Triggering a Task on the Completion of Other Tasks

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

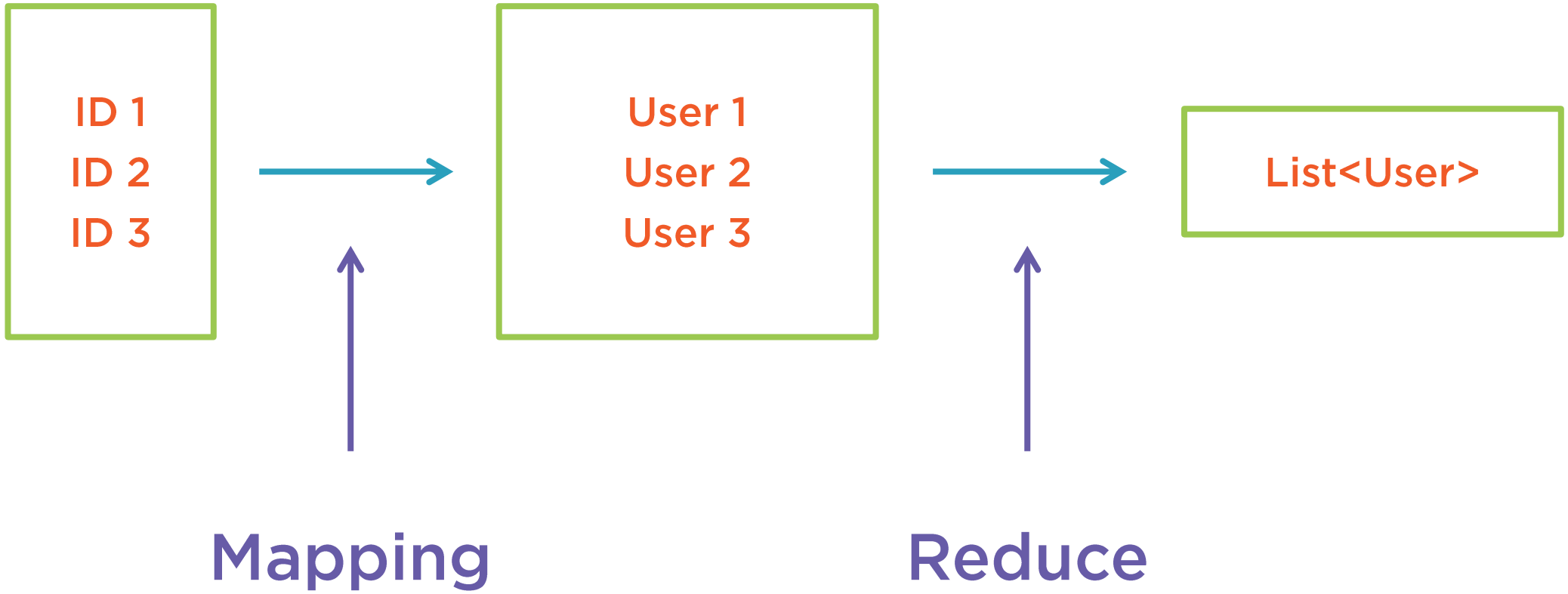
- Chaining of tasks
- Create pipelines of tasks
- Combine and compose them
- Run them asynchronously
- And control in which thread everything is executed

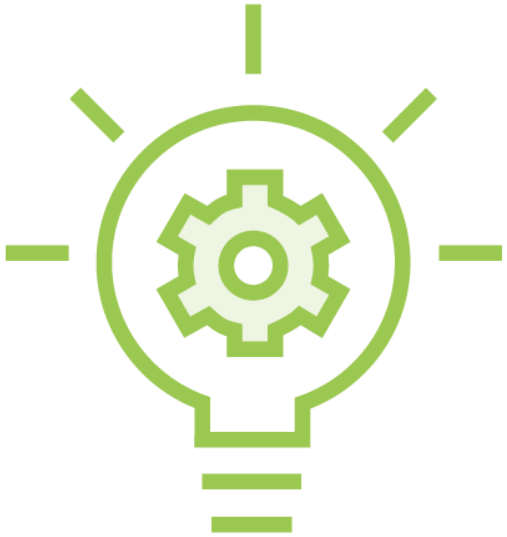
Defining a Pipeline of Tasks

Suppose you have a list of primary keys

And you need to query a database

To get the corresponding users


ID 1
ID 2
ID 3
User 1
User 2
User 3
List<User>
Mapping
Reduce



The first task is a supplier

We need a model for a subsequent task

- launched when the supplier is done

- that takes the value supplied and maps it

This is a function!

```
CompletableFuture<List<Long>> cf1 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L));
```

The first task is to provide the list of the primary keys

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


Function<List<Long>, List<User>> mapper =
   list -> readUsers(list);
```

The first task is to provide the list of the primary keys

You can chain a subsequent task on the completion of the first one

It returns another completable future

A completable future can send its result

To a mapper

Modeled by a function

Passing the result when it is available is handled by the API

More Tasks for CompletableFuture

```
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"));
```

Suppose you need to log a message once the list is available

You can add another subsequent task as a runnable

And if you need to log the number of users, you can use a consumer

CompletableFuture Supported Tasks

	Example	Method	CF method
Runnable	`() -> logger.info("User read")`	`void run()`	`thenRun()`
Consumer	`n -> logger.info(n + " users read")`	`void accept(Long)`	`thenAccept()`
Function	`id -> readUserFromDB(id)`	`User apply(Long)`	`thenApply()`

Single Task Chaining Patterns

```
CompletableFuture<Void> cf =
   CompletableFuture.runAsync(() -> updateDB())
                    .thenRun(() -> logger.info("Update done!"));
```

Suppose you need to update a database with a long running process

It makes sense to chain a runnable task

```
CompletableFuture<Void> cf =
   CompletableFuture.runAsync(() -> updateDB())
                    .thenAccept(value -> ...);
```

It is also possible to chain a task that takes a parameter

Question: what can be the value of this parameter?

Answer: the null value...

Does chaining a runnable with a consumer or a function make sense?

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

On the other hand, you can chain anything on a supplier

- a function

```
CompletableFuture<Void> cf =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenAccept(list -> logger.info(...));
```

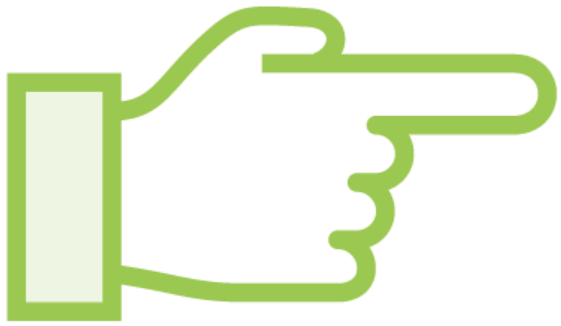
On the other hand, you can chain anything on a supplier

- a consumer

```
CompletableFuture<Void> cf =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenRun(() -> logger.info(...));
```

On the other hand, you can chain anything on a supplier

- a runnable



So the first task can be:

- a supplier: supplyAsync()

- or a runnable: runAsync()

The next task can be:

- a runnable: thenRun()

- a consumer: thenAccept()

- a function: thenApply()

But some combinations may be irrelevant

Completable Future Composition

Suppose you have a first task that fetches a list of User Ids from a remote service

And a second one that fetches the user objects from a database

Both tasks should be run asynchronously

```
Supplier<List<Long>> userIdsSupplier =
   () -> remoteService();   // returns the user IDs
Function<List<Long>, List<User>> usersFromIds =
   ids -> fetchFromDB(ids); // returns the user objects


CompletableFuture<List<User>> cf =
   CompletableFuture.supplyAsync(userIdsSupplier)
                    .thenApply(usersFromIds);
```

In this pattern, fetching the user objects with the user IDs is a synchronous operation

It is conducted synchronously when the list of user IDs is available

What you need is an asynchronous operation

```
Supplier<List<Long>> userIdsSupplier =
   () -> remoteService();   // returns the user IDs
Function<List<Long>, CompletableFuture<List<User>>> usersFromIds =
   ids -> fetchFromDB(ids); // returns the user objects


CompletableFuture<List<User>> cf =
   CompletableFuture.supplyAsync(userIdsSupplier)
                    .thenCompose(usersFromIds);
```

The thenCompose() method works the same as the flatMap() method from the Stream API and the Optional API

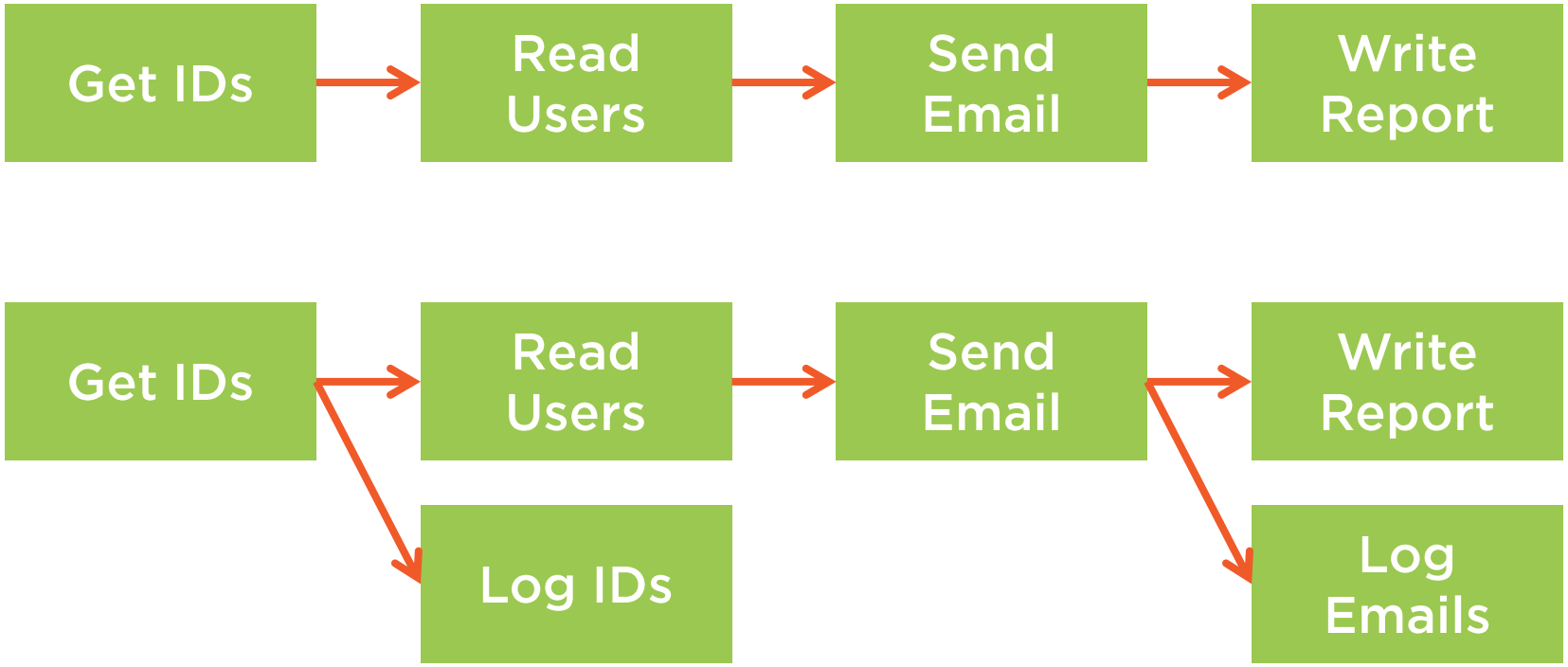
It just composes completable futures

Completable futures are composable

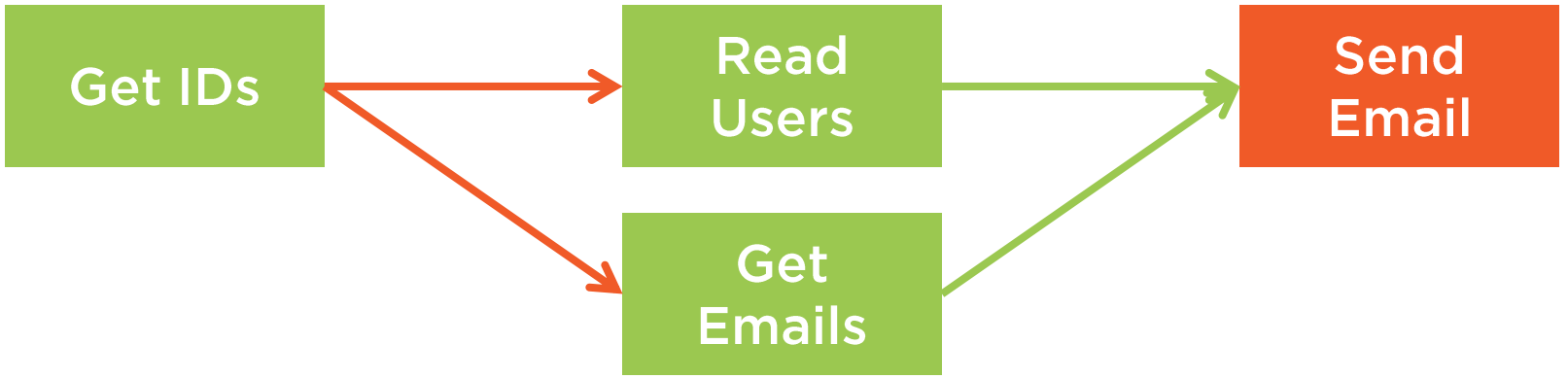
With the thenCompose() method

Triggering More Than One Task

CompletableFuture Task Chaining

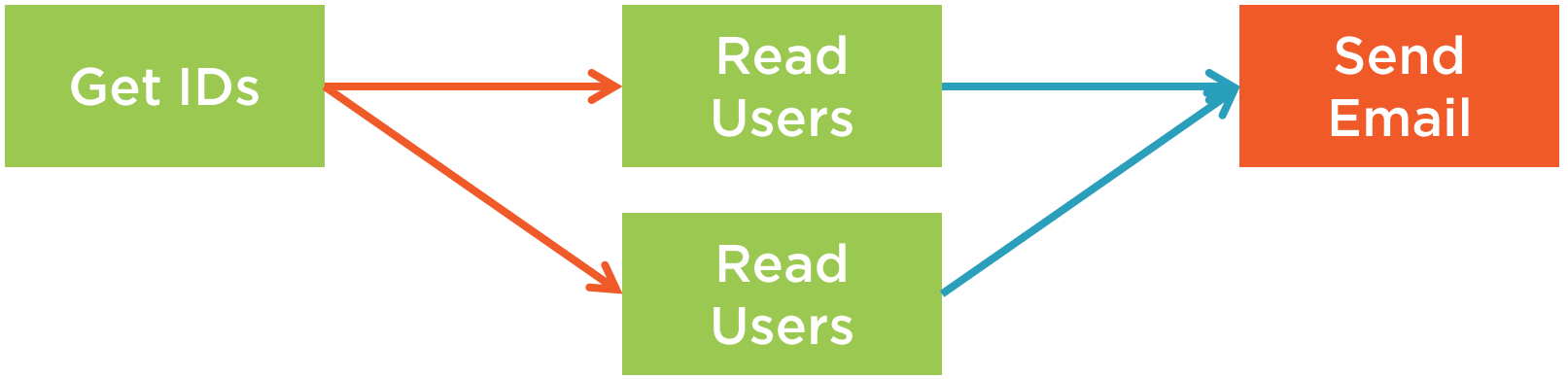


CompletableFuture Task Chaining



Send Email is launched if Read Users and Get Emails are done

CompletableFuture Task Chaining



Send Email is launched if any of Read Users is done

When both tasks complete, you can:

- execute a Runnable

- execute a BiConsumer

- execute a BiFunction

The two combined completable futures can return objects of different types

```
CompletableFuture<Long> cf1 = ...;
CompletableFuture<User> cf2 = ...;

CompletableFuture<Void> cf3 =
   cf1.thenAcceptBoth(cf2, (id, name) -> logger.info(...));
```

Two patterns are available:

- thenAccept() takes another completable future and a biconsumer

```
CompletableFuture<Long> cf1 = ...;
CompletableFuture<User> cf2 = ...;

CompletableFuture<List<User>> cf3 =
   cf1.thenCombine(cf2, (id, user) -> query(...));
```

Two patterns are available:

- thenCombine() takes another completable future and a bifunction

When either tasks complete, you can:

- execute a Runnable

- execute a Consumer

- execute a Function

In that case, both completable futures must return objects of the same type

```
CompletableFuture<Long> cf1 = ...;
CompletableFuture<Long> cf2 = ...;

CompletableFuture<Void> cf3 =
   cf1.thenAcceptEither(cf2, id -> logger.info(...));
```

In this case thenAcceptEither() takes another completable future and a consumer

The first completable future to produce a result is the winner!

```
CompletableFuture<Long> cf1 = ...;
CompletableFuture<Long> cf2 = ...;

CompletableFuture<User> cf3 =
   cf1.thenApplyToEither(cf2, id -> readUser(id));
```

The thenApplyToEither() takes another completable future and a function

```
CompletableFuture<Long> cf1 = ...;
CompletableFuture<Long> cf2 = ...;

CompletableFuture<User> cf3 =
   cf1.thenRunAfterEither(cf2, () -> logger.info(...));
```

The thenRunAfterEither() takes another completable future and a runnable

In the case of N completable futures, there is no combining of the results offered by the API

But the resulting completable future can complete either:

- on the completion of all the CF

- on the completion of the first CF

```
CompletableFuture<Long>       cf1 = ...;
CompletableFuture<User>       cf2 = ...;
CompletableFuture<String>     cf3 = ...;
CompletableFuture<List<User>> cf4 = ...;

CompletableFuture<Void> cf =
   CompletableFuture.allOf(cf1, cf2, cf3, cf4, ...);
```

Two patterns are available:

- allOf() takes a varargs of completable futures

```
CompletableFuture<Long>        cf1 = ...;
CompletableFuture<User>        cf2 = ...;
CompletableFuture<String>      cf3 = ...;
CompletableFuture<List<User>>  cf4 = ...;

CompletableFuture<?> cf =
   CompletableFuture.anyOf(cf1, cf2, cf3, cf4, ...);
```

Two patterns are available:

- anyOf() also takes a varargs of completable futures

What About Threads?

In which thread your tasks are executed?

Can you force this execution to be conducted in a given thread?

By default, asynchronous tasks are conducted in the Common Fork / Join pool of threads

A task triggered by another task is executed in the same thread as the triggering task

And you can change that

You can change it in two ways:

- allow the task to be executed in another thread of the same pool of threads as the triggering task

- have the task to be executed in another pool of threads

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


cf2.thenApply(
         list -> readUsers(list));
```

- This is executed in the Common FJ Pool

- This is executed in the same thread

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


cf2.thenApply(
         list -> readUsers(list));
```

- This is executed in the provided pool of threads

- This is executed in the same thread

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


cf2.thenApplyAsync(
          list -> readUsers(list));
```

- This is executed in the Common FJ Pool

- This is executed in the same pool of threads

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


cf2.thenApplyAsync(
         list -> readUsers(list),
         executor);
```

- This is executed in the Common FJ Pool

- This is executed in the provided pool of threads

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


cf2.thenApplyAsync(
         list -> readUsers(list),
         executor);

cf2.thenRun(
         () -> logger.info("..."));

cf2.thenAcceptAsync(
         users -> logger.info("..."));
```

- This is executed in the common FJ Pool
- This is executed in the provided pool of threads
- This is executed in the same thread
- This is executed in the same pool of thread

You can finely control the thread or pool of threads that executes each task

It makes the API complex at a first glance, since it adds many methods

Be careful: moving data from one thread to another is costly

Demo

Let us see some code!

Let us create simple tasks

Then chain and compose them

And move them from one thread to another

Module Wrapup

What did you learn?

Chaining and composition!

1 – 1 CompletableFuture Chaining

	Parameter type	Async	Executor
thenRun()	`Runnable`	yes	yes
thenAccept()	`Consumer<T>`	yes	yes
thenApply()	`Function<T, U>`	yes	yes
thenCompose()	`Function<T, CompletionStage<U>>`	yes	yes

2 – 1 CompletableFuture Chaining

	Parameter type	Async	Executor
runAfterBoth()	`Runnable`	yes	yes
thenAcceptBoth()	`BiConsumer<T, U>`	yes	yes
thenCombine()	`BiFunction<T, U, V>`	yes	yes

2 – 1 CompletableFuture Chaining

	Parameter type	Async	Executor
runAfterEither()	`Runnable`	yes	yes
acceptEither()	`Consumer<T>`	yes	yes
applyToEither()	`Function<T, V>`	yes	yes

Module Wrap Up

What did you learn?

Chaining and composition!

Patterns to trigger the execution of a task with another task

And with two tasks

How to control which thread should execute which task