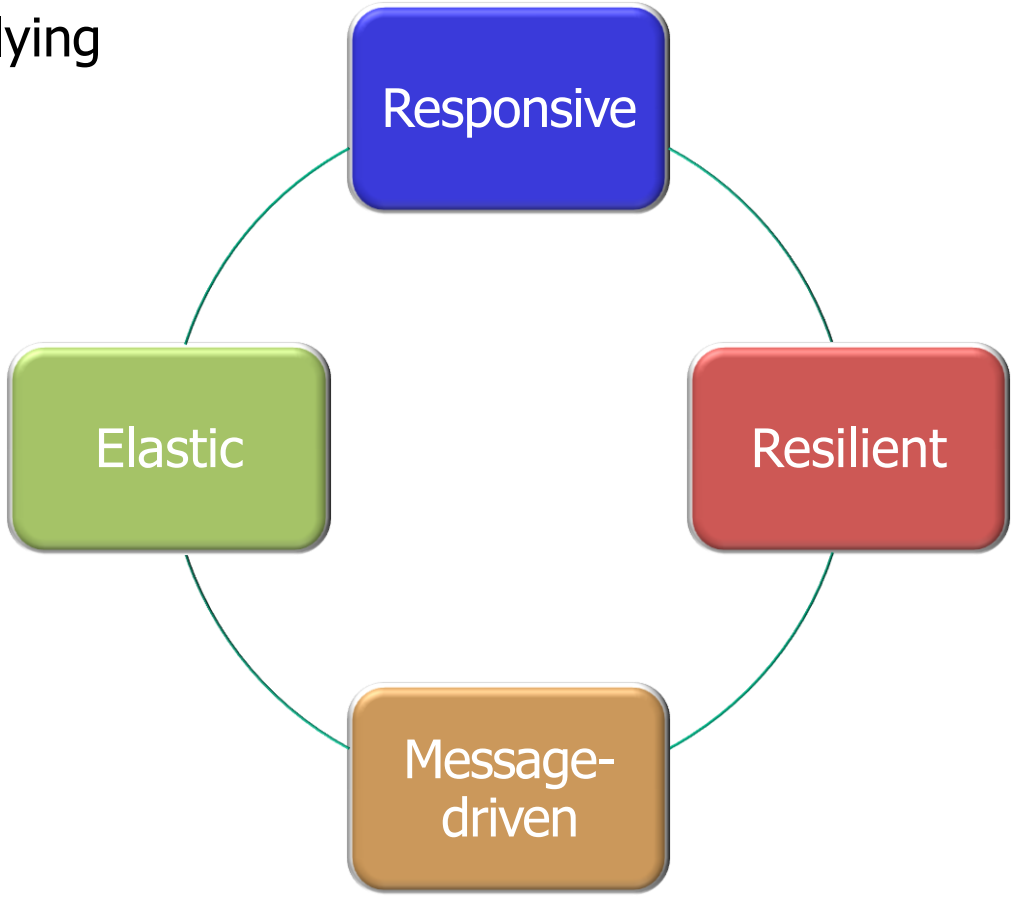
Overview of Reactive Programming

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Recognize the key principles underlying reactive programming



Overview of Reactive Programming

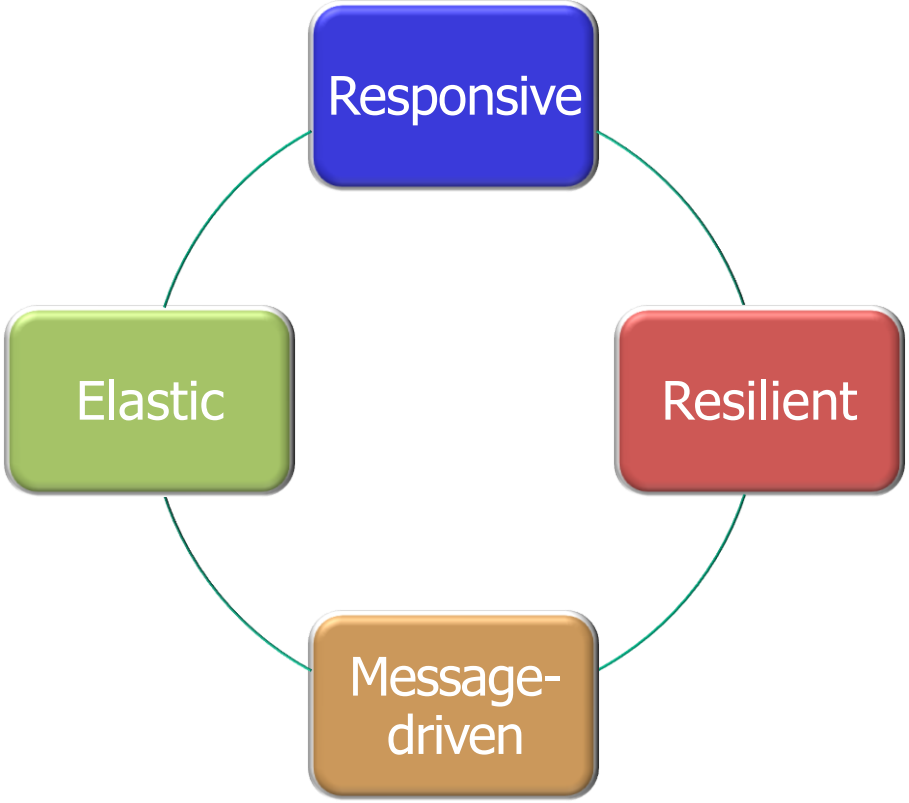
Overview of Reactive Programming

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

See en.wikipedia.org/wiki/Reactive_programming

Overview of Reactive Programming

- Reactive programming is based on four key principles.



See www.reactivemanifesto.org

Overview of Reactive Programming

- Reactive programming is based on four key principles.

1. **Responsive**
   - Provide rapid and consistent response times.

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

See en.wikipedia.org/wiki/Responsiveness

Overview of Reactive Programming

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

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

See en.wikipedia.org/wiki/Resilience_(network)

Overview of Reactive Programming

- Reactive programming is based on four key principles.
  1. Responsive
  2. Resilient
  3. **Elastic**
     - A system should remain responsive, even under varying workload.

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

See en.wikipedia.org/wiki/Autoscaling

Overview of Reactive Programming

- Reactive programming is based on four key principles.
  1. Responsive
  2. Resilient
  3. Elastic
  4. **Message-driven**
     - Asynchronous message passing ensures loose coupling, isolation, and location transparency between components.

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

See en.wikipedia.org/wiki/Message-oriented_middleware

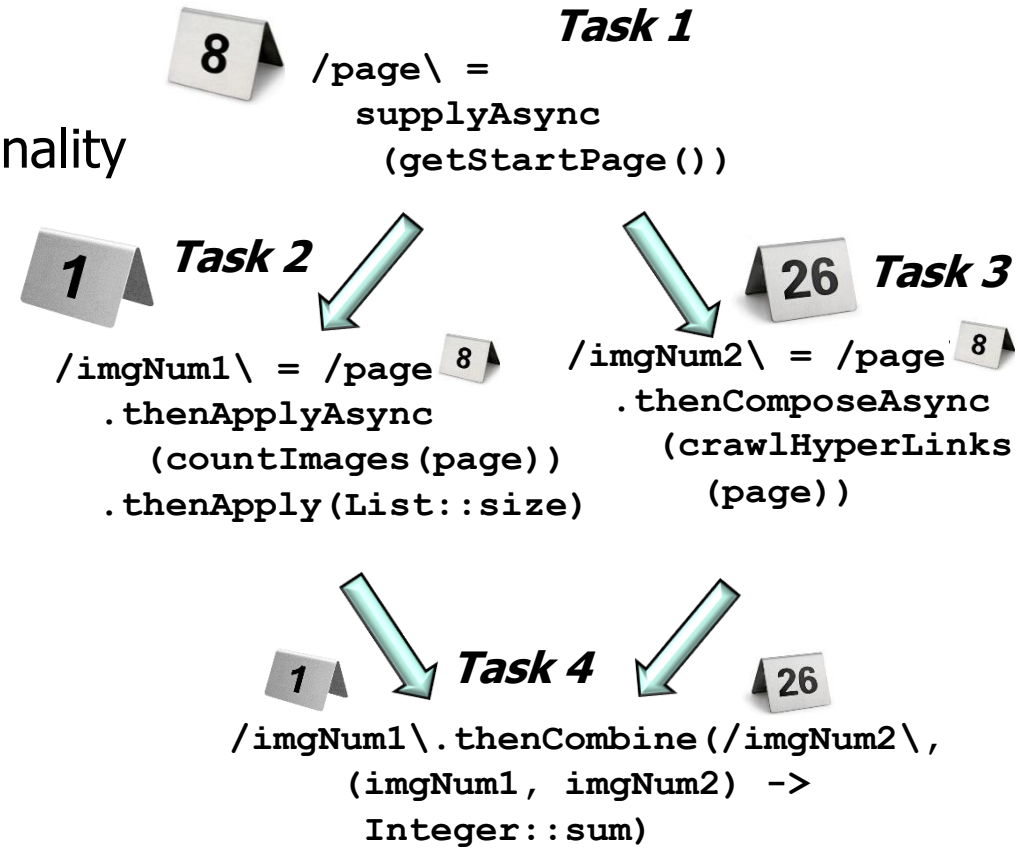
The End

Overview of the Java Completable Futures Framework

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

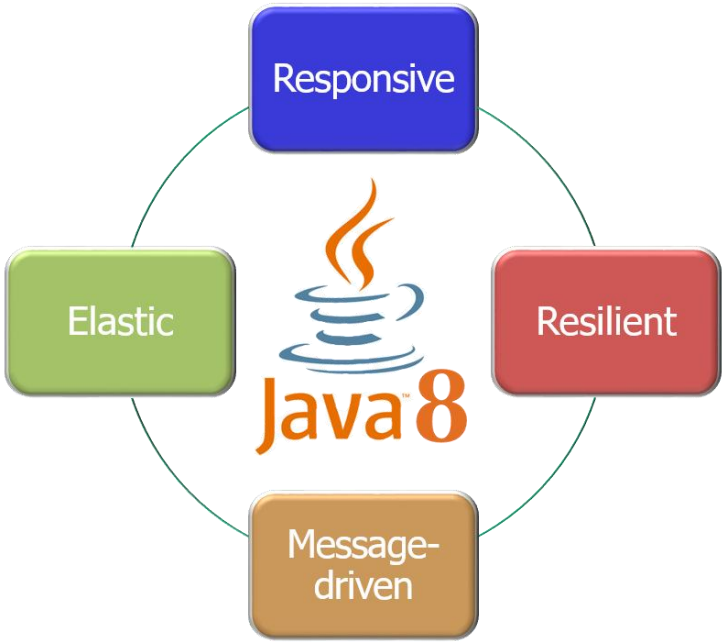
- Recognize the key principles underlying reactive programming
- Be aware of structure and functionality of the Java completable futures framework



Overview of the Java Completable Futures Framework

Overview of Completable Futures

- Java's completable futures framework provides an asynchronous and 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.

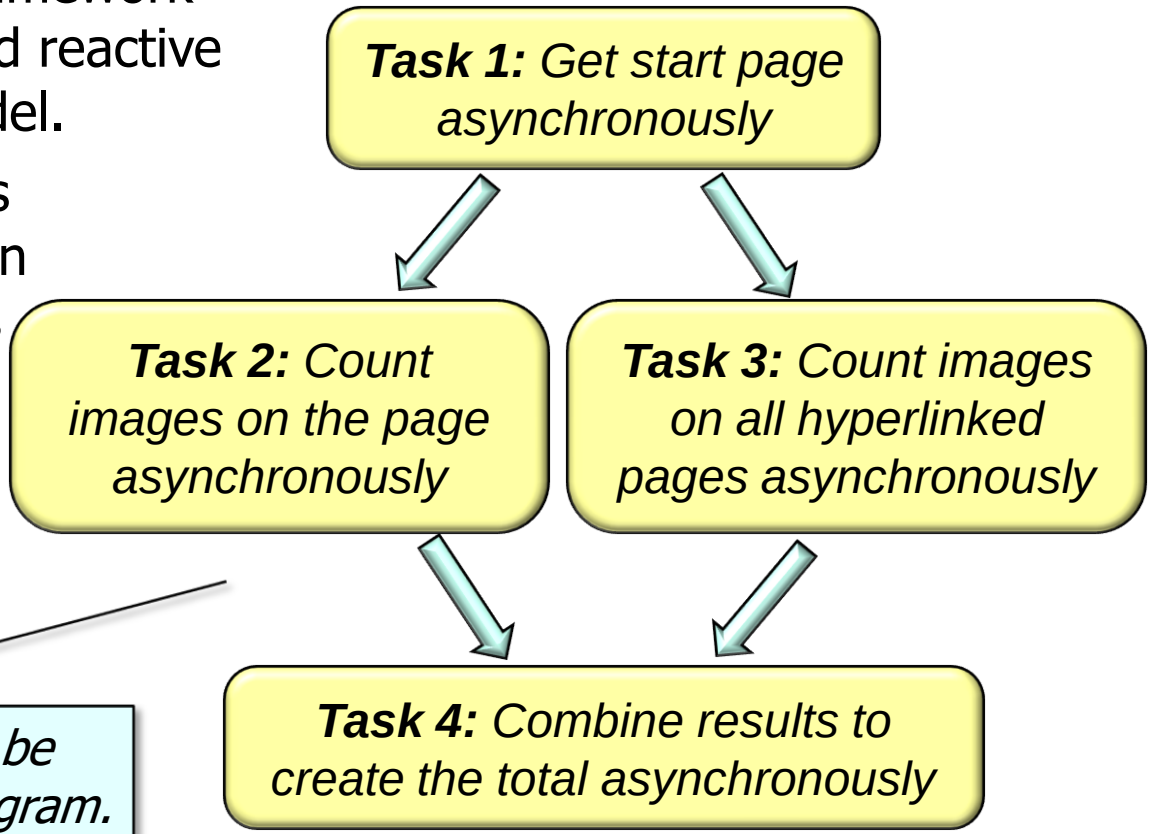
In addition to these and related methods for directly manipulating status and results, CompletableFuture implements interface `CompletionStage` with the following policies:

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

Overview of Completable Futures

- Java's completable futures framework provides an asynchronous and reactive concurrent programming model.
  - Supports dependent actions that trigger upon completion of asynchronous operations

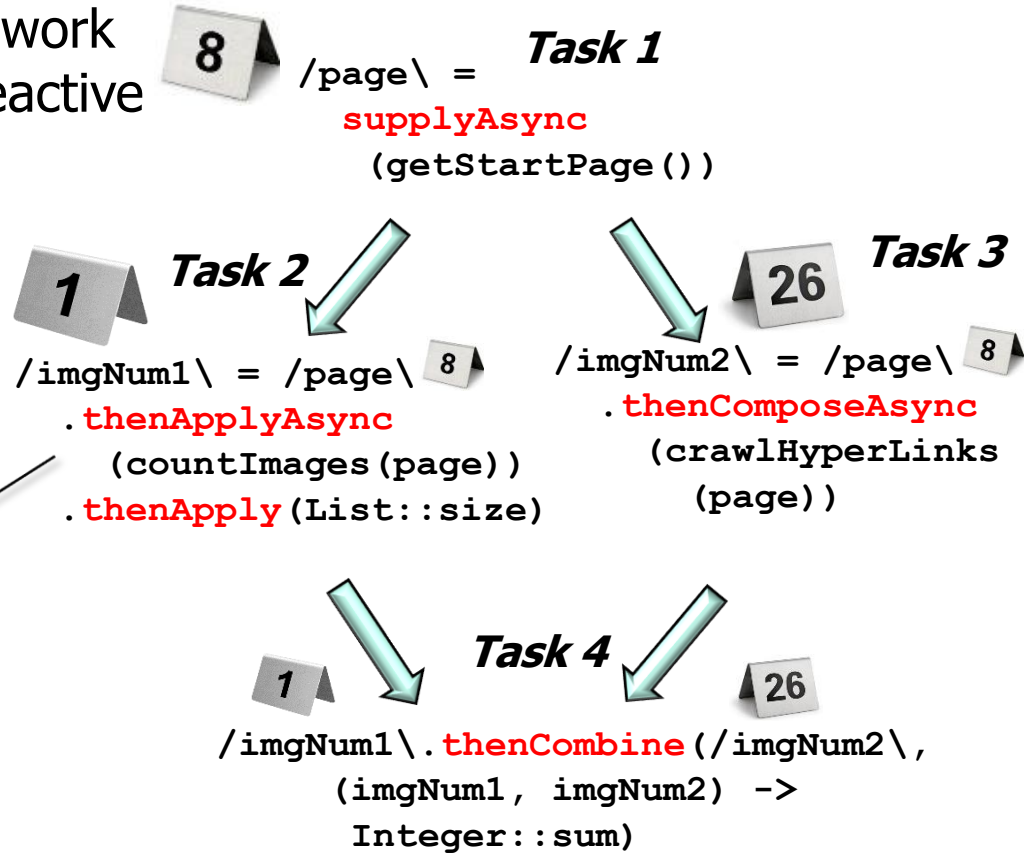
These dependencies can be modeled via a data flow diagram.



See en.wikipedia.org/wiki/Web_crawler

Overview of Completable Futures

- Java's completable futures framework provides an asynchronous and reactive concurrent programming model.
  - Supports dependent actions that trigger upon completion of asynchronous operations

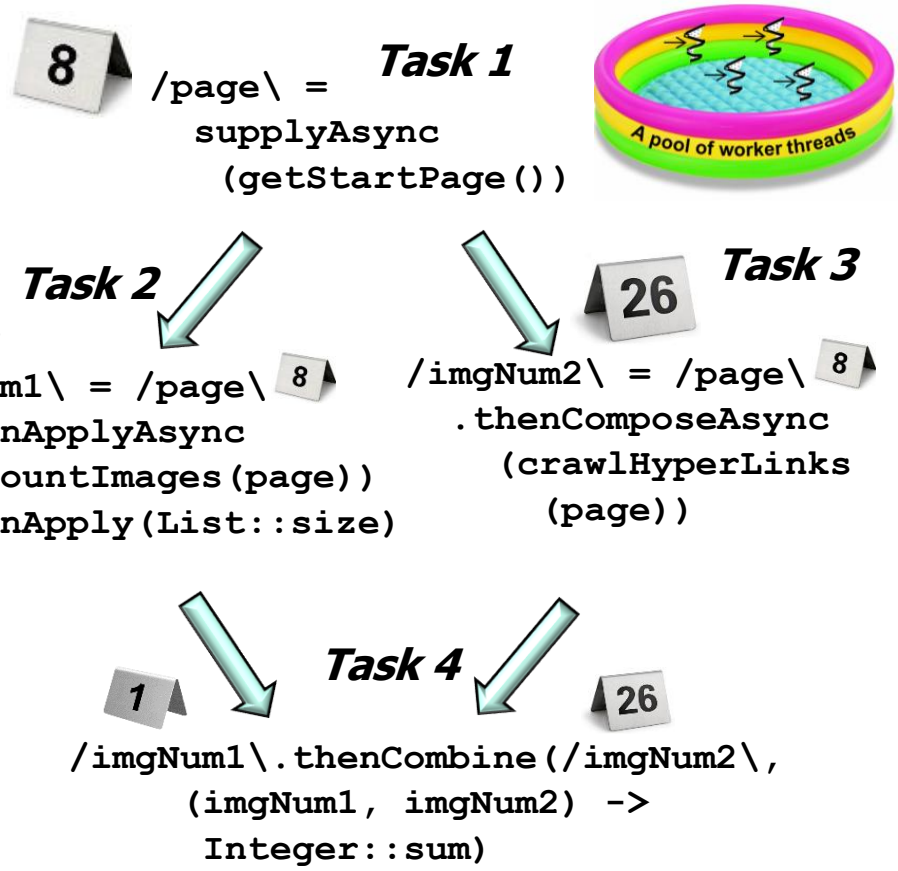


Asynchronous operations can be forked, chained, and joined.

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

Overview of Completable Futures

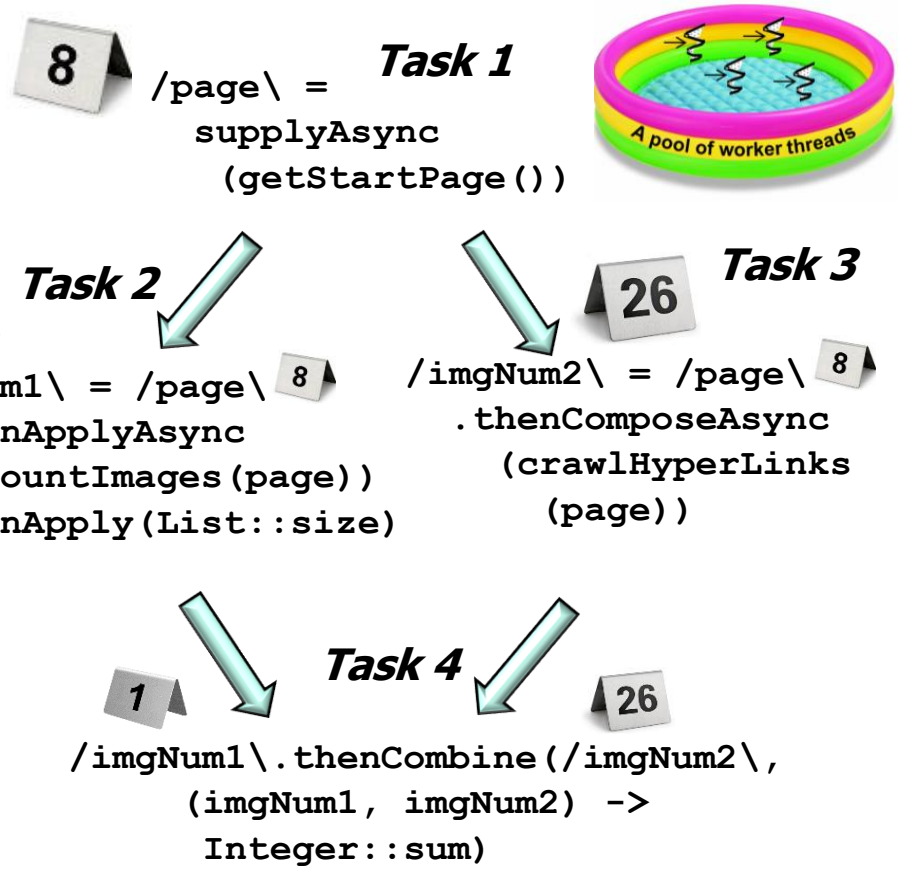
- Java's completable futures framework provides an asynchronous and reactive concurrent programming model.
  - Supports dependent actions that trigger upon completion of asynchronous operations
  - Asynchronous operations can run concurrently in thread pools



See www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html

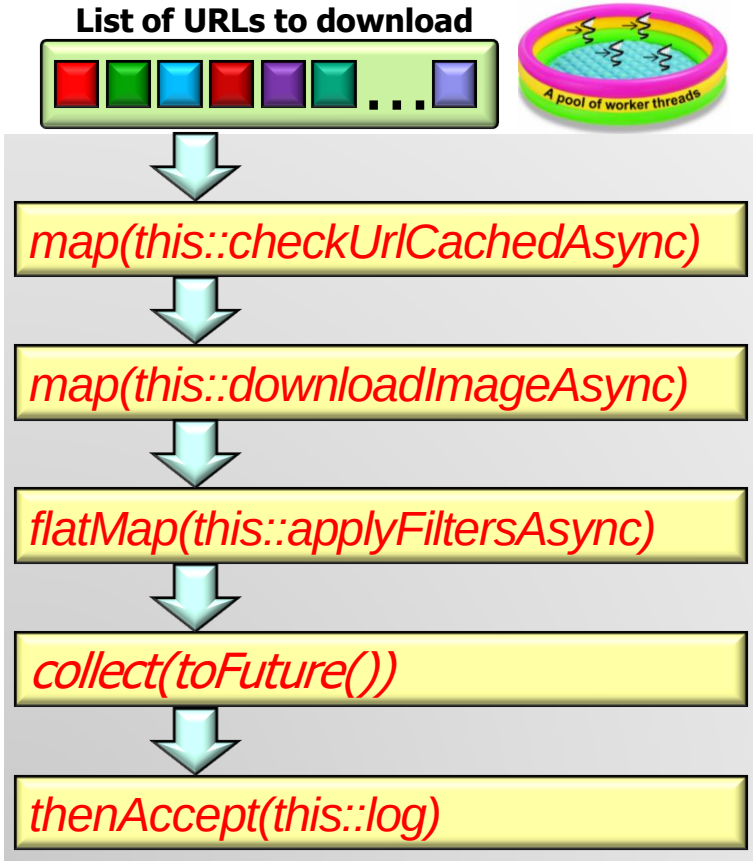
Overview of Completable Futures

- Java's completable futures framework provides an asynchronous and reactive concurrent programming model.
  - Supports dependent actions that trigger upon completion of asynchronous operations
  - Asynchronous operations can run concurrently in thread pools
    - Either a (common) fork-join pool or various types of pre- or user-defined thread pools



Overview of Completable Futures

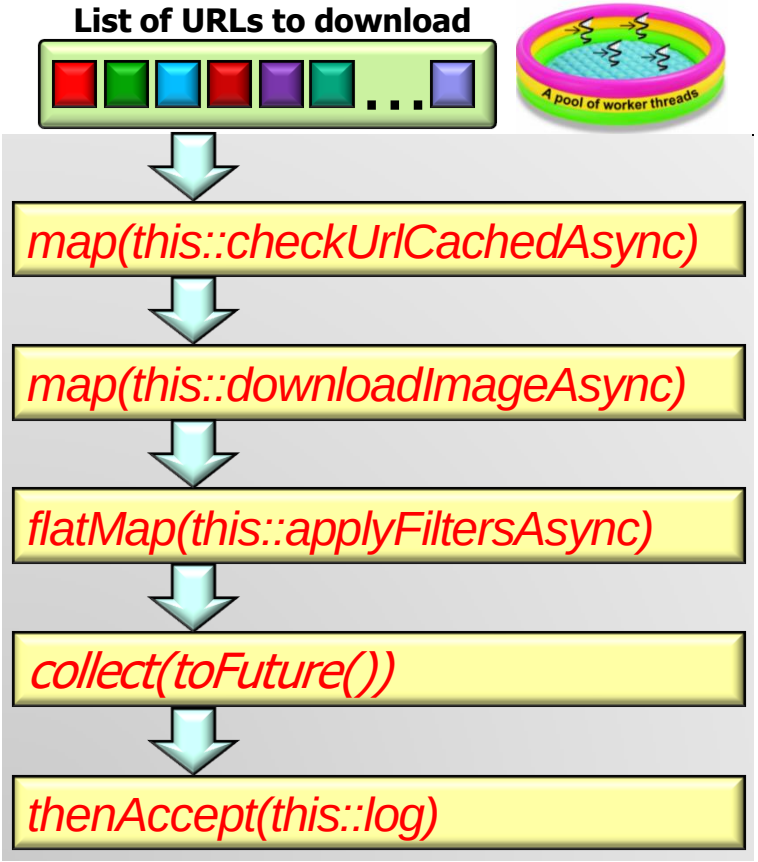
- Java completable futures, sequential streams, and functional programming features can be combined nicely!



See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

Overview of Completable Futures

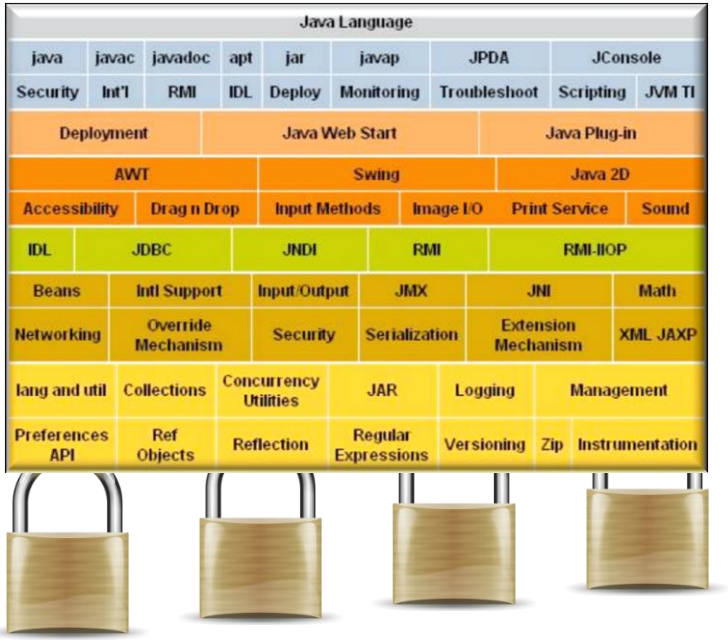
- Java completable futures often need no explicit synchronization or threading when developing concurrent apps!

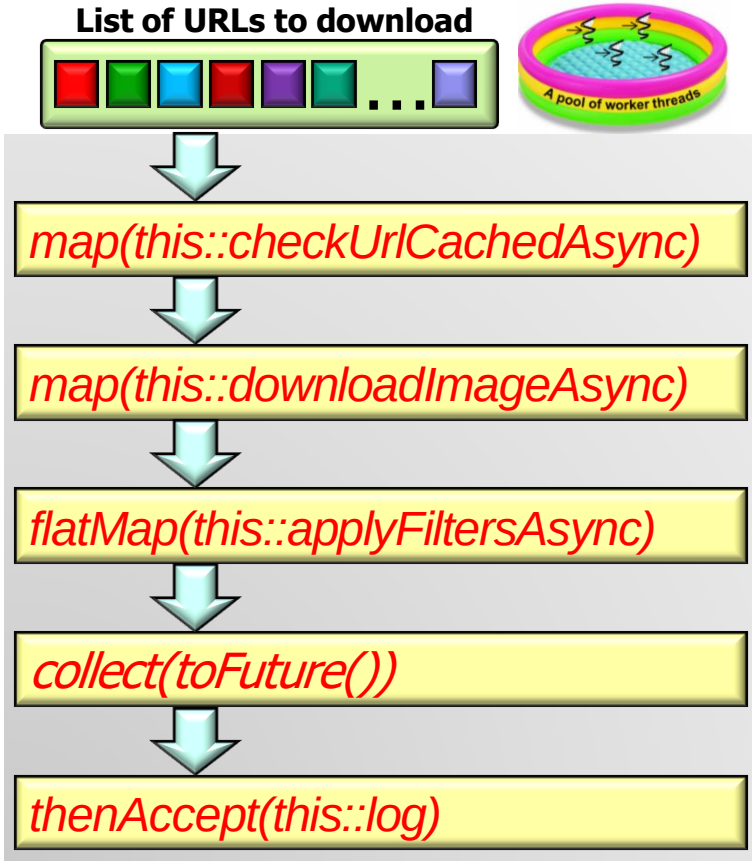


Alleviates many accidental and inherent complexities of concurrent programming

Overview of Completable Futures

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





Java class libraries handle locking needed to protect shared mutable state.

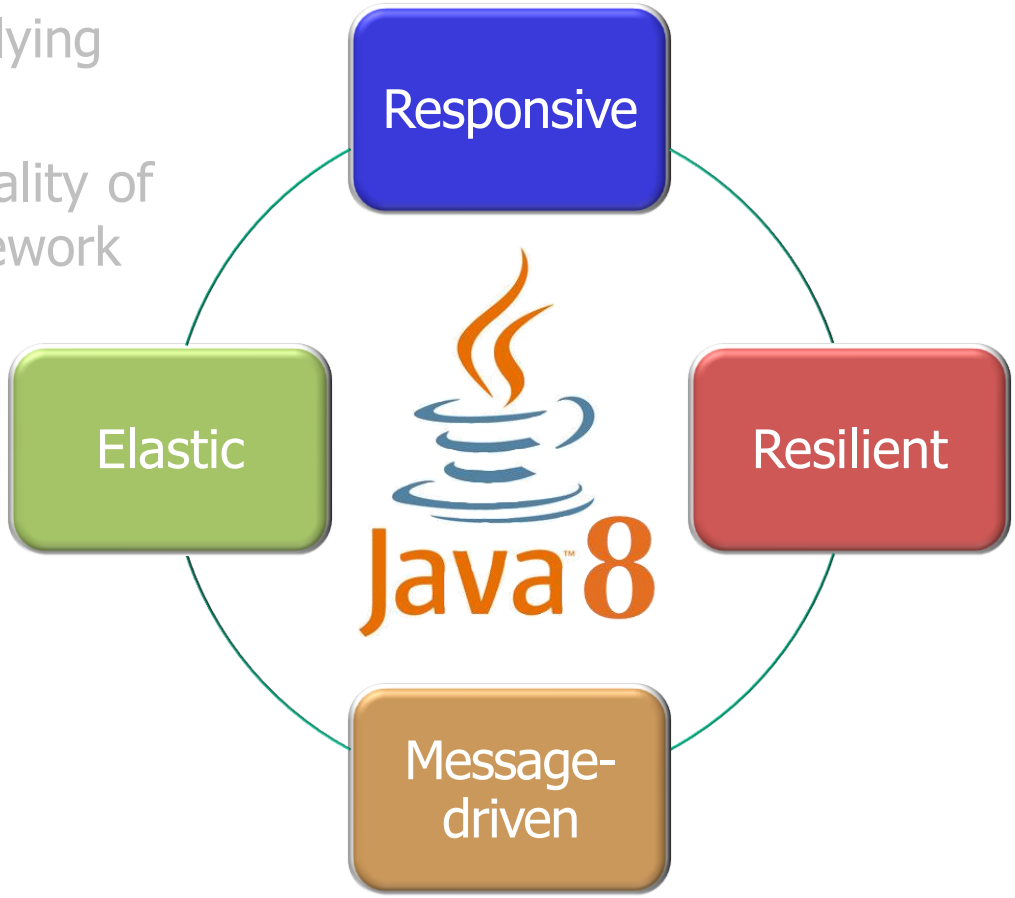
The End

Reactive Programming and Java Completable Futures

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

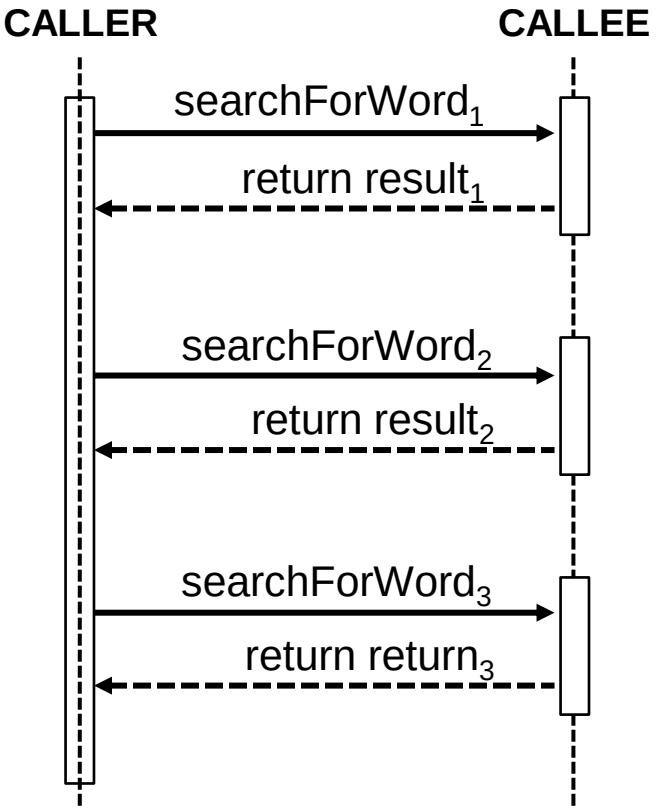
- Recognize the key principles underlying reactive programming
- Be aware of structure and functionality of the Java completable futures framework
- Understand how Java completable futures maps to key principles of reactive programming



Reactive Programming and Java Completable Futures

Reactive Programming and Java Completable Futures

- Java completable futures map onto key reactive programming principles.
  - **Responsive**
    - Avoid blocking in user code.
      - Blocking underutilizes cores, impedes inherent parallelism, and complicates program structure.

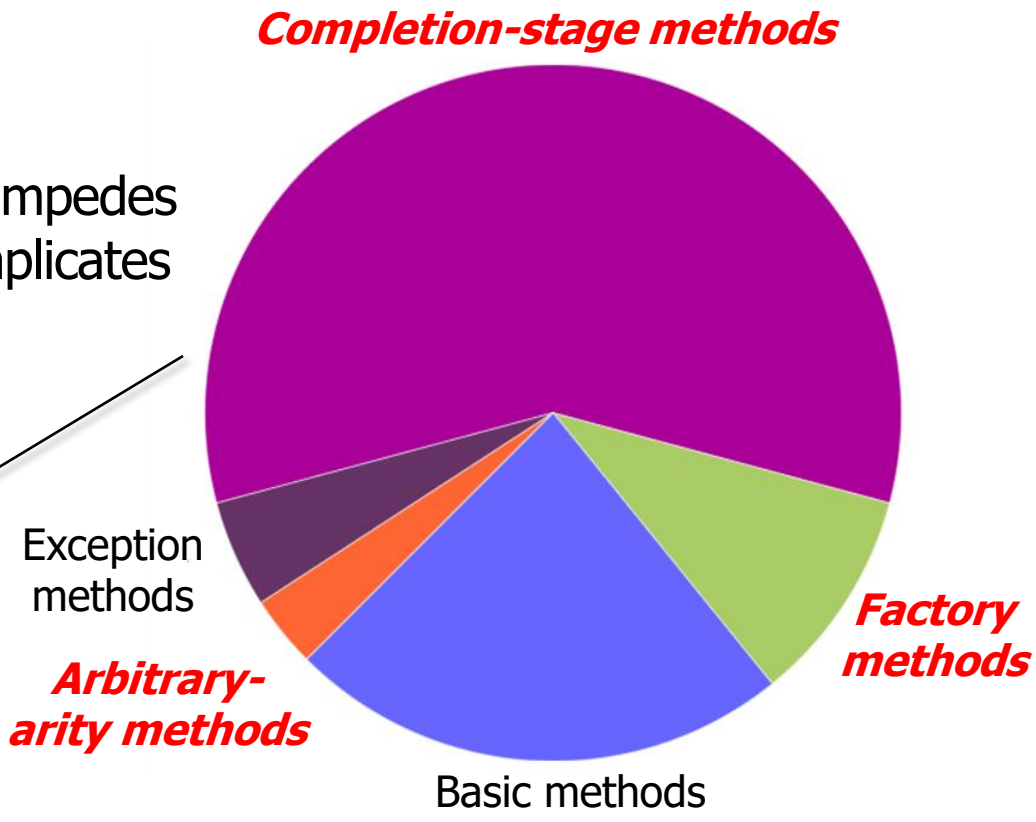


See www.ibm.com/developerworks/library/j-jvmc3

Reactive Programming and Java Completable Futures

- Java completable futures map onto key reactive programming principles.
  - **Responsive**
    - Avoid blocking in user code.
      - Blocking underutilizes cores, impedes inherent parallelism, and complicates program structure.

Factory, completion-stage, and arbitrary-arity methods avoid blocking threads.



Reactive Programming and Java Completable Futures

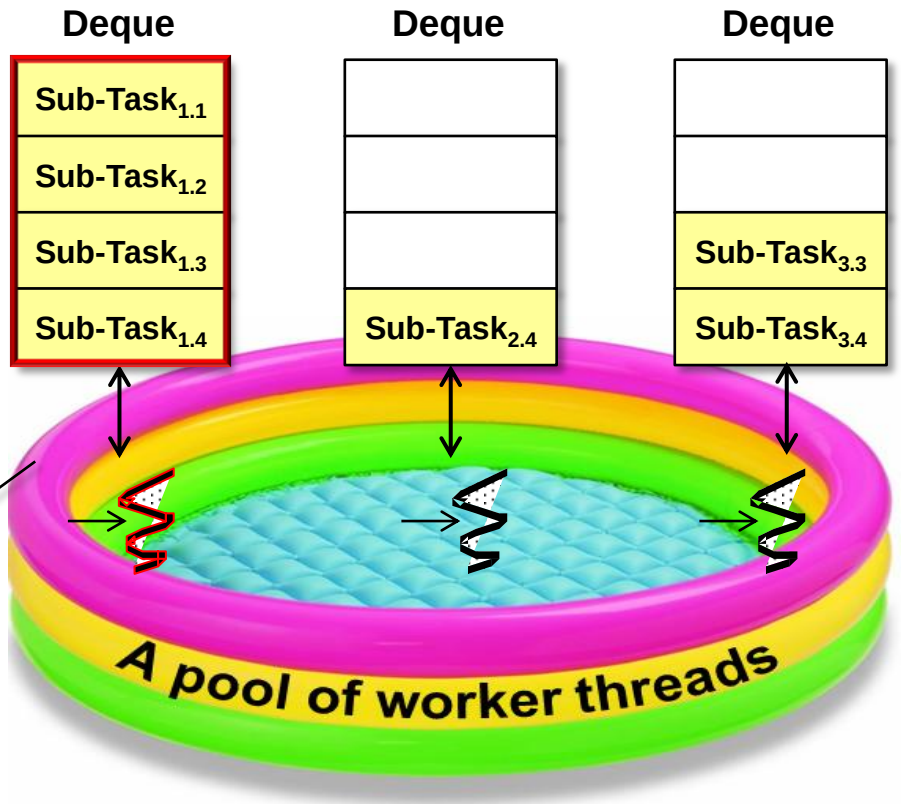
- Java completable futures map onto key reactive programming principles.
  - **Responsive**
    - Avoid blocking in user code.
    - Avoid changing threads.
      - Incurs excessive overhead wrt synchronization, context switching, and memory/cache management

See gee.cs.oswego.edu/dl/papers/fj.pdf

Reactive Programming and Java Completable Futures

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

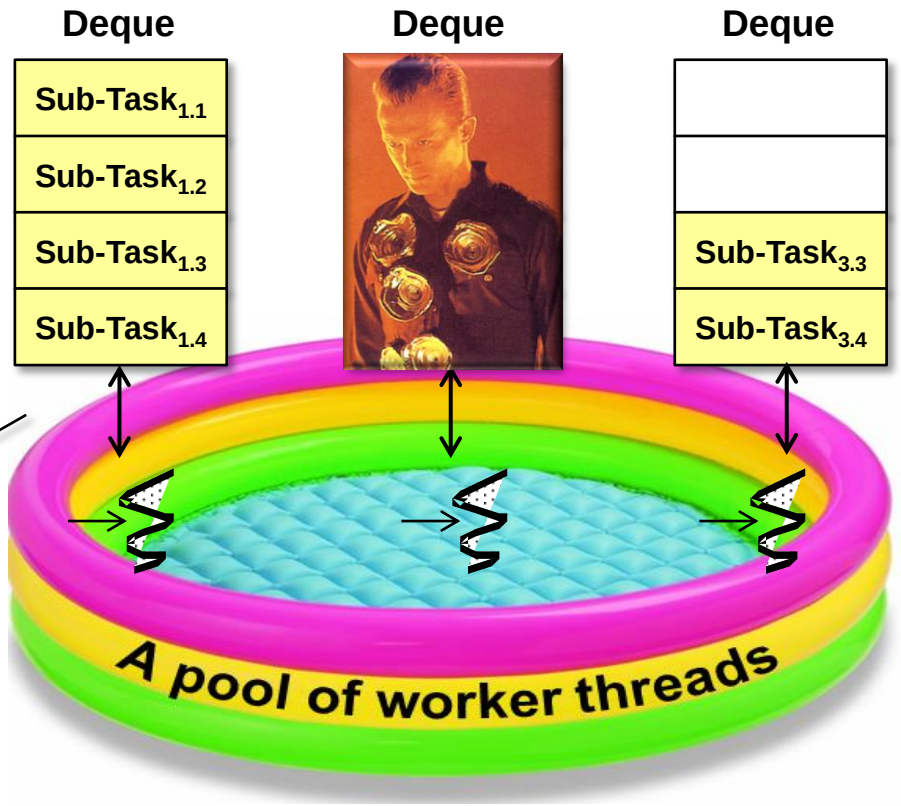
*The fork-join pool and non-*Async() methods avoid changing threads.*



See gee.cs.oswego.edu/dl/papers/fj.pdf

Reactive Programming and Java Completable Futures

- Java completable futures map onto key reactive programming principles.
  - Responsive
  - **Resilient**
    - Exception methods make more programs are resilient to failures.

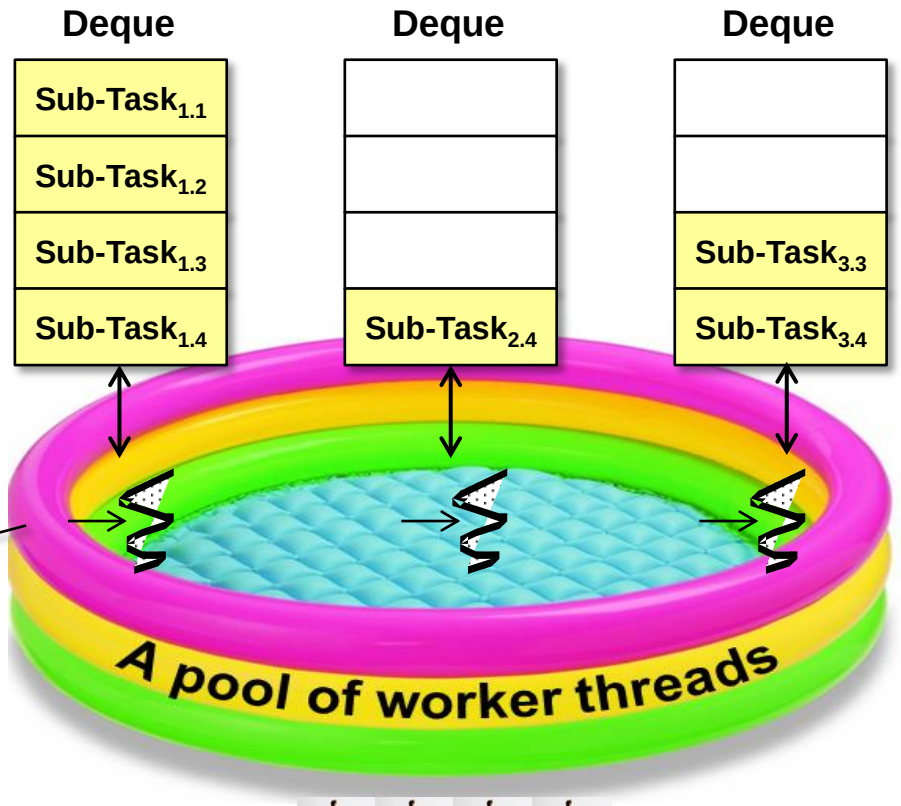


Exceptions decouple error processing from normal operations.

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

Reactive Programming and Java Completable Futures

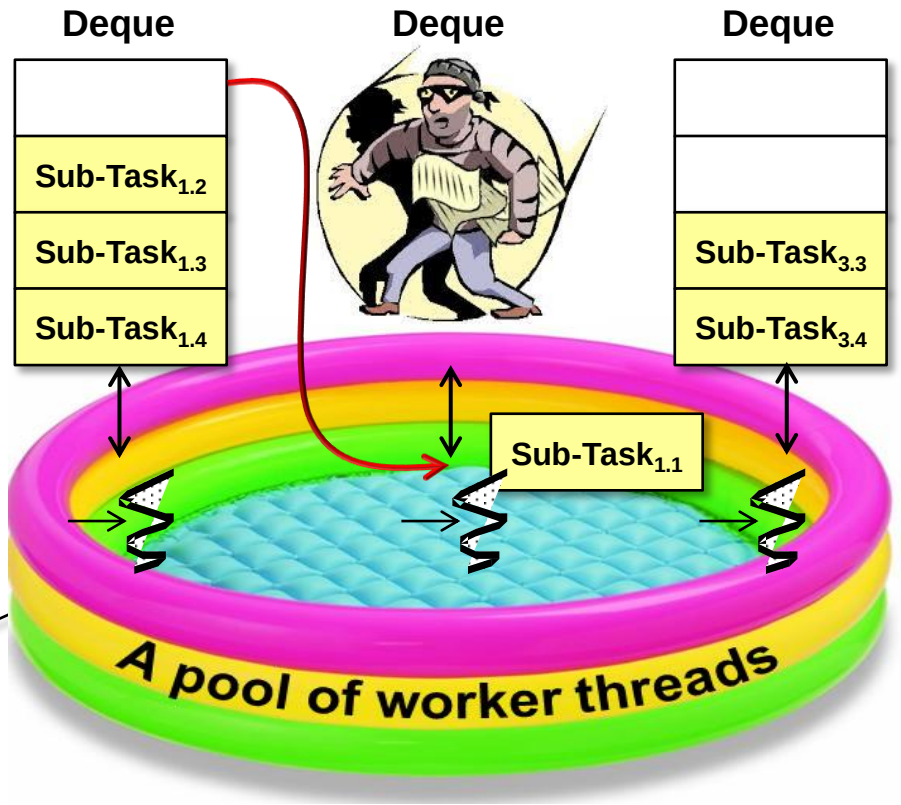
- Java completable futures map onto key reactive programming principles.
  - Responsive
  - Resilient
  - **Elastic**
    - Asynchronous computations can run scalably in a pool of threads atop a set of cores.



Can be a (common) fork-join pool or a pre- or user-defined thread pool

Reactive Programming and Java Completable Futures

- Java completable futures map onto key reactive programming principles.
  - Responsive
  - Resilient
  - Elastic
  - **Message-driven**
    - Java's thread pools pass messages between threads in the pool internally.



Java's fork-join pool supports "work stealing" between deques.

See en.wikipedia.org/wiki/Work_stealing

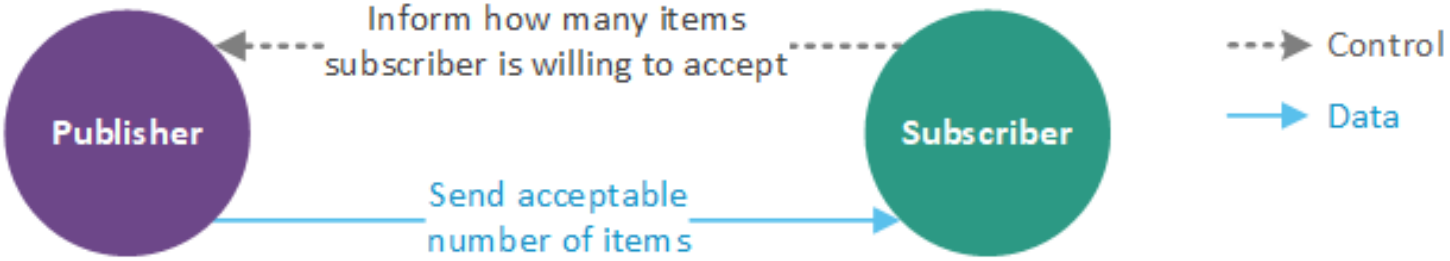
The End

Reactive Programming and Java Reactive Streams

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Recognize the key principles underlying reactive programming
- Be aware of structure and functionality of the Java completable futures framework
- Understand how Java completable futures maps to key principles of reactive programming
- Know the relationship between reactive programming and Java reactive streams



See community.oracle.com/docs/DOC-1006738

Reactive Programming and Java Reactive Streams

Reactive Programming and Java Reactive Streams

- Java 9 supports reactive programming via “Reactive Streams” and the Flow API.

Class Flow

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

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

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

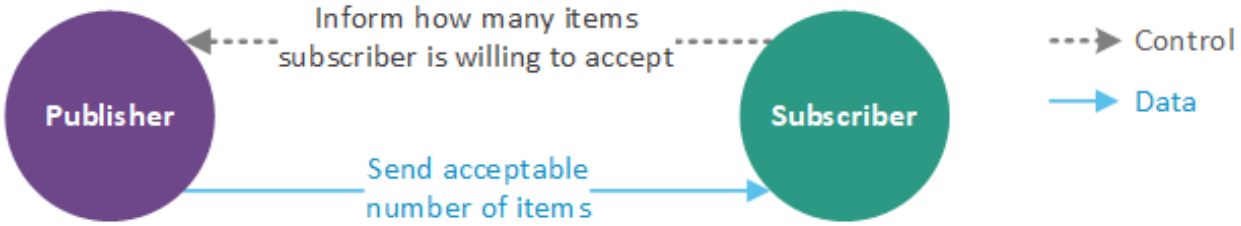
These interfaces correspond to the reactive-streams specification. They apply in both concurrent and distributed asynchronous settings: All (seven) methods are defined in `void` "one-way" message style. Communication relies on a simple form of flow control (method `Flow.Subscription.request(long)`) that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

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

See docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html

Reactive Programming and Java Reactive Streams

- Java 9 supports reactive programming via "Reactive Streams" and the Flow API.
  - Adds support for stream-oriented pub/sub patterns

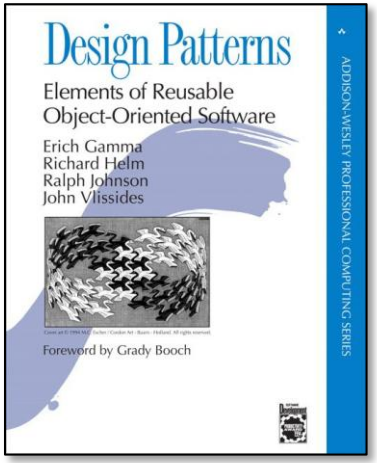


See javasampleapproach.com/java/java-9/java-9-flow-api-example-publisher-and-subscriber

Reactive Programming and Java Reactive Streams

- Java 9 supports reactive programming via "Reactive Streams" and the Flow API.
  - Adds support for stream-oriented pub/sub patterns

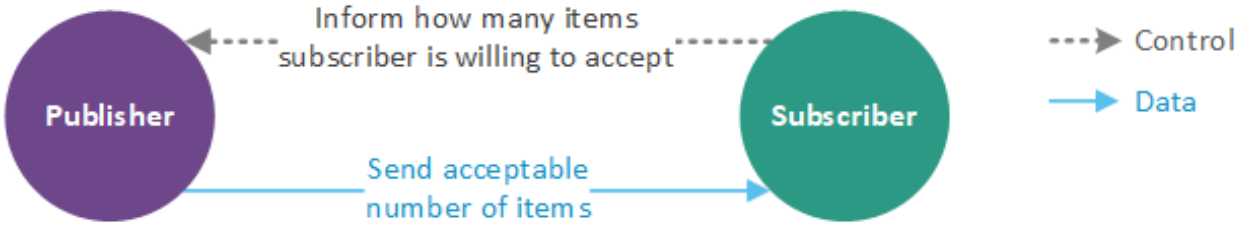
  - Combines two patterns



See www.journaldev.com/20723/java-9-reactive-streams

Reactive Programming and Java Reactive Streams

- Java 9 supports reactive programming via "Reactive Streams" and the Flow API.
  - Adds support for stream-oriented pub/sub patterns

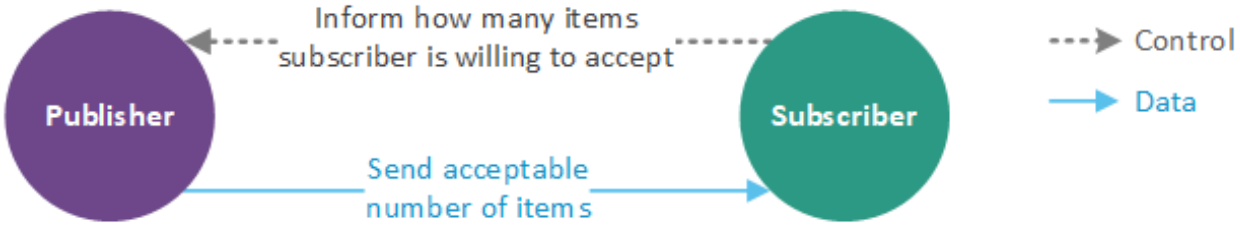


  - Combines two patterns
    - *Iterator*, which applies a pull model where apps pull items from a source

See en.wikipedia.org/wiki/Iterator_pattern

Reactive Programming and Java Reactive Streams

- Java 9 supports reactive programming via "Reactive Streams" and the Flow API.
  - Adds support for stream-oriented pub/sub patterns

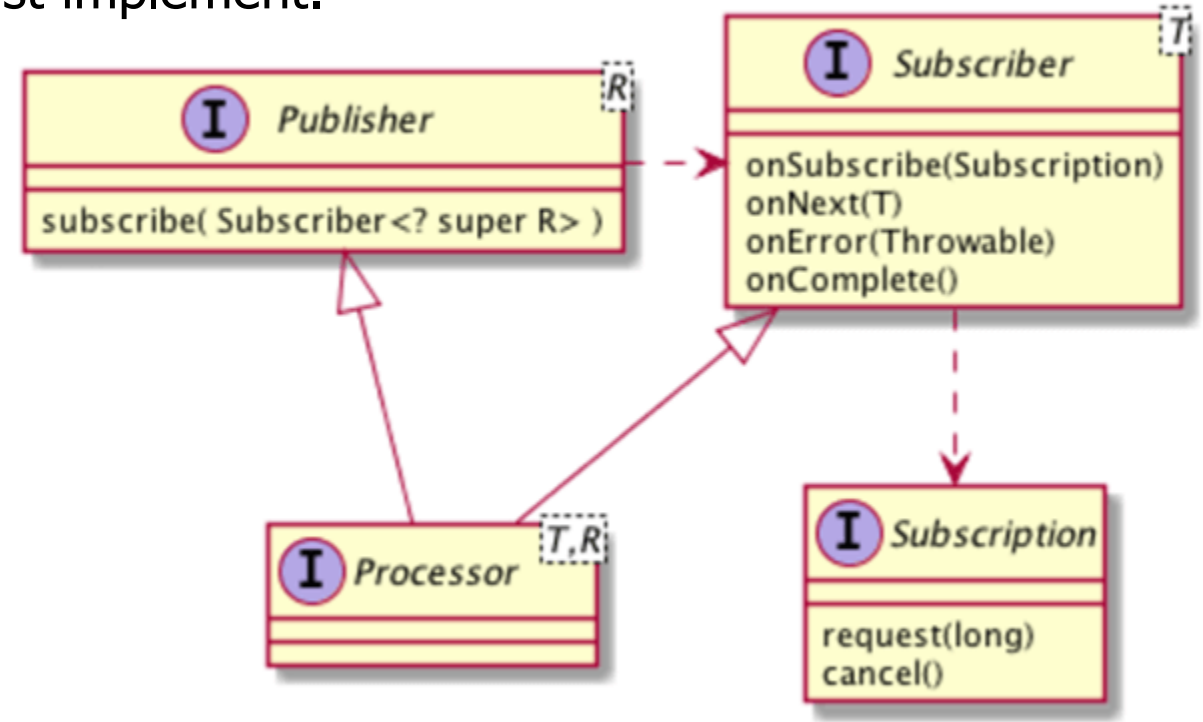


  - Combines two patterns
    - *Iterator*, which applies a pull model where apps pull items from a source
    - *Observer*, which applies a push model that reacts when item is pushed from a source to a subscriber

See en.wikipedia.org/wiki/Observer_pattern

Reactive Programming and Java Reactive Streams

- The Java Flow API is merely a set of interfaces that others must implement.



See www.baeldung.com/java-9-reactive-streams

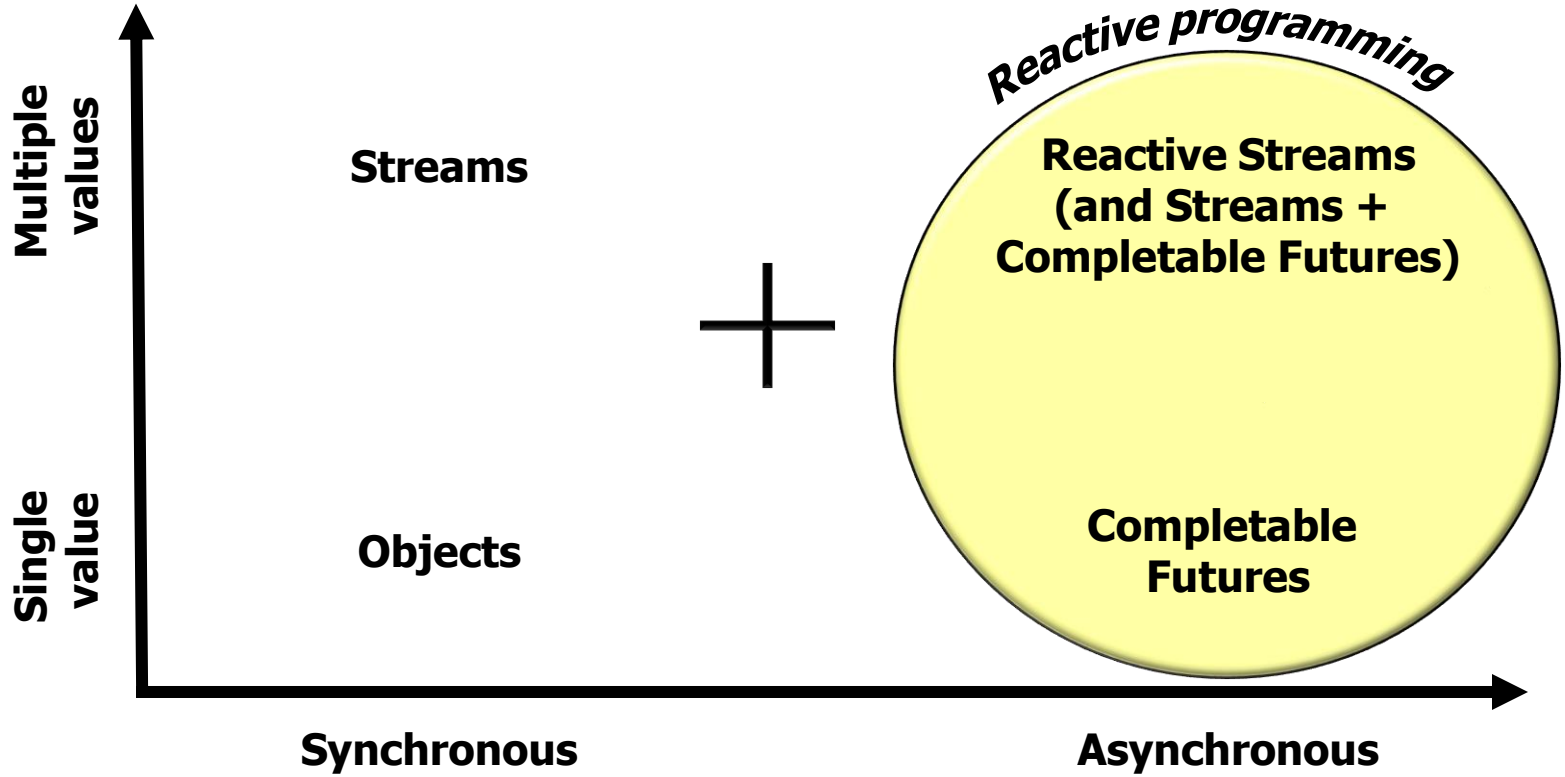
Reactive Programming and Java Reactive Streams

- The Java Flow API is merely a set of interfaces that others must implement.
  - Provides an interoperable foundation that's implemented by other reactive programming frameworks

See www.baeldung.com/rx-java & akka.io

Reactive Programming and Java Reactive Streams

- Comparing reactive programming with other Java programming paradigms



The End

The Pros and Cons of Synchrony

Douglas C. Schmidt

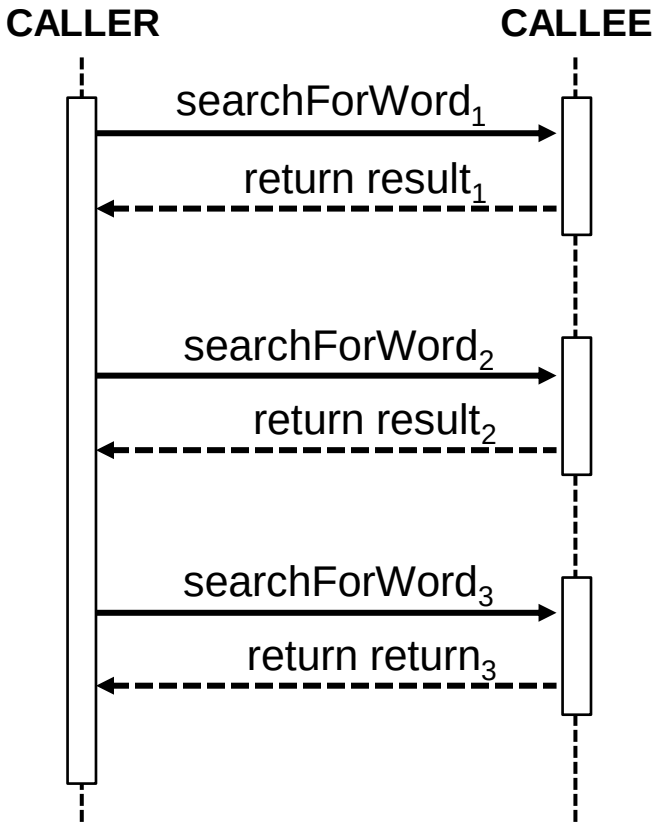
Learning Objectives in This Part of the Lesson

- Motivate the need for Java futures by understanding the pros and cons of synchrony

Overview of Synchrony and Synchronous Operations

Overview of Synchrony and Synchronous Operations

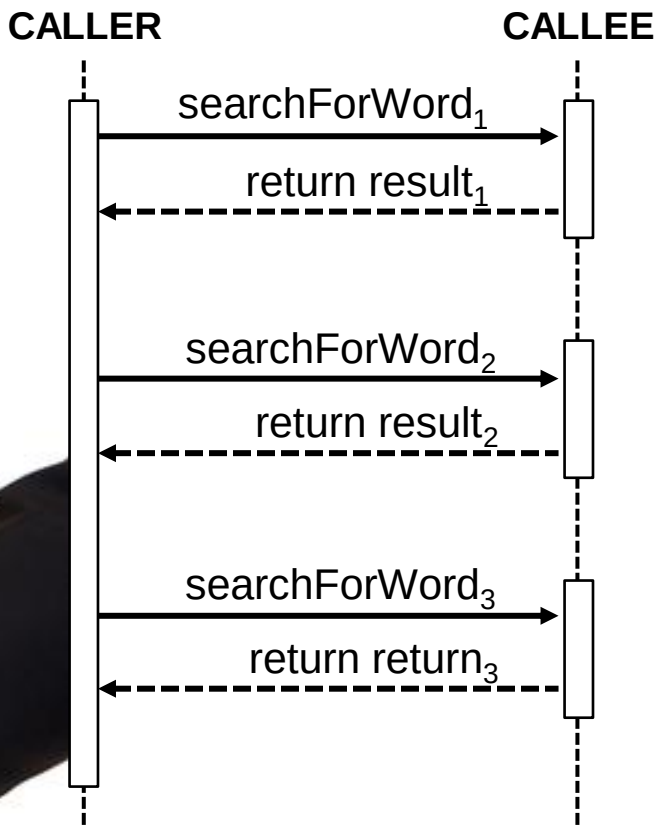
- Method calls in typical Java programs are largely *synchronous.*



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

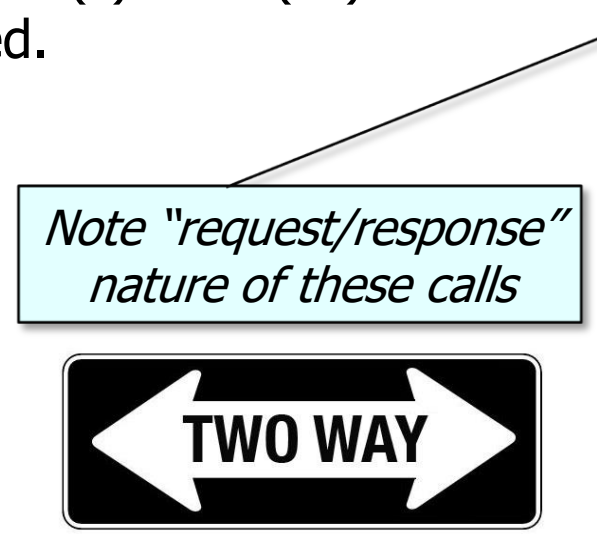
Overview of Synchrony and Synchronous Operations

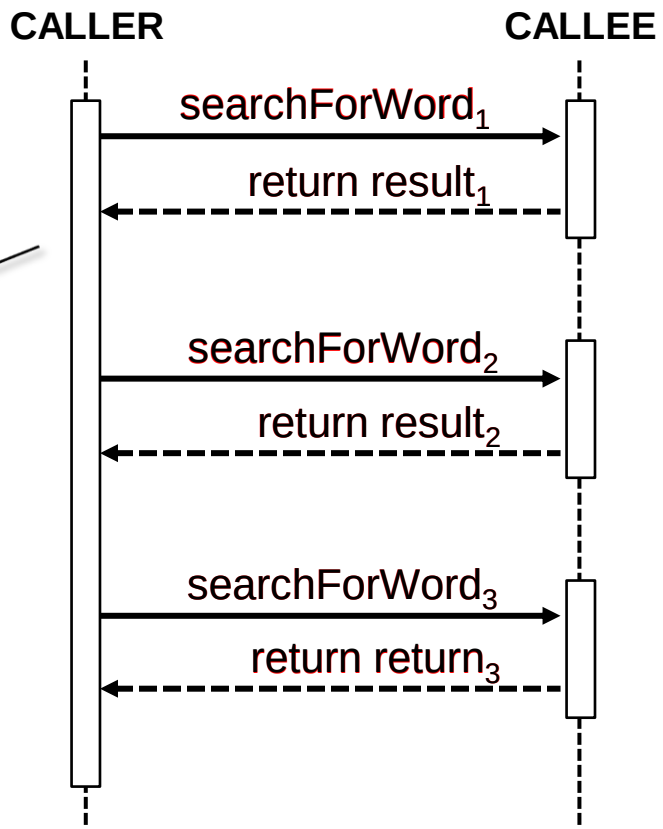
- Method calls in typical Java programs are largely *synchronous.*
  - A callee borrows the thread of its caller until its computation(s) finish(es).



Overview of Synchrony and Synchronous Operations

- Method calls in typical Java programs are largely *synchronous.*
  - A callee borrows the thread of its caller until its computation(s) finish(es) and a result is returned.





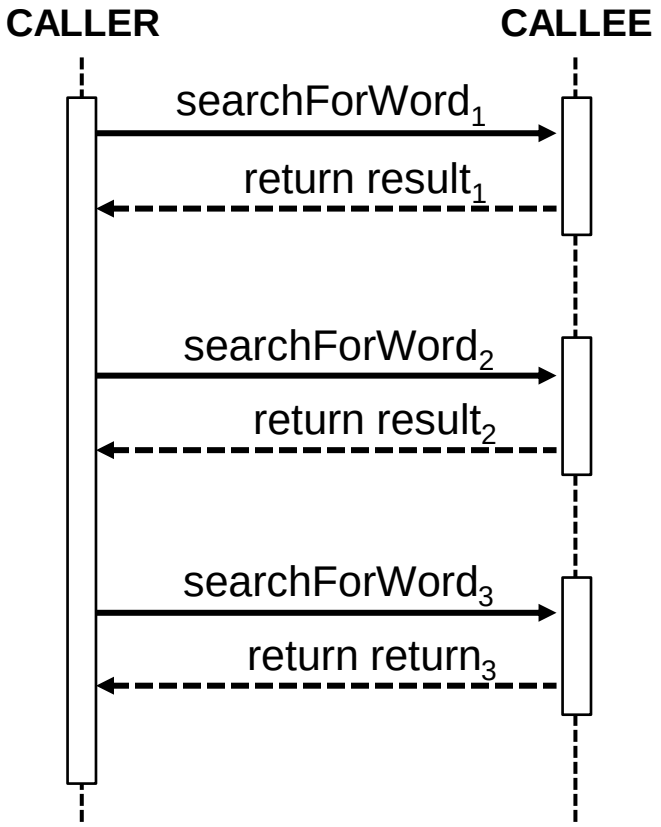
The Pros of Synchrony

The Pros of Synchrony

- Pros of synchronous calls

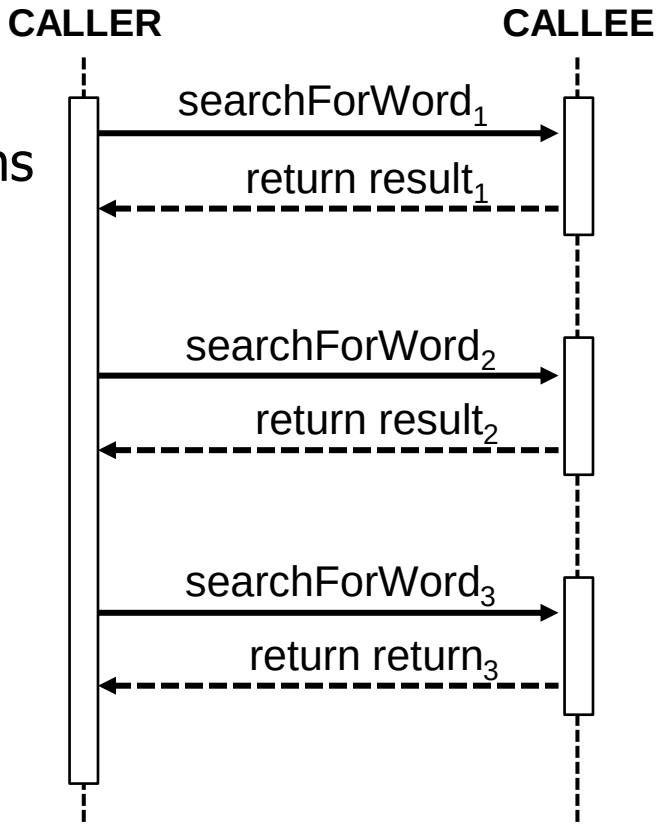
The Pros of Synchrony

- Pros of synchronous calls
  - "Intuitive" to program and debug



The Pros of Synchrony

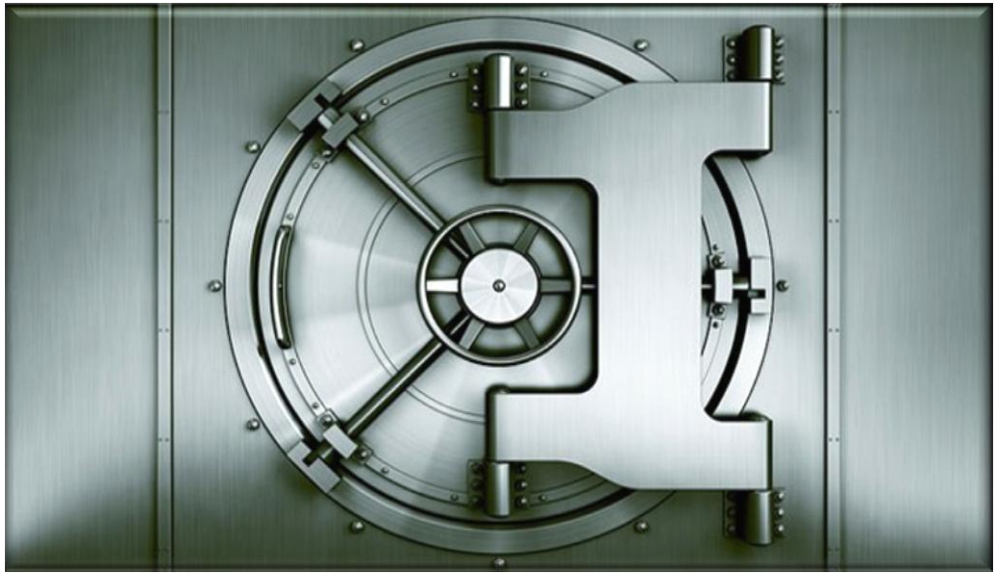
- Pros of synchronous calls
  - "Intuitive" to program and debug
    - Maps onto common two-way method patterns

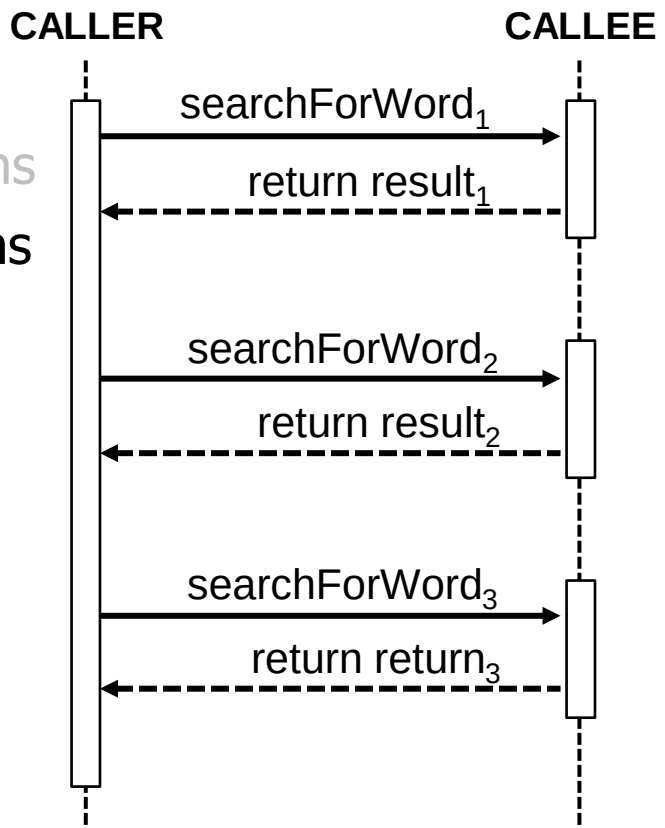


See www.iro.umontreal.ca/~keller/Layla/remote.pdf

The Pros of Synchrony

- Pros of synchronous calls
  - "Intuitive" to program and debug
    - Maps onto common two-way method patterns
    - Local caller state retained when callee returns



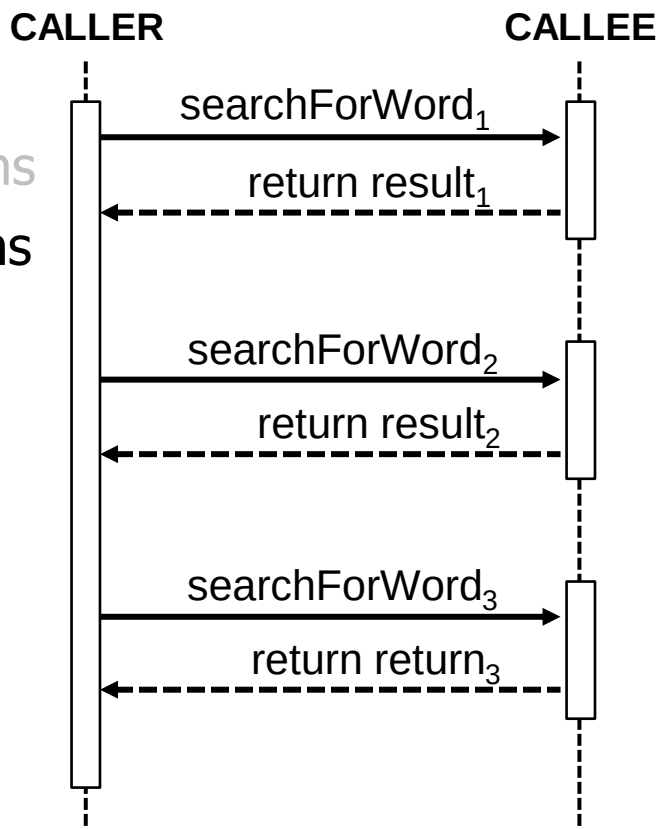


See wiki.c2.com/?ActivationRecord

The Pros of Synchrony

- Pros of synchronous calls
  - "Intuitive" to program and debug
    - 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); ...
```



See wiki.c2.com/?ActivationRecord

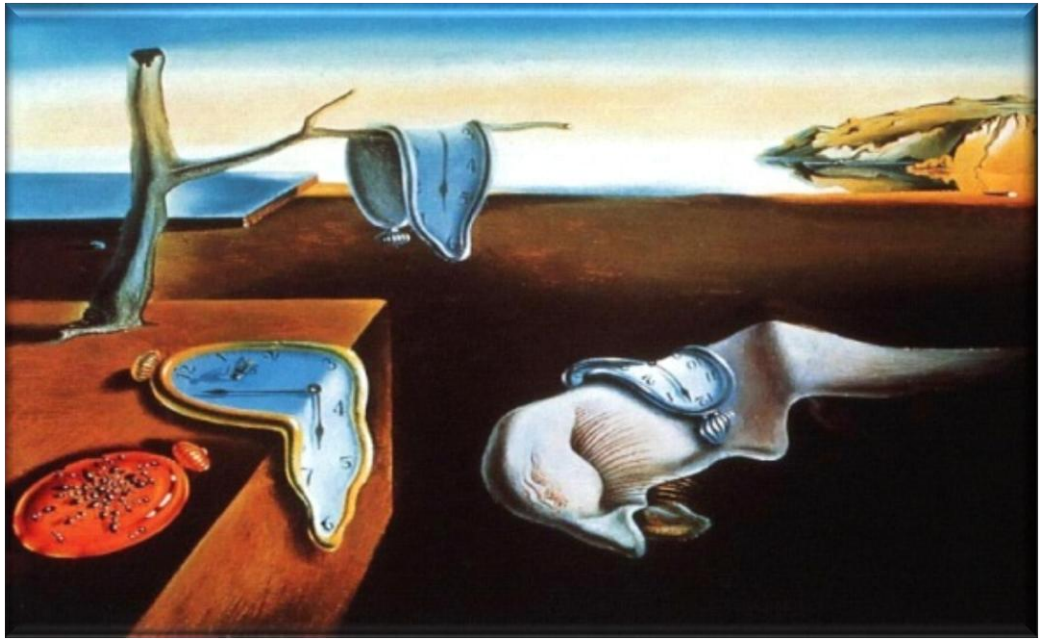
The Cons of Synchrony

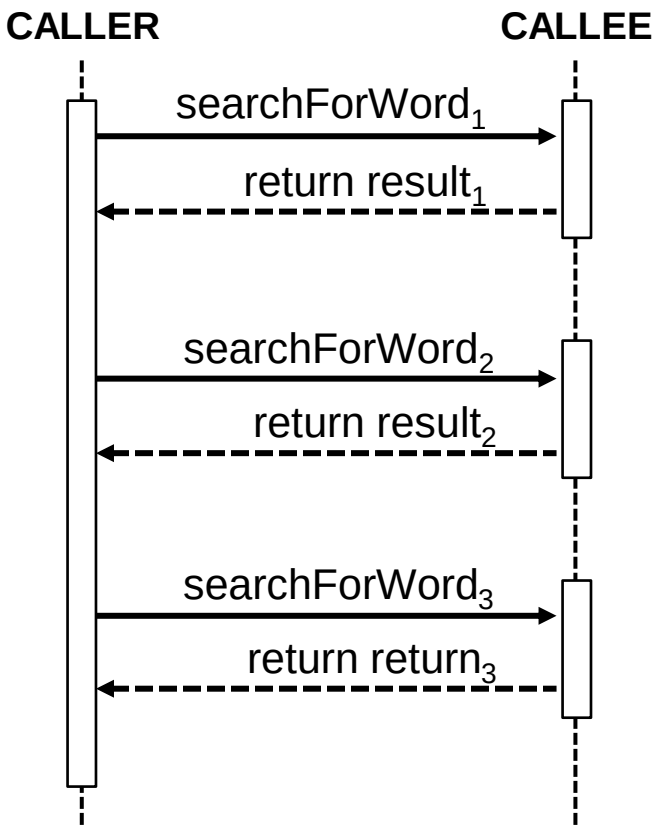
The Cons of Synchrony

- Cons of synchronous calls

The Cons of Synchrony

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

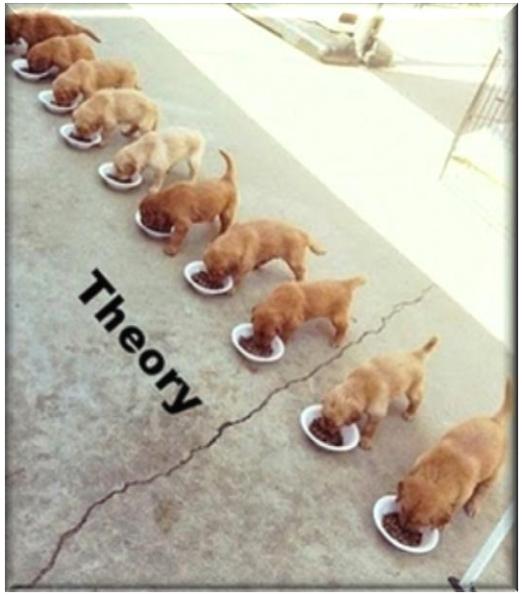




See www.ibm.com/developerworks/library/j-jvmc3

The Cons of Synchrony

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



See www.ibm.com/developerworks/library/j-jvmc3

The Cons of Synchrony

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

Efficient performance

Efficient resource utilization

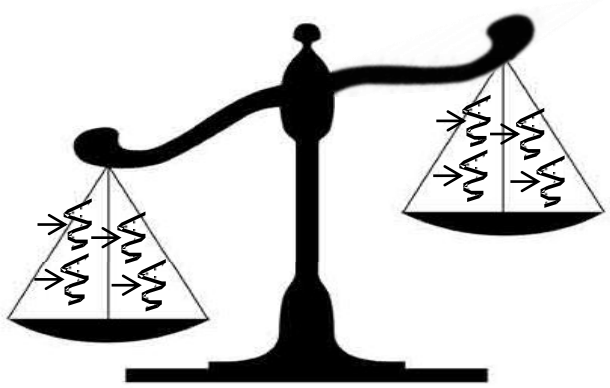
```
List<Image> filteredImages = urls
   .parallelStream()
   .filter(not(this::urlCached))
   .map(this::downloadImage)
   .flatMap(this::applyFilters)
   .collect(toList());
```


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

See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

The Cons of Synchrony

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



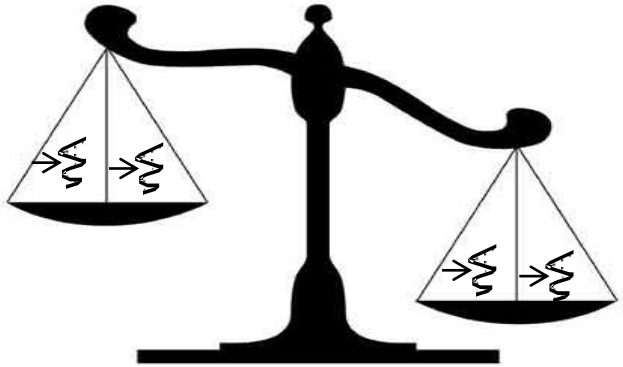
Efficient performance / *Efficient resource utilization*

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

A large number of threads may help to improve performance but can also waste resources.

The Cons of Synchrony

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



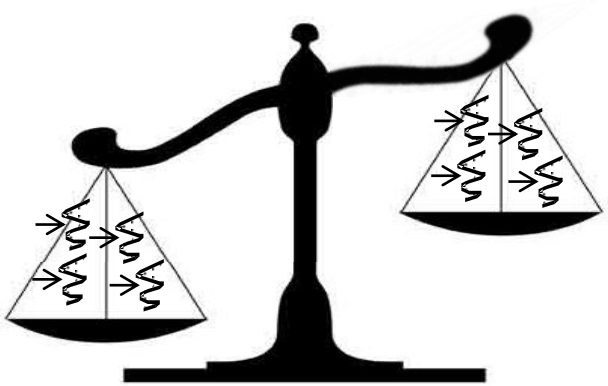
Efficient performance / *Efficient resource utilization*

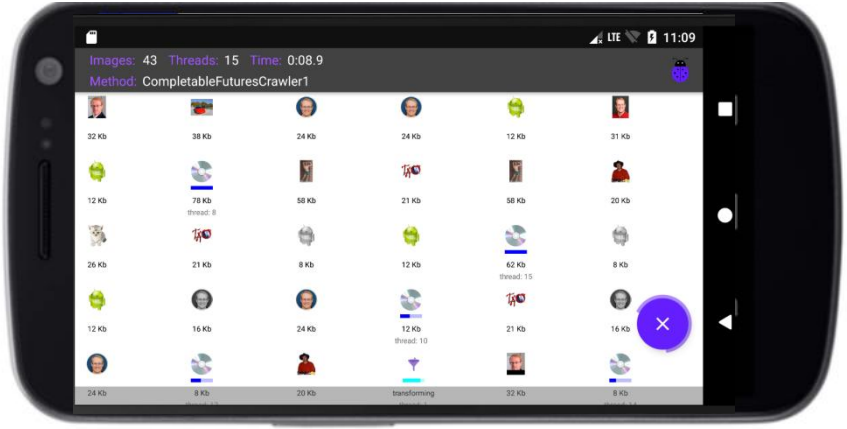
```
List<Image> filteredImages = urls
   .parallelStream()
   .filter(not(this::urlCached))
   .map(this::downloadImage)
   .flatMap(this::applyFilters)
   .collect(toList());
```

A small number of threads may conserve resources at the cost of performance.

The Cons of Synchrony

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





Efficient performance

Efficient resource utilization

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

The Cons of Synchrony

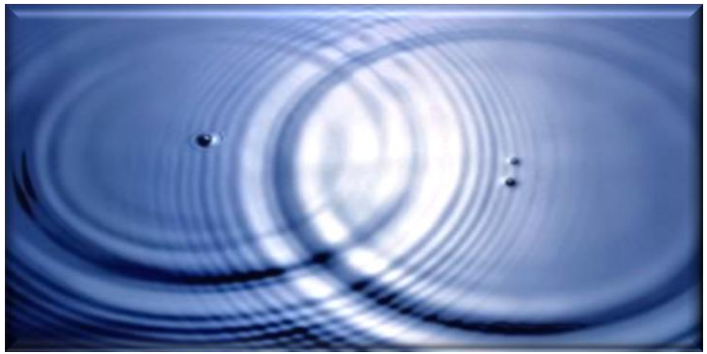
- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
  - May need to change the size of the common fork-join pool

See lesson: *The Java Fork-Join Pool: Maximizing Core Utilization With the Common Fork-Join Pool*

The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
  - May need to change the size of the common fork-join pool
    - Set a system property

```
String desiredThreads = "10";
System.setProperty
  ("java.util.concurrent." +
   "ForkJoinPool.common." +
   "parallelism",
   desiredThreads);
```



It's hard to estimate total number of threads to set in common fork-join pool.

The Cons of Synchrony

- Cons of synchronous calls
  - May not leverage all parallelism available in multi-core systems
  - May need to change the size of the common fork-join pool
    - Set a system property
    - Or use the ManagedBlocker to increase common pool size automatically/temporarily

ManageBlockers can only be used with the common fork-join pool.

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

The End

The Pros and Cons of Asynchrony

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Motivate the need for Java futures by understanding the pros and cons of synchrony
- Motivate the need for Java futures by understanding the pros and cons of asynchrony

Overview of Asynchrony and Asynchronous Operations

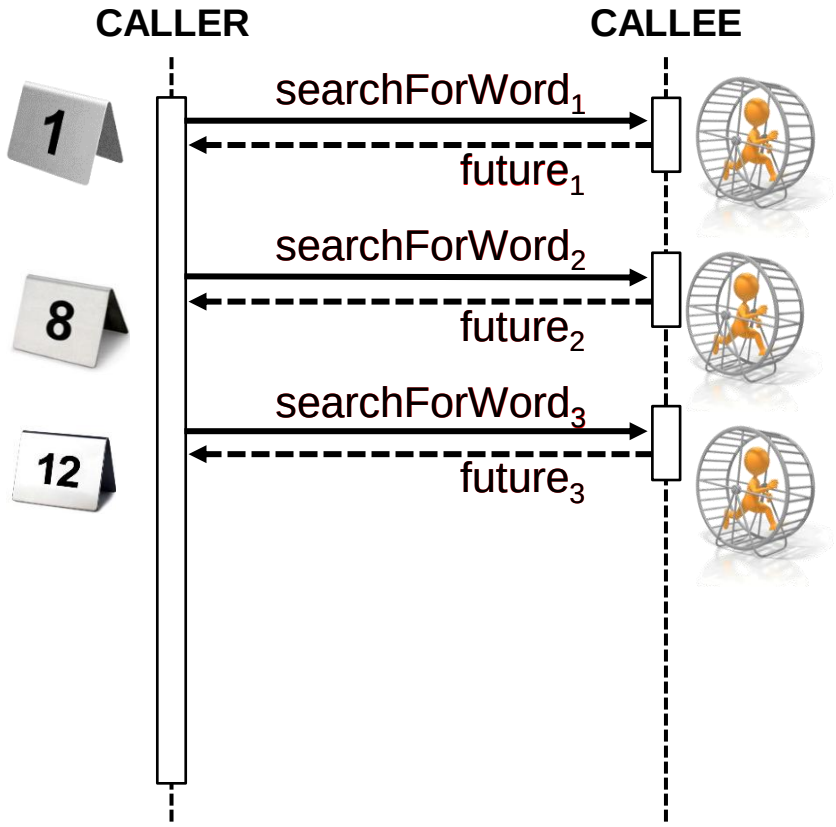
Overview of Asynchrony and Asynchronous Operations

- Asynchrony is a means of concurrent programming where the caller does not block waiting for called code to complete.

See en.wikipedia.org/wiki/Asynchrony_(computer_programming)

Overview of Asynchrony and Asynchronous Operations

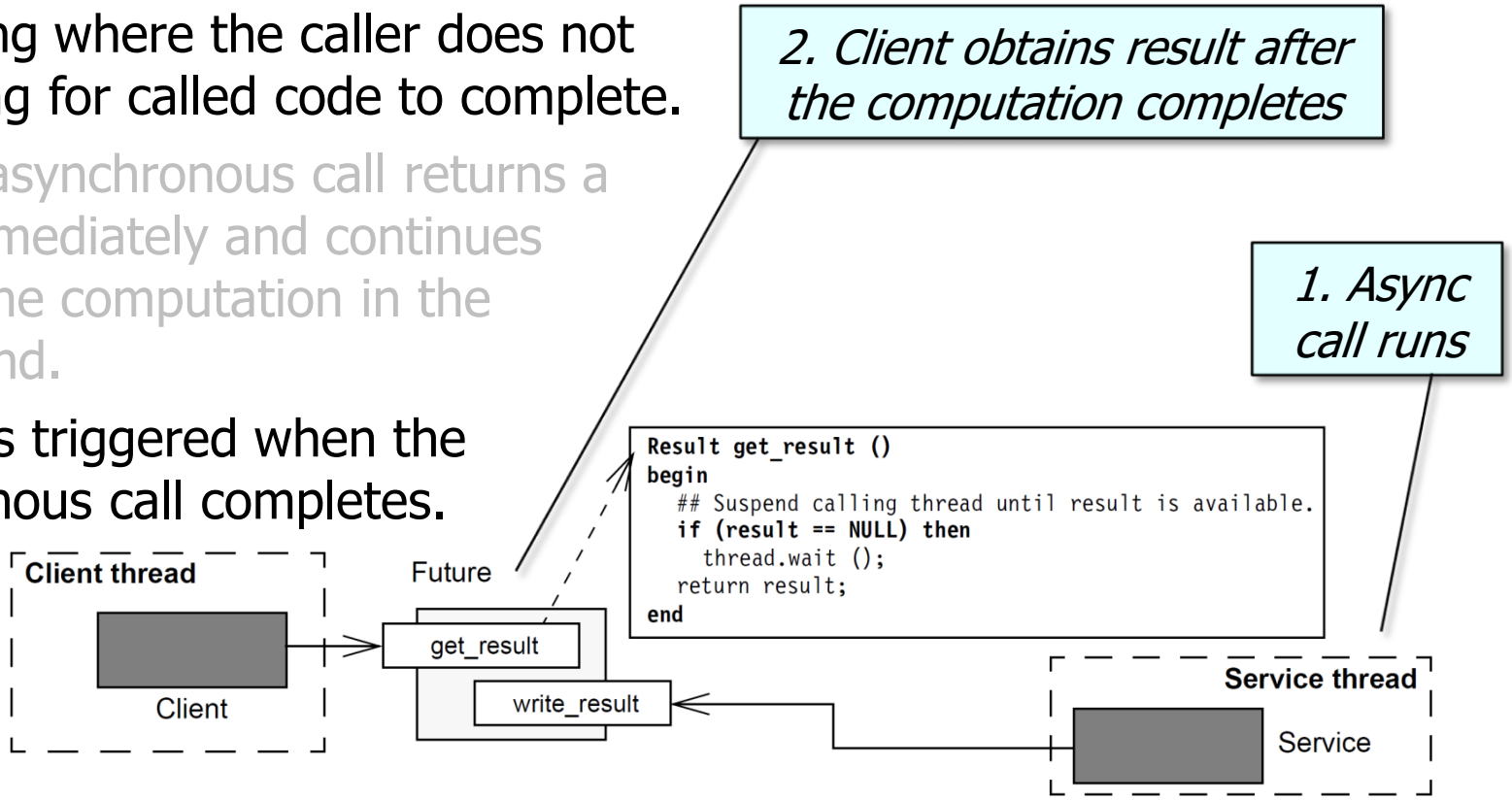
- Asynchrony is a means of concurrent programming where the caller does not block waiting for called code to complete.
  - Instead, asynchronous call returns a future immediately and continues running the computation in the background.



See en.wikipedia.org/wiki/Asynchronous_method_invocation

Overview of Asynchrony and Asynchronous Operations

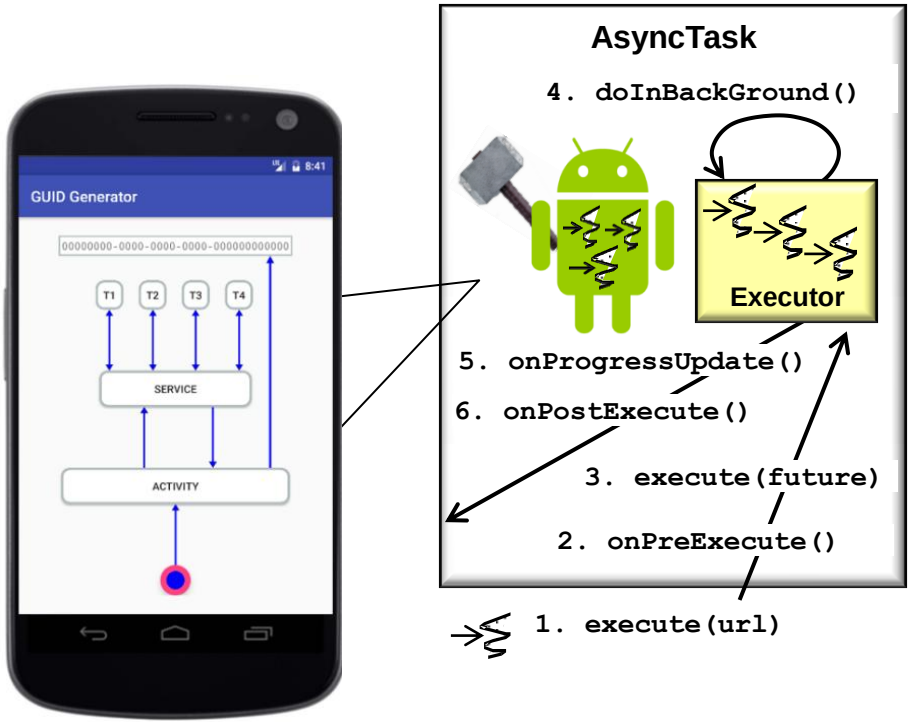
- Asynchrony is a means of concurrent programming where the caller does not block waiting for called code to complete.
  - Instead, asynchronous call returns a future immediately and continues running the computation in the background.
  - A future is triggered when the asynchronous call completes.



See upcoming lessons on *Overview of Java Futures*

Overview of Asynchrony and Asynchronous Operations

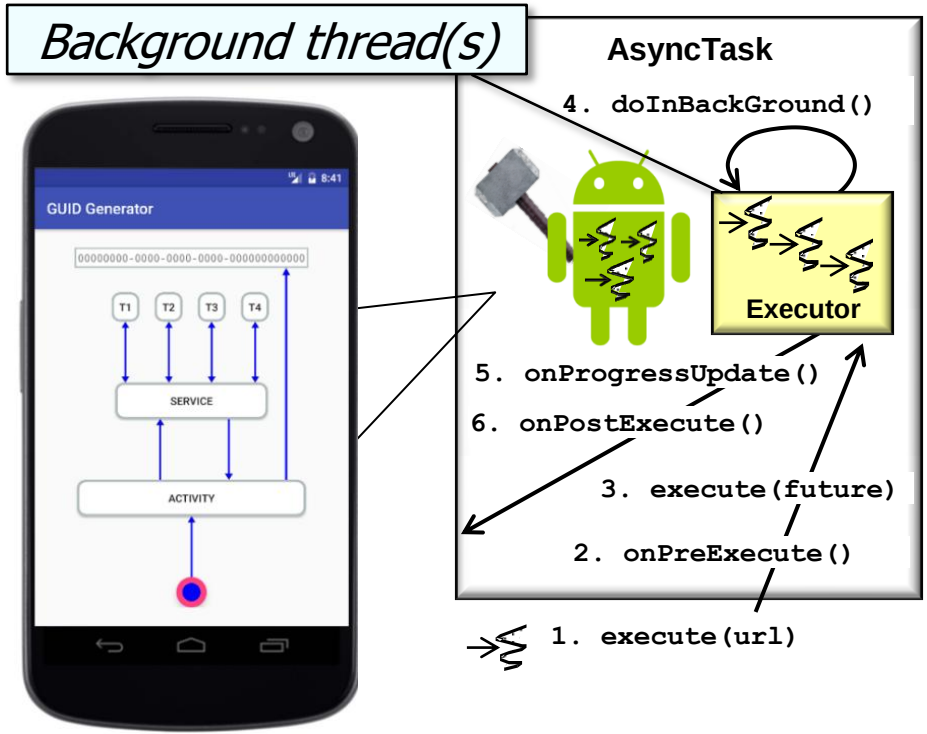
- Android's AsyncTask framework performs background operations and publishes results on the user-interface (UI) thread without having to manipulate threads and/or handlers.



See developer.android.com/reference/android/os/AsyncTask

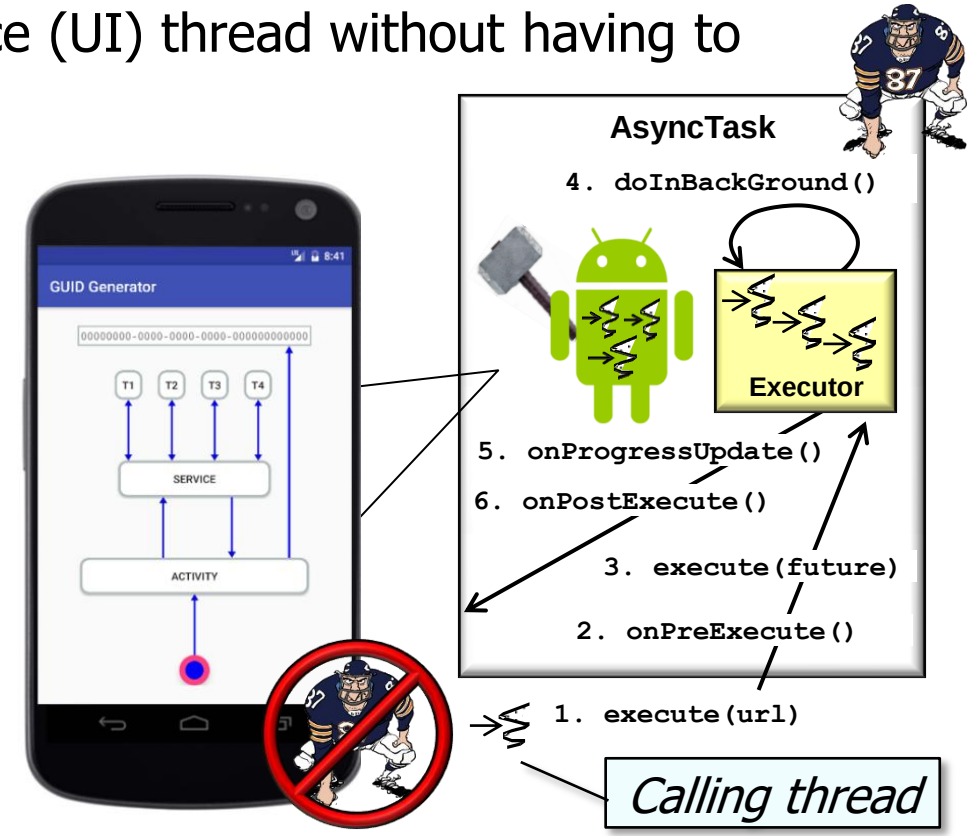
Overview of Asynchrony and Asynchronous Operations

- Android's AsyncTask framework performs background operations and publishes results on the user-interface (UI) thread without having to manipulate threads and/or handlers.
  - AsyncTask executes long-duration operations asynchronously in one or more background threads.



Overview of Asynchrony and Asynchronous Operations

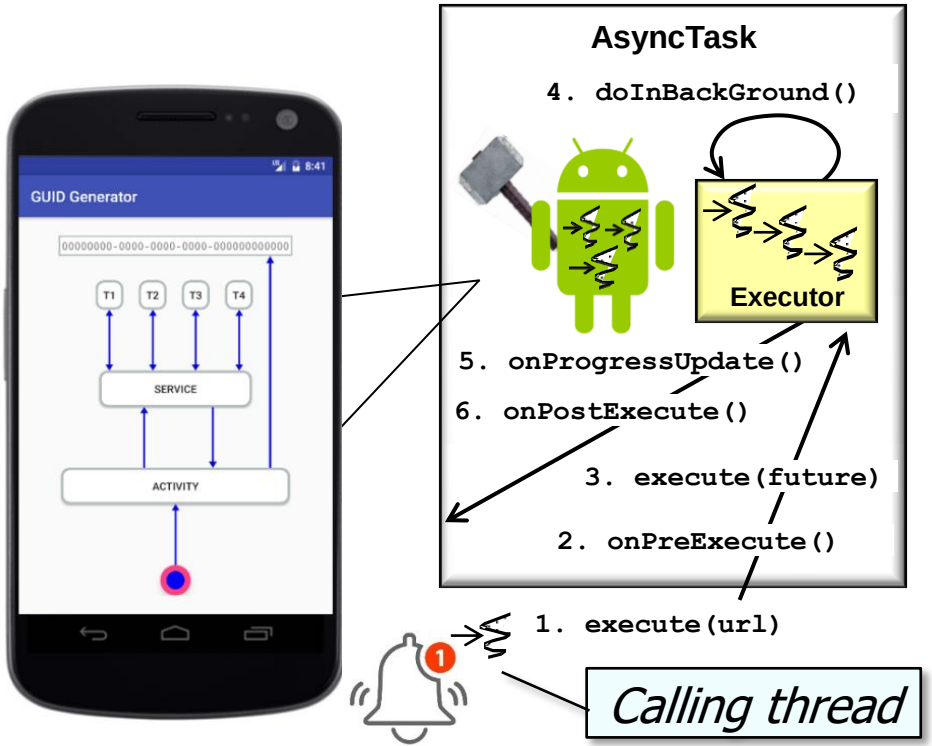
- Android's AsyncTask framework performs background operations and publishes results on the user-interface (UI) thread without having to manipulate threads and/or handlers.
  - AsyncTask executes long-duration operations asynchronously in one or more background threads.
  - Blocking operations in background threads don't block the calling (e.g., UI) thread.



See developer.android.com/training/multiple-threads/communicate-ui

Overview of Asynchrony and Asynchronous Operations

- Android's AsyncTask framework performs background operations and publishes results on the user-interface (UI) thread without having to manipulate threads and/or handlers.
  - AsyncTask executes long-duration operations asynchronously in one or more background threads.
  - Blocking operations in background threads don't block the calling (e.g., UI) thread.
  - The calling (UI) thread can be notified upon completion, failure, or progress of the AsyncTask.



AsyncTask shields client code from details of programming futures.

The Pros of Asynchrony

The Pros of Asynchrony

- Pros of asynchronous operations

The Pros of Asynchrony

- Pros of asynchronous operations
  - Responsiveness
    - A calling thread needn't block waiting for the asynchronous request to complete.

See en.wikipedia.org/wiki/Asynchronous_method_invocation

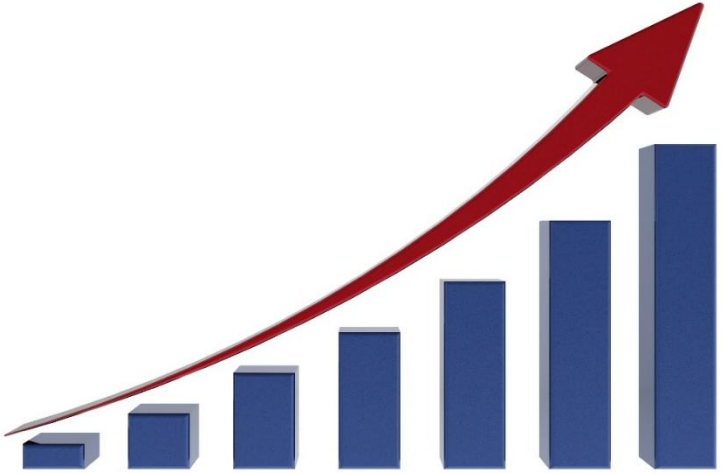
The Pros of Asynchrony

- Pros of asynchronous operations
  - Responsiveness
    - A calling thread needn't block waiting for the asynchronous request to complete.
      - Better leverage the parallelism available in multi-core systems

See headcrashing.wordpress.com/2015/07/20/iobound-completablefuture

The Pros of Asynchrony

- Pros of asynchronous operations
  - Responsiveness
  - Elasticity
    - Multiple requests can run scalably and concurrently on multiple cores.



See en.wikipedia.org/wiki/Elasticity_(cloud_computing)

The Pros of Asynchrony

- Pros of asynchronous operations
  - Responsiveness
  - Elasticity
    - Multiple requests can run scalably and concurrently on multiple cores.
      - Elasticity is particularly useful to auto-scale computations in cloud environments.

See en.wikipedia.org/wiki/Elasticity_(cloud_computing) and en.wikipedia.org/wiki/Autoscaling

The Cons of Asynchrony

The Cons of Asynchrony

- Cons of asynchronous operations

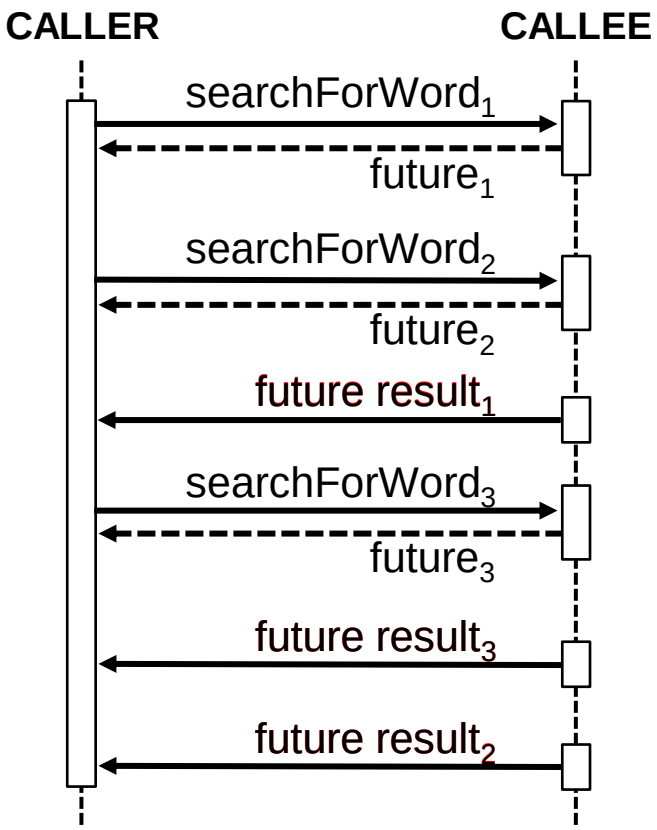
The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
    - Response times may be unpredictable due to non-determinism of asynchronous operations.

Non-determinism is a general problem with concurrency and not just asynchrony.

The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
    - Response times may be unpredictable due to non-determinism of asynchronous operations.
    - Results can occur in a different order than the original calls were made.



Additional time and effort may be required if results must be ordered somehow.

The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
  - Complicated programming and debugging

The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
  - Complicated programming and debugging
    - The patterns and best practices of asynchronous programming are not well understood.

Parallel and Asynchronous Programming in Java 8

Java 8 offered a boon to parallel and asynchronous programming. Let's check out the lessons Java learned from JavaScript and how JDK 8 changed the game.

by Lisa Steendam · May. 11, 18 · Java Zone · Tutorial

Like (16) Comment (0) Save Tweet 45.66k Views

Parallel code, which is code that runs on more than one thread, was once the nightmare of many an experienced developer, but Java 8 brought a lot of changes that should make this performance-boosting trick a lot more manageable.

CompletableFuture

`CompletableFuture` implements both the `Future` and the `CompletionStage` interface. `Future` already existed pre-Java8, but it wasn't very developer-friendly by itself. You could only get the result of the asynchronous computation by using the `.get()` method, which blocked the rest (making the async part pretty pointless most of the time) and you needed to implement each possible scenario manually. Adding the `CompletionStage` interface was the breakthrough that made asynchronous programming in Java workable.

`CompletionStage` is a promise, namely the promise that the computation will eventually be done. It contains a bunch of methods that let you attach callbacks that will be executed on that completion. Now we can handle the result without blocking.

There are two main methods that let you start the asynchronous part of your code: `supplyAsync` if you want to do something with the result of the method, and `runAsync` if you don't.

See dzone.com/articles/parallel-and-asynchronous-programming-in-java-8

The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
  - Complicated programming and debugging
    - The patterns and best practices of asynchronous programming are not well understood.
      - Asynchronous programming is tricky without proper abstractions.

See dzone.com/articles/callback-hell

The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
  - Complicated programming and debugging
    - The patterns and best practices of asynchronous programming are not well understood.
    - Errors can be hard to track due to unpredictability.

See www.jetbrains.com/help/idea/tutorial-java-debugging-deep-dive.html

The Cons of Asynchrony

- Cons of asynchronous operations
  - Unpredictability
  - Complicated programming and debugging
    - The patterns and best practices of asynchronous programming are not well understood.
    - Errors can be hard to track due to unpredictability.

Again, this non-determinism is a general problem with concurrent processing.

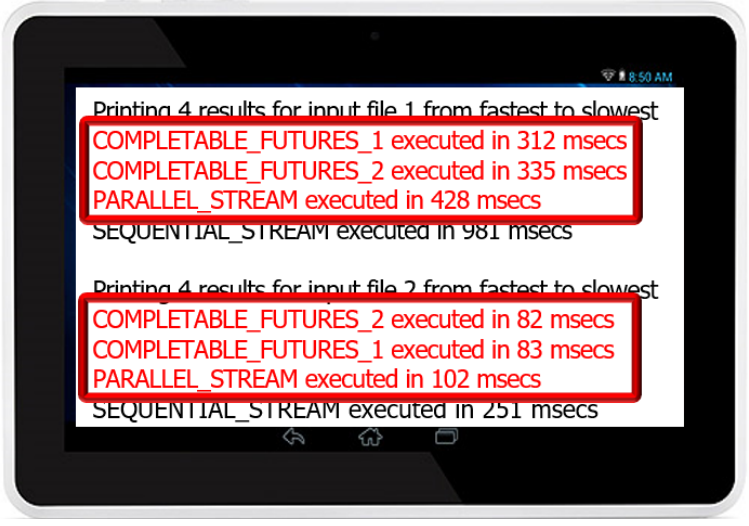
Weighing the Pros and Cons of Asynchrony

Weighing the Pros and Cons of Asynchrony

- Two things are necessary for the pros of asynchrony to outweigh the cons.

Weighing the Pros and Cons of Asynchrony

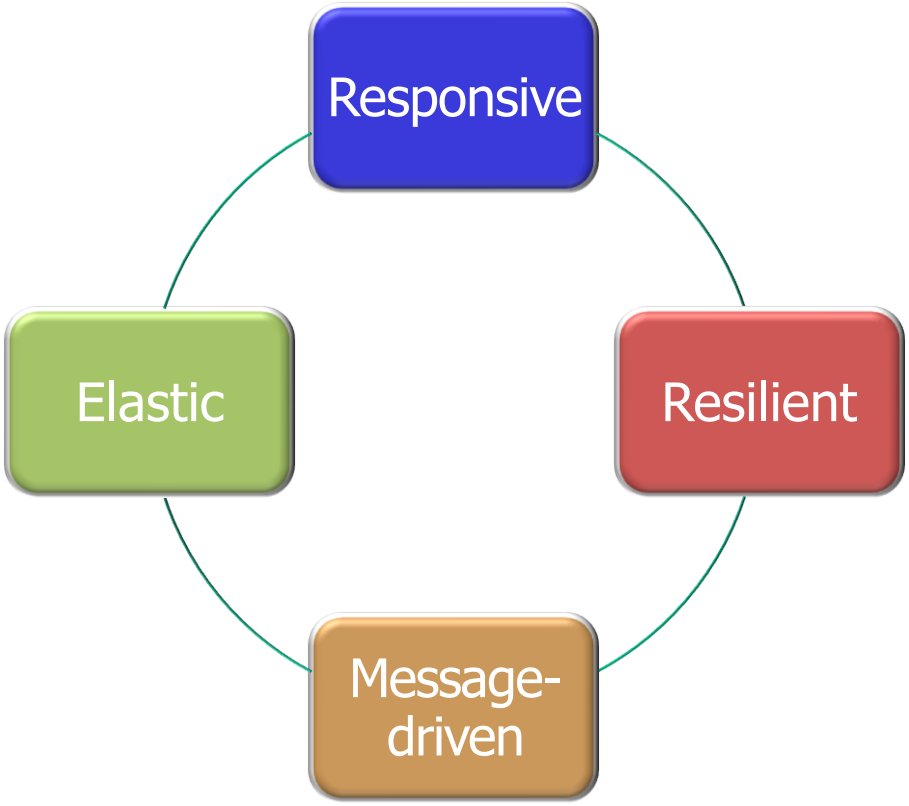
- Two things are necessary for the pros of asynchrony to outweigh the cons.
  1. Performance should improve to offset the increased complexity of programming and debugging.



See upcoming lesson on *Java Completable Futures ImageStreamGang Example*

Weighing the Pros and Cons of Asynchrony

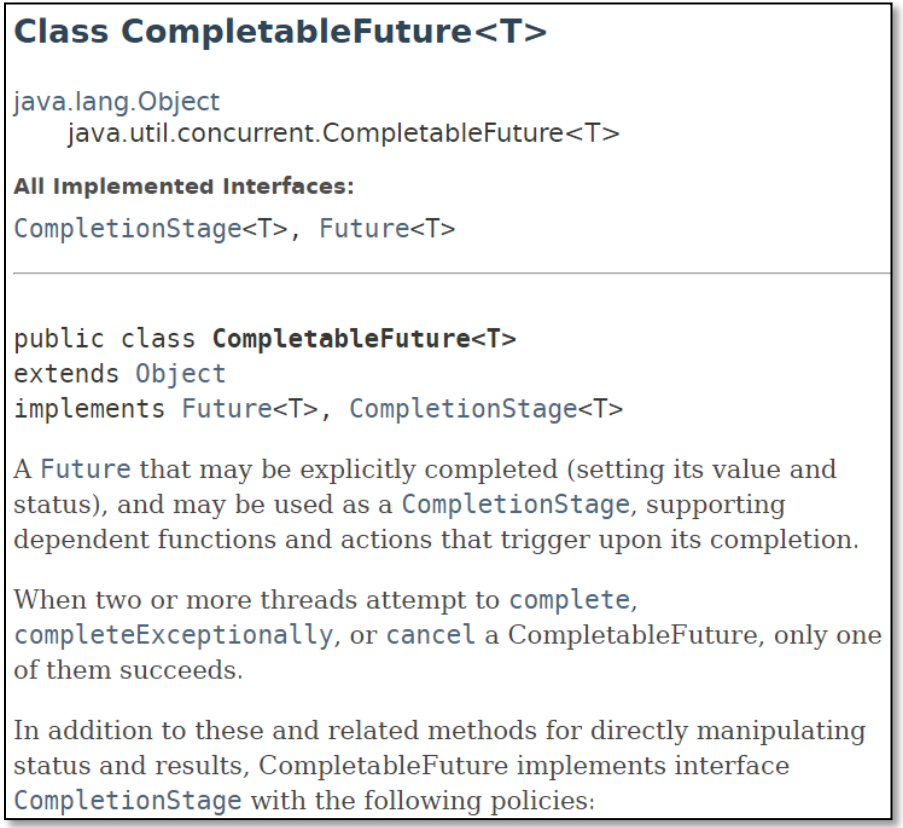
- Two things are necessary for the pros of asynchrony to outweigh the cons.
  1. Performance should improve to offset the increased complexity of programming and debugging.
  2. An asynchronous programming model should reflect the key principles of the reactive paradigm.



See earlier lesson on *Overview of Reactive Programming*

Weighing the Pros and Cons of Asynchrony

- Java's completable future framework provides an asynchronous concurrent programming model that performs well and supports the reactive paradigm.



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:

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

The End

Overview of Java Futures

Part I

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

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

<<Java Class>>
CompletableFuture<T>

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

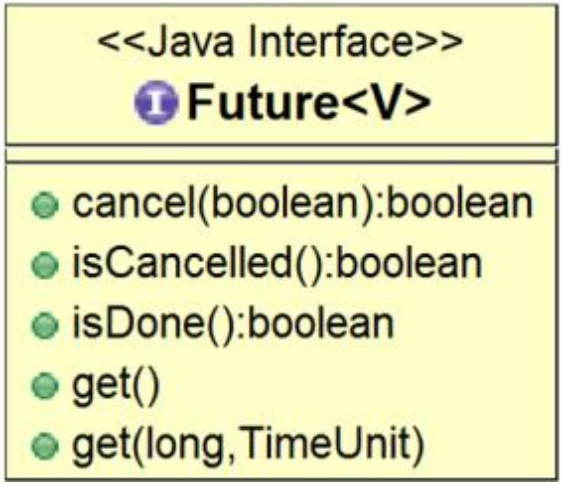
See en.wikipedia.org/wiki/Java_version_history

Learning Objectives in This Part of the Lesson

- Motivate the need for Java futures by understanding the pros and cons of synchrony and asynchrony
- Know how Java futures provide the foundation for completable futures in Java
  - Understand a human-known use of Java futures

Learning Objectives in This Part of the Lesson

- Motivate the need for Java futures by understanding the pros and cons of synchrony and asynchrony
- Know how Java futures provide the foundation for completable futures in Java
  - Understand a human-known use of Java futures
  - Recognize the methods in the Future interface

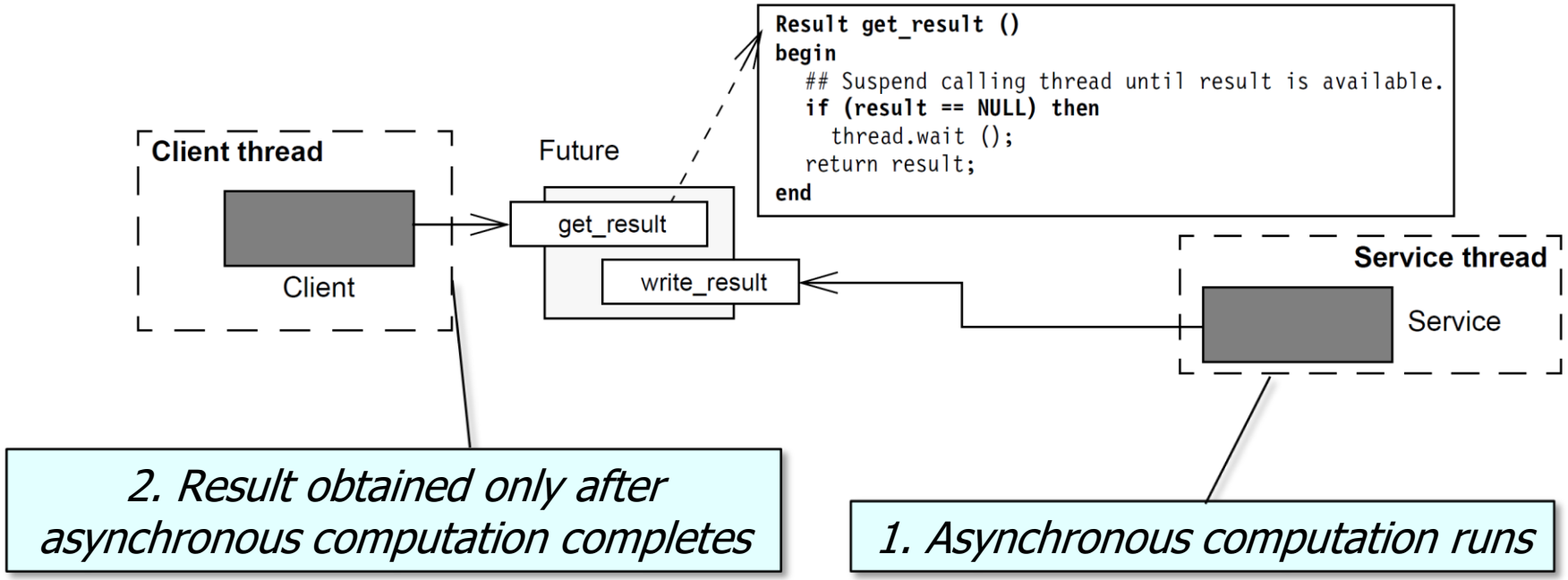


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

A Human-Known Use of Java Futures

A Human-Known Use of Java Futures

- A future is essentially a proxy that represents the result(s) of an asynchronous call.



See en.wikipedia.org/wiki/Futures_and_promises

A Human-Known Use of Java Futures

- A future is essentially a proxy that represents the result(s) of an asynchronous call.

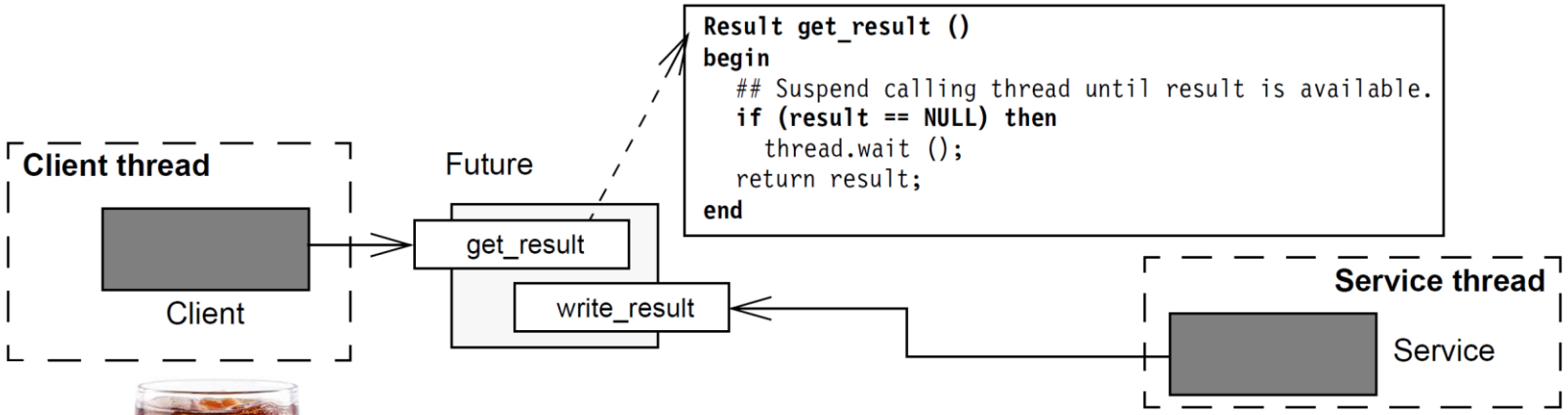
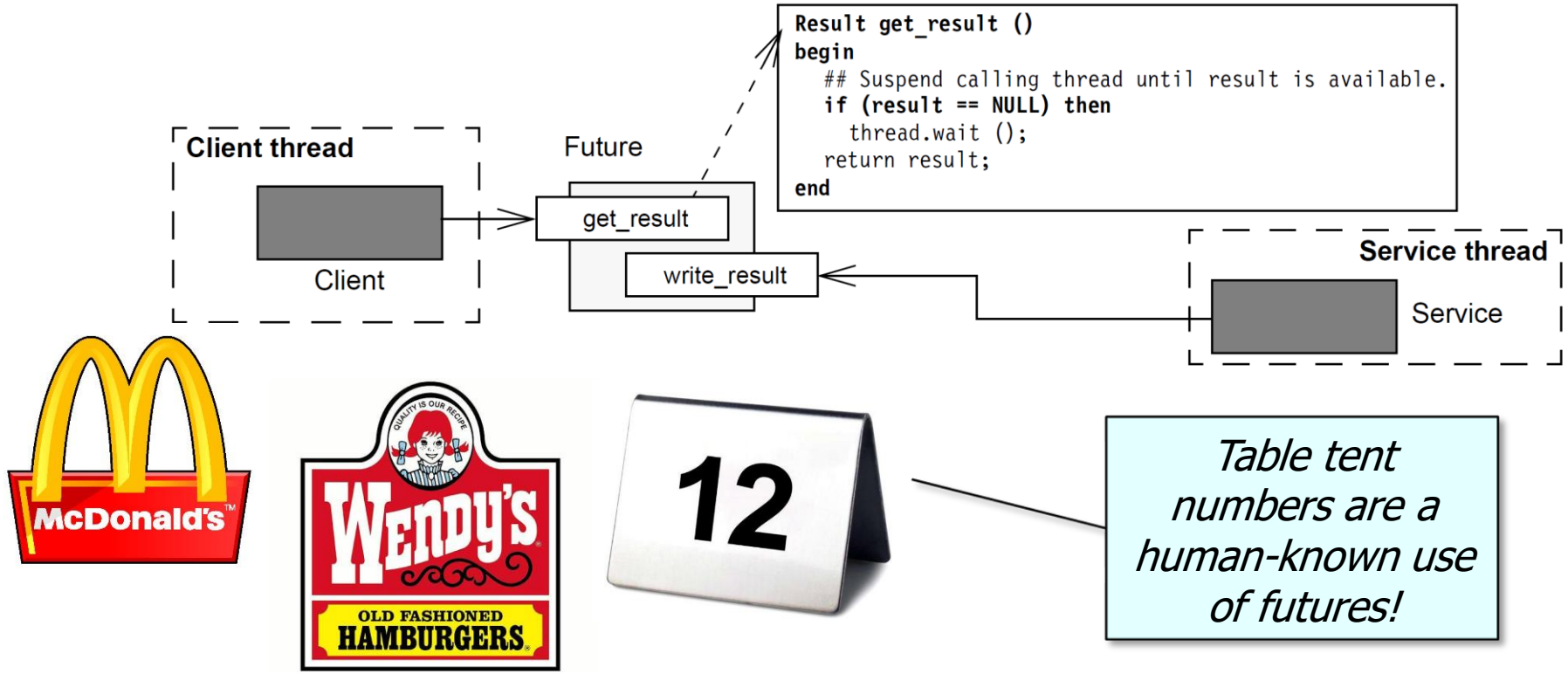


Table tent numbers are a human-known use of futures!

See www.citygrafx.com/table-numbers-table-markers

A Human-Known Use of Java Futures

- A future is essentially a proxy that represents the result(s) of an asynchronous call.

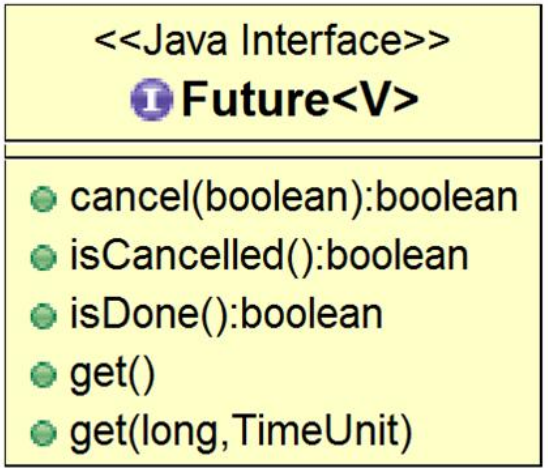


McDonald's vs. Wendy's model of preparing fast food

Overview of the Java Future API

Overview of the Java Future API

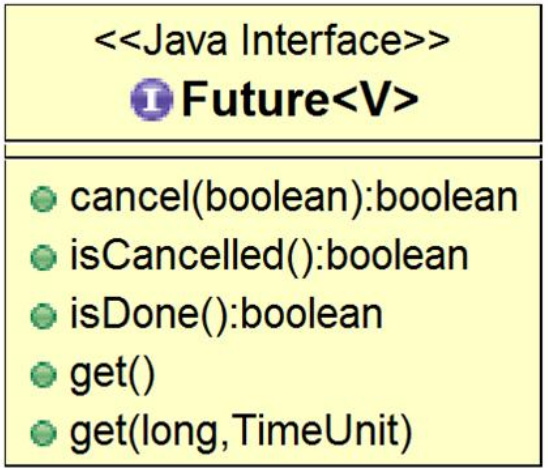
- Java 5 added asynchronous call support via the Java Future interface.

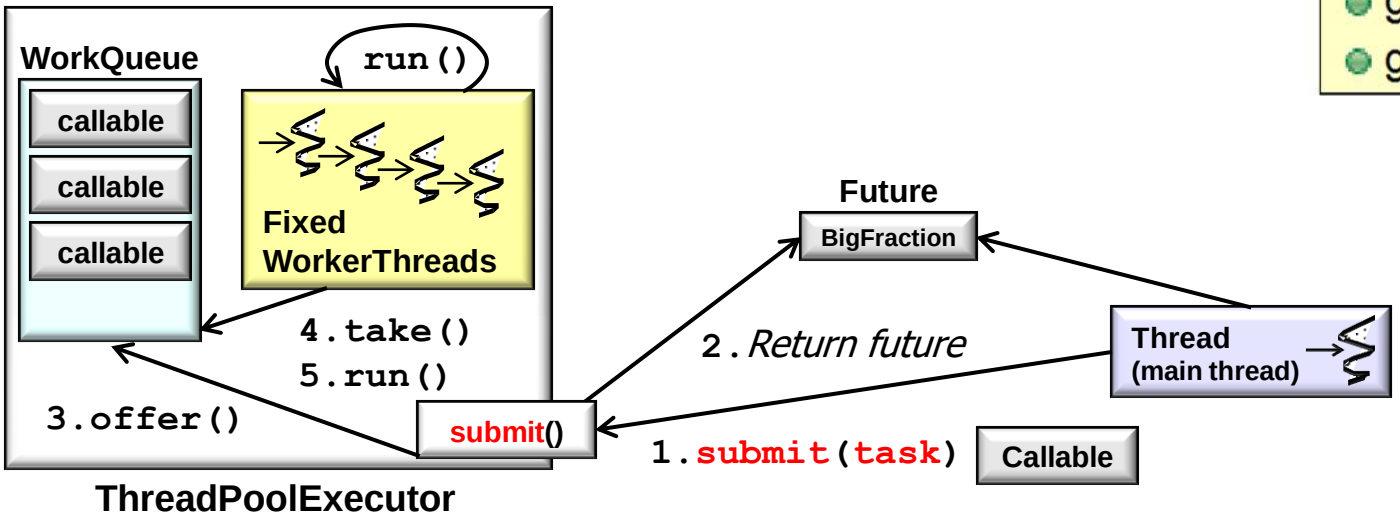


See en.wikipedia.org/wiki/Java_version_history

Overview of the Java Future API

- Java Future methods can manage a task's life cycle after it's submitted to run asynchronously.

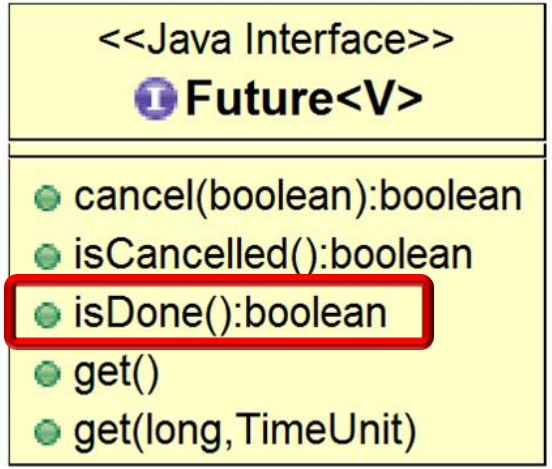


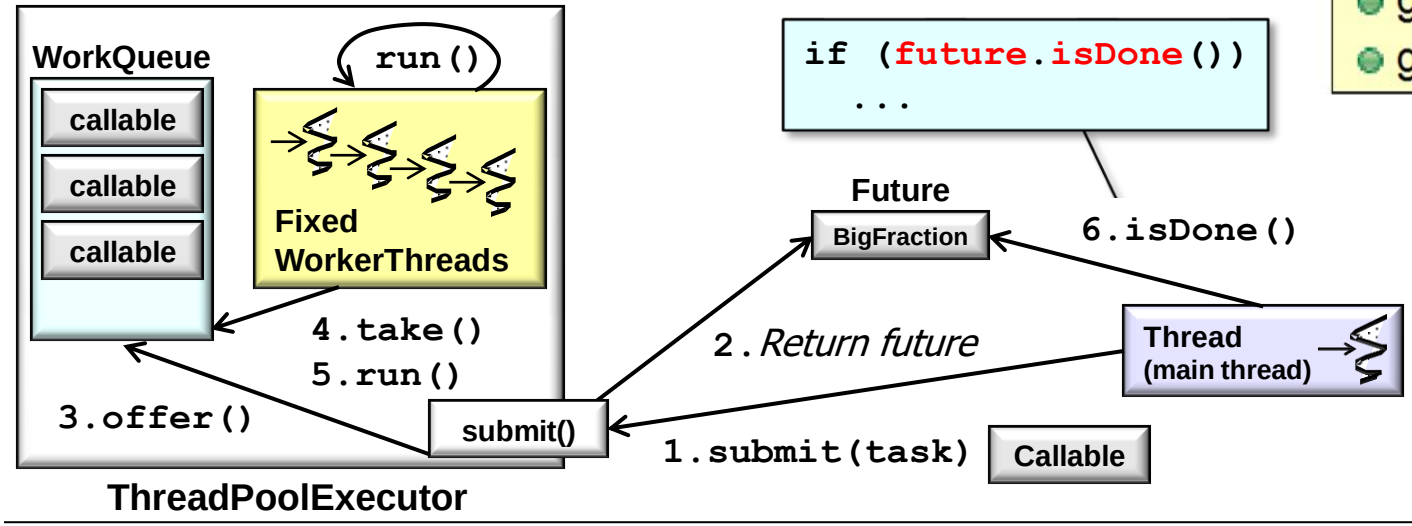


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

Overview of the Java Future API

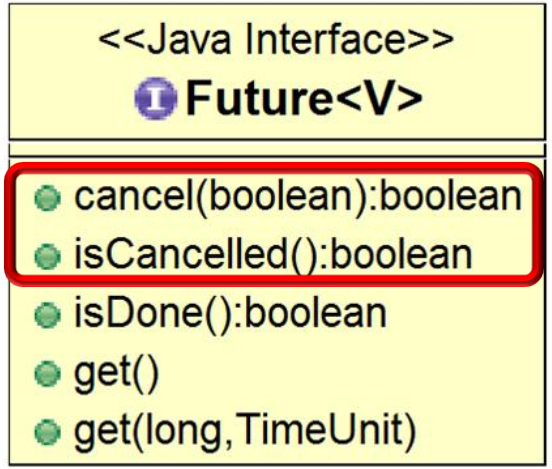
- Java Future methods can manage a task's life cycle after it's submitted to run asynchronously.
  - A future can be tested for completion.

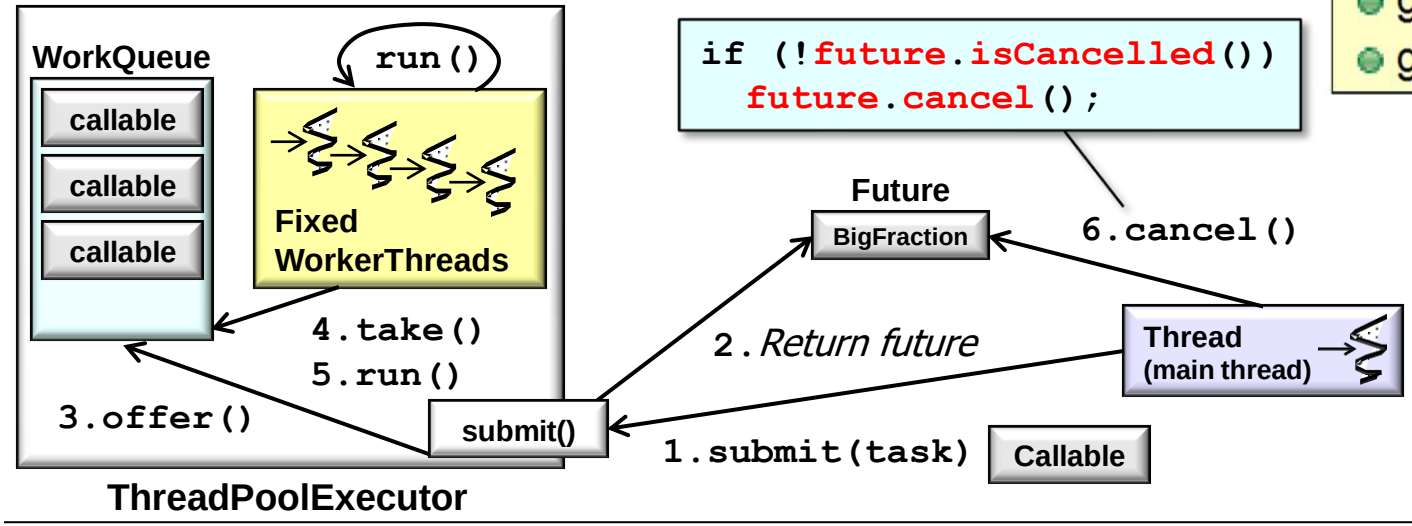




Overview of the Java Future API

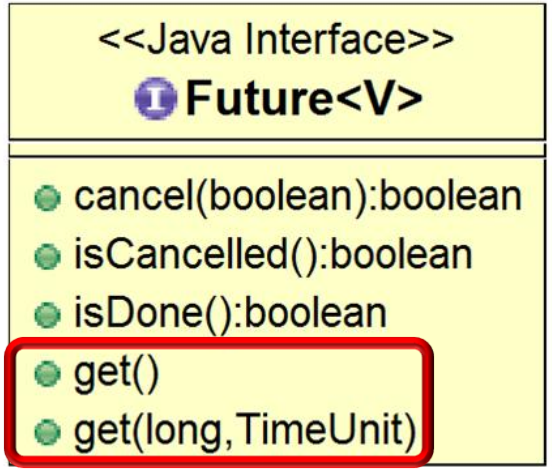
- Java Future methods can manage a task's life cycle after it's submitted to run asynchronously.
  - A future can be tested for completion.
  - A future be tested for cancellation and cancelled.

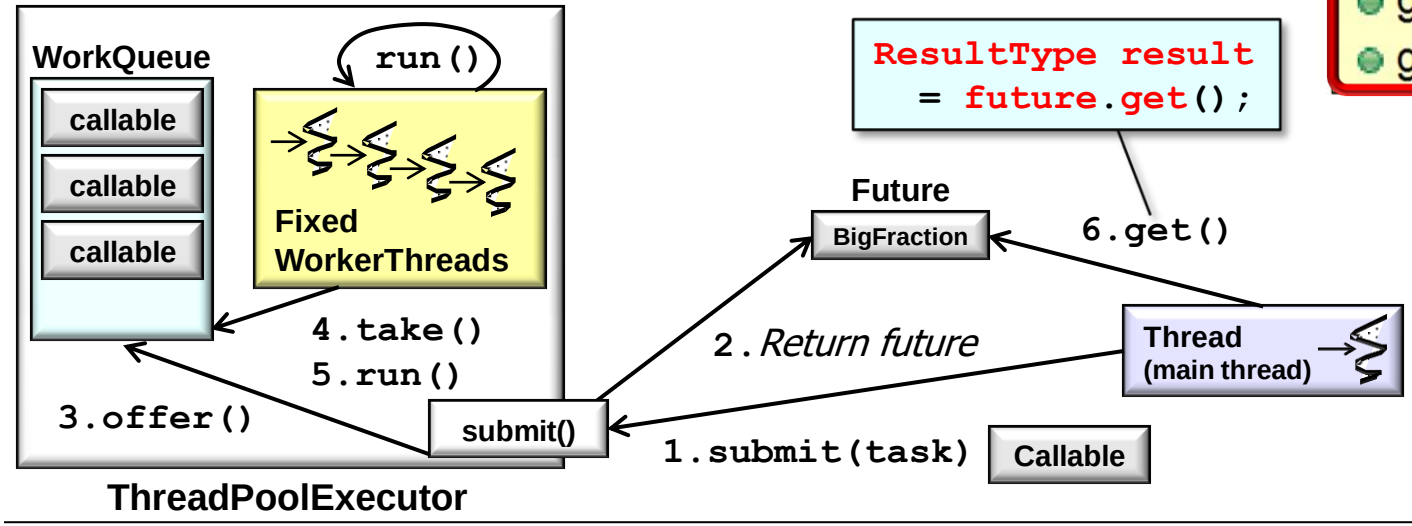




Overview of the Java Future API

- Java Future methods can manage a task's life cycle after it's submitted to run asynchronously.
  - A future can be tested for completion.
  - A future be tested for cancellation and cancelled.
  - A future can retrieve a two-way task's result.





The End

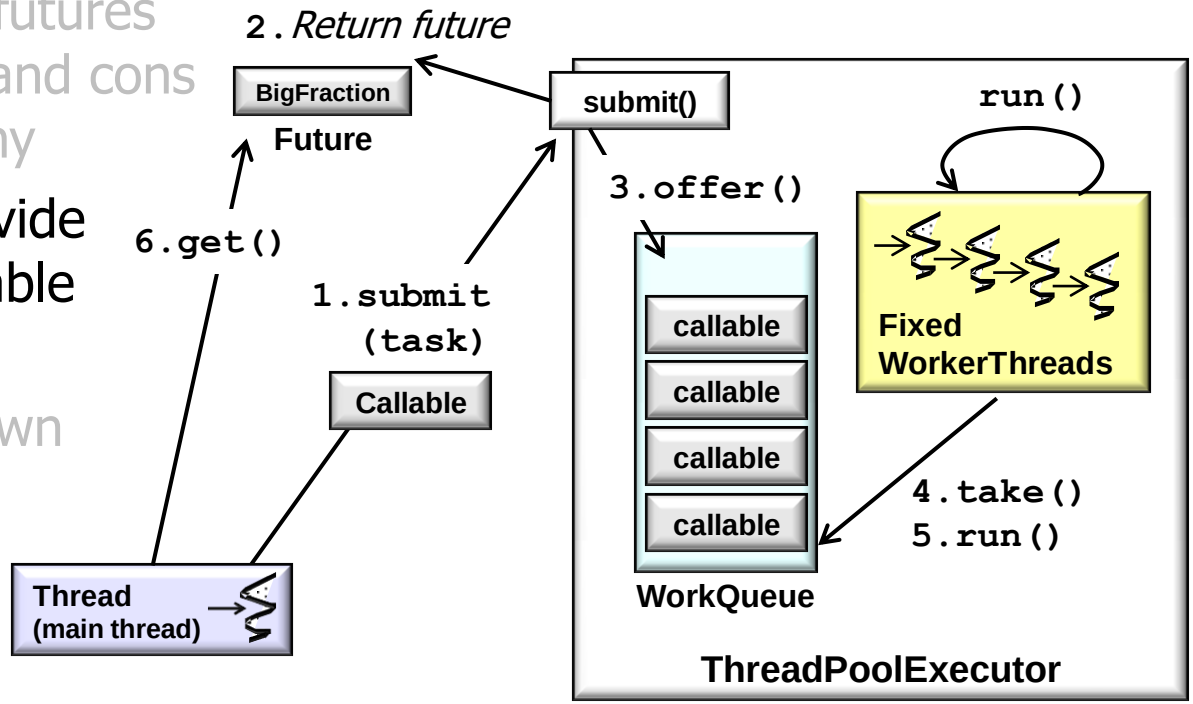
Overview of Java Futures

Part II

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

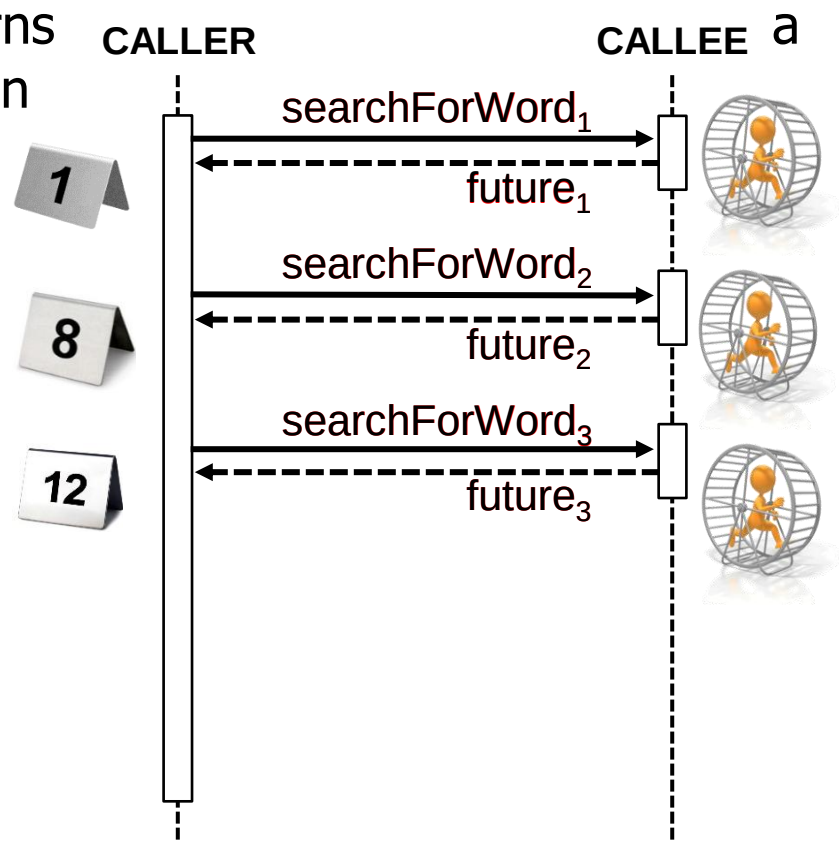
- Motivate the need for Java futures by understanding the pros and cons of synchrony and asynchrony
- Know how Java futures provide the foundation for completable futures in Java
  - Understand a human-known use of Java futures
  - Visualize Java futures in action



Visualizing Java Futures in Action

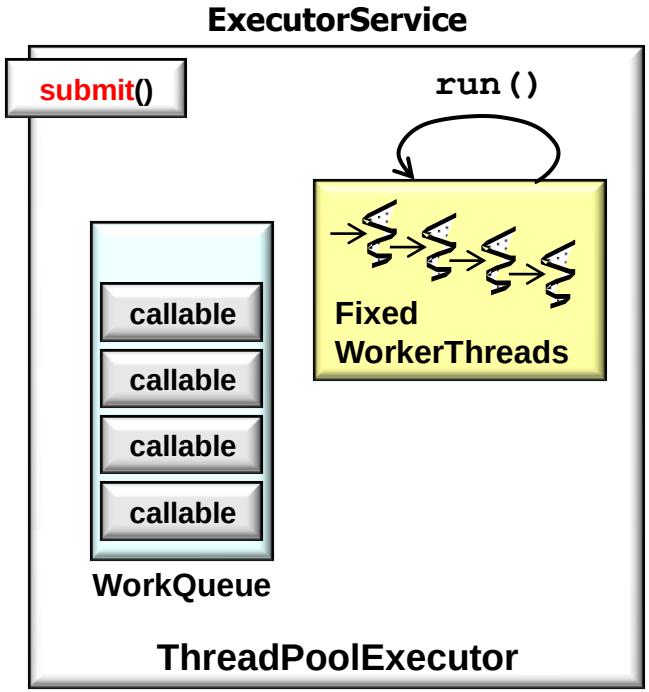
Visualizing Java Futures in Action

- A Java asynchronous call immediately returns future and continues to run the computation in a background thread.



Visualizing Java Futures in Action

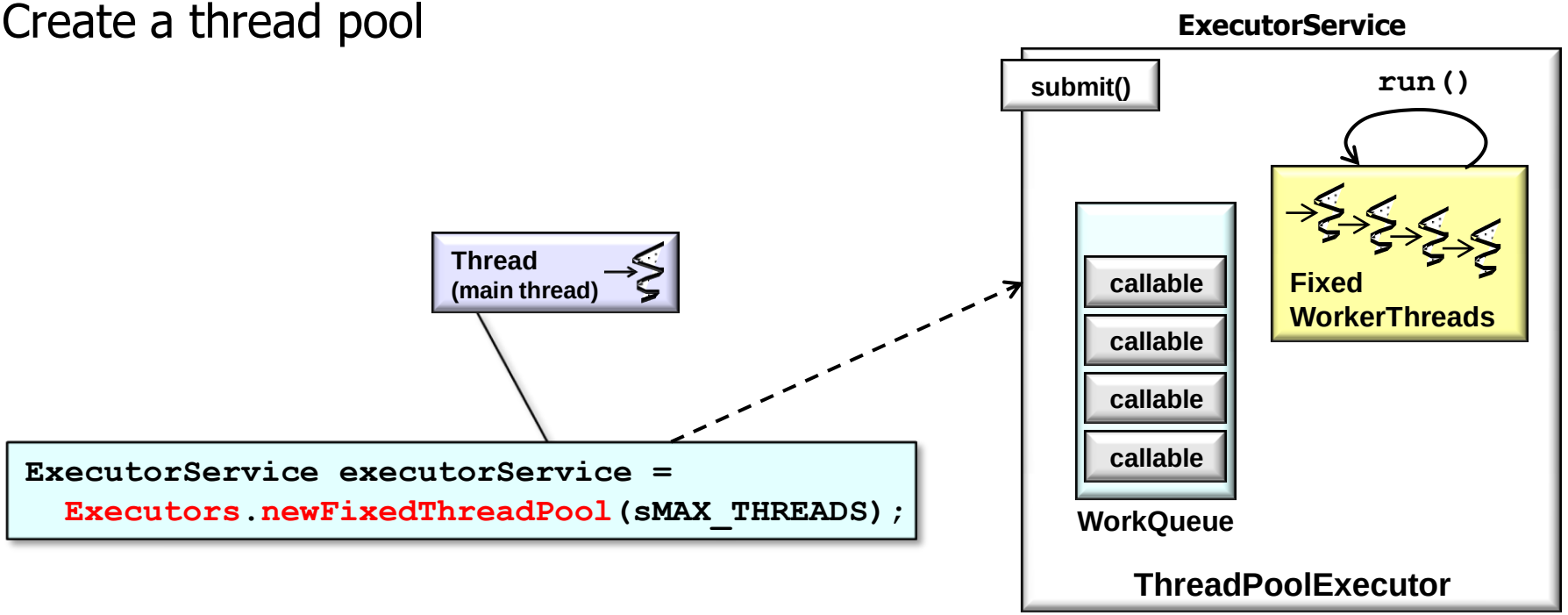
- ExecutorService.submit() can initiate an asynchronous call in Java.



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit

Visualizing Java Futures in Action

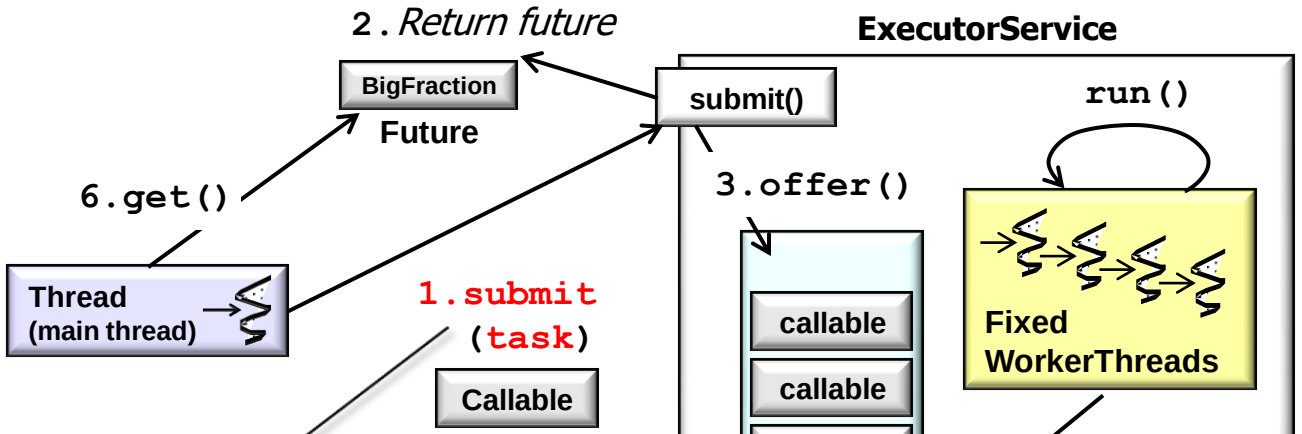
- ExecutorService.submit() can initiate an asynchronous call in Java.
  - Create a thread pool



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool

Visualizing Java Futures in Action

- ExecutorService.submit() can initiate an asynchronous call in Java.
  - Create a thread pool
  - Submit a task

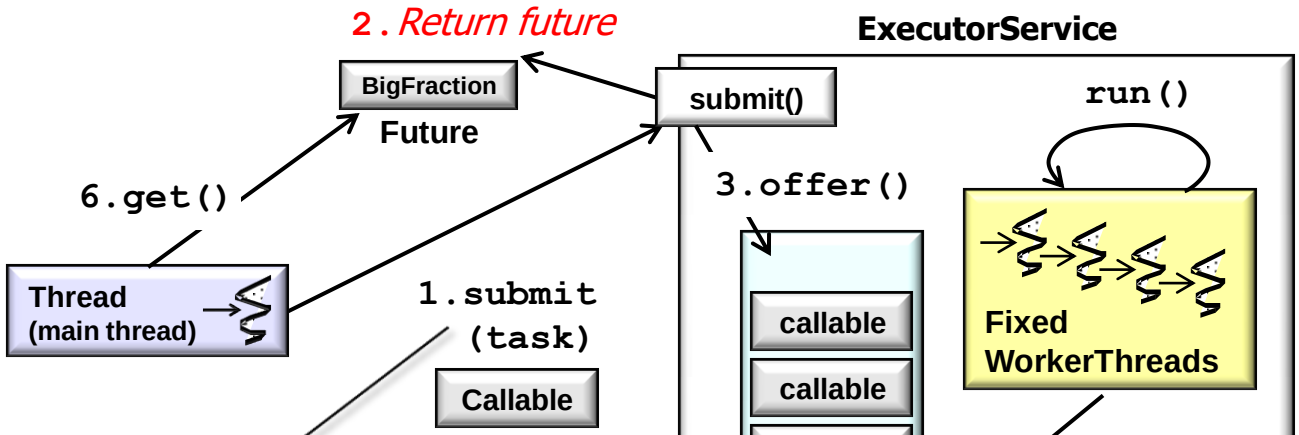



```
Callable<BigFraction> task = () -> {
  BigFraction bf1 = new BigFraction(f1);
  BigFraction bf2 = new BigFraction(f2);
  return bf1.multiply(bf2);
};

Future<BigFraction> future = executorService.submit(task);
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit

Visualizing Java Futures in Action

- ExecutorService.submit() can initiate an asynchronous call in Java.
  - Create a thread pool
  - Submit a task
  - Return a future
    - Implemented as a FutureTask

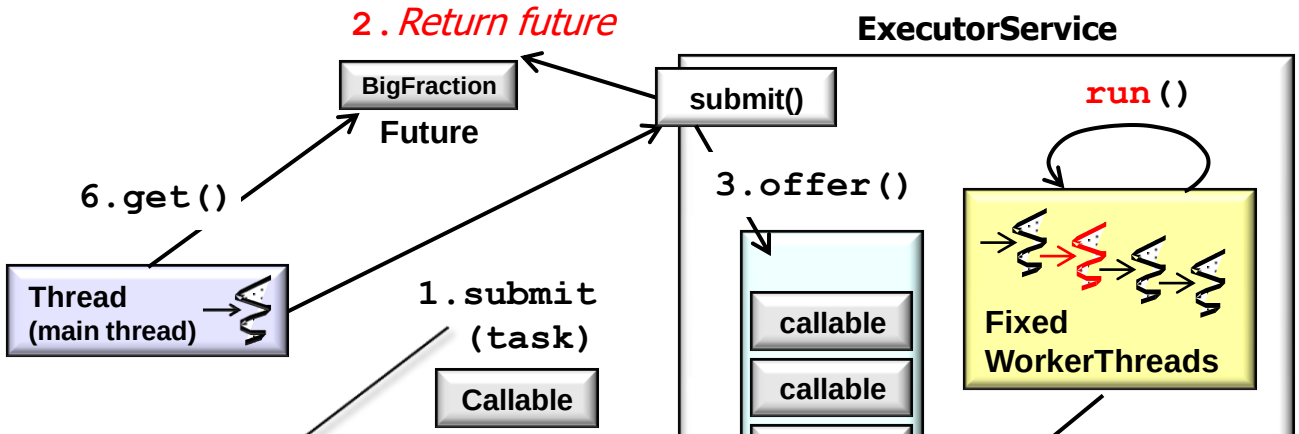



```
Callable<BigFraction> task = () -> {
  BigFraction bf1 = new BigFraction(f1);
  BigFraction bf2 = new BigFraction(f2);
  return bf1.multiply(bf2);
};

Future<BigFraction> future = executorService.submit(task);
```

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

Visualizing Java Futures in Action

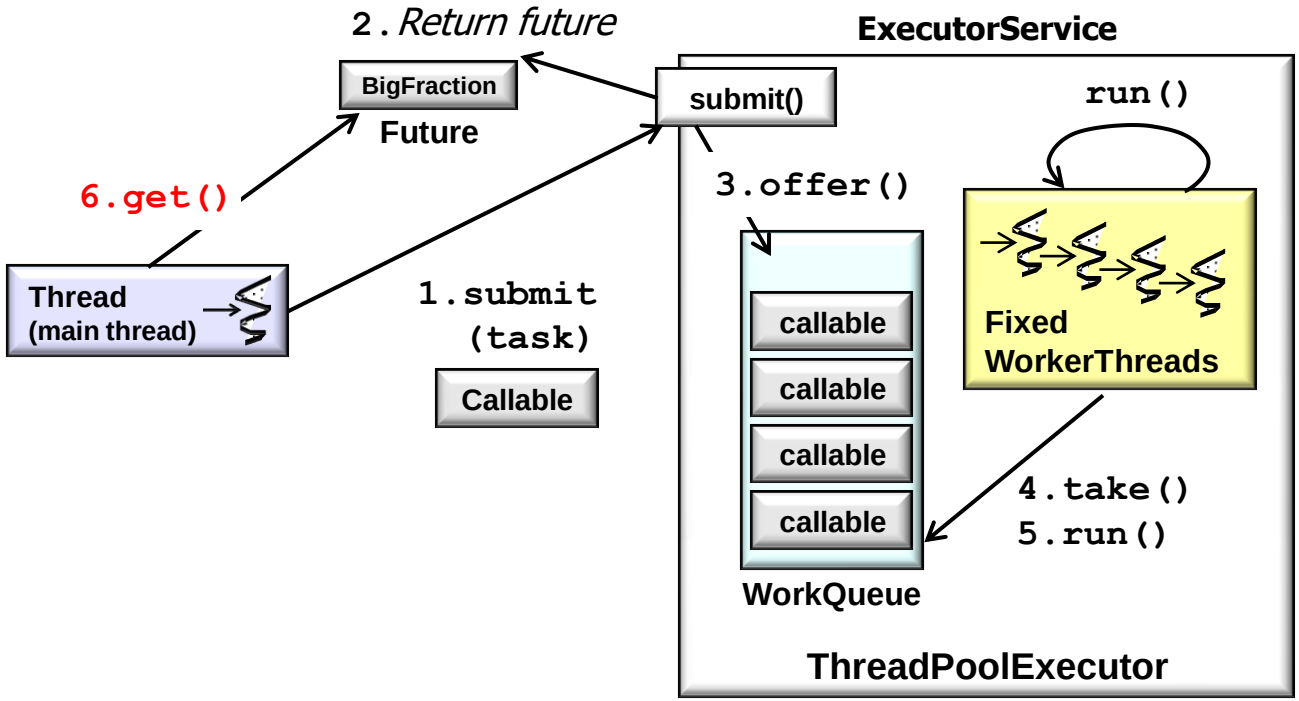
- ExecutorService.submit() can initiate an asynchronous call in Java.
  - Create a thread pool
  - Submit a task
  - Return a future
  - Run computation asynchronously




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

Future<BigFraction> future = executorService.submit(task);
```

Visualizing Java Futures in Action

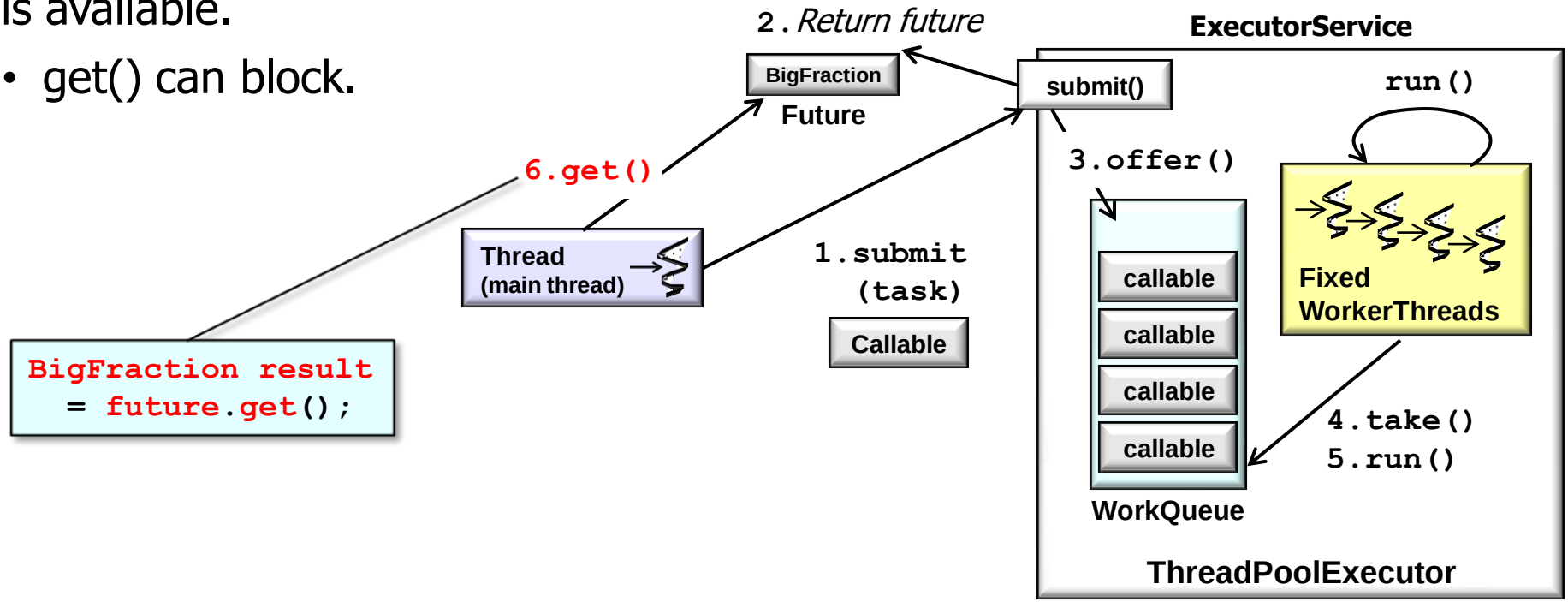
- When the asynchronous call completes, the future is triggered and the result is available.



See www.nurkiewicz.com/2013/02/javautilconcurrentfuture-basics.html

Visualizing Java Futures in Action

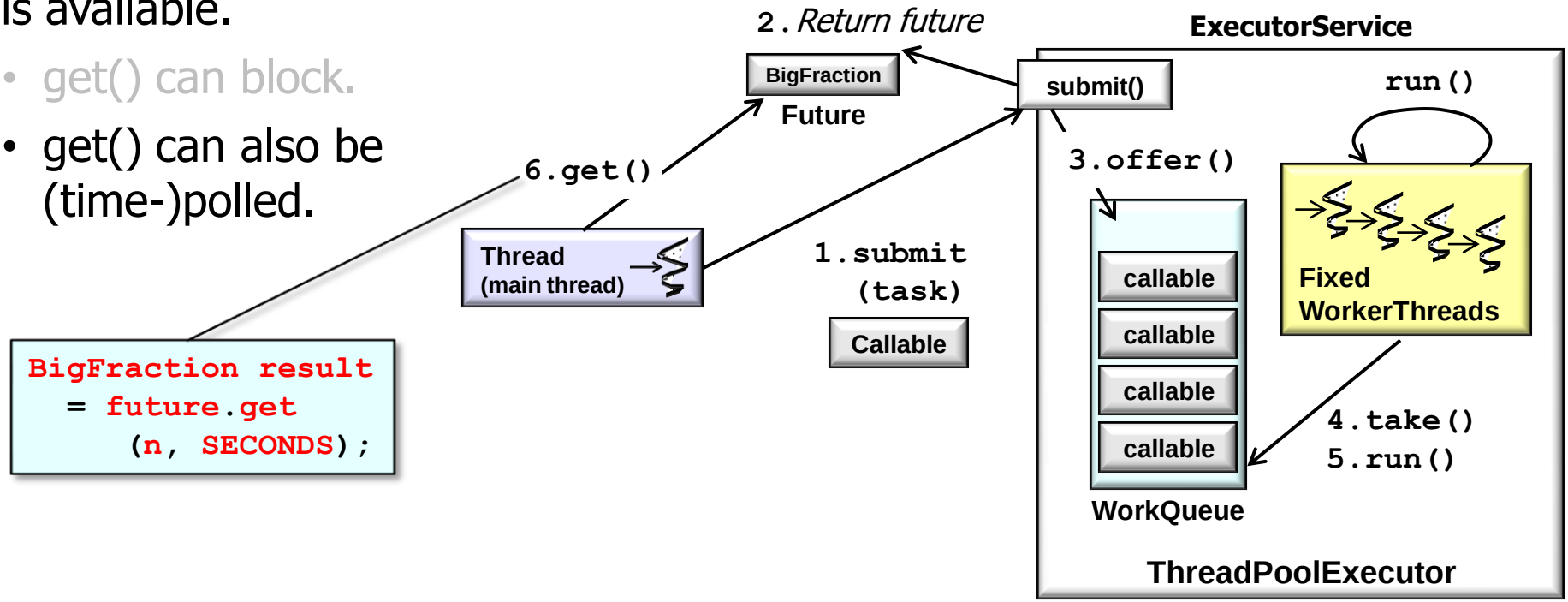
- When the asynchronous call completes, the future is triggered and the result is available.
  - get() can block.



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get

Visualizing Java Futures in Action

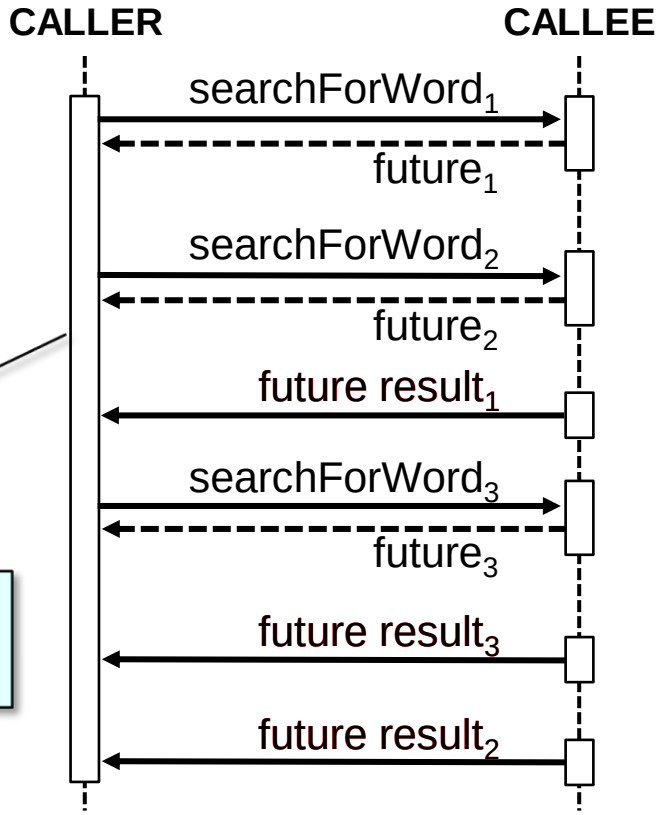
- When the asynchronous call completes, the future is triggered and the result is available.
  - get() can block.
  - get() can also be (time-)polled.



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get

Visualizing Java Futures in Action

- When the asynchronous call completes, the future is triggered and the result is available.
  - get() can block.
  - get() can also be (time-)polled.



Computations can complete in a different order than the asynchronous calls were made.

The End

Programming With Java Futures

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

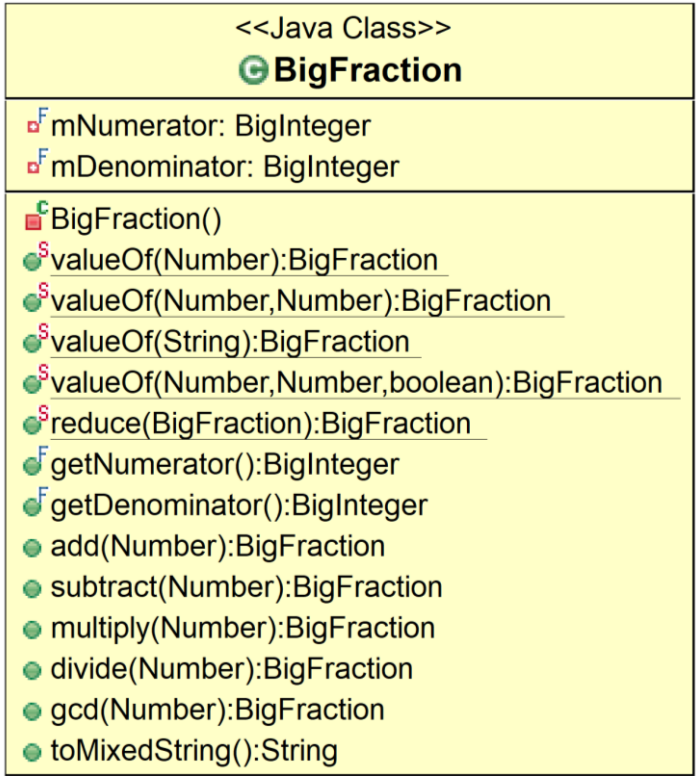
- Motivate the need for Java futures by understanding the pros and cons of synchrony and asynchrony
- Know how Java futures provide the foundation for completable futures in Java
- Understand how to multiply BigFraction objects concurrently via Java futures

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

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

Future<BigFraction> future =
   commonPool().submit(task);
...
BigFraction res = future.get();
```

Overview of BigFraction

Overview of BigFraction

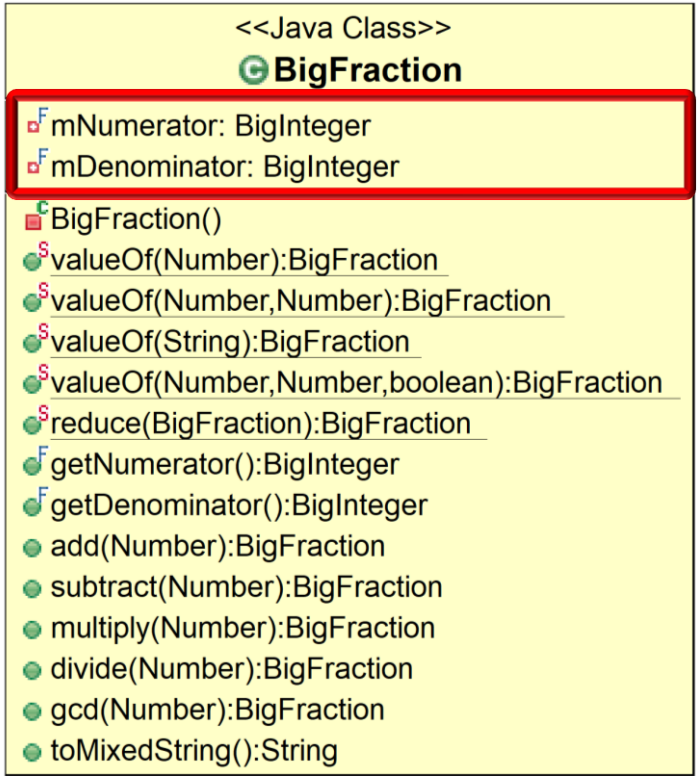
- We show how to apply Java futures in the context of a BigFraction class.



See LiveLessons/blob/master/Java8/ex8/src/utils/BigFraction.java

Overview of BigFraction

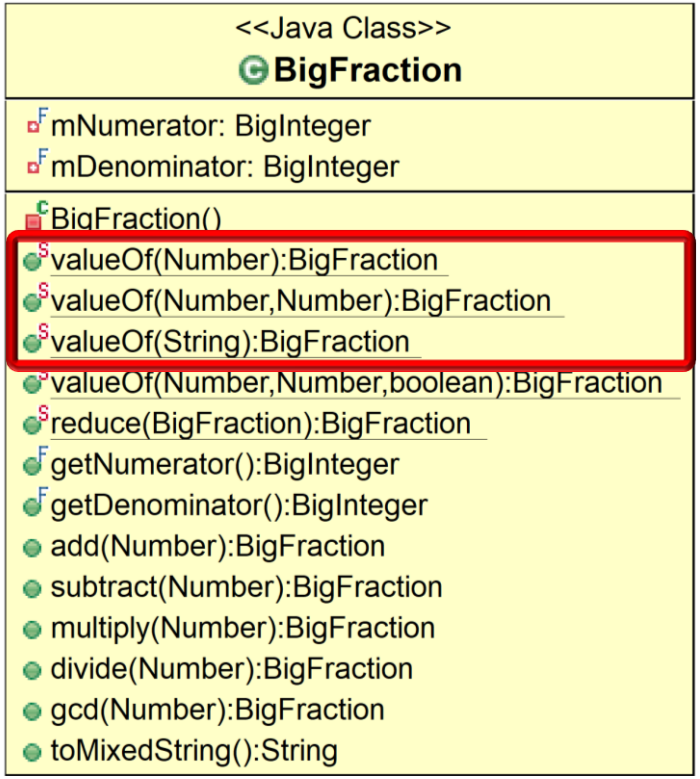
- We show how to apply Java futures in the context of a BigFraction class.
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator and denominator



See docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html

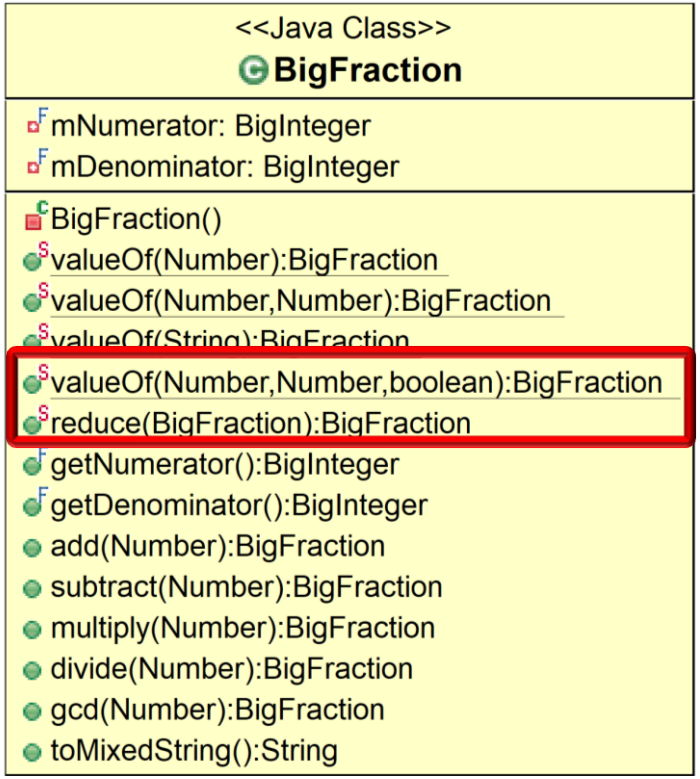
Overview of BigFraction

- We show how to apply Java futures in the context of a BigFraction class.
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator and denominator
  - Factory methods for creating "reduced" fractions
    - 44/55 → 4/5
    - 12/24 → 1/2
    - 144/216 → 2/3



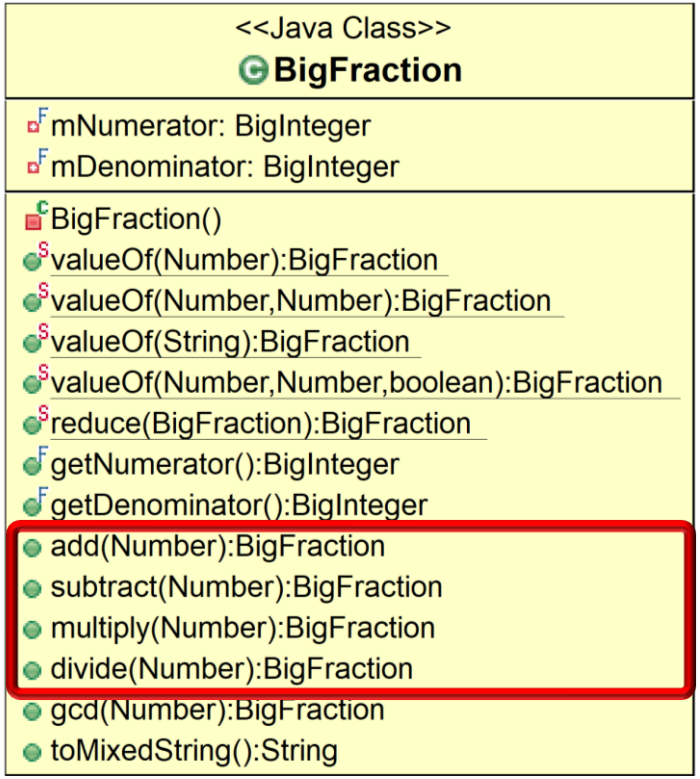
Overview of BigFraction

- We show how to apply Java futures in the context of a BigFraction class.
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator and denominator
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (and then reducing them)
    - 12/24 (→ 1/2)



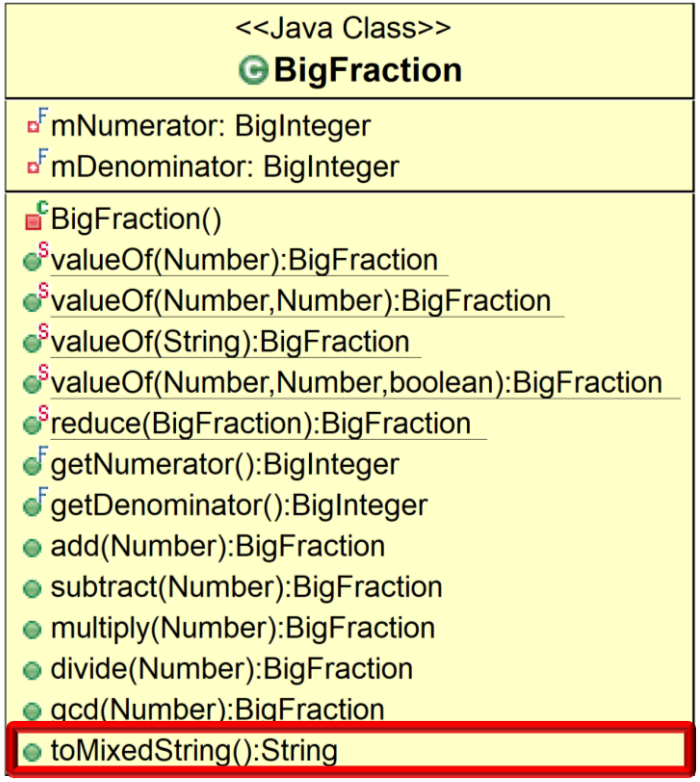
Overview of BigFraction

- We show how to apply Java futures in the context of a BigFraction class.
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator and denominator
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (and then reducing them)
  - Arbitrary-precision fraction arithmetic
    - 18/4 × 2/3 = 3



Overview of BigFraction

- We show how to apply Java futures in the context of a BigFraction class.
  - Arbitrary-precision fraction, utilizing BigIntegers for numerator and denominator
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (and then reducing them)
  - Arbitrary-precision fraction arithmetic
  - Create a mixed fraction from an improper fraction
    - 18/4 → 4 1/2



See www.mathsisfun.com/improper-fractions.html

Programming BigFraction Objects With Java Futures

Programming BigFraction Objects With Java Futures

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

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

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

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

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

Programming BigFraction Objects With Java Futures

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

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

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

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

Callable is a two-way task that returns a result via a single method with no arguments.

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

Programming BigFraction Objects With Java Futures

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

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

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

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

Java enables the initialization of a callable via a supplier lambda.

See lesson on "*Overview of Java Lambda Expressions and Method References*"

Programming BigFraction Objects With Java Futures

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

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

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

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

Can pass values to a callable via effectively final variables

See javarevisited.blogspot.com/2015/03/what-is-effectively-final-variable-of.html

Programming BigFraction Objects With Java Futures

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

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

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

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

Submit a two-way task to run in a thread pool (in this case, the common fork-join pool)

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

Programming BigFraction Objects With Java Futures

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

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

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

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

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

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

Programming BigFraction Objects With Java Futures

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

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

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

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

Other code can run here concurrently wrt the task running in the background.

Programming BigFraction Objects With Java Futures

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

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

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

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

get() blocks, if necessary, for the computation to complete and then retrieves its result.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get

Programming BigFraction Objects With Java Futures

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

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

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

Future<BigFraction> future =
  commonPool().submit(task);
...
BigFraction result =
  future.get(n, SECONDS);
```

get() can also perform polling and timed blocks.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get

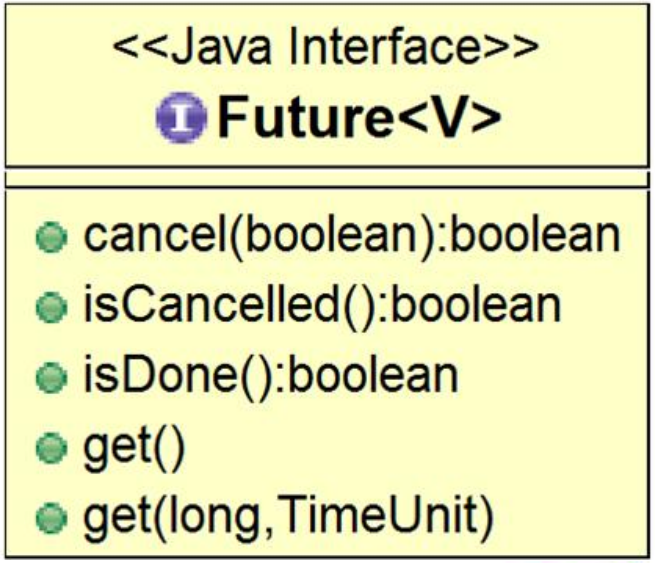
The End

Evaluating the Pros and Cons of Java Futures

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Motivate the need for Java futures by understanding the pros and cons of synchrony and asynchrony
- Know how Java futures provide the foundation for completable futures in Java
- Understand how to multiply BigFraction objects concurrently via Java futures
- Motivate the need for Java completable futures by evaluating the pros and cons with Java futures



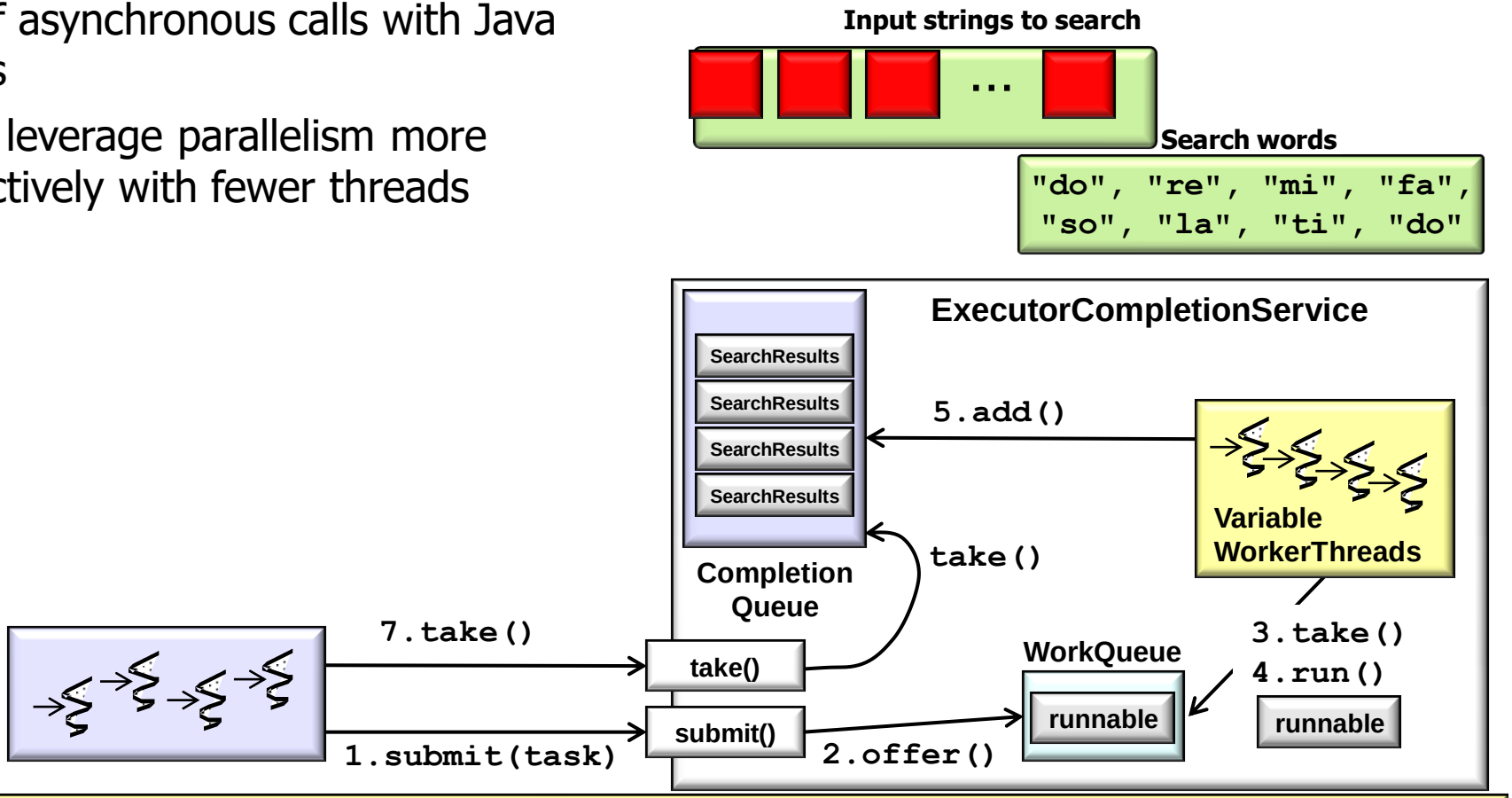
The Pros of Java Futures

The Pros of Java Futures

- Pros of asynchronous calls with Java futures

The Pros of Java Futures

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

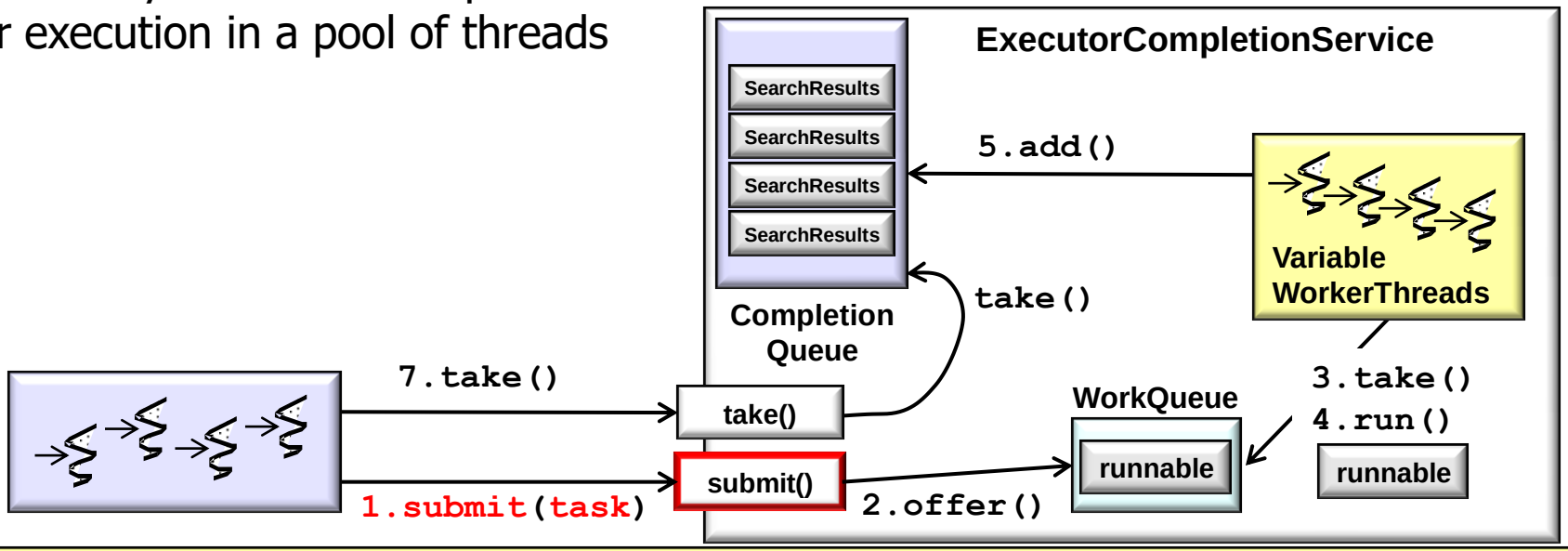


See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchTaskGang

The Pros of Java Futures

- Pros of asynchronous calls with Java futures
  - May leverage parallelism more effectively with fewer threads
    - Queue asynchronous computations for execution in a pool of threads

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

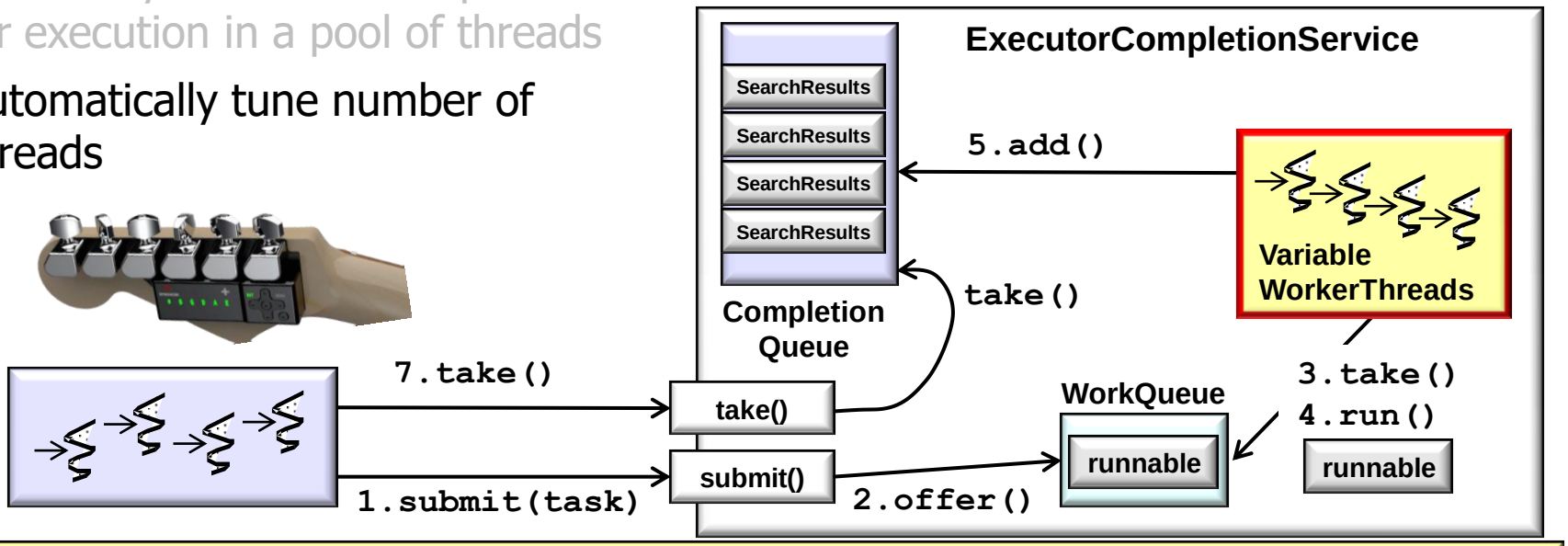


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#submit

The Pros of Java Futures

- Pros of asynchronous calls with Java futures
  - May leverage parallelism more effectively with fewer threads
    - Queue asynchronous computations for execution in a pool of threads
    - Automatically tune number of threads

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



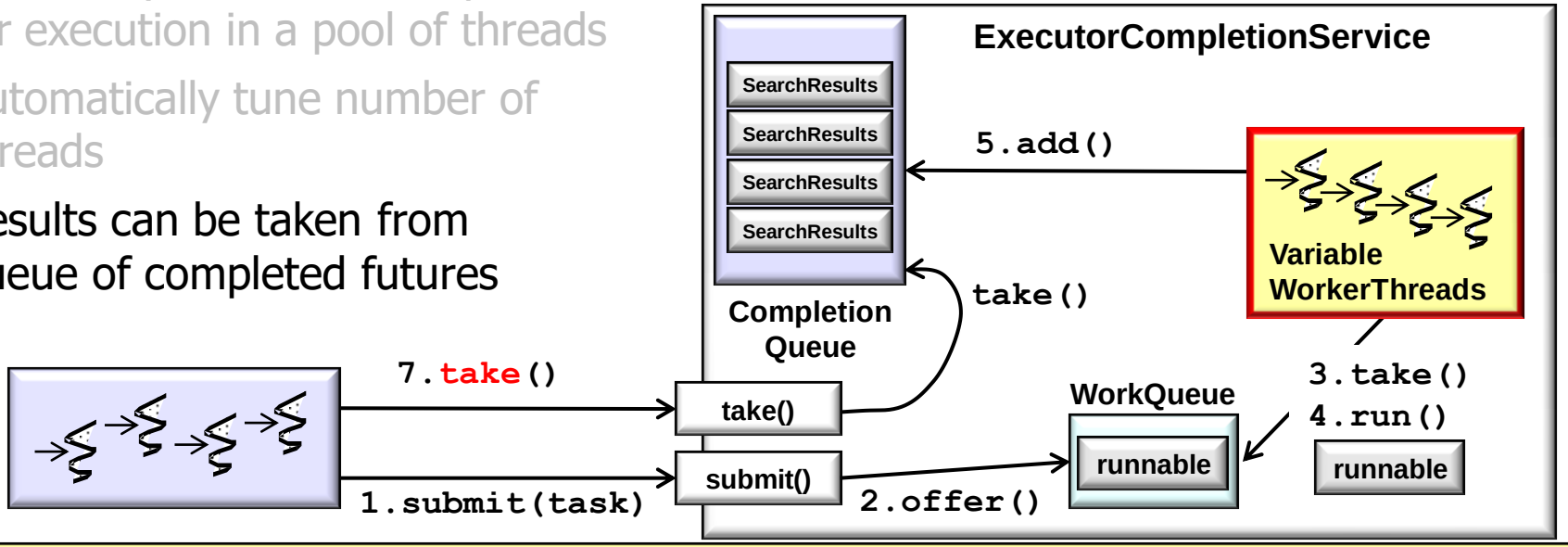
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool

The Pros of Java Futures

- Pros of asynchronous calls with Java futures
  - May leverage parallelism more effectively with fewer threads
    - Queue asynchronous computations for execution in a pool of threads
    - Automatically tune number of threads
  - Results can be taken from queue of completed futures

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

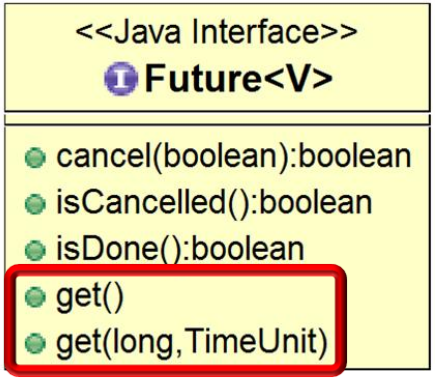
take() blocks, but get() doesn't



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#take

The Pros of Java Futures

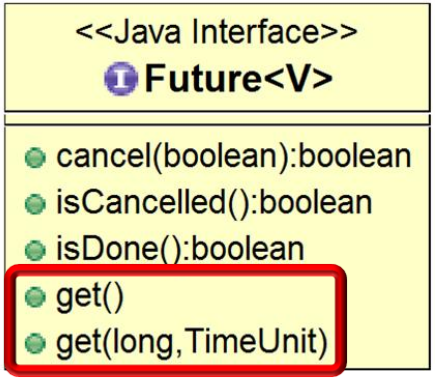
- Pros of asynchronous calls with Java futures
  - May leverage parallelism more effectively with fewer threads
  - Can block until the result of an asynchronous two-way task is available




```
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();
```

The Pros of Java Futures

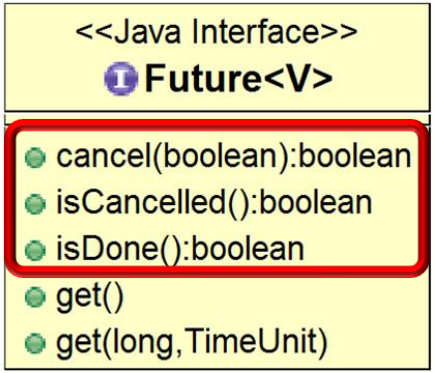
- Pros of asynchronous calls with Java futures
  - May leverage parallelism more effectively with fewer threads
  - Can block until the result of an asynchronous two-way task is available
    - Can also poll or time-block




```
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);
```

The Pros of Java Futures

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




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

Future<BigFraction> f =
   commonPool().submit(() -> {
     BigFraction bf1 =
       new BigFraction(f1);
     BigFraction bf2 =
       new BigFraction(f2);
     return bf1.multiply(bf2);
   });
...
if (!(f.isDone()
        || !f.isCancelled()))
   f.cancel();
```

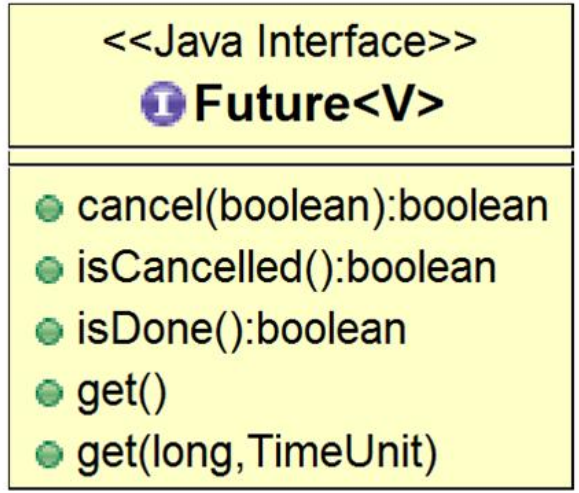
The Cons of Java Futures

The Cons of Java Futures

- Cons of asynchronous calls with Java futures

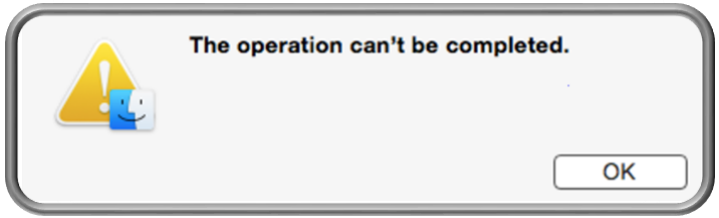
The Cons of Java Futures

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



The Cons of Java Futures

- Cons of asynchronous calls with Java futures
  - Limited feature set
    - *Cannot* be completed explicitly
      - Additional mechanisms like FutureTask are needed.



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

The Cons of Java Futures

- Cons of asynchronous calls with Java futures
  - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently
      - Trigger-dependent actions to handle results of asynchronous processing

See en.wikipedia.org/wiki/Fluent_interface

The Cons of Java Futures

- Cons of asynchronous calls with Java futures
  - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently
    - *Cannot* be triggered reactively
      - 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);
```

The Cons of Java Futures

- Cons of asynchronous calls with Java futures
  - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently
    - *Cannot* be triggered reactively
      - 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);
```

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

The Cons of Java Futures

- Cons of asynchronous calls with Java futures
  - Limited feature set
    - *Cannot* be completed explicitly
    - *Cannot* be chained fluently
    - *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 asynchronous computation finishes first

The Cons of Java Futures

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

In general, it's awkward and inefficient to "compose" multiple futures.

The Cons of Java Futures

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

Class CompletableFuture<T>

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

All Implemented Interfaces:

CompletionStage<T>, Future<T>

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

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

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

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

The End

Overcoming Limitations of Java Futures via Java Completable Futures

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Know how Java completable futures overcome limitations with Java futures

See earlier lesson on "*Overview of the Java Completable Futures Framework*"

Overcoming Limitations With Java Futures

Overcoming Limitations With Java Futures

- The completable future framework overcomes Java future limitations.

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

Overcoming Limitations With Java Futures

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


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

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

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

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

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

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 and 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.

See en.wikipedia.org/wiki/Fluent_interface

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 and cleanly
  - *Can* be triggered reactively/ efficiently as a *collection* of futures without 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

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex8

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 and cleanly
  - *Can* be triggered reactively/ efficiently as a *collection* of futures without undue overhead


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

Print out results after all asynchronous 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 and cleanly
  - *Can* be triggered reactively/ efficiently as a *collection* of futures without undue overhead

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

Java completable futures can also be combined with Java sequential streams.

Overcoming Limitations of
Java Futures via Java Completable Futures

The End

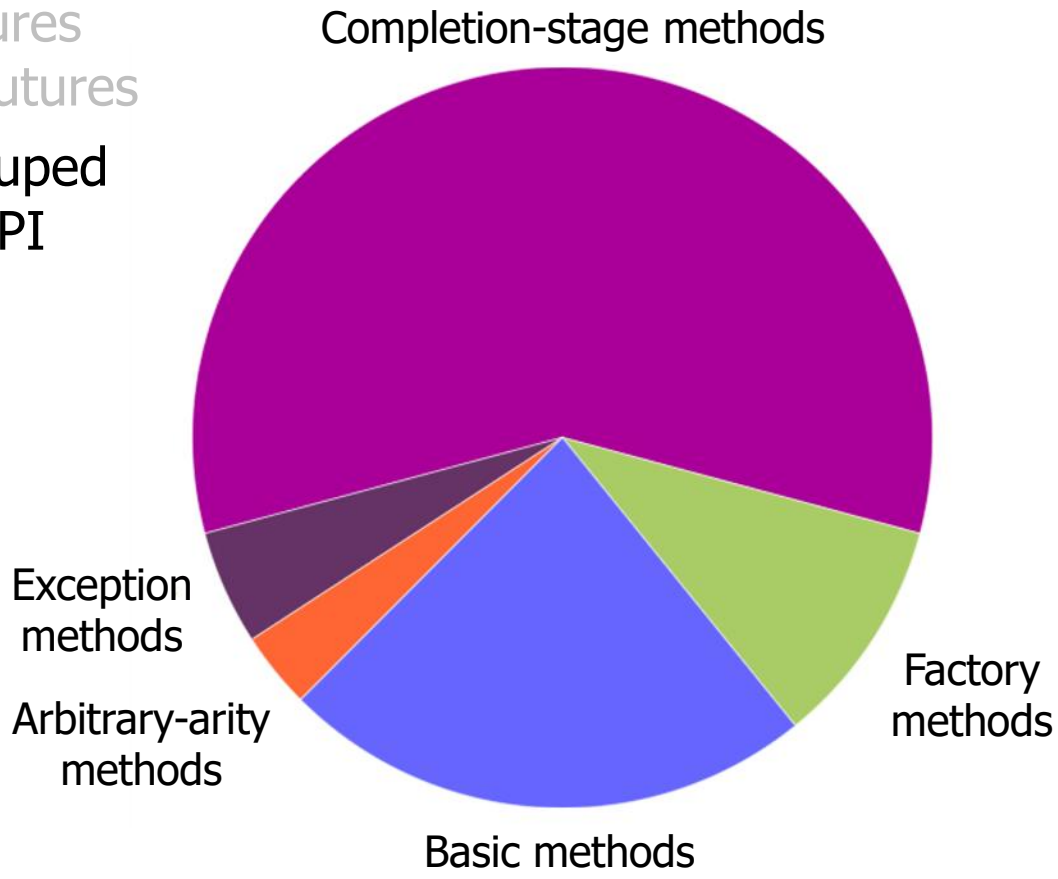
Java Completable Futures

API Overview

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

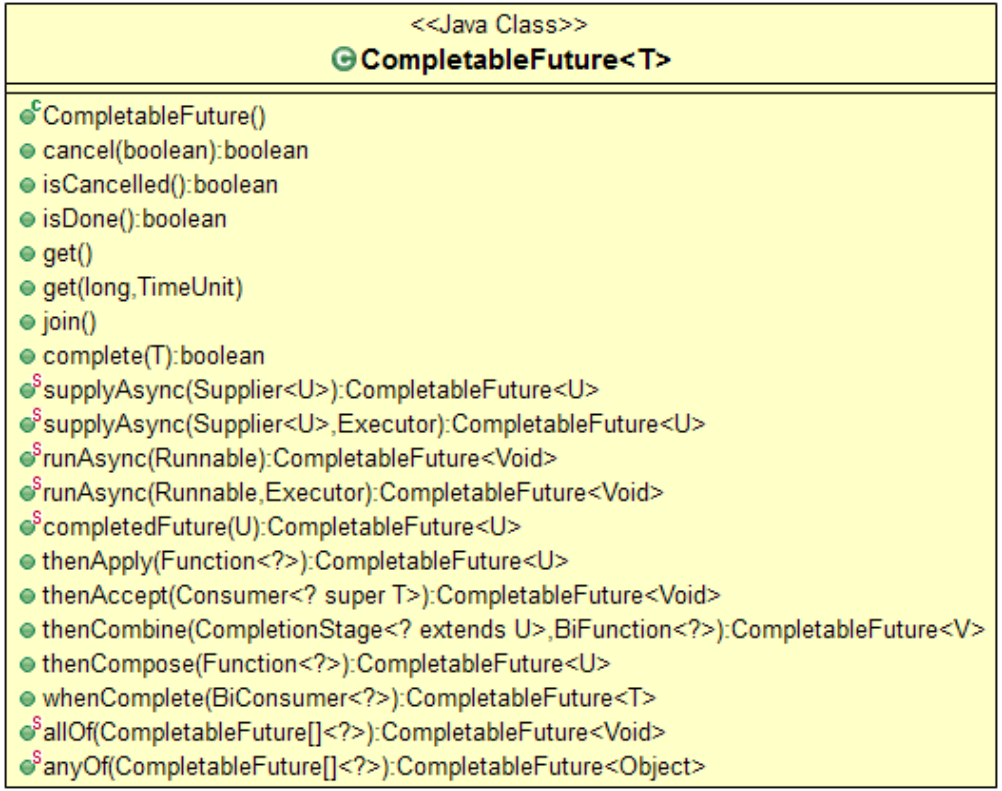
- Know how Java completable futures overcome limitations with Java futures
- Recognize how methods are grouped in the Java completable future API



Grouping the Java Completable Future API

Grouping the Java Completable Future API

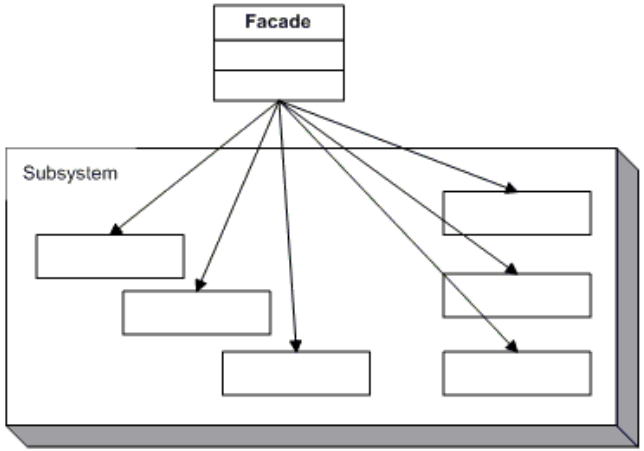
- The entire completable future framework resides in one public class with more than 60 methods!

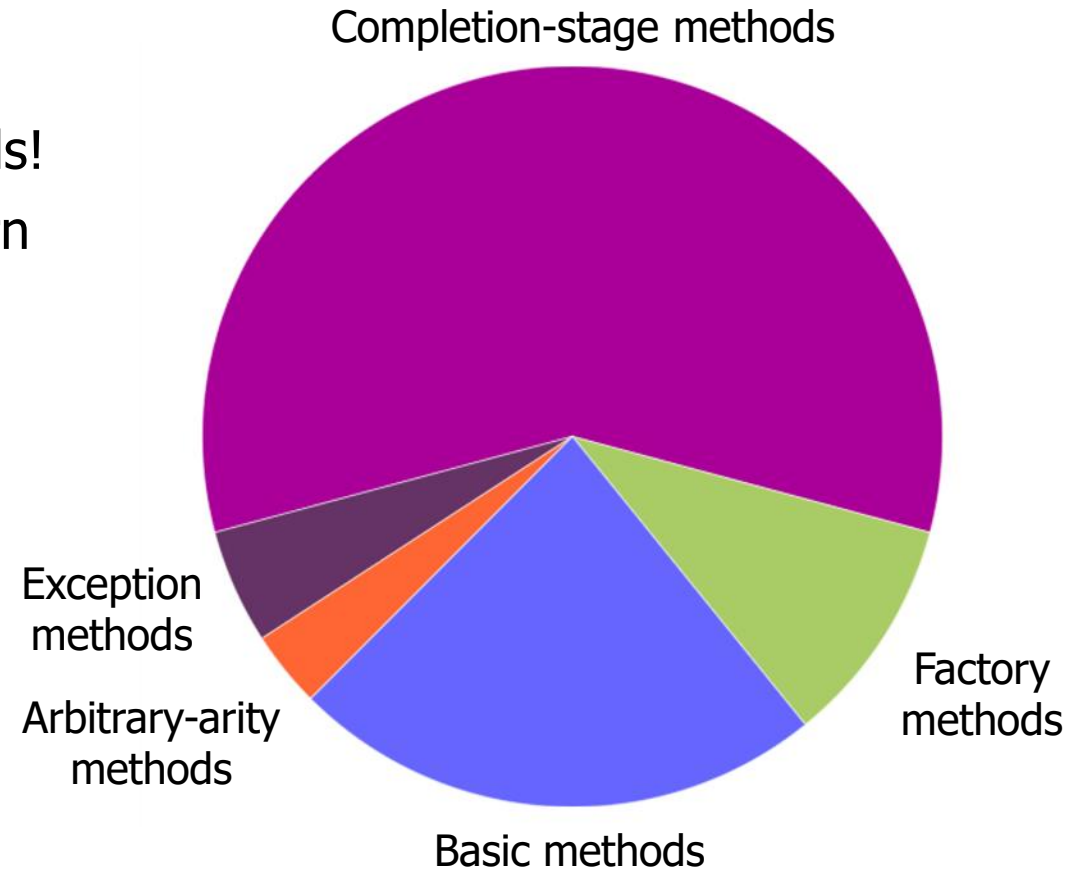


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

Grouping the Java Completable Future API

- The entire completable future framework resides in one public class with more than 60 methods!
  - Implements the Façade pattern

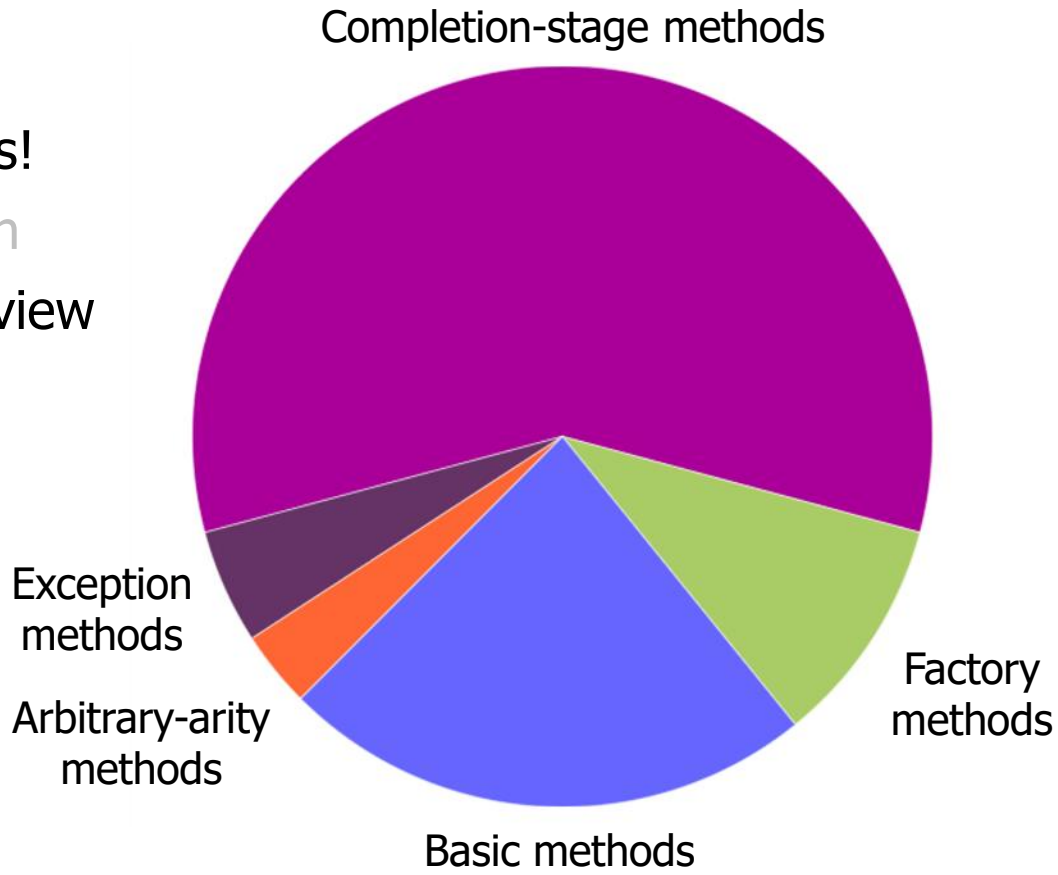




See en.wikipedia.org/wiki/Facade_pattern

Grouping the Java Completable Future API

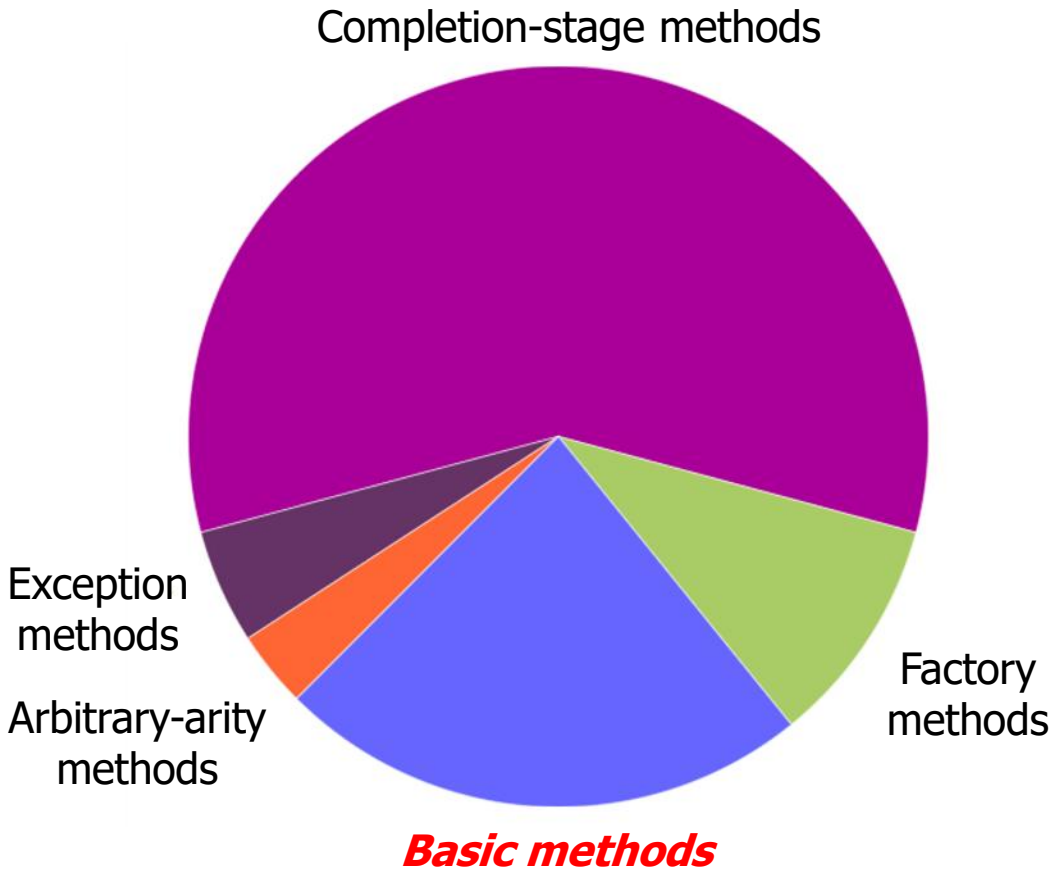
- The entire completable future framework resides in one public class with more than 60 methods!
  - Implements the Façade pattern
  - It helps to have a "birds-eye" view of this method-intensive class



See en.wikipedia.org/wiki/Earthrise

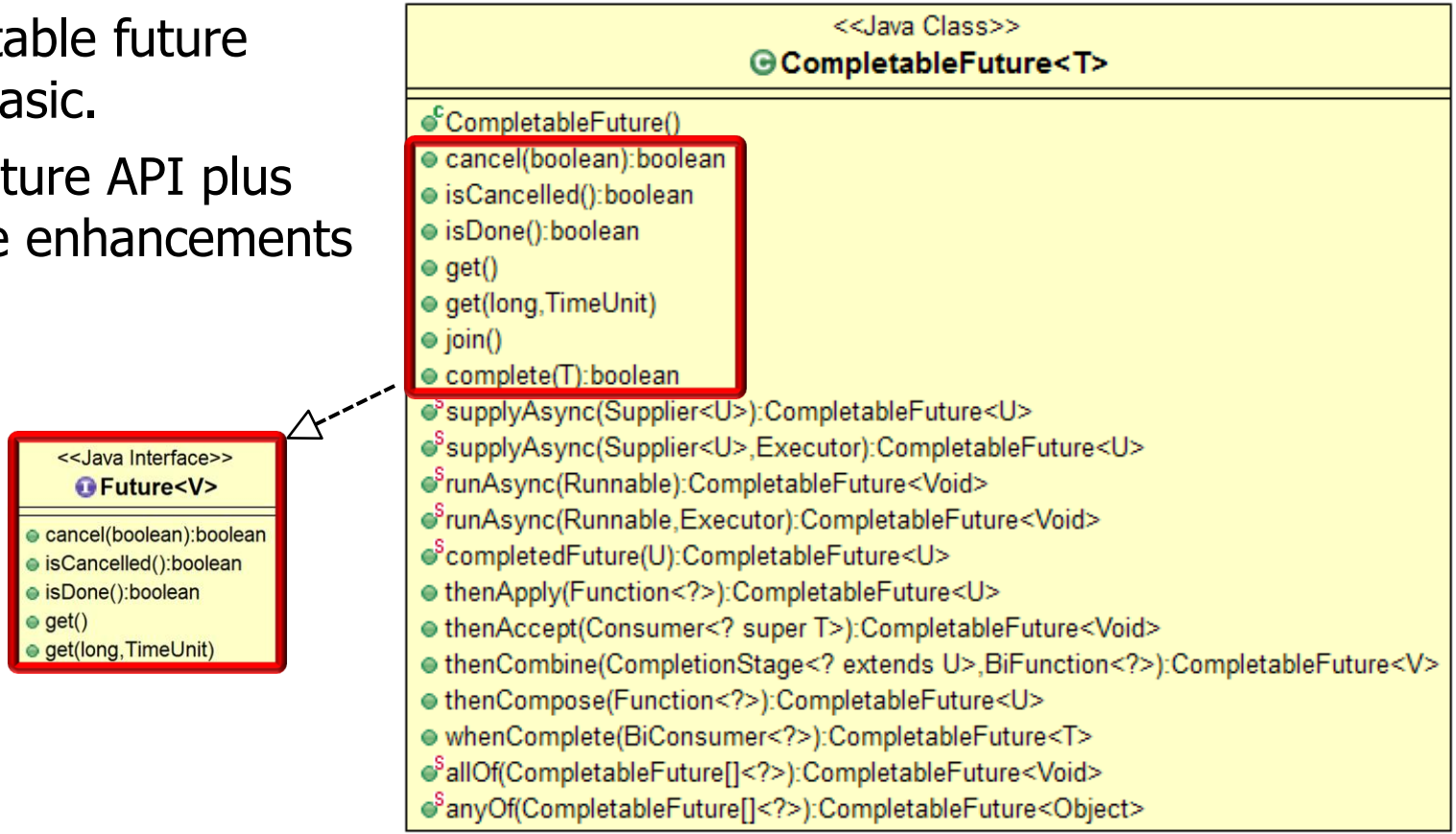
Grouping the Java Completable Future API

- Some completable future features are basic.



Grouping the Java Completable Future API

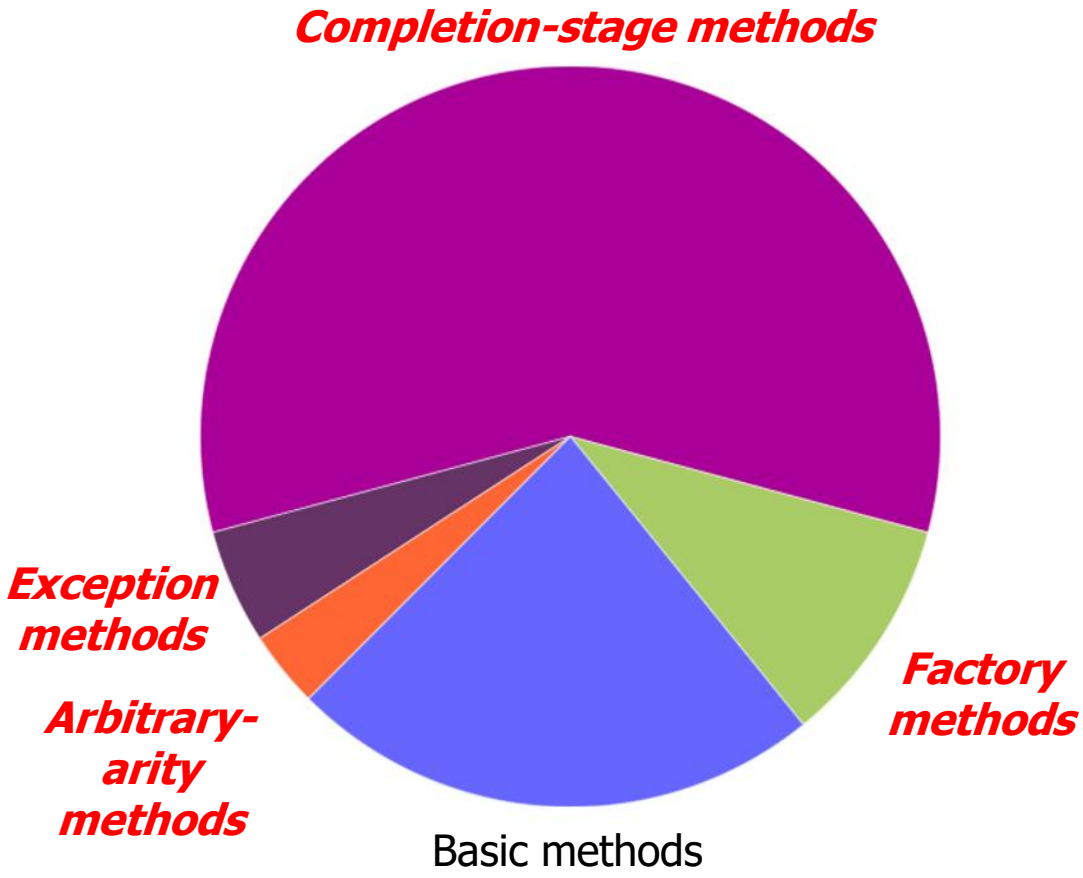
- Some completable future features are basic.
  - The Java Future API plus some simple enhancements



Only slightly better than the conventional Future interface

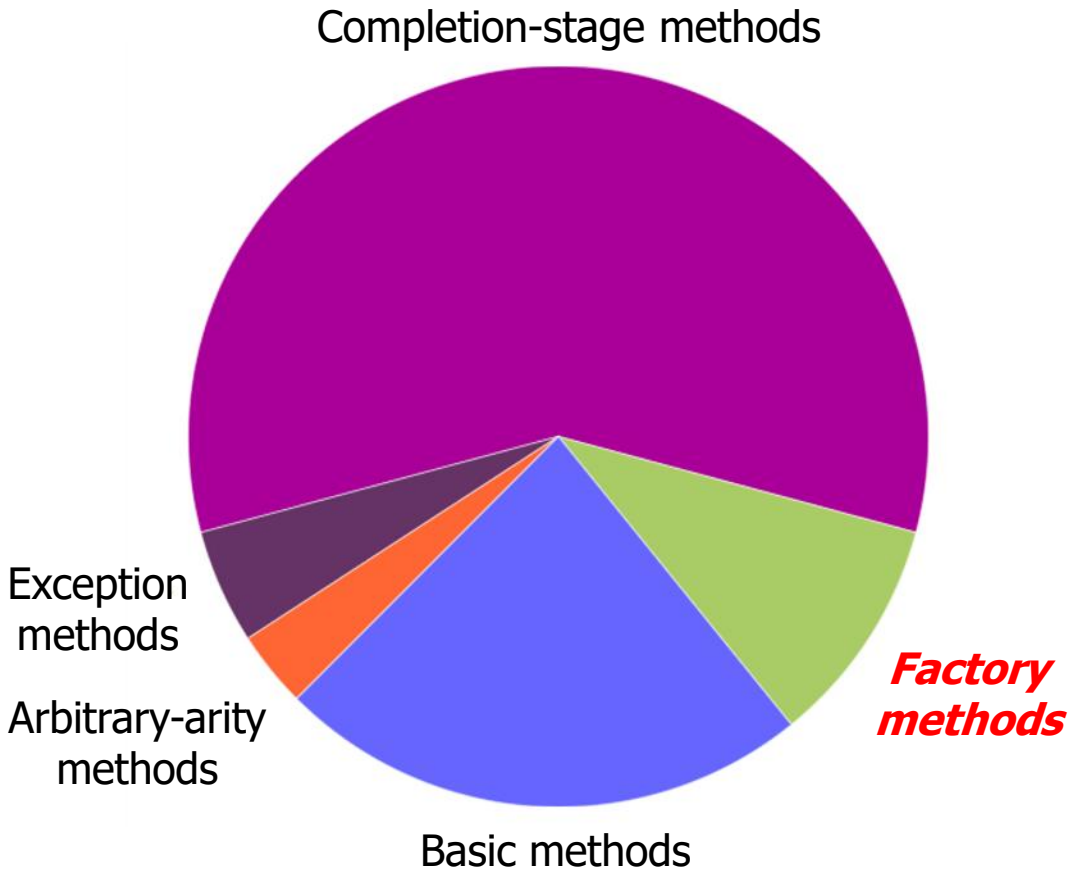
Grouping the Java Completable Future API

- Other completable future features are more advanced.



Grouping the Java Completable Future API

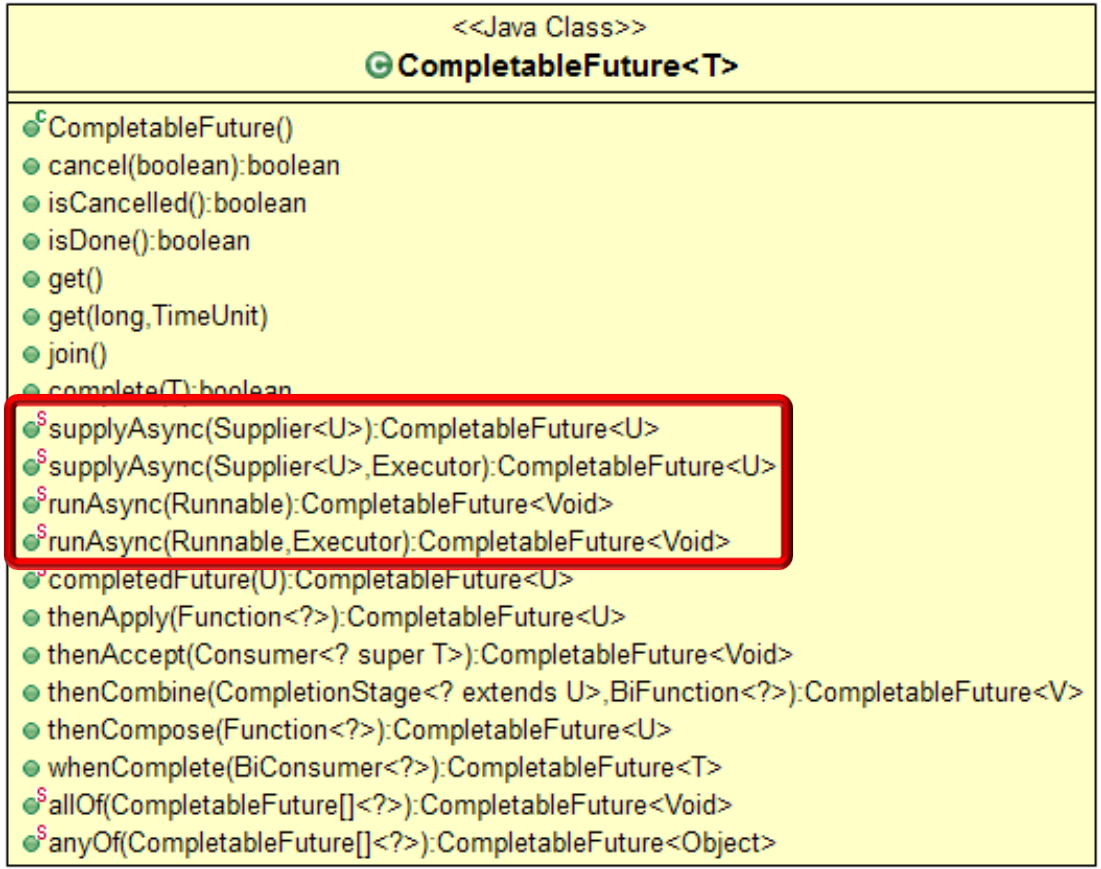
- Other completable future features are more advanced.
  - Factory methods



See en.wikipedia.org/wiki/Factory_method_pattern

Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
    - Initiate asynchronous two-way or one-way computations without using threads explicitly



Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
    - Initiate asynchronous two-way or one-way computations without using threads explicitly

<<Java Class>>

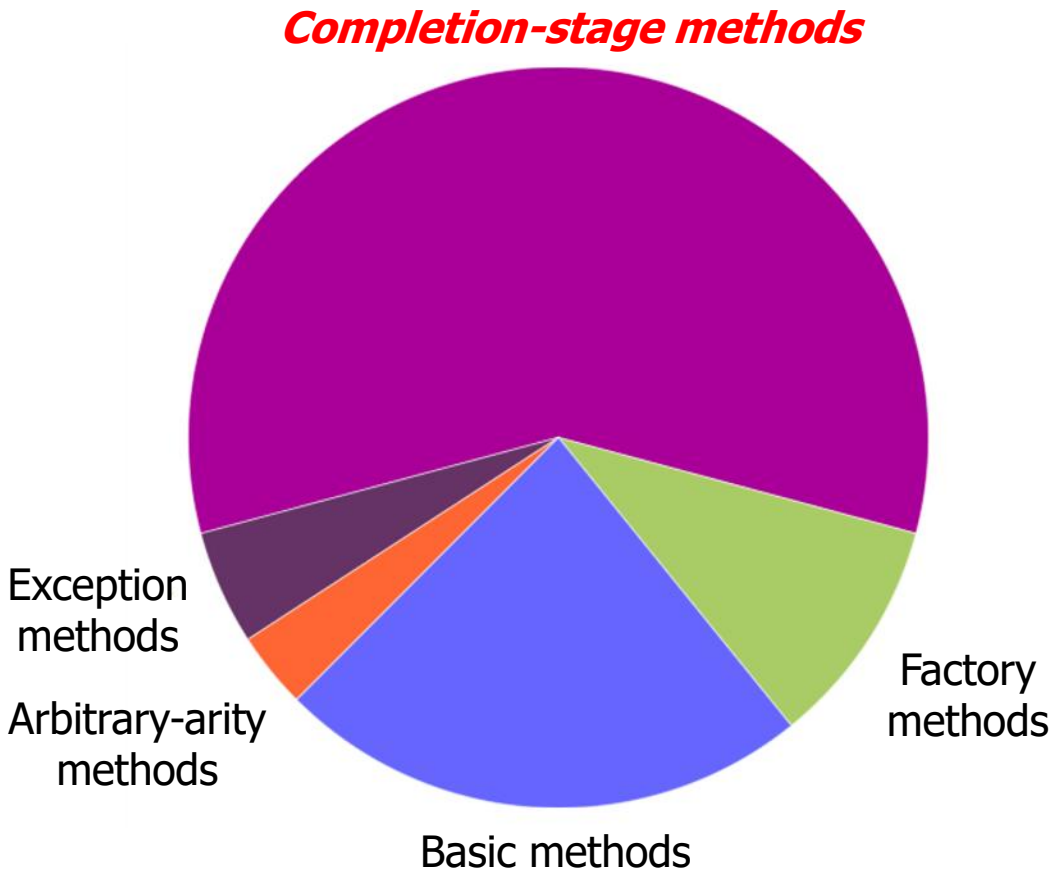
CompletableFuture<T>

- 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 *elastic* by leveraging a pool of worker threads

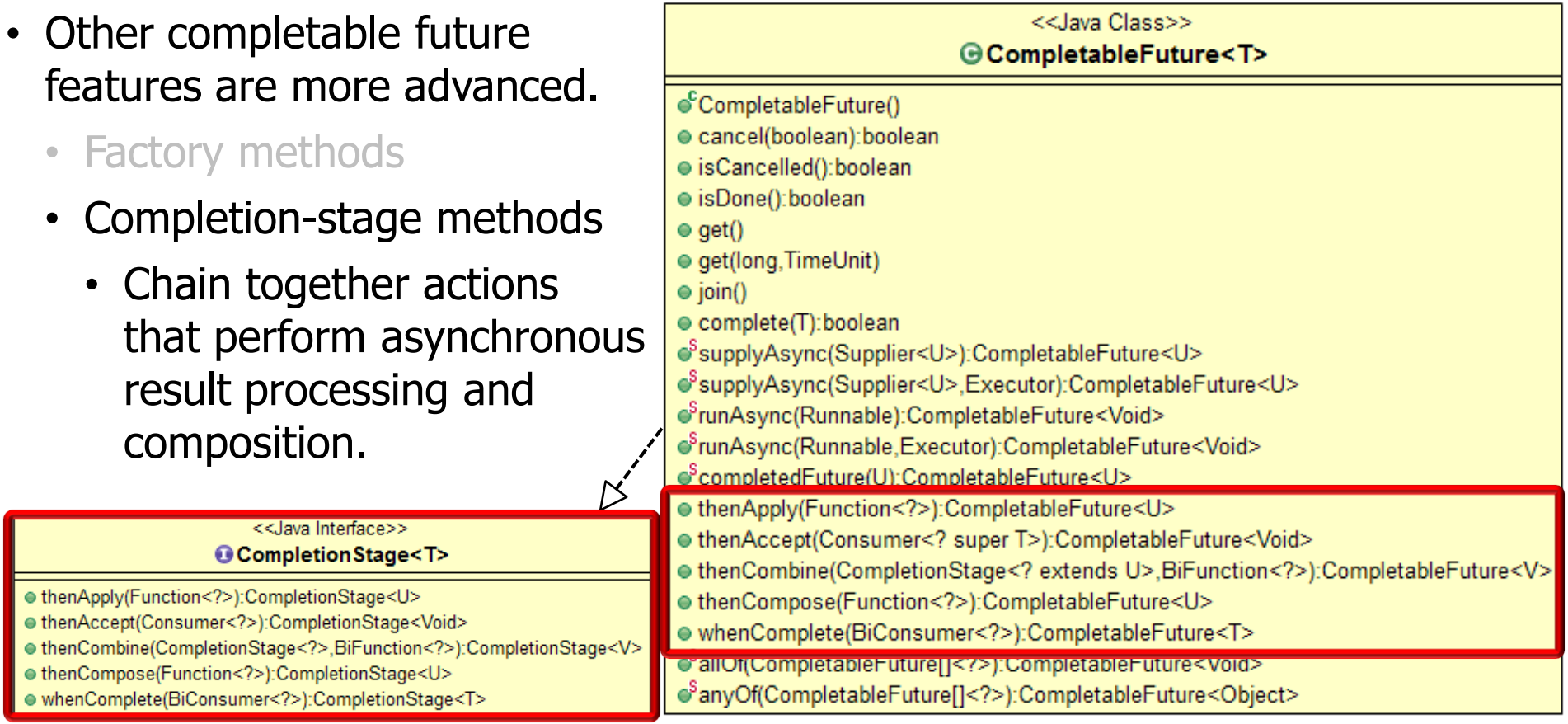
Grouping the Java Completable Future API

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



Grouping the Java Completable Future API

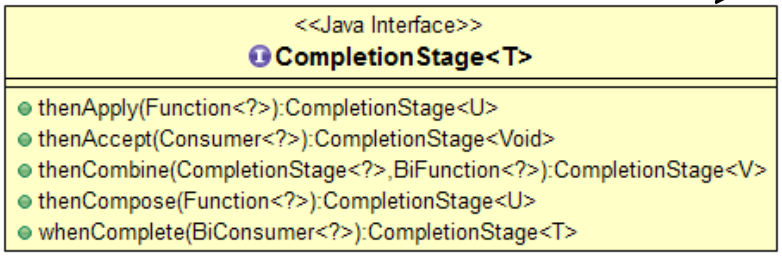
- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
    - Chain together actions that perform asynchronous result processing and composition.

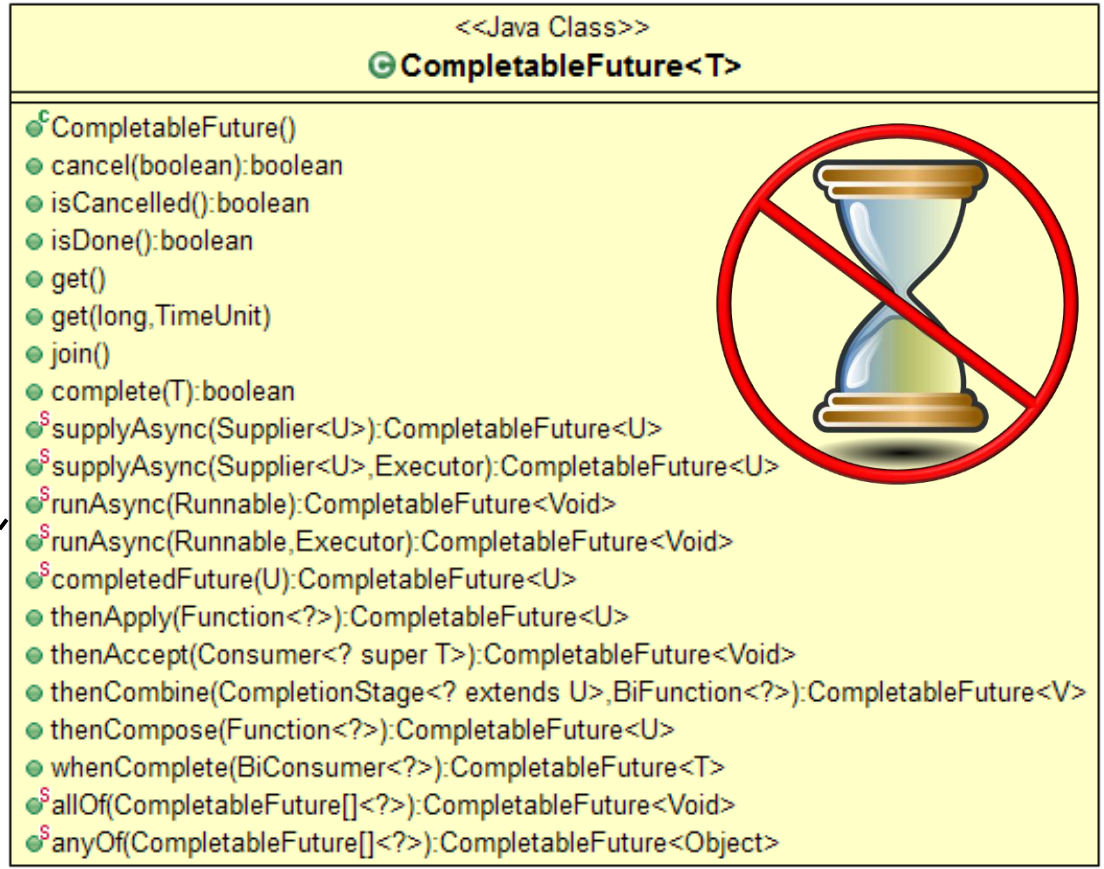


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html

Grouping the Java Completable Future API

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

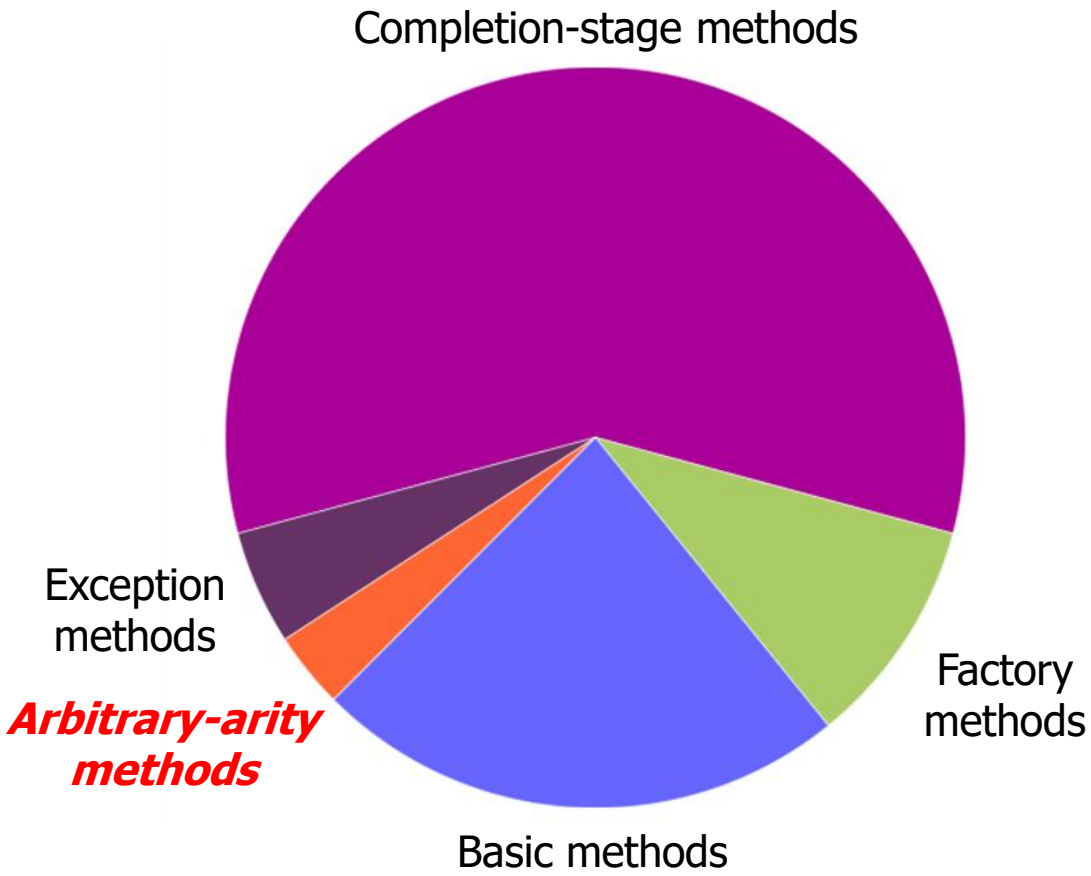




Help make programs more *responsive* by not blocking user code

Grouping the Java Completable Future API

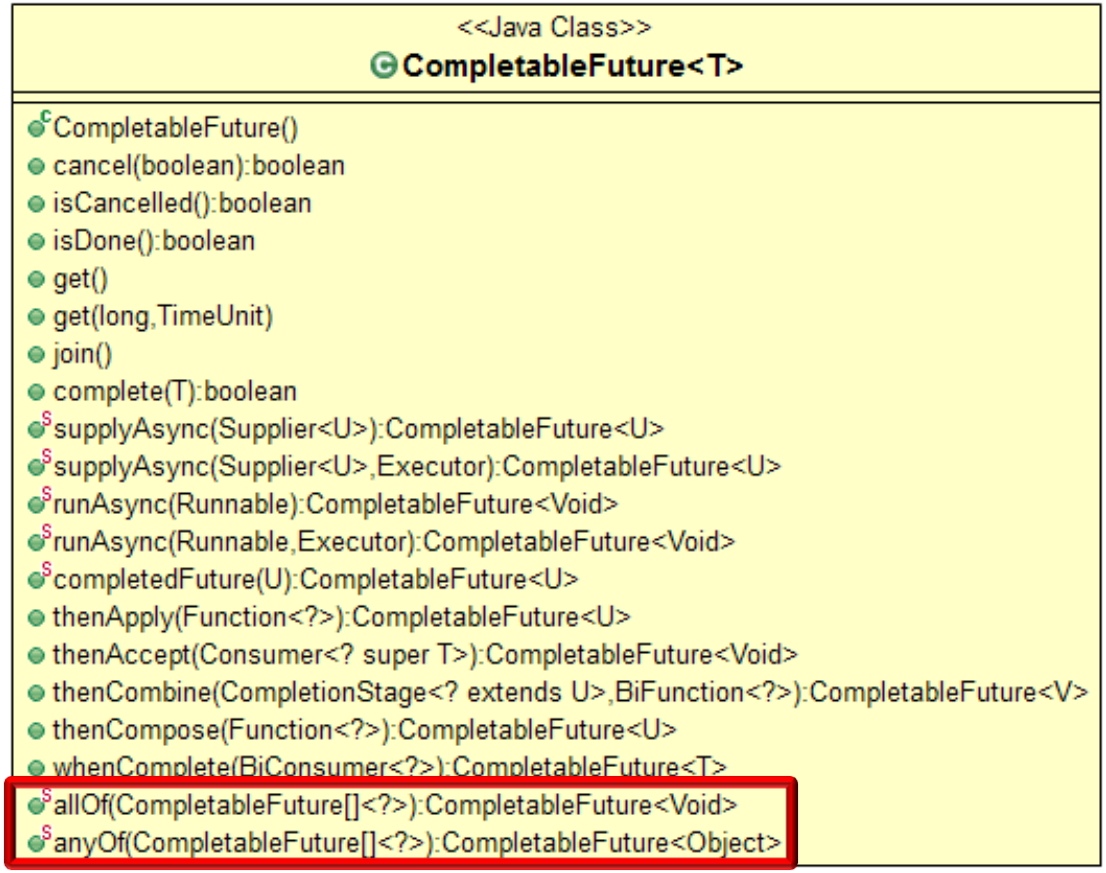
- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
  - "Arbitrary-arity" methods



See en.wikipedia.org/wiki/Arity

Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
  - "Arbitrary-arity" methods
    - Process futures in bulk by combining multiple futures into a single future



Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
  - "Arbitrary-arity" methods
    - Process futures in bulk by combining multiple futures into a single future

<<Java Class>>

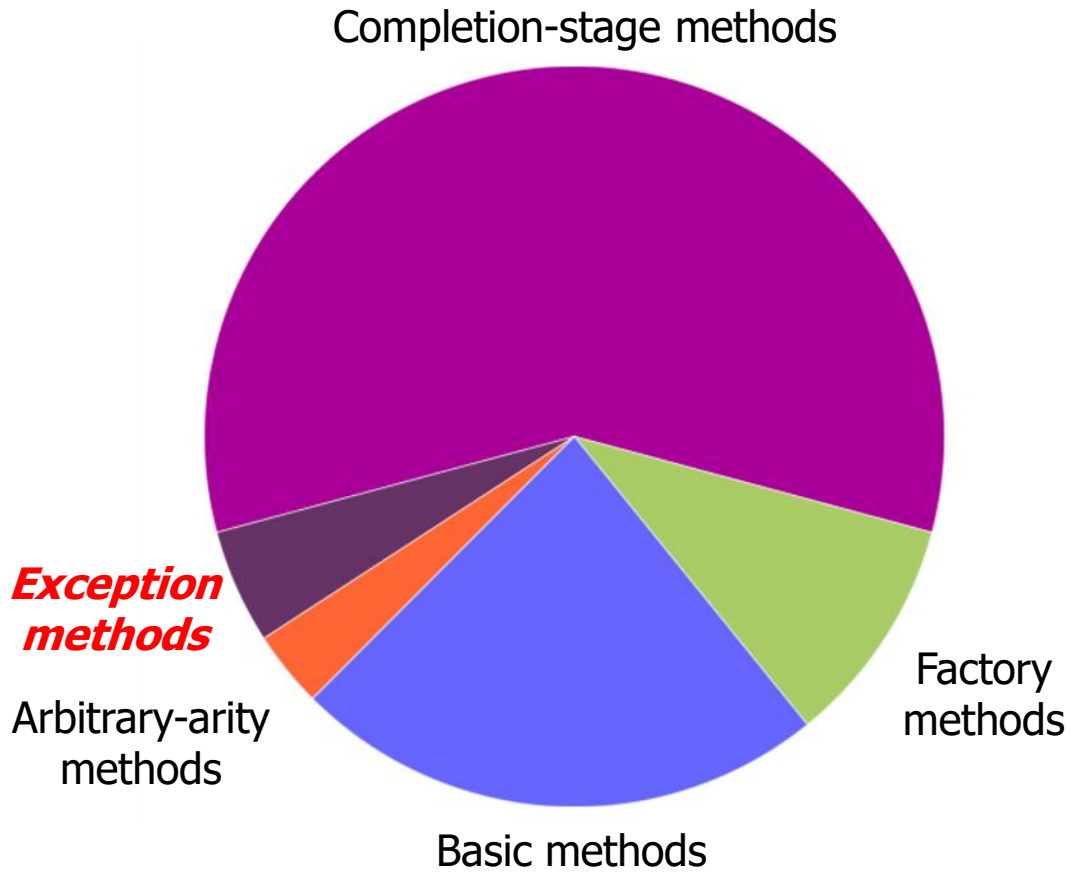
CompletableFuture<T>

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

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

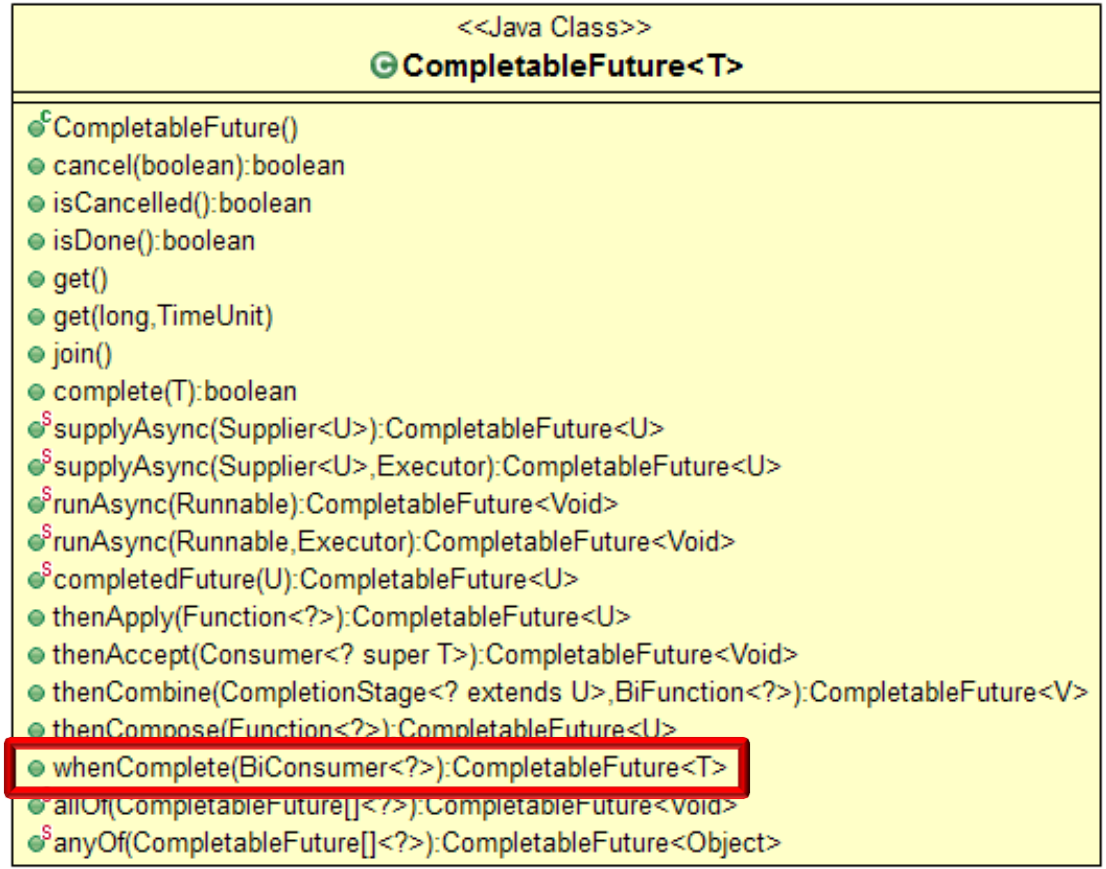
Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
  - "Arbitrary-arity" methods
  - Exception methods



Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
  - "Arbitrary-arity" methods
  - Exception methods
    - Handle exceptional conditions at runtime



Grouping the Java Completable Future API

- Other completable future features are more advanced.
  - Factory methods
  - Completion-stage methods
  - "Arbitrary-arity" methods
  - 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

Grouping the Java Completable Future API

- All methods are implemented internally via message passing that's ultimately connected to Java-based thread pools.

<<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>

Ensure loose coupling, isolation, and location transparency between components

The End