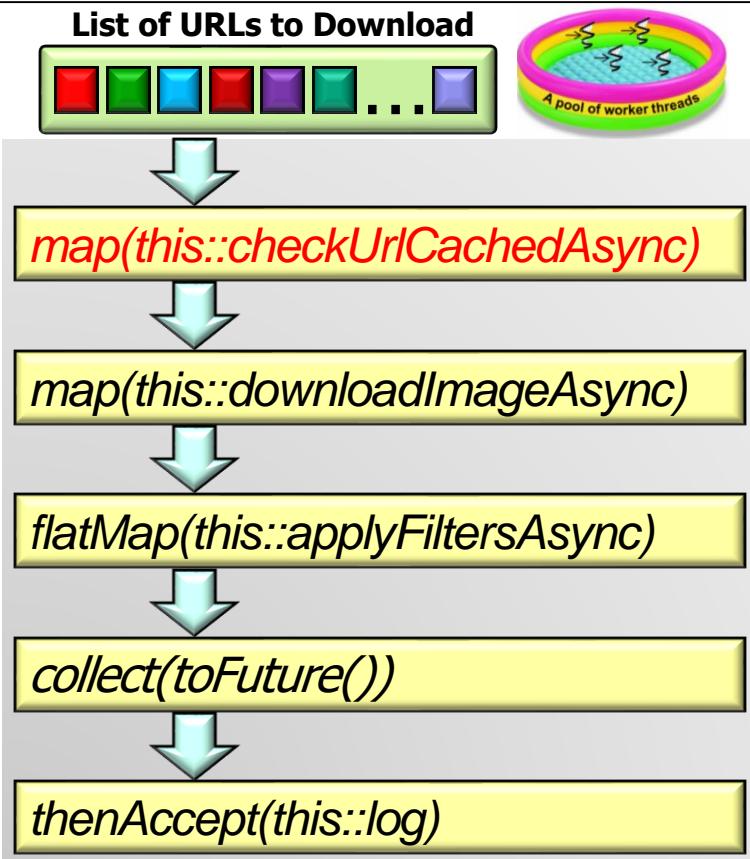
*Parallel Streams*



*Completable Futures*

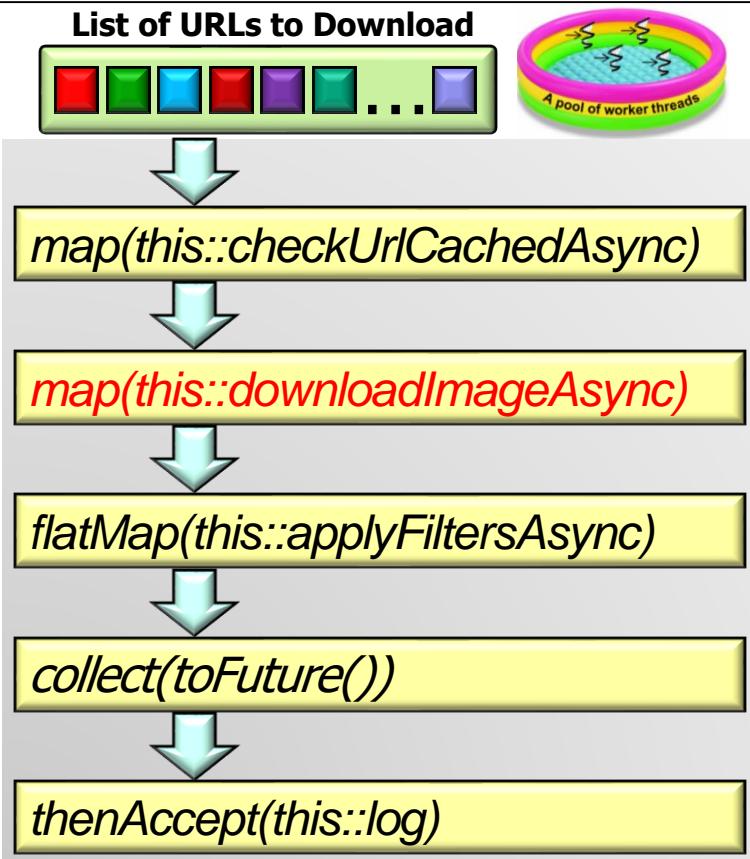
# Overview of Completable Futures ImageStreamGang

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



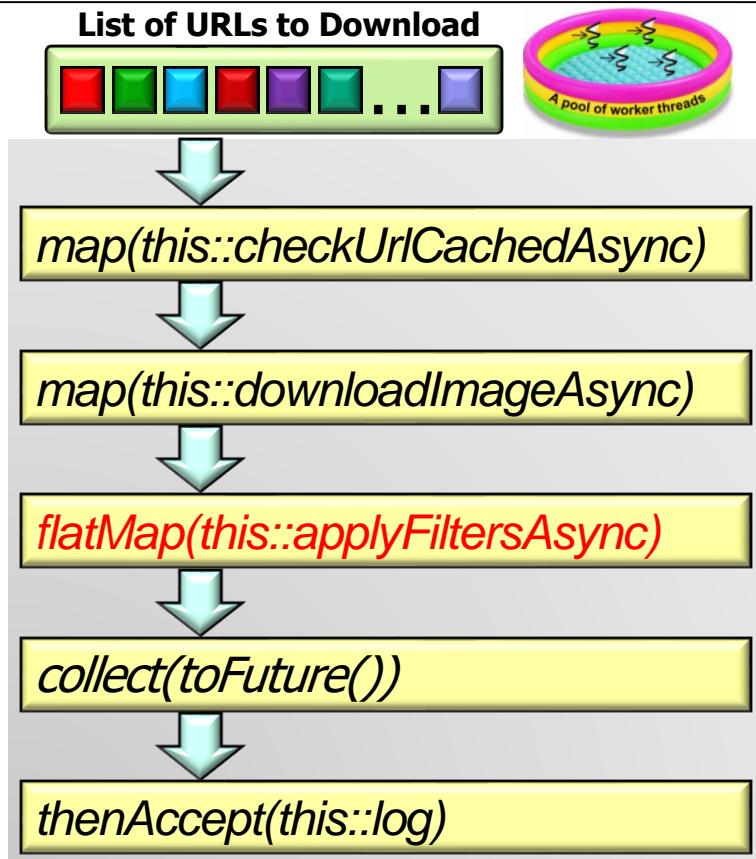
# Overview of Completable Futures ImageStreamGang

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



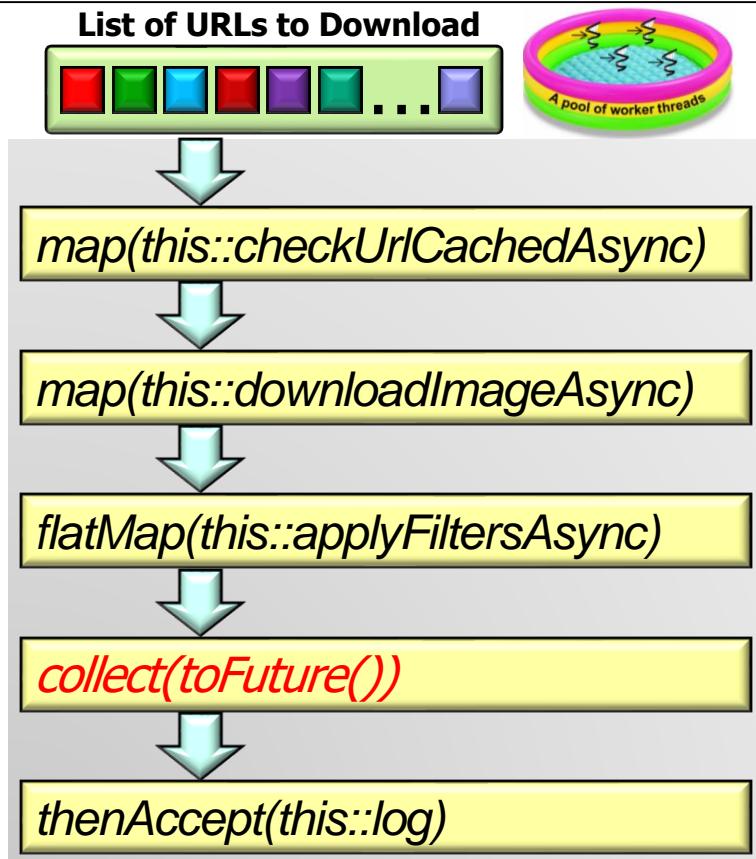
# Overview of Completable Futures ImageStreamGang

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



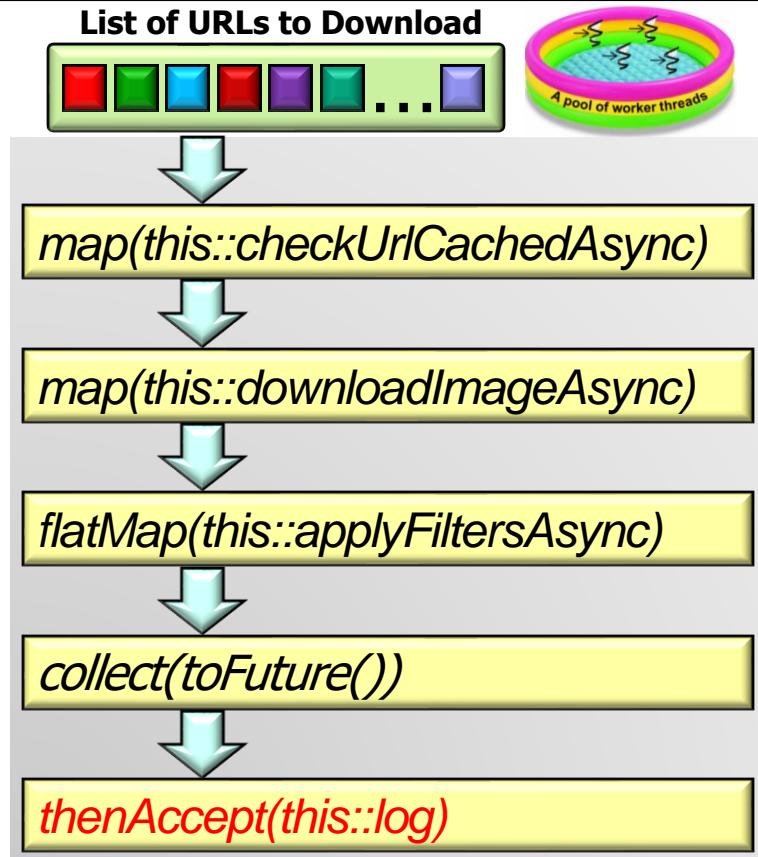
# Overview of Completable Futures ImageStreamGang

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



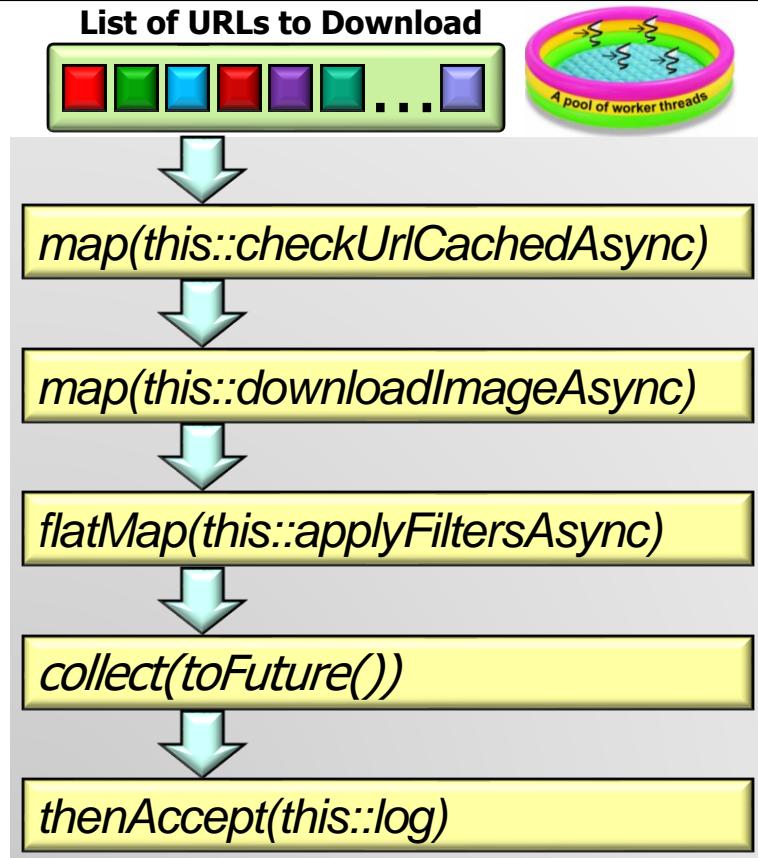
# Overview of Completable Futures ImageStreamGang

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



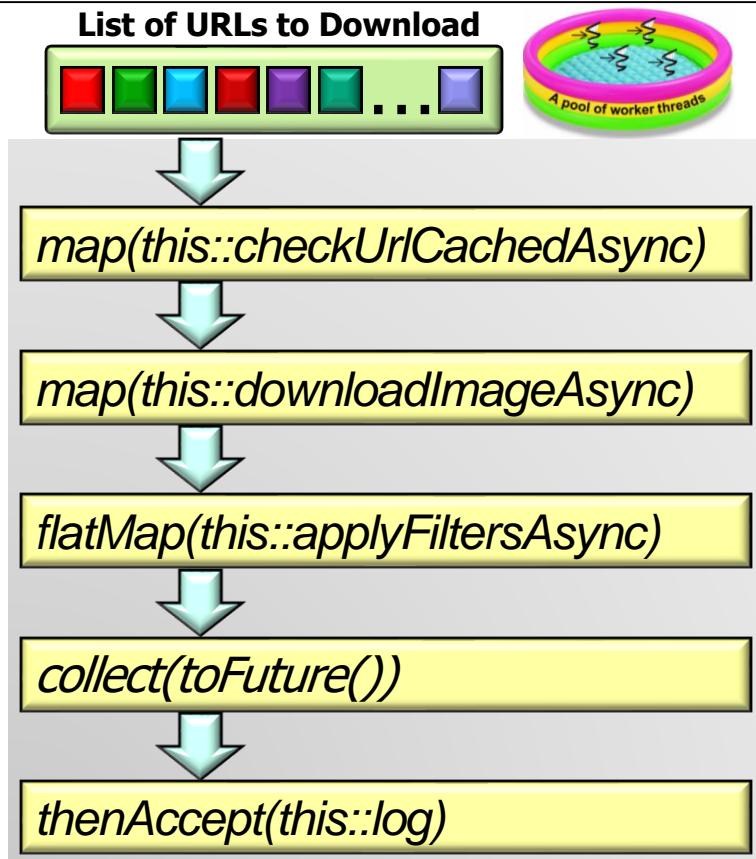
# Overview of Completable Futures ImageStreamGang

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



# Overview of Completable Futures ImageStreamGang

- Combining completable futures & streams helps to close the gap between the design intent & the implementation



---

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

# Java 8 CompletableFuture

## ImageStreamGang Example (Part 2)

Douglas C. Schmidt

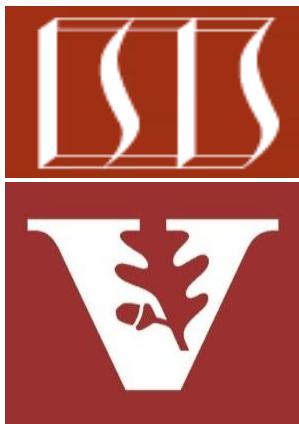
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

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

Professor of Computer Science

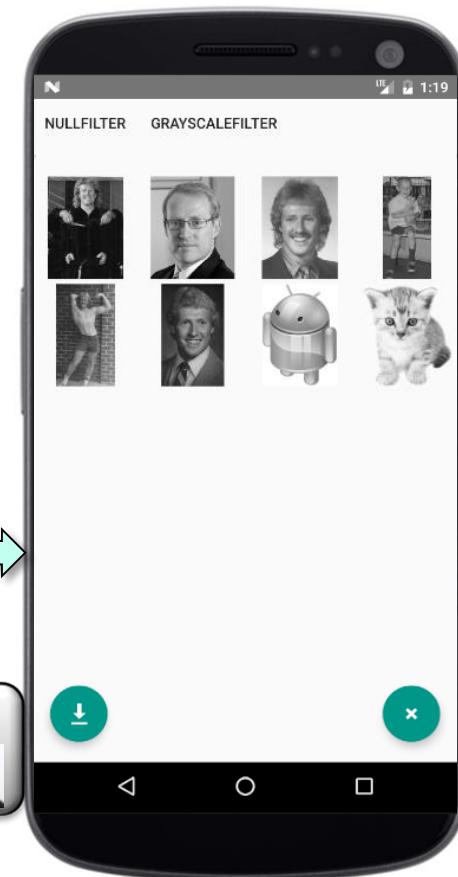
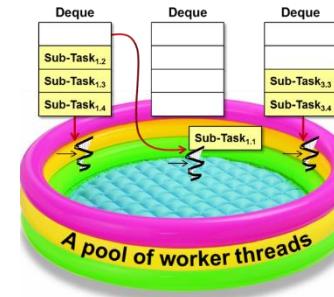
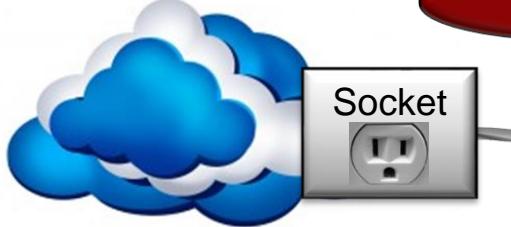
Institute for Software  
Integrated Systems

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 to apply completable futures to ImageStreamGang



# Learning Objectives in this Part of the Lesson

- Understand the design of the Java 8 completable future version of ImageStreamGang
- Know how to apply completable futures to ImageStreamGang, e.g.
  - 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<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

See [imagestreamgang/streams/ImageStreamCompletableFuture1.java](#)

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()

*Get the list of URLs  
input by the user*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();
```

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

*Combines a Java 8  
sequential stream with  
completable futures*

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()

*Factory method creates  
a stream of URLs*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();
```

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams

*Asynchronously check if images  
are already cached locally*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

map() converts a stream of URLs to a stream of futures to optional URLs

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams

*Asynchronously download  
an image at each given URL*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();
```

map() converts URL futures (completed) to image futures (downloading)

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams

*Asynchronously filter & store downloaded images on the local file system*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

flatMap() converts image futures (completed) to filtered image futures (xforming/storing)

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

*Create a future used to wait for all  
async operations associated w/the  
stream of futures to complete*

See next part on “arbitrary-arity” methods in CompletableFuture

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams

*This lambda logs the results  
when all the futures in stream  
complete their async processing*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

# Applying Completable Futures to ImageStreamGang

- Focus on processStream()
  - This implementation is very different from parallel streams

*Wait until all the images have been downloaded, processed, & stored*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

This is the one & only call to join() in this implementation strategy

---

# Applying Factory Methods in ImageStreamGang

# Applying Factory Methods in ImageStreamGang

---

- Initiate an async check to see if images are cached locally

*map() calls the behavior  
checkUrlCachedAsync()*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

# Applying Factory Methods in ImageStreamGang

- Initiate an async check to see if images are cached locally

*Asynchronously check if a URL is already downloaded*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

# Applying Factory Methods in ImageStreamGang

- Initiate an async check to see if images are cached locally

*Returns a stream of completable futures to optional URLs, which have a value if the URL is not cached or are empty if it is cached*



```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

Later behaviors simply ignore “empty” optional URL values

# Applying Factory Methods in ImageStreamGang

---

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

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url)  
                ? null  
                : url),  
            getExecutor());  
}
```

# Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url))  
                ? null  
                : url),  
        getExecutor());  
}
```

*This factory method registers an action that runs asynchronously*

# Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url))  
                ? null  
                : url),  
        getExecutor());  
}
```

*supplyAsync() runs action in a worker thread from the common fork-join pool*

```
void initiateStream() {  
    // Set the executor to the common fork-join pool.  
    setExecutor(ForkJoinPool.commonPool());  
    ...  
}
```

# Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url))  
                ? null  
                : url),  
        getExecutor());  
}
```

*ofNullable() is a factory method that returns a completable future to an optional URL, which has a value if the URL is not cached or is empty if it is cached*

# Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url))  
                ? null  
                : url),  
        getExecutor());  
}
```

*Returns true if the image has already been filtered before*

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

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

# Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url))  
                ? null  
                : url),  
        getExecutor());  
}
```

*Returns true if image file already exists*

```
boolean urlCached(URL url, String filterName) {  
    File file = new File(getPath(), filterName);  
    File imageFile = new File(file, getNameForUrl(url));  
    return !imageFile.createNewFile();  
}
```

See [imagestreamgangstreams/ImageStreamGang.java](#)

# Applying Factory Methods in ImageStreamGang

- checkUrlCachedAsync() uses the supplyAsync() factory method internally

```
CompletableFuture<Optional<URL>> checkUrlCachedAsync (URL url) {  
    return CompletableFuture  
        .supplyAsync(() ->  
            Optional.ofNullable(urlCached(url))  
                ? null  
                : url),  
        getExecutor());  
}
```



ClearlyBetter®  
SOLUTIONS

```
boolean urlCached(URL url, String filterName) {  
    File file = new File(getPath(), filterName);  
    File imageFile = new File(file, getNameForUrl(url));  
    return !imageFile.createNewFile();  
}
```

There are clearly better ways of implementing an image cache!

---

# Applying Completion Stage Methods in ImageStreamGang

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously download an image at each given URL

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

*map() calls the behavior downloadImageAsync()*

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously download an image at each given URL

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

*Asynchronously downloads an image & stores it in memory*



Later behaviors simply ignore “empty” optional images

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously download an image at each given URL

*Returns a stream of futures to optional images, which have a value if the image is being downloaded or are empty if it is already cached*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

# Applying Completion Stage Methods in ImageStreamGang

- downloadImageAsync() uses the thenApplyAsync() method internally

```
CompletableFuture<Optional<Image>> downloadImageAsync  
        (CompletableFuture<Optional<URL>> urlFuture) {  
    return urlFuture  
        .thenApplyAsync(urlOpt ->  
            urlOpt  
            .map(this::blockingDownload),  
            getExecutor());  
}
```

*Asynchronously download  
image when future completes*

See [imagestreamgang/streams/ImageStreamCompletableFuture1.java](https://github.com/imagestreamgang/streams/blob/main/src/main/java/com/github/imagestreamgang/streams/ImageStreamCompletableFuture1.java)

# Applying Completion Stage Methods in ImageStreamGang

- downloadImageAsync() uses the thenApplyAsync() method internally

```
CompletableFuture<Optional<Image>> downloadImageAsync  
    (CompletableFuture<Optional<URL>> urlFuture) {  
    return urlFuture  
        .thenApplyAsync(urlOpt ->  
            urlOpt  
            .map(this::blockingDownload),  
            getExecutor());  
}
```

*This completion stage method registers an action that's not executed immediately, but only after the future completes*

# Applying Completion Stage Methods in ImageStreamGang

- downloadImageAsync() uses the thenApplyAsync() method internally

```
CompletableFuture<Optional<Image>> downloadImageAsync  
        (CompletableFuture<Optional<URL>> urlFuture) {  
    return urlFuture  
        .thenApplyAsync(urlOpt ->  
            urlOpt  
            .map(this::blockingDownload),  
            getExecutor());  
}
```

*If a url is present when the future completes download it & return an optional describing the result; otherwise return an empty optional*

# Applying Completion Stage Methods in ImageStreamGang

- downloadImageAsync() uses the thenApplyAsync() method internally

```
CompletableFuture<Optional<Image>> downloadImageAsync  
    (CompletableFuture<Optional<URL>> urlFuture) {  
    return urlFuture  
        .thenApplyAsync(urlOpt ->  
            urlOpt  
            .map(this::blockingDownload),  
            getExecutor());  
}
```

*Asynchronously run blockingDownload() if the url is non-empty*

# Applying Completion Stage Methods in ImageStreamGang

- downloadImageAsync() uses the thenApplyAsync() method internally

```
CompletableFuture<Optional<Image>> downloadImageAsync  
    (CompletableFuture<Optional<URL>> urlFuture) {  
    return urlFuture  
        .thenApplyAsync(urlOpt ->  
            urlOpt  
            .map(this::blockingDownload),  
            getExecutor());  
}
```

*Use the common fork-join pool & its ManagedBlocker mechanism*

You could also use a ThreadPoolExecutor (fixed-size or cached)

# Applying Completion Stage Methods in ImageStreamGang

- `downloadImageAsync()` uses the `thenApplyAsync()` method internally

```
Image blockingDownload(URL url) {  
    return BlockingTask  
        .callInManagedBlock(() ->  
            downloadImage(url));  
}
```

*Transform a URL into an Image by  
downloading each image via the URL*

See [imagestreamgang/streams/ImageStreamGang.java](https://github.com/imagestreamgang/streams/tree/main/src/main/java/com/github/imagestreamgang/streams/ImageStreamGang.java)

# Applying Completion Stage Methods in ImageStreamGang

- downloadImageAsync() uses the thenApplyAsync() method internally

```
Image blockingDownload(URL url) {  
    return BlockingTask  
        .callInManagedBlock(() ->  
            downloadImage(url));  
}
```

*BlockingTask.callInManagedBlock() wraps the ManagedBlocker interface, which expands the common fork/join thread pool to handle the blocking image download*

See [imagestreamgang/utils/BlockingTask.java](#)

# Applying Completion Stage Methods in ImageStreamGang

- `downloadImageAsync()` uses the `thenApplyAsync()` method internally

```
CompletableFuture<Optional<Image>> downloadImageAsync  
    (CompletableFuture<Optional<URL>> urlFuture) {  
    return urlFuture  
        .thenApplyAsync(urlOpt ->  
            urlOpt  
            .map(this::blockingDownload),  
            getExecutor());  
}
```

*Returns a future to an image that completes  
when the image is finished downloading*

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously filter & store downloaded images on the local file system

*flatMap() calls behavior applyFiltersAsync()*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously filter & store downloaded images on the local file system

*Asynchronous filter images & store them into files*



```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

Later operations ignore “empty” optional images

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously filter & store downloaded images on the local file system



*"Flatten" all filtered/stored images into a single output stream*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously filter & store downloaded images on the local file system

Returns a stream of futures to optional images, which have a value if the image is being filtered or are empty if it is already cached

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image) .run() ) ,  
            getExecutor() ) ;  
}
```

*Asynchronously filter images & then store them into files*

See [imagestreamgangstreamsImageStreamCompletableFuture1.java](#)

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
    (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image) .run()) ,  
                getExecutor()));  
}
```

*Convert the list of filters into a stream*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image) .run() ) ,  
                getExecutor() ) );  
}
```



*Asynchronously apply a filter action  
after the previous stage completes*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image).run())  
                .getExecutor()));  
}
```

*This completion stage method registers an action that's not executed immediately, but runs only after the future completes*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image) .run() ) ,  
                getExecutor() ) ); }
```

*If an image is present then perform the action & return optional containing result; otherwise return an empty optional*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image).run())  
                .getExecutor()));  
}
```

*If an image is non-null then asynchronously filter the image & store it in an output file*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                    (filter, image) .run() ) ,  
            getExecutor() ) ); }
```

*thenApplyAsync() runs the actions in a thread from the common fork-join pool*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image).run()) ,  
                getExecutor()));  
}
```

*It also returns a new completable future that will trigger when the image has been filtered/stored*

# Applying Completion Stage Methods in ImageStreamGang

- `applyFiltersAsync()` uses the `thenApplyAsync()` method internally

```
Stream<CompletableFuture<Optional<Image>>> applyFiltersAsync  
        (CompletableFuture<Optional<Image>> imageFuture) {  
    return mFilters  
        .stream()  
  
        .map(filter -> imageFuture  
            .thenApplyAsync(imageOpt ->  
                imageOpt  
                .map(image ->  
                    makeFilterDecoratorWithImage  
                        (filter, image) .run() ) ,  
                getExecutor() ) );  
}
```

*applyFiltersAsync() returns  
a stream of completable  
futures to optional images*

# Applying Completion Stage Methods in ImageStreamGang

- Asynchronously filter & store downloaded images on the local file system



*flatMap() merges the stream of futures returned by applyFilters Async() into a single stream*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

This stream is processed by collect(), as discussed in the next part of the lesson

---

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

# Java 8 CompletableFuture

## ImageStreamGang Example (Part 3)

Douglas C. Schmidt

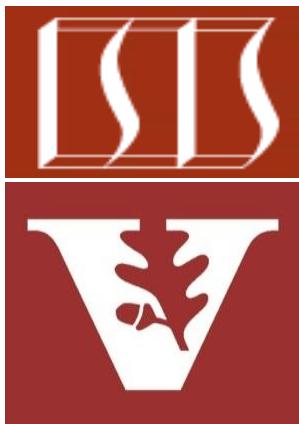
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[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 to apply completable futures to ImageStreamGang, e.g.
  - Factory methods & completion stage methods
  - Arbitrary-arity 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>`

---

# Applying Arbitrary-Arity Methods in ImageStreamGang

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored

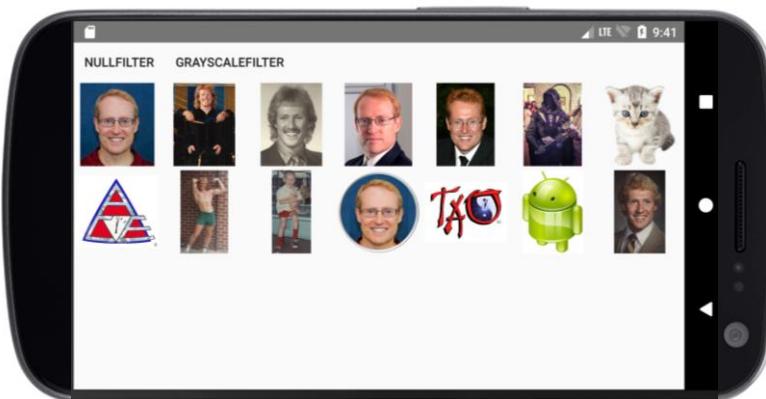
```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
            .join();  
}
```

*Provides a single means to await completion of a set of futures before continuing with the program*

collect() also triggers processing of all the intermediate operations

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored
  - These images are displayed after processing completes



```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored
  - These images are displayed after processing completes
  - StreamOfFuturesCollector wraps “arbitrary-arity” allOf() method

*Return a future that completes when all futures in the stream complete*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored
  - These images are displayed after processing completes
  - StreamOfFuturesCollector wraps “arbitrary-arity” allOf() method

*Log the results after the final future completes*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored
  - These images are displayed after processing completes
  - StreamOfFuturesCollector wraps “arbitrary-arity” allOf() method

*Remove empty optional values from the stream in Java 9*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored
  - These images are displayed after processing completes
  - StreamOfFuturesCollector wraps “arbitrary-arity” allOf() method

*Remove empty optional values  
from the stream in Java 8*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream -> log(stream  
            .filter(Optional::isPresent)  
            .map(Optional::get),  
            urls.size()))  
        .join();  
}
```

# Applying Arbitrary-Arity Methods in ImageStreamGang

- collect() returns a completable future to a stream of futures to images being asynchronously downloaded, filtered, & stored
  - These images are displayed after processing completes
  - StreamOfFuturesCollector wraps “arbitrary-arity” allOf() method

*Wait until all asynchronous processing is completed*

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
    resultsFuture = urls  
        .stream()  
        .map(this::checkUrlCachedAsync)  
        .map(this::downloadImageAsync)  
        .flatMap(this::applyFiltersAsync)  
        .collect(toFuture())  
        .thenApply(stream ->  
            log(stream.flatMap  
                (Optional::stream),  
                urls.size()))  
        .join();
```

---

# Implementing the Class StreamOfFuturesCollector

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()



<<Java Interface>>

I Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



<<Java Class>>

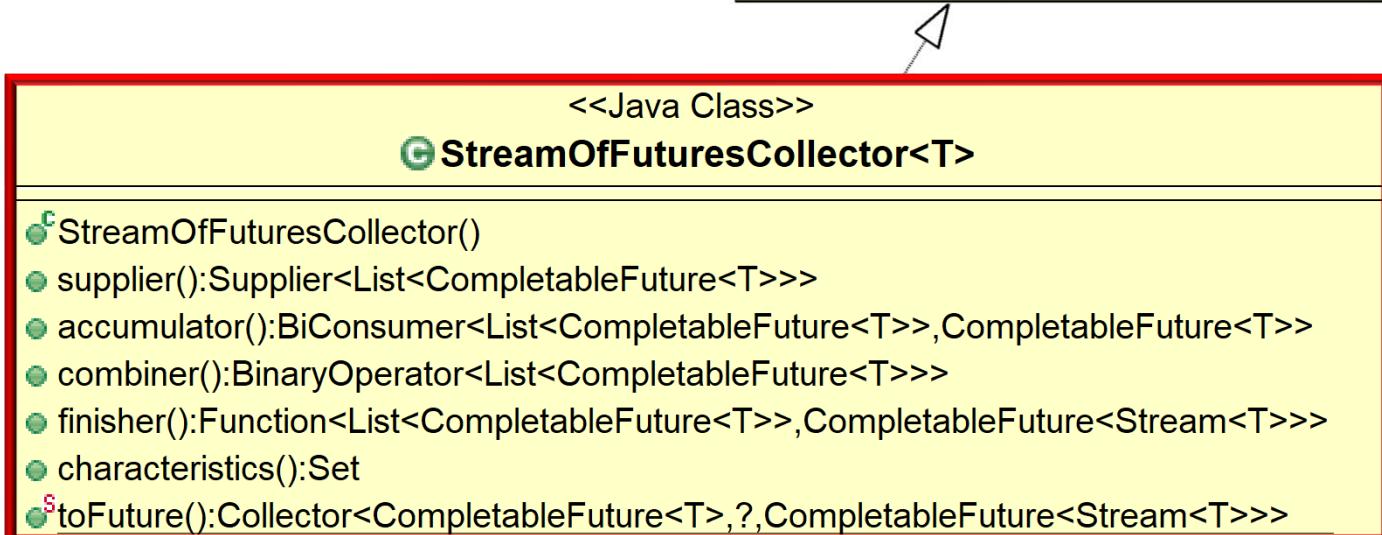
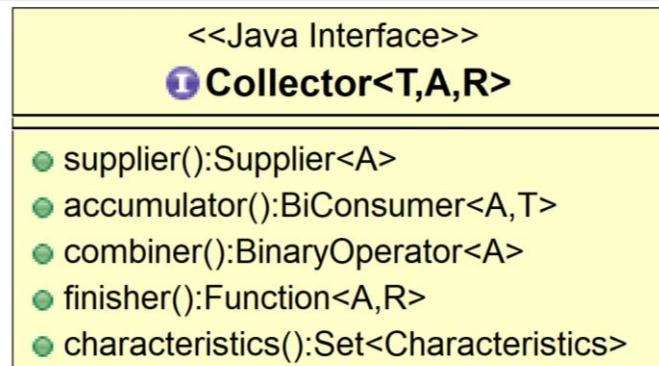
G StreamOfFuturesCollector<T>

- StreamOfFuturesCollector()
- supplier():Supplier<List<CompletableFuture<T>>>
- accumulator():BiConsumer<List<CompletableFuture<T>>,CompletableFuture<T>>
- combiner():BinaryOperator<List<CompletableFuture<T>>>
- finisher():Function<List<CompletableFuture<T>>,CompletableFuture<Stream<T>>>
- characteristics():Set
- <sup>S</sup>toFuture():Collector<CompletableFuture<T>,>?,CompletableFuture<Stream<T>>>

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

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()
  - Converts a *stream* of completable futures into a *single* completable future that's triggered when *all* futures in the stream complete



StreamOfFuturesCollector is a non-concurrent collector (supports parallel & sequential streams)

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps 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

<<Java Interface>>

Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



<<Java Class>>

StreamOfFuturesCollector<T>

- StreamOfFuturesCollector()
- supplier():Supplier<List<CompletableFuture<T>>>
- accumulator():BiConsumer<List<CompletableFuture<T>>,CompletableFuture<T>>
- combiner():BinaryOperator<List<CompletableFuture<T>>>
- finisher():Function<List<CompletableFuture<T>>,CompletableFuture<Stream<T>>>
- characteristics():Set
- toFuture():Collector<CompletableFuture<T>,>?,CompletableFuture<Stream<T>>>

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)

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()



<<Java Interface>>

**I Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



<<Java Class>>

**G StreamOfFuturesCollector<T>**

- StreamOfFuturesCollector()
- supplier():Supplier<List<CompletableFuture<T>>>
- accumulator():BiConsumer<List<CompletableFuture<T>>,CompletableFuture<T>>
- combiner():BinaryOperator<List<CompletableFuture<T>>>
- finisher():Function<List<CompletableFuture<T>>,CompletableFuture<Stream<T>>>
- characteristics():Set
- <sup>S</sup>toFuture():Collector<CompletableFuture<T>,?,CompletableFuture<Stream<T>>>

StreamOfFuturesCollector provides a powerful wrapper for some complex code!

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

...

*Implements a  
custom collector*

# Implementing the Class StreamOfFuturesCollector

---

- StreamOfFuturesCollector wraps allOf()

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

...

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

# Implementing the Class StreamOfFuturesCollector

---

- StreamOfFuturesCollector wraps allOf()

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

...

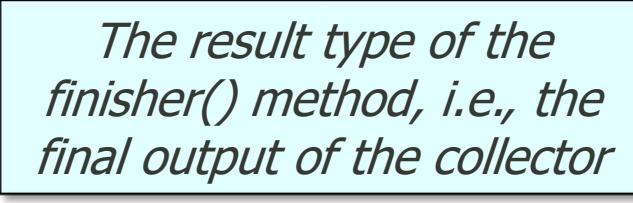
*The mutable accumulation type  
of the accumulator() method*

# Implementing the Class StreamOfFuturesCollector

---

- StreamOfFuturesCollector wraps allOf()

```
public class StreamOfFuturesCollector<T>  
    implements Collector<CompletableFuture<T>,  
              List<CompletableFuture<T>>,  
              CompletableFuture<Stream<T>> {  
    ...  
}
```



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

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

The Stream<T> parameter differs from the List<T> parameter used by the earlier FuturesCollector

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

```
public class StreamOfFuturesCollector<T>
    implements Collector<CompletableFuture<T>,
               List<CompletableFuture<T>>,
               CompletableFuture<Stream<T>>> {
    public Supplier<List<CompletableFuture<T>>> supplier() {
        return ArrayList::new;
    }
```

*This factory method returns a supplier used by the Java 8 streams collector framework to create a new mutable array list container*

```
public BiConsumer<List<CompletableFuture<T>>,
                  CompletableFuture<T>> accumulator()
{ return List::add; }
...
```

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

```
public class StreamOfFuturesCollector<T>
    implements Collector<CompletableFuture<T>,
               List<CompletableFuture<T>>,
               CompletableFuture<Stream<T>> {
    public Supplier<List<CompletableFuture<T>>> supplier() {
        return ArrayList::new;
    }
```

*This factory method returns a bi-consumer used by the Java 8 streams collector framework to add a new completable future into the mutable array list container*

```
public BiConsumer<List<CompletableFuture<T>>,
                  CompletableFuture<T>> accumulator()
{ return List::add; }
...
```

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

*This factory method returns a binary operator that merges two partial array list results into a single array list (only relevant for parallel streams)*

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

*This factory method returns a function used by the Java 8 streams collector framework to transform the array list accumulation type to the completable future result type*

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

```
public class StreamOfFuturesCollector<T>
{
    ...
    public Function<List<CompletableFuture<T>>,  

                    CompletableFuture<Stream<T>>> finisher() {  

        return futures -> CompletableFuture  

            .allOf(futures.toArray(new CompletableFuture[0]))  

            .thenApply(v -> futures.stream()  

                .map(CompletableFuture::join));  

    }
    ...
}
```

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

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

```
public class StreamOfFuturesCollector<T>  
{  
    ...  
    public Function<List<CompletableFuture<T>>,  
                      CompletableFuture<Stream<T>>> finisher() {  
        return futures -> CompletableFuture  
            .allOf(futures.toArray(new CompletableFuture[0]))  
  
            When all futures have completed get a single  
            future to a stream of joined elements of type T  
            .thenApply(v -> futures.stream()  
                .map(CompletableFuture::join));  
    }  
    ...  
}
```

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

```
public class StreamOfFuturesCollector<T>  
{  
    ...  
    public Function<List<CompletableFuture<T>>,  
                      CompletableFuture<Stream<T>>> finisher() {  
        return futures -> CompletableFuture  
            .allOf(futures.toArray(new CompletableFuture[0]))  
    }  
    ...  
}
```

*Convert the array list of futures into a stream of futures*

```
.thenApply(v -> futures.stream()  
            .map(CompletableFuture::join));
```

}

...

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

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

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

*Return a future to a stream of elements of T*

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

*Returns a set indicating the characteristics  
of the StreamOfFutureCollector class*

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

StreamOfFuturesCollector is thus a *non-concurrent* collector

# Implementing the Class StreamOfFuturesCollector

- StreamOfFuturesCollector wraps allOf()

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

*This static factory method creates a new StreamOfFuturesCollector*

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

# Implementing the Class StreamOfFuturesCollector

- `toFuture()` returns a future to a stream of futures to images that are being downloaded, filtered, & stored

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(stream ->  
                log(stream.flatMap  
                    (Optional::stream),  
                    urls.size()))  
  
    .join();
```

*Provides a single means to await completion of a set of futures before continuing with the program*

---

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

# Pros & Cons of Java 8 CompletableFuture

Douglas C. Schmidt

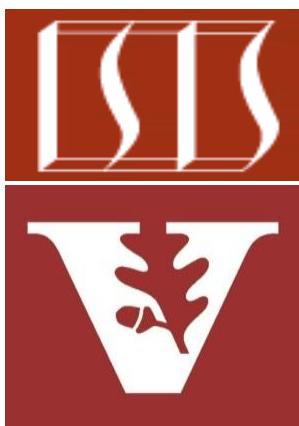
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[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

---

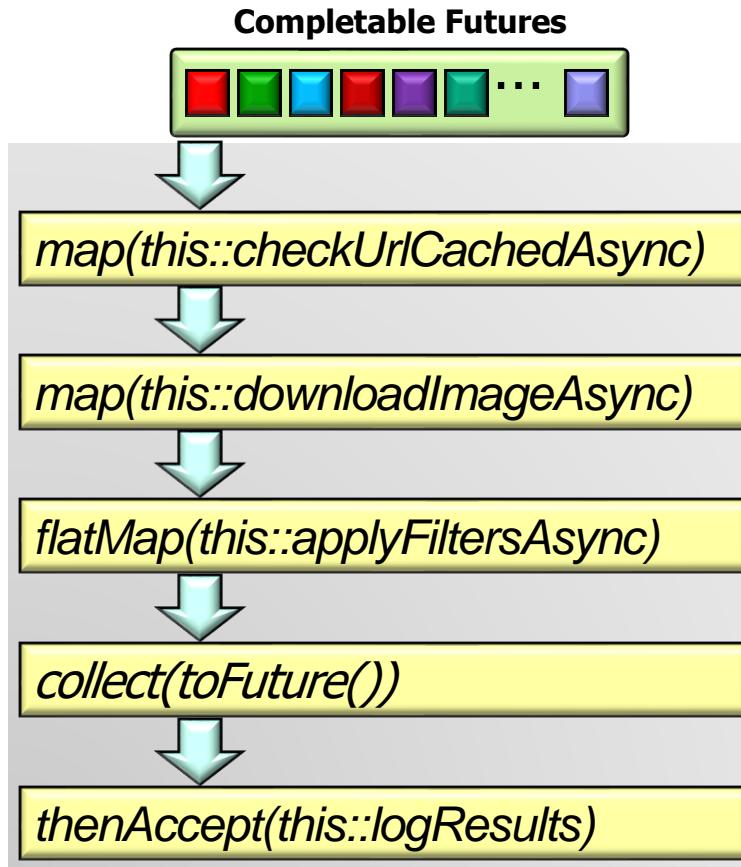
- Understand the pros & cons of using the completable futures framework



---

# Pros & Cons of Java 8 CompletableFuture

# Pros & Cons of Java 8 Completable Futures



No explicit synchronization or threading is required in this implementation

# Pros & Cons of Java 8 Completable Futures

## Completable Futures



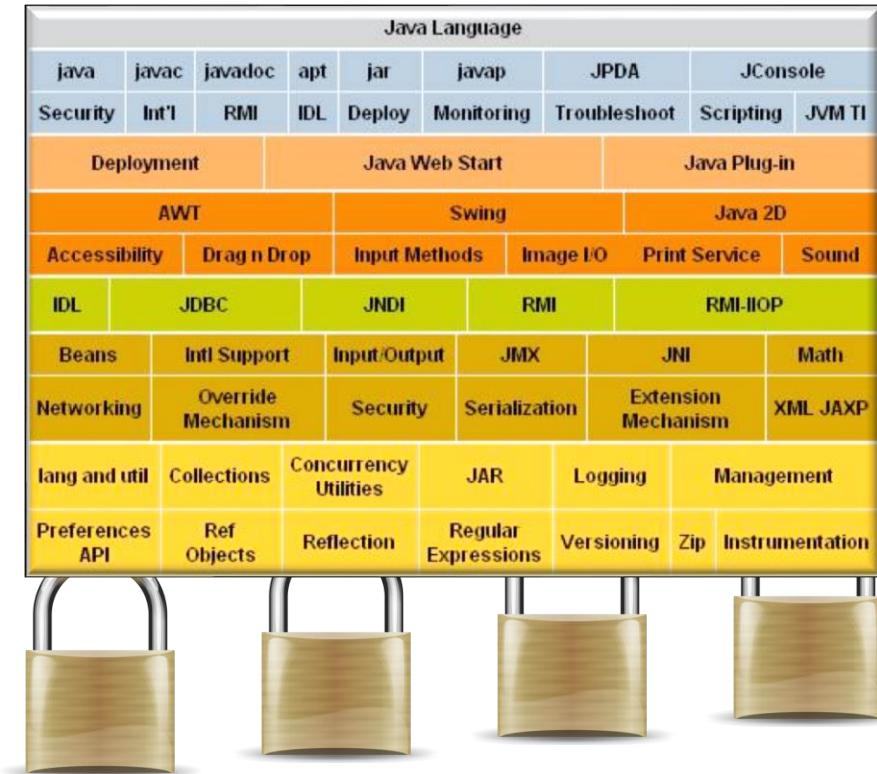
`map(this::checkUrlCachedAsync)`

`map(this::downloadImageAsync)`

`flatMap(this::applyFiltersAsync)`

`collect(toFuture())`

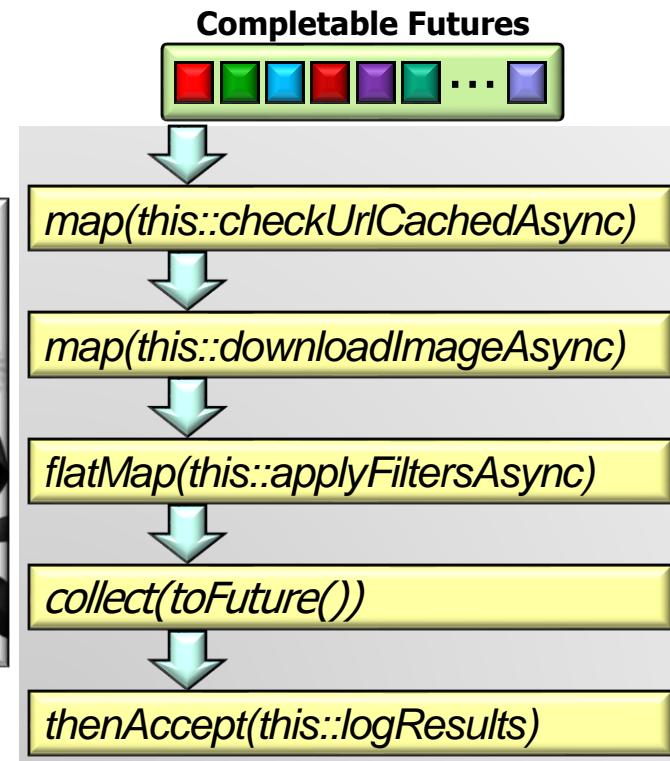
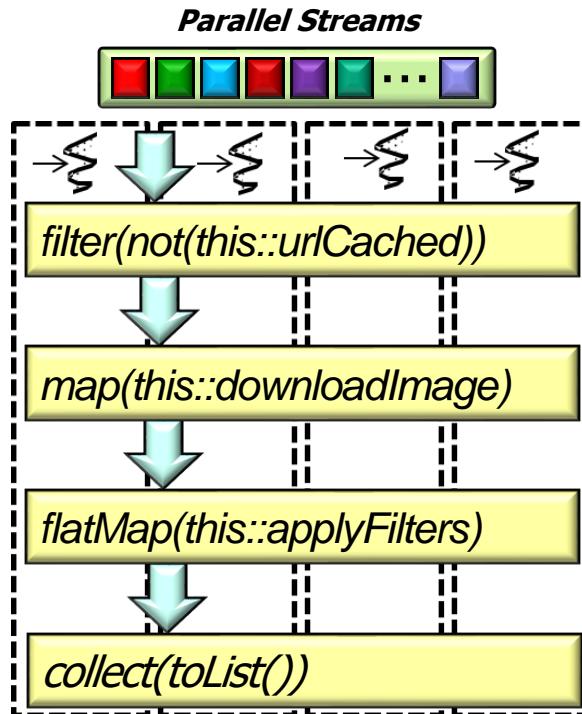
`thenAccept(this::logResults)`



Java libraries handle any locking needed to protect shared mutable state

# Pros & Cons of Java 8 Completable Futures

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



# Pros & Cons of Java 8 Completable Futures

---

- It's easier to program Java 8 parallel streams than completable futures

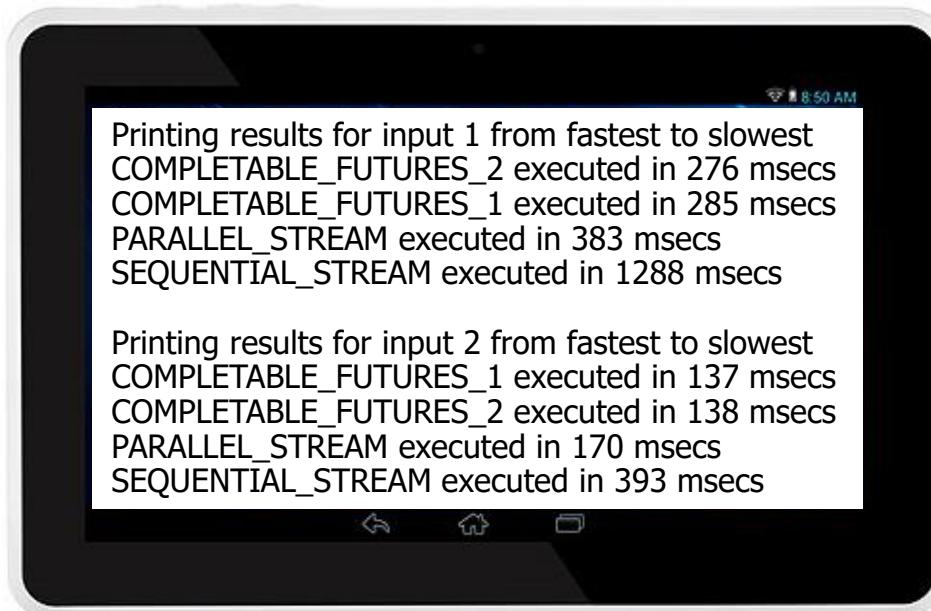
```
void processStream() {  
    List<URL> urls = getInput();  
  
    List<Image> filteredImages =  
        urls  
            .parallelStream()  
            .filter(not(this::urlCached))  
            .map(this::blockingDownload)  
            .flatMap(this::applyFilters)  
            .collect(toList());  
  
    logResults(filteredImages);  
}
```

```
void processStream() {  
    List<URL> urls = getInput();  
  
    CompletableFuture<Stream<Image>>  
        resultsFuture = urls  
            .stream()  
            .map(this::checkUrlCachedAsync)  
            .map(this::downloadImageAsync)  
            .flatMap(this::applyFiltersAsync)  
            .collect(toFuture())  
            .thenApply(this::logResults)  
            .join();  
    ...
```

In general, asynchrony patterns aren't as well understood by many developers

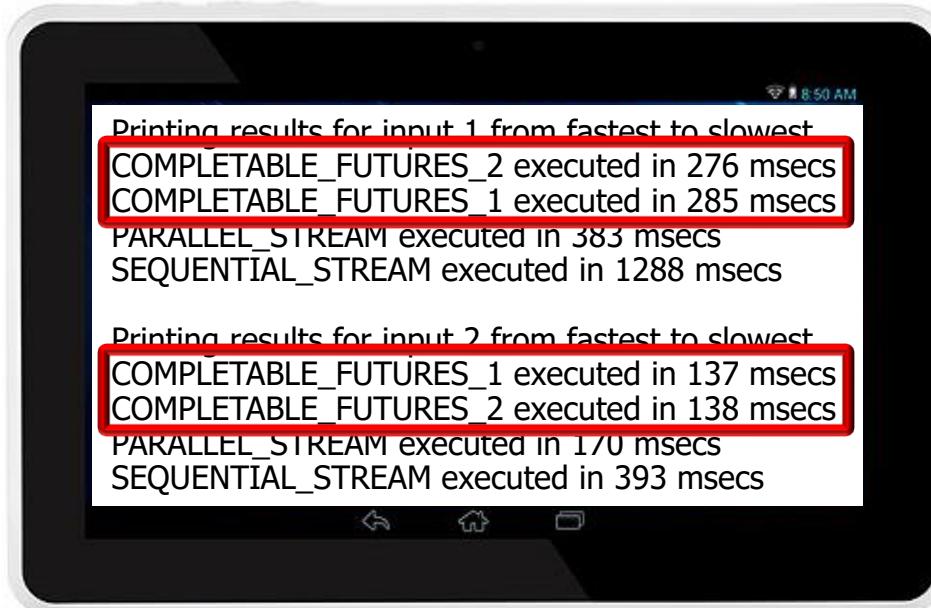
# Pros & Cons of Java 8 Completable Futures

- 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

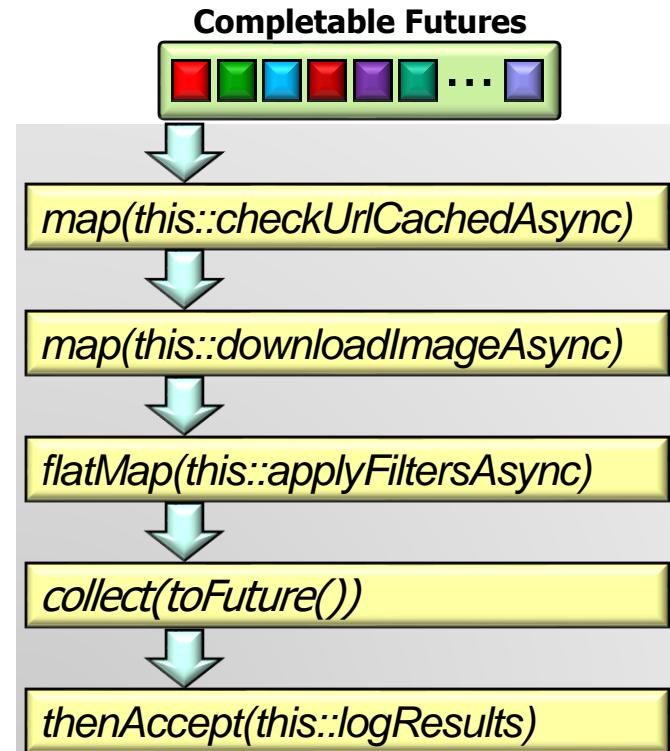
---

- 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
  - Combining sequential streams & completable futures is often a win



# 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
  - Combining sequential streams & completable futures is often a win
    - However, it's overkill to combine parallel streams & completable futures

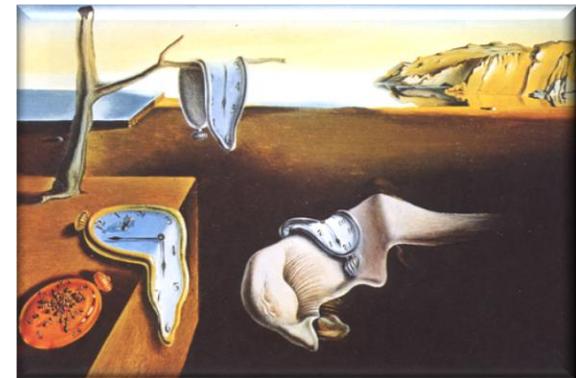


# Pros & Cons of Java 8 Completable Futures

- 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, ex) -> {  
    if (ex == null)  
        { System.out.println("The price is: " + amount + "GBP"); }  
    else { System.out.println(ex.getMessage()); }  
});
```



---

# End of Pros & Cons of Java 8 CompletableFuture