Dealing with Exceptions in the CompletionStage Pipeline

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

How does completable futures deal with exceptions?

Exceptions can be caught in the pipeline

And then processed or rethrown

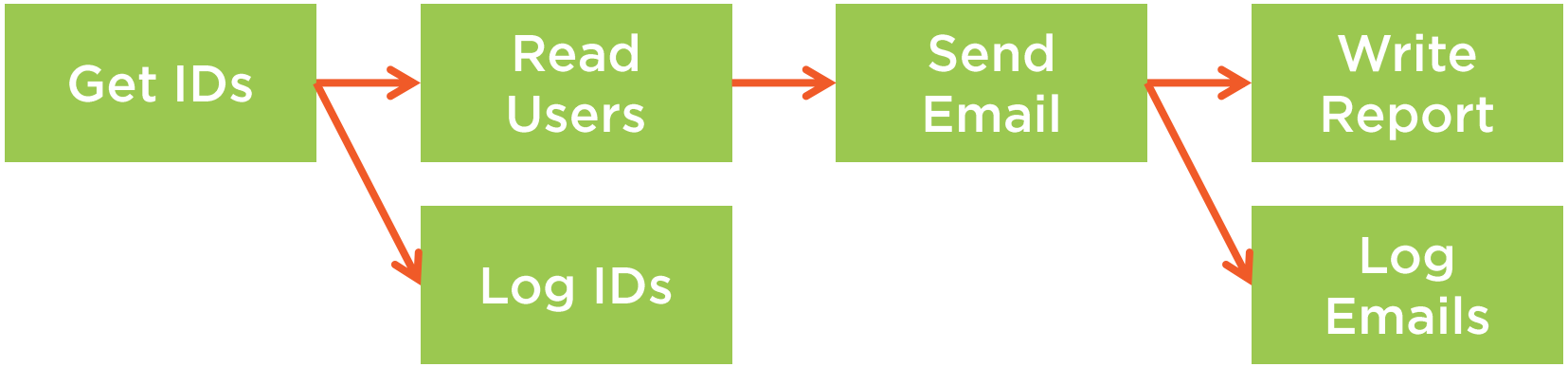
When Things Do Not Work as Expected

In fact exceptions are expected in the CompletableFuture API

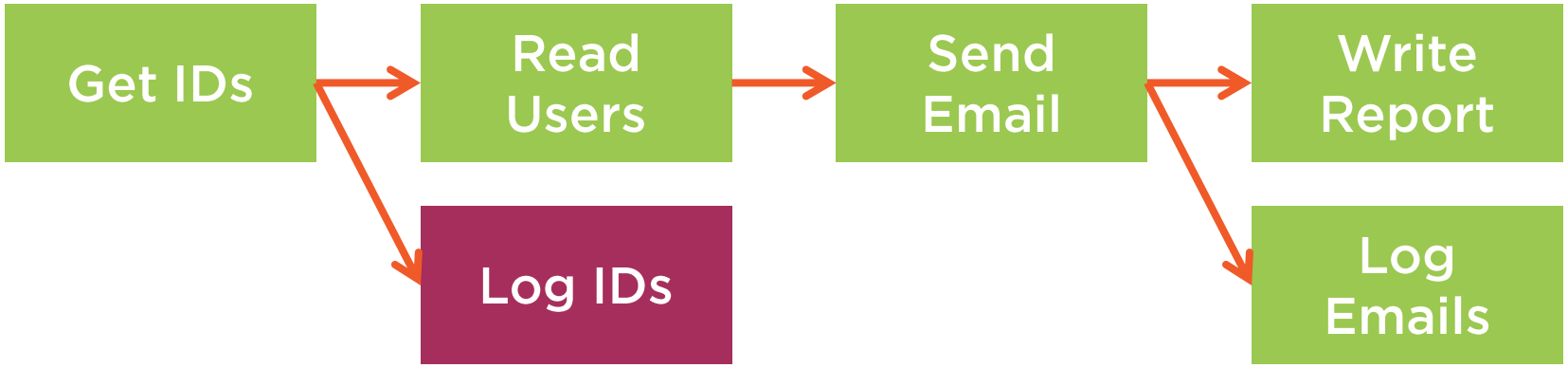
The behavior of the downstream tasks is specified

And there are several ways to catch exceptions

Setting up a Pipeline

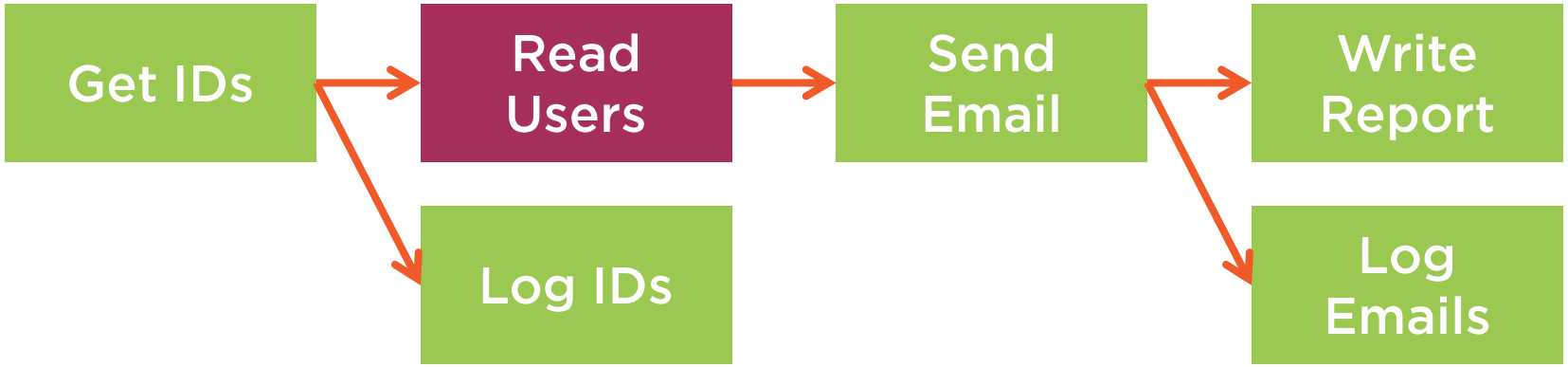


A Task with No Descendant



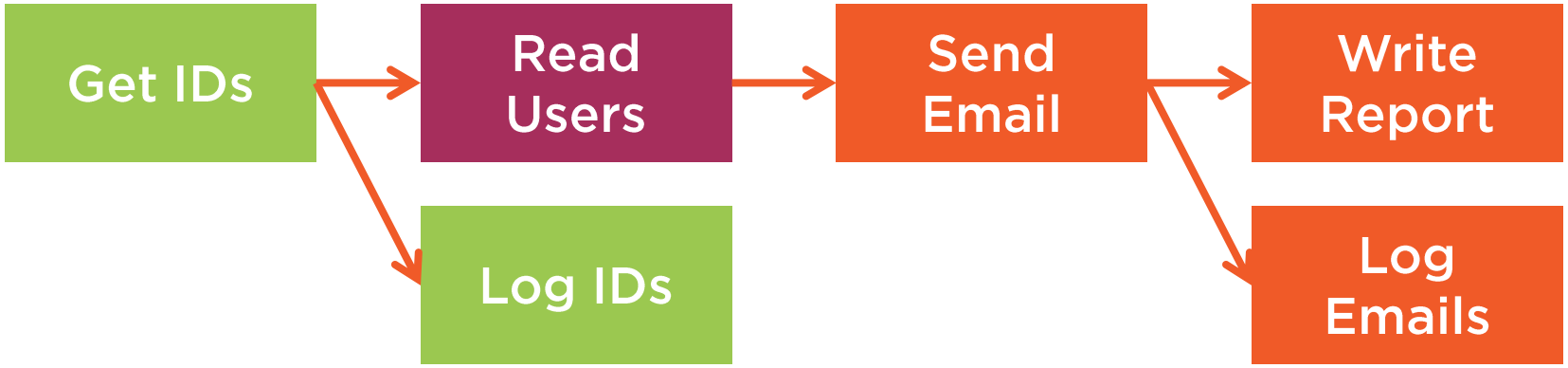
In this case Log IDs completes with an exception

A Task with Descendants



In this case Read Users completes with an exception

A Task with Descendants



In this case Read Users completes with an exception

And so do all the downstream tasks

```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list));
```

Suppose that a SQL exception is thrown when the list of users is fetched from the database

Then cf2 cannot produce any result

```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list));


List<User> usersFromJoin = cf2.join(); // from CompletableFuture
List<User> usersFromGet = cf2.get();   // from Future
```

Calling join() to get the result will throw a CompletionException

Calling get() will throw an ExecutionException

Both with the SQLException as the cause of this exception

```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list));

cf2.thenRun(() -> logger.info("The list of users has been read"));
cf2.thenAccept(
    users -> logger.info(users.size() + " users have been read"));
```

Since the readUsers() call threw an exception, no subsequent task can be executed

So neither the runnable, neither the consumer are called

The completable future they return completes exceptionally

So an exception may complete a completable future

Preventing it from providing a result

Instead, it will forward this exception to all its downstream completable futures

Since calling get() or join() is not the classical way of getting the result, what can be done?

Exception Handling

The Completable Future API can handle exceptions without any try catch

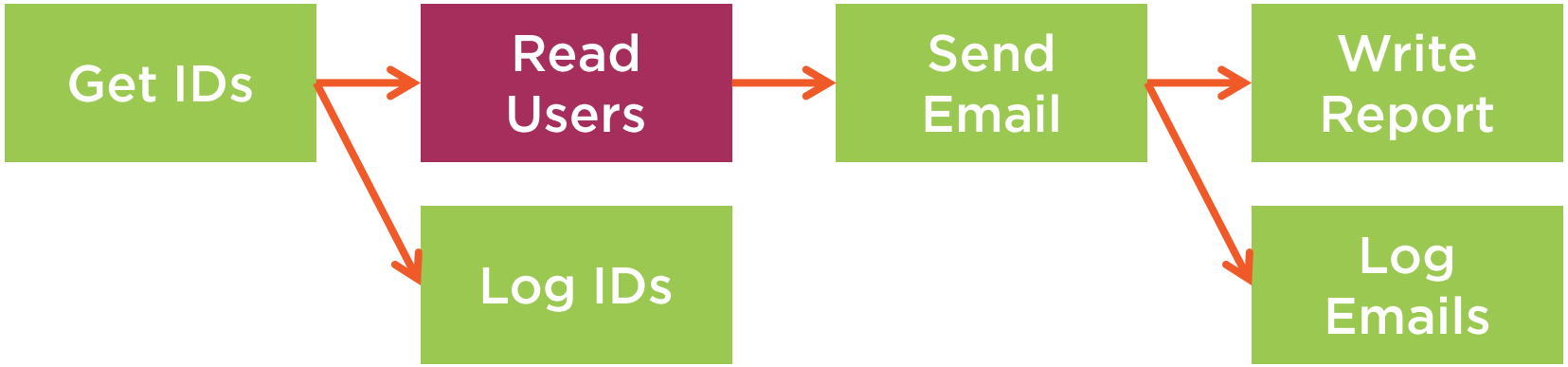
There are three patterns available:

- exceptionally()

- whenComplete()

- handle()

A Task with Descendants



Suppose Read Users completes with an exception

```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list))
                    .exceptionally(exception -> List.of());
```

Let us call exceptionally() on this pattern

exceptionally() returns a CompletableFuture of the same type as the one it is called on

```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list))
                    .exceptionally(exception -> List.of());
```

- If no exception is thrown, it returns the provided value

- If an exception is thrown, the function is executed and the result it gives is sent to the subsequent tasks

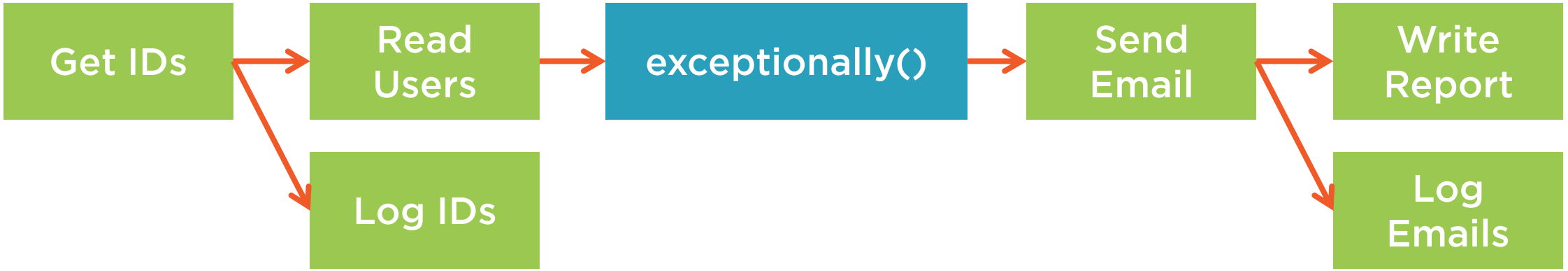
```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list))
                    .exceptionally(exception -> List.of());

cf2.thenRun(() -> logger.info("The list of users has been read"));
cf2.thenAccept(
    users -> logger.info(users.size() + " users have been read"));
```

If an exception is thrown:

- thenRun() will execute as usual

- thenAccept() will “see” an empty list

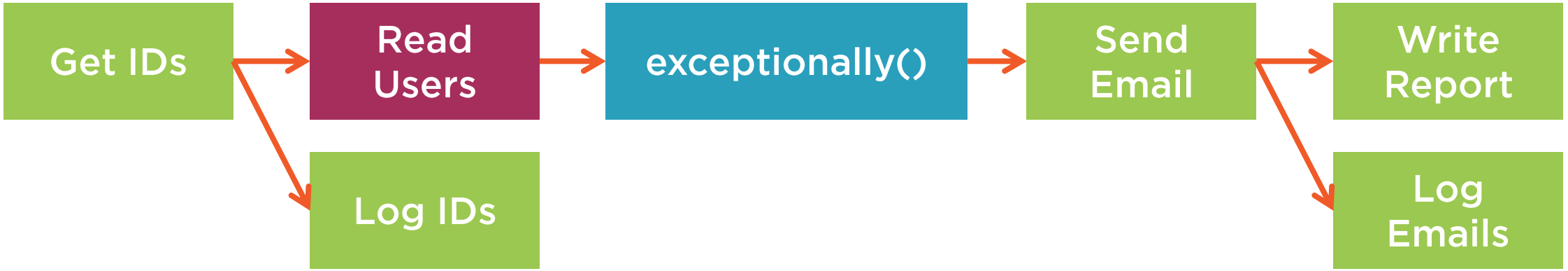
Inserting an Exception Handling Task



Let us insert an exceptionally() task in our pipeline

If no exception is thrown, this task will be transparent

Inserting an Exception Handling Task

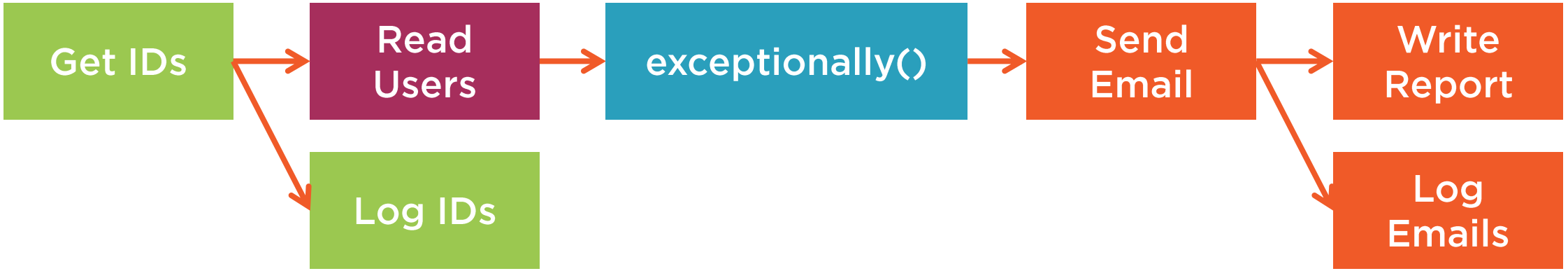


If an exception is thrown in task Read Users

This exception will be passed to the exceptionally() task

And a default value will be passed to Send Email

Inserting an Exception Handling Task



Note that this exception can also be rethrown

In that case, Send Email and the others will complete exceptionally

More Exception Handling Patterns

The whenComplete() pattern takes a result and the exception, if thrown

One of these two objects is null

They are passed to a biconsumer

The returned completable future returns the same thing as the calling one

```
CompletableFuture<List<User>> cf =
    supplyAsync(() -> List.of(1L, 2L, 3L))
        .thenApply(list -> readUsers(list))
        .whenComplete(
            (list, exception) -> {
                if (list != null)
                    logger.info("The list of users has been read");
                else
                    logger.error("An exception has been raised");
            });
```

You need to test which object is null to further process it

The exception raised by thenApply(), if any, will be also raised by whenComplete()

If no exception is raised by thenApply(), the result will be also returned by whenComplete()

The handle() pattern takes
a result and the exception, if thrown

One of these two objects is null

They are passed to a bifunction, that
produces a result

This result is returned by this completable
future

```
CompletableFuture<List<User>> cf2 =
    supplyAsync(() -> List.of(1L, 2L, 3L))
        .thenApply(list -> readUsers(list))
        .handle(
            (list, exception) -> {
                if (list != null)
                    return list;
                else {
                    logger.error("An exception has been raised");
                    return new ArrayList<>();
            }});
```

You need to test which object is null to further process it

The bifunction can either process and swallow the exception, or rethrow it as needed

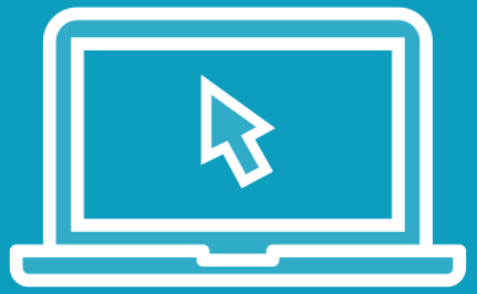
The methods whenComplete() and handle() have asynchronous versions:

- whenCompleteAsync()

- handleAsync()

Both come in two versions, one that takes an executor and one that does not

Demo



Time to see some code in action!

Let us create tasks and throw exceptions

And handle them properly

Module Wrapup

What did you learn?

Exceptions!

Exception Handling

	Parameter type	Can recover	Async
exceptionally()	`Function<Throwable, T>`	yes	no
handle()	`BiFunction<T, Throwable, U>`	yes	yes
whenComplete()	`BiConsumer<T, Throwable>`	no	yes

Module Wrap Up

What did you learn?

Exceptions!

Patterns to handle them, either by letting them go

Or by catching them and continue the process with default values

Both in asynchronous or synchronous mode