

# Virtual Technology Summit

**Hands-On Learning With Oracle and Community Experts**

Where Technology and Community Meet



## Asynchronous programming in Java 8: how to use CompletableFuture

José Paumard

PhD, Java Champion, Java Rock Star

Assistant Professor (Paris), independent contractor

Feb 2016

**Please Stand By. This session will begin promptly at the time indicated on the agenda. Thank You.**

## Safe Harbor Statement

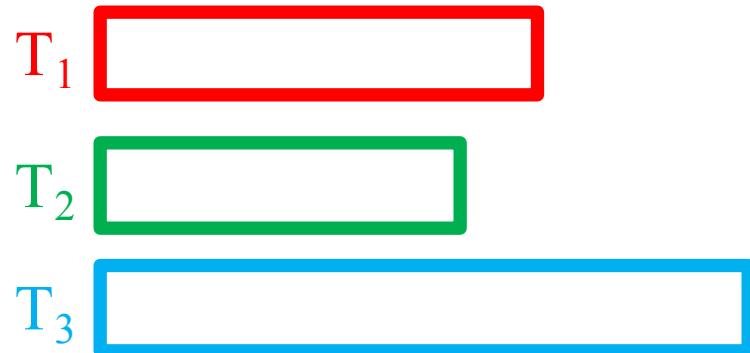
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Asynchronous programming in Java 8: how to use CompletableFuture



# Asynchronous

Suppose we have three tasks to execute



# Asynchronous

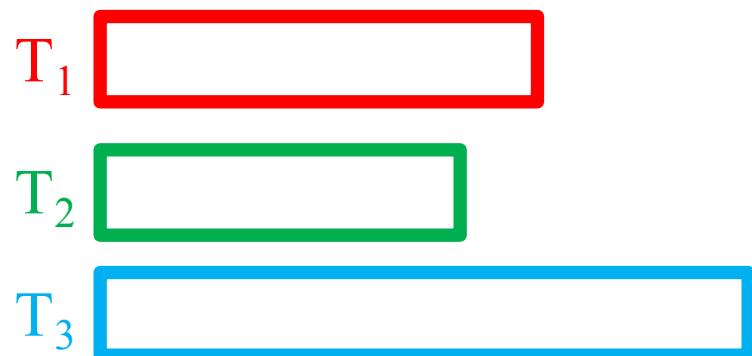
Suppose we have three tasks to execute



“synchronous execution”

# Asynchronous

2<sup>nd</sup> way to do it:



“multithreaded execution”

# Asynchronous

2<sup>nd</sup> way to do it:



“multithreaded execution” on only one core

# Asynchronous

3<sup>rd</sup> way to do it:



Asynchronous execution...

# Asynchronous

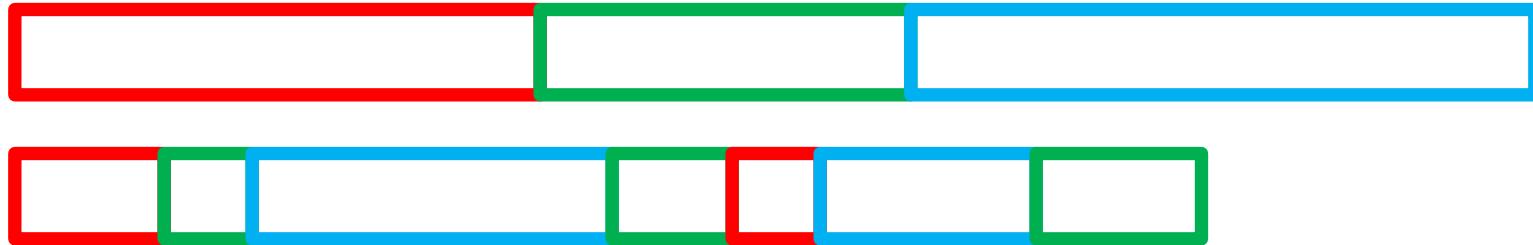
3<sup>rd</sup> way to do it:



Asynchronous execution... even on a multicore

# Asynchronous

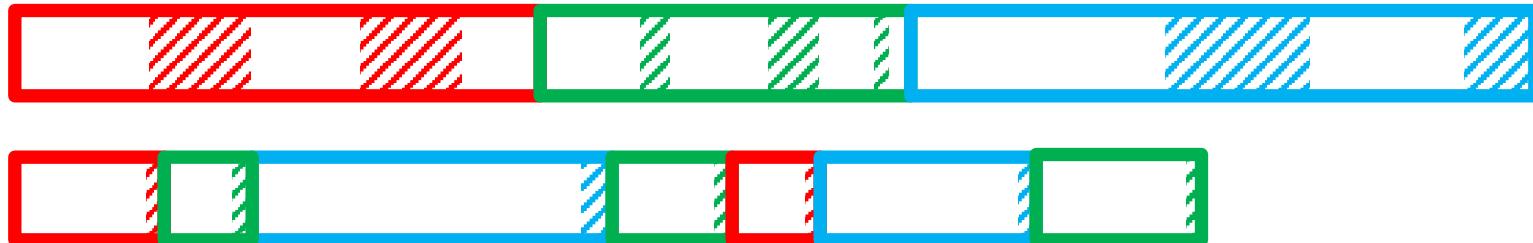
Synchronous vs asynchronous:



Is asynchronous any faster?

# Asynchronous

Synchronous vs asynchronous:



Is asynchronous any faster?

Well it can be

Because it is “non blocking”

# Asynchronous

Difference with the synchronous multithreaded model?

- 1) The async engine decides to switch from one context to another
- 2) Single threaded = no issue with atomicity or visibility

Performances?

No multithreaded “context switch”

# Asynchronous Pattern

```
queryEngine.select("select user from User")
    .foreach(user -> System.out.println(user)) ;
```

# Asynchronous Pattern

```
queryEngine.select("select user from User")
    .forEach(user -> System.out.println(user)) ;
```

Callback or task: lambda expression

# Asynchronous Pattern

```
queryEngine.select("select user from User")
    .forEach(user -> System.out.println(user)) ;
```

Callback or task: lambda expression

When the result is available, then we can continue with the next task

# Asynchronous Pattern

```
queryEngine.select("select user from User")
    .forEach(user -> System.out.println(user)) ;
```

How can we write that in Java?

# A task in Java

Since Java 1: Runnable

Since Java 5: Callable

In Java 5 we have the ExecutorService (pool of threads)

We give a task and get back a Future

# A task in Java Pattern

```
Callable<String> task = () -> "select user from User" ;  
Future<String> future = executorService.submit(task) ;
```

# A task in Java Pattern

```
Callable<String> task = () -> "select user from User" ;  
Future<String> future = executorService.submit(task) ;
```

```
List<User> users = future.get() ; // blocking  
users.forEach(System.out::println) ;
```

# A task in Java Pattern

```
Callable<String> task = () -> "select user from User" ;  
Future<String> future = executorService.submit(task) ;
```

```
List<User> users = future.get() ; // blocking  
users.forEach(System.out::println) ;
```

Passing an object from one task to another has to be handled in the “master” thread

# Asynchronous programming

We have new tools in Java 8 to handle this precise case

It brings new solutions to chain tasks

And can handle both asynchronous and multithreaded programming

# Creation of an asynchronous task

Let us see an example first

# Creation of an asynchronous task

The Jersey way to create an asynchronous call

```
@Path("/resource")
public class AsyncResource {
    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        new Thread(new Runnable() {
            public void run() {
                String result = longOperation();
                asyncResponse.resume(result);
            }
        }).start();
    }
}
```

# Creation of an asynchronous task

(let us fix this code, this is Java 8)

```
@Path("/resource")
public class AsyncResource {

    @Inject private Executor executor;

    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        executor.execute(() -> {
            String result = longOperation();
            asyncResponse.resume(result);
        });
    }
}
```

# How can we test this code?

The question is: how can we test that code?

We want to check if the result object  
is passed to the resume( ) method of the asyncResponse

# How can we test this code?

We have mocks for that!

It is a very basic test, but tricky to write since we are in an asynchronous world

# How can we test this code?

Let us give one more look at the code

```
@Path("/resource")
public class AsyncResource {
    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        executor.execute(() -> { // executed in the main thread
            String result = longOperation(); // executed in another thread
            asyncResponse.resume(result);
        });
    }
}
```

# How can we test this code?

We have mocks to check if resume() is properly called with the result object

# How can we test this code?

We can inject a mock AsyncResponse, even mock the result

```
@Path("/resource")
public class AsyncResource {
    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        executor.execute(() -> {
            String result = longOperation();
            asyncResponse.resume(result);
        });
    }
}
```

# How can we test this code?

We can inject a mock AsyncResponse, even mock the result

Then verify the correct interaction:

```
Mockito.verify(mockAsyncResponse).resume(result);
```

But:

- we need to verify this once the run() method has been called

# How can we test this code?

We can inject a mock AsyncResponse, even mock the result

Then verify the correct interaction:

```
Mockito.verify(mockAsyncResponse).resume(result);
```

But:

- we need to verify this once the run() method has been called
- and take into account the multithreaded aspect... the read / writes on the mock should be “visible”

# How can we test this code?

So our constraints are the following:

- we need to verify this once the run() method has been called
- we need to read / write on our mocks in the same thread as the one which runs the task we want to test

# How can we test this code?

This is where CompletionStage comes to the rescue!

```
@Path("/resource")
public class AsyncResource {

    @Inject ExecutorService executor;

    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        executor.submit(() -> {
            String result = longOperation();
            asyncResponse.resume(result);
        });
    }
}
```

# How can we test this code?

This pattern:

```
executor.submit(() -> {
    String result = longOperation();
    asyncResponse.resume(result);
});
```

# How can we test this code?

This pattern:

```
executor.submit(() -> {
    String result = longOperation();
    asyncResponse.resume(result);
});
```

Becomes this one: (and does basically the same thing)

```
CompletableFuture.runAsync(() -> {
    String result = longOperation();
    asyncResponse.resume(result);
}, executor);
```

# How can we test this code?

And the nice thing is:

```
CompletableFuture<Void> completableFuture =
CompletableFuture.runAsync(() -> {
    String result = longOperation();
    asyncResponse.resume(result);
}, executor);
```

# How can we test this code?

And the nice thing is:

```
CompletableFuture<Void> completableFuture =
CompletableFuture.runAsync(() -> {
    String result = longOperation();
    asyncResponse.resume(result);
}, executor);
```

And on this object, we can call:

```
completableFuture
    .thenRun(() -> {
        Mockito.verify(mockAsyncResponse).resume(result);
    }
);
```

# How can we test this code?

Because of visibility issues

- 1) It's simpler to run everything in the same thread
- 2) Create, train and check our mocks in this thread

# CompletionStage / CompletableFuture

Two elements in this API:

- an interface: CompletionStage
- an implementing class: CompletableFuture

The interface depends on CompletableFuture:

```
public CompletableFuture<T> toCompletableFuture();
```

# What is a CompletionStage?

A model for a task:

- that performs an action and may return a value when another completion stage completes
- that may trigger other tasks

So a completion stage is an element of a chain

# What is a CompletableFuture?

A class that implements both Future and CompletionStage

# What is a CompletableFuture?

A class that implements both Future and CompletionStage

It has a state:

- the task may be running
- the task may have complete normally
- the task may have complete exceptionnaly

# Methods from Future

Five methods from Future:

```
boolean cancel(boolean mayInterruptIfRunning) ;
```

# Methods from Future

Five methods from Future :

```
boolean cancel(boolean mayInterruptIfRunning) ;
```

```
boolean isCanceled() ;  
boolean isDone() ;
```

# Methods from Future

Five methods from Future :

```
boolean cancel(boolean mayInterruptIfRunning) ;
```

```
boolean isCanceled() ;  
boolean isDone() ;
```

```
V get() ;                                // blocking call  
V get(long timeout, TimeUnit timeUnit) ; // may throw a checked exception  
                                         throws InterruptedException, ExecutionException, TimeoutException ;
```

# More from CompletableFuture

Future-like methods:

```
V join() ; // may throw an unchecked exception  
V getNow(V valueIfAbsent) ; // returns immediately
```

# More from CompletableFuture

Future-like methods:

```
V join() ; // may throw an unchecked exception  
V getNow(V valueIfAbsent) ; // returns immediately
```

```
boolean complete(V value) ; // sets the returned value if not returned  
void obtrudeValue(V value) ; // resets the returned value
```

# More from CompletableFuture

Future-like methods:

```
V join() ; // may throw an unchecked exception  
V getNow(V valueIfAbsent) ; // returns immediately
```

```
boolean complete(V value) ; // sets the returned value if not returned  
void obtrudeValue(V value) ; // resets the returned value
```

```
boolean completeExceptionally(Throwable t) ; // sets an exception  
void obtrudeException(Throwable t) ; // resets with an exception
```

# How to create a CompletableFuture?

A completed CompletableFuture

```
public static <U> CompletableFuture<U> completedFuture(U value) ;
```

# How to create a CompletableFuture?

A CompletableFuture from a Runnable or a Supplier

```
public static CompletableFuture<Void>
    runAsync(Runnable runnable, Executor executor) ;

public static <U> CompletableFuture<U>
    supplyAsync(Supplier<U> value, Executor executor) ;
```

# Building CompletionStage chains

A CompletionStage is a step in a chain

- it can be triggered by a previous CompletionStage
- it can trigger another CompletionStage
- it can be executed in a given Executor

# Building CompletionStage chains

What is a task?

- it can be a Function
- it can be a Consumer
- it can be a Runnable

# Building CompletionStage chains

What kind of operation does it support?

- chaining (1 – 1)
- composing (1 – 1)
- combining, waiting for both result (2 – 1)
- combining, triggered on the first available result (2 – 1)

# Building CompletionStage chains

In what thread can it be executed?

- In the same executor as the caller
- In a new executor, passed as a parameter
- Asynchronously, ie in the common fork join pool

All this gives 36 methods!

# CompletionStage – patterns

## Some 1 – 1 patterns

```
public <U> CompletionStage<U>
    thenApply(Function<? super T,? extends U> fn);
```

# CompletionStage – patterns

## Some 1 – 1 patterns

```
public <U> CompletionStage<U>
    thenApply(Function<? super T,? extends U> fn);
```

```
public CompletionStage<Void>
    thenRunAsync(Runnable action, Executor executor);
```

# CompletionStage – patterns

## Some 1 – 1 patterns

```
public <U> CompletionStage<U>
    thenApply(Function<? super T, ? extends U> fn);
```

```
public CompletionStage<Void>
    thenRunAsync(Runnable action, Executor executor);
```

```
public CompletionStage<Void>
    thenComposeAsync(
        Function<? super T, ? extends CompletionStage<U>> fn);
```

# CompletionStage – patterns

## Some 2 – 1 patterns

```
public <U, V> CompletionStage<V> thenCombineAsync  
(CompletionStage<U> other,  
 BiFunction<T, U, V> function) ;
```

# CompletionStage – patterns

## Some 2 – 1 patterns

```
public <U, V> CompletionStage<V> thenCombineAsync  
(CompletionStage<U> other,  
 BiFunction<T, U, V> function) ;
```

```
public <U> CompletionStage<Void> thenAcceptBoth  
(CompletionStage<U> other,  
 BiConsumer<T, U> action) ;
```

# CompletionStage – patterns

## Some 2 – 1 patterns

```
public <U, V> CompletionStage<V> thenCombineAsync  
(CompletionStage<U> other,  
 BiFunction<T, U, V> function) ;
```

```
public <U> CompletionStage<Void> thenAcceptBoth  
(CompletionStage<U> other,  
 BiConsumer<T, U> action) ;
```

```
public CompletionStage<Void> runAfterBothAsync  
(CompletionStage<?> other,  
 Runnable action, Executor executor) ;
```

# CompletionStage – patterns

Some more 2 – 1 patterns

```
public <U> CompletionStage<U> applyToEither  
    (CompletionStage<? extends T> other,  
     Function<T, U> function) ;
```

# CompletionStage – patterns

Some more 2 – 1 patterns

```
public <U> CompletionStage<U> applyToEither  
(CompletionStage<? extends T> other,  
Function<T, U> function) ;
```

```
public CompletionStage<Void> acceptEitherAsync  
(CompletionStage<? extends T> other,  
Consumer<? extends T> consumer) ;
```

# CompletionStage – patterns

Some more 2 – 1 patterns

```
public <U> CompletionStage<U> applyToEither  
(CompletionStage<? extends T> other,  
Function<T, U> function) ;
```

```
public CompletionStage<Void> acceptEitherAsync  
(CompletionStage<? extends T> other,  
Consumer<? extends T> consumer) ;
```

```
public CompletionStage<Void> runAfterEitherAsync  
(CompletionStage<U> other,  
Runnable action, Executor executor) ;
```

# Back to our first example

So the complete pattern becomes this one

## 1) First we create our mocks

```
String result = Mockito.mock(String.class);
AsyncResponse response = Mockito.mock(AsyncResponse.class);

Runnable train = () -> Mockito.doReturn(result).when(response).longOperation();
Runnable verify = () -> Mockito.verify(response).resume(result);
```

## Back to our first example

So the complete pattern becomes this one

2) Then we create the call & verify

```
Runnable callAndVerify = () -> {  
    asyncResource.executeAsync(response).thenRun(verify);  
}
```

# Back to our first example

So the complete pattern becomes this one

3) Then we create the task

```
ExecutorService executor = Executors.newSingleThreadExecutor();

AsyncResource asyncResource = new AsyncResource();
asyncResource.setExecutorService(executor);

CompletableFuture
    .runAsync(train, executor)
    .thenRun(callAndVerify);                                // this trains our mocks
                                                        // this verifies our mocks
```

## Back to our first example

Since a CompletableFuture is also a Future, we can fail with a timeout if the test does not complete fast enough

```
ExecutorService executor = Executors.newSingleThreadExecutor();

AsyncResource asyncResource = new AsyncResource();
asyncResource.setExecutorService(executor);

CompletableFuture
    .runAsync(train, executor)                      // this trains our mocks
    .thenRun(callAndVerify)                          // this verifies our mocks
    .get(10, TimeUnit.SECONDS);
```

# A second example

## Async analysis of a web page

```
CompletableFuture.supplyAsync(  
    () -> readPage("http://whatever.com/"))  
)
```

# A second example

## Async analysis of a web page

```
CompletableFuture.supplyAsync(  
    () -> readPage("http://whatever.com/"))  
)  
.thenApply(page -> linkParser.getLinks(page))
```

# A second example

## Async analysis of a web page

```
CompletableFuture.supplyAsync(  
    () -> readPage("http://whatever.com/"))  
    )  
.thenApply(page -> linkParser.getLinks(page))  
.thenAccept(  
    links -> displayPanel.display(links) // in the right thread!  
)  
;
```

# A second example

## Async analysis of a web page

```
CompletableFuture.supplyAsync(  
    () -> readPage("http://whatever.com/"))  
)  
.thenApply(page -> linkParser.getLinks(page))  
.thenAcceptAsync(  
    links -> displayPanel.display(links),  
    executor  
) ;
```

# A second example

## Async analysis of a web page

```
public interface Executor {  
    void execute(Runnable command);  
}
```

# A second example

## Async analysis of a web page

```
public interface Executor {  
    void execute(Runnable command);  
}
```

```
Executor executor = runnable -> SwingUtilities.invokeLater(runnable) ;
```

# A second example

So the complete pattern is the following:

```
CompletableFuture.supplyAsync(  
    () -> readPage("http://whatever.com/"))  
)  
.thenApply(page -> linkParser.getLinks(page))  
.thenAcceptAsync(  
    links -> displayPanel.display(links),  
    runnable -> SwingUtilities.invokeLater(runnable)  
) ;
```

## A second example

Which can also be written with method references:

```
CompletableFuture.supplyAsync(  
    () -> readPage("http://whatever.com/"))  
    .thenApply(Parser::getLinks)  
    .thenAcceptAsync(  
        DisplayPanel::display,  
        SwingUtilities::invokeLater  
    ) ;
```

# A last example

## Async events in CDI

```
@Inject  
Event<String> event ;  
  
event.fire("some event") ; // returns void
```

```
public void observes(@Observes String payload) {  
    // handle the event, called in the firing thread  
}
```

# A last example

## Async events in CDI

```
public void observes(@Observes String payload) {  
    // handle the event, called in the firing thread  
    CompletableFuture.anyOf(/* some task */);  
}
```

# A last example

## Async events in CDI

```
@Inject  
Event<String> event ;  
  
event.fireAsync("some event") ; // returns CompletionStage<Object>
```

```
public void observes(@ObservesAsync String payload) {  
    // handle the event in another thread  
}
```

# A last example

## Async events in CDI

```
@Inject  
Event<String> event ;  
  
Executor executor = SwingUtilities::invokeLater  
  
event.fireAsync("some event", executor) ;
```

# A last example

## Async events in CDI

```
@Inject  
Event<String> event ;  
  
Executor executor = SwingUtilities::invokeLater  
  
CompletionStage<Object> cs =  
event.fireAsync("some event", executor) ;  
  
cs.whenComplete(...); // handle the exceptions
```

# CompletionStage – last patterns

## Static methods

```
public static CompletableFuture<Void>
    allOf(CompletableFuture<?>... cfs) ;
```

```
public static CompletableFuture<Object>
    anyOf(CompletableFuture<?>... cfs) ;
```

# Exception handling

So, a CompletableFuture can depend on:

- 1) one CompletableFuture
- 2) two CompletableFuture
- 3)  $N$  CompletableFuture

# Exception handling

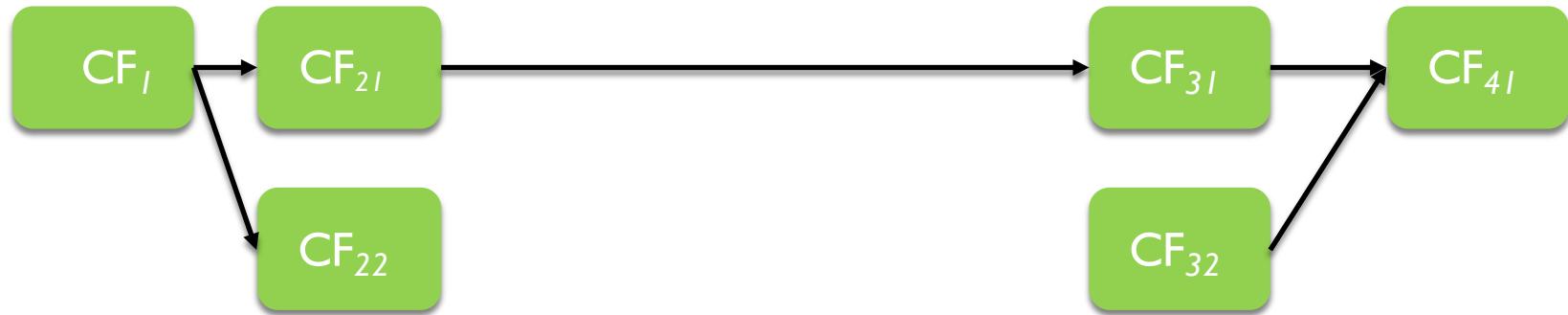
So, a CompletableFuture can depend on:

- 1) one CompletableFuture
- 2) two CompletableFuture
- 3)  $N$  CompletableFuture

What happens when an exception is thrown?

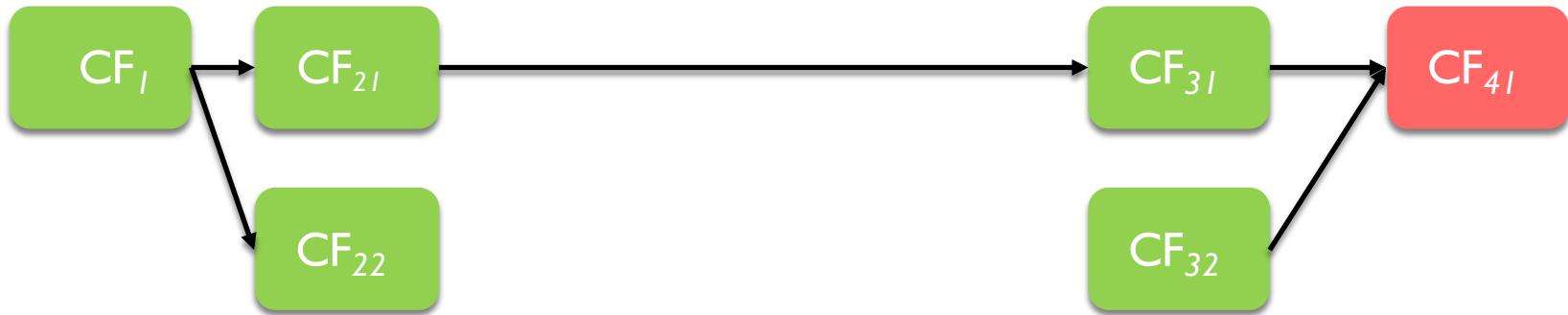
# Exception handling

Suppose we have this CF pipeline



# Exception handling

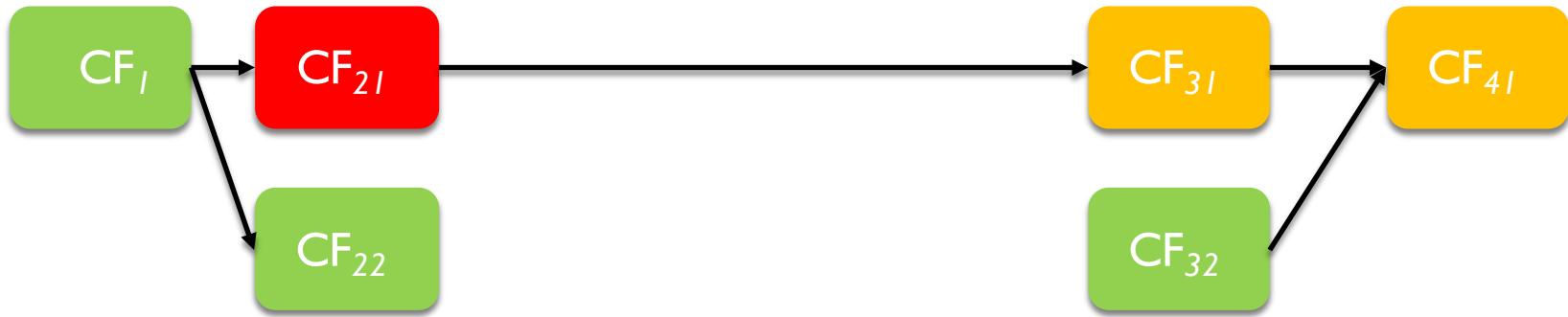
Suppose we have this CF pipeline



And CF<sub>21</sub> raises an exception

# Exception handling

Suppose we have this CF pipeline



And CF<sub>21</sub> raises an exception

Then all the depending CF are in error

# Exception handling

Which means that:

- the call to `isCompletedExceptionnaly()` returns true
- the call to `get()` throws an `ExecutionException` which cause is the root Exception

# Exception handling

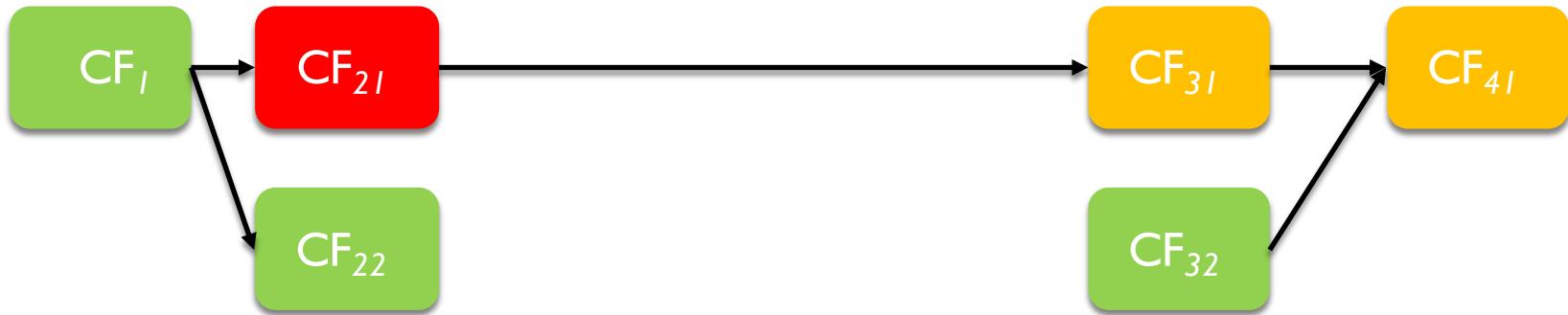
Which means that:

- the call to `isCompletedExceptionally()` returns true
- the call to `get()` throws an `ExecutionException` whose cause is the root Exception

But `CompletableFuture` can also handle exceptions

# Exception handling

Suppose we have this CF pipeline

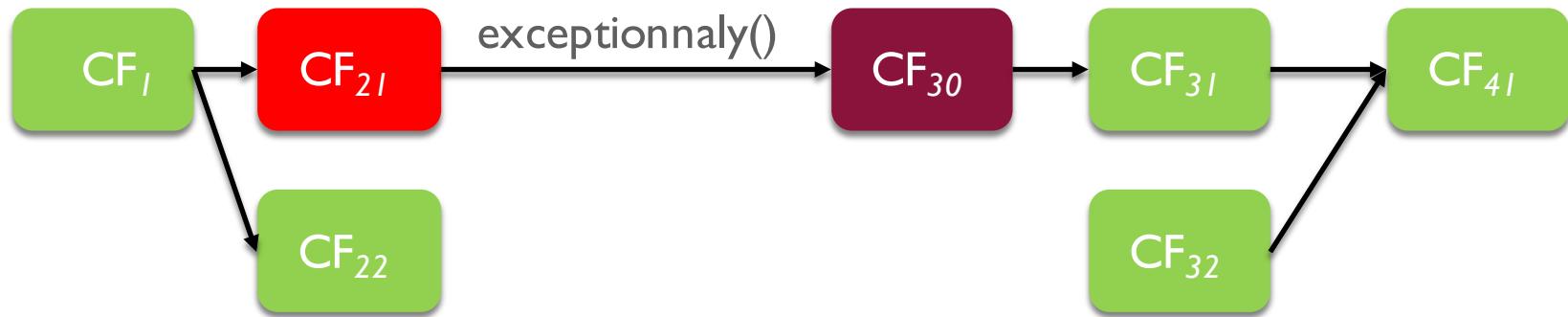


And CF<sub>21</sub> raises an exception

Then all the depending CF are in error

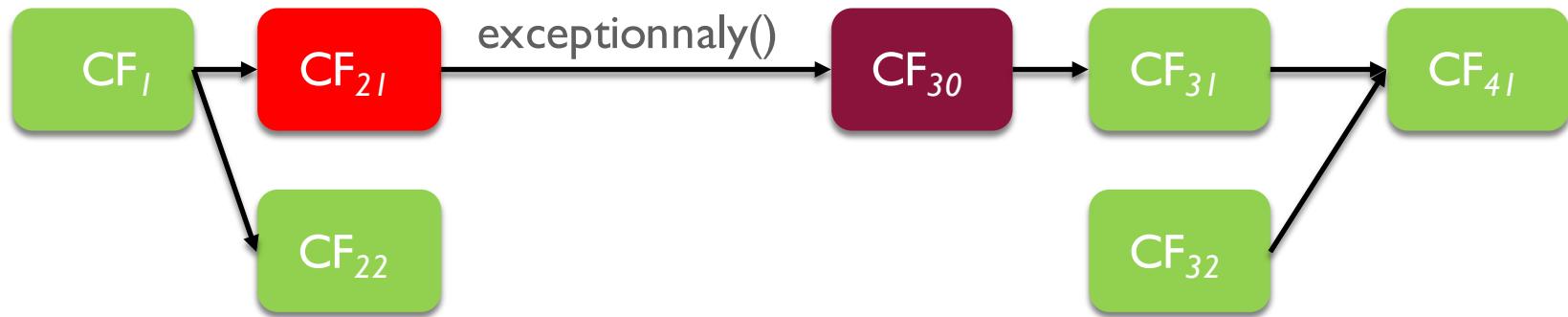
# Exception handling

Suppose  $CF_{30}$  has been created with `exceptionnaly()`



# Exception handling

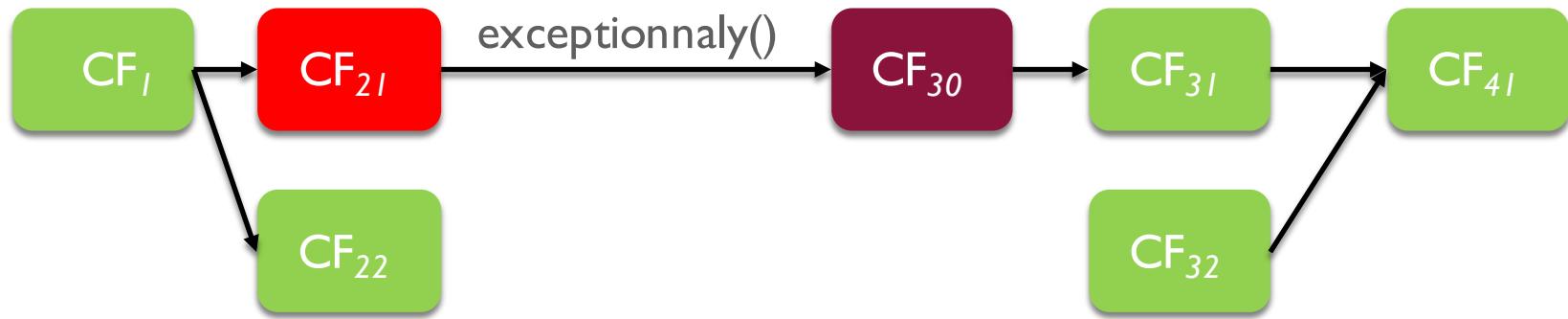
Suppose  $CF_{30}$  has been created with `exceptionnaly()`



If  $CF_{21}$  completes normally, then  $CF_{30}$  just transmits the value

# Exception handling

Suppose  $CF_{30}$  has been created with `exceptionnaly()`



If  $CF_{21}$  completes normally, then  $CF_{30}$  just transmits the value

If it raises an exception, then  $CF_{30}$  handles it and generate a value for  $CF_{31}$

# Exception handling

There are three methods to handle an exception

```
CompletionStage<T> exceptionally(  
    Function<Throwable, ? extends T> function);
```

# Exception handling

There are three methods to handle an exception

```
CompletionStage<T> exceptionally(  
    Function<Throwable, ? extends T> function);
```

```
<U> CompletionStage<U> handle(  
    BiFunction<? super T, Throwable, ? extends U> bifunction);
```

handle() has also asynchronous versions

# Exception handling

There are three methods to handle an exception

```
CompletionStage<T> exceptionally(  
    Function<Throwable, ? extends T> function);
```

```
<U> CompletionStage<U> handle(  
    BiFunction<? super T, Throwable, ? extends U> bifunction);
```

```
CompletionStage<T> whenComplete(  
    BiConsumer<? super T, Throwable> action);
```

whenComplete() has also asynchronous versions

# Conclusion

We have an API for async computations in the JDK!

# Conclusion

We have an API for async computations in the JDK!

Very rich, many methods which makes it complex

# Conclusion

We have an API for async computations in the JDK!

Very rich, many methods which makes it complex

Built on lambdas

# Conclusion

We have an API for async computations in the JDK!

Very rich, many methods which makes it complex

Built on lambdas

Gives a fine control over threads

# Conclusion

We have an API for async computations in the JDK!

Very rich, many methods which makes it complex

Built on lambdas

Gives a fine control over threads

Handle chaining, composition

# Conclusion

We have an API for async computations in the JDK!

Very rich, many methods which makes it complex

Built on lambdas

Gives a fine control over threads

Handle chaining, composition

Very clean way of handling exceptions

# Thank you!



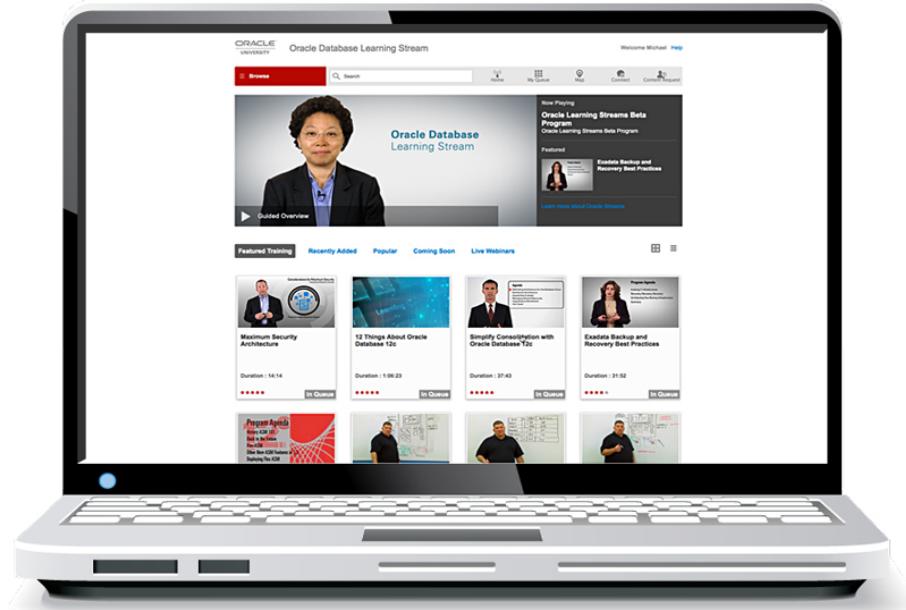
Copyright © 2014 Oracle and/or its affiliates. All rights reserved. |

# Q+A

ORACLE®

Copyright © 2014 Oracle and/or its affiliates. All rights reserved. |

# Keep Learning with Training from Oracle University



- Hands-on training delivered in-class or online by tenured instructors around the world
- New subscription-based learning services to give you any-time access to training
- Certification programs to validate your skills

[education.oracle.com](http://education.oracle.com)



**Thank You for Joining Us Today  
Please Move to the Next Session.**

# **Hardware and Software Engineered to Work Together**

# ORACLE®