DEVELHOPE

#codeforimpact

Async Programming

DEVELHOPE

#codeforimpact

Sync vs Async

Sync programming means that Java can execute just one task at a time.
In order to be executed, the following task has to wait for the previous task to finish.

Aysnc programming means that Java can execute multiple tasks in the same time.
Two or more tasks can be executed in the exact instant, without waiting each other.

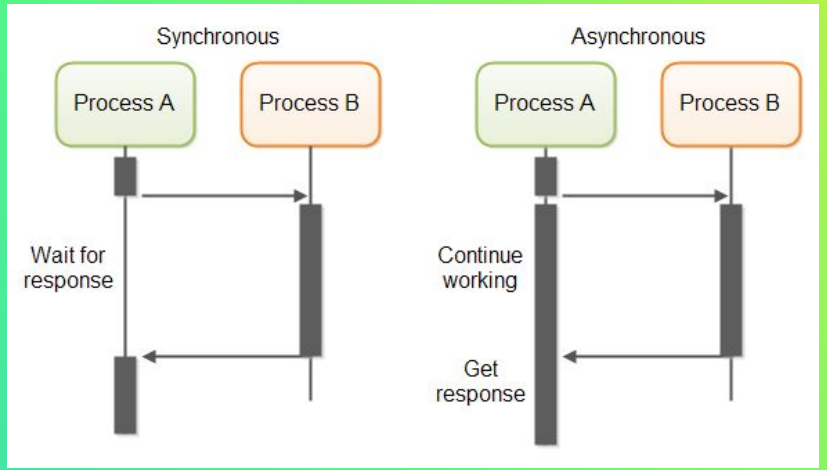

Image source

When to implement async programming

Async programming can be useful when:

- there are loops with large number of iterations involved
- there are I/O tasks as reading files
- there are network calls
- there's a critical need of efficiency optimisation, using parallel programming and multithreading
- there's demand of non-blocking code
- there are a large number of tasks that are independent of one another

Lambda expressions

You will see often the –> syntax in the following slide. This kind of arrow is called *lambda expression*.

A *lambda expression* takes one or more parameters and returns a value: `parameter -> expression`

Lambdas are similar to methods, but they don't need a name.

```
parameter -> expression

(p1, p2, …, pn) -> expression // multiple parameters

p1 -> { … }                   // more complex code can be written inside the brackets

(p1, p2) -> { … }             // the code will need to return a value
```

#codeforimpact

`CompletableFuture` API

Java 8 introduced the `CompletableFuture` *API* that helps programmers with async programming.

The `CompletableFuture` API basically has 4 main features:

- Async task creation
- Chaining (callbacks)
- Completion
- Exception handling

`CompletableFuture` API: async task creation

The `CompletableFuture` *API* lets you create the initial async task in two ways:

- with `runAsync`, that takes a `Runnable` as parameter and returns a `Void`;
- with `supplyAsync`, that takes a `Supplier` as parameter and returns the same type of the `Supplier` argument

```
Runnable task = () -> {
     System.out.println("Runnable Message");
}

CompletableFuture<Void> testTask = CompletableFuture.runAsync(task);


Supplier<String> supplier = () -> "Supplier Message";

CompletableFuture<String> testSupply = CompletableFuture.supplyAsync(supplier);
```

`CompletableFuture` API: chaining callbacks

A newly created `CompletableFuture` has 4 methods, also called *callbacks*, that let you chain different tasks, creating pipelines:

- `thenRunAsync(Runnable action)` - executes the given `action`
- `thenApplyAsync(Function fn)` - takes a `Function` as argument, being a kind of intermediate chain element that consumes the input of the previous stage and provides the output for the next stage. This callback can be used for transform the result of `CompletableFuture`
- `thenAcceptAsync(Consumer action)` - takes a `Consumer` as argument, receives input of the previous stage and doesn't return anything
- `thenComposeAsync(Function fn)` - is a callback that can be used for chaining `CompletableFuture`.

`CompletableFuture` API: completion

`CompletableFuture` API offers different completion methods, we are covering just 3:

- `join()` - returns the result value when completed. If completed exceptionally, it throws an unchecked exception
- `getNow(T valueIfAbsent)` - if completed returns the result value, else returns the `valueIfAbsent`
- `completeExceptionally(Throwable ex)` - if not already completed, causes invocations of `get()` and related method to throw the `ex` exception. It returns `true` if the invocation caused this `CompletableFuture` to transition to a completed state, else `false`.

`CompletableFuture` API: exception handling

The *state* of a `CompletableFuture` can be:

- `Running`
- `Completed`
- `CompletedExceptionally`

If a task throws an `Exception`, the *state* of the `CompletableFuture` will transition to `CompletedExceptionally`.

The `CompletableFuture` *API* offers some methods to handle exceptions occurred during the chaining:

- `exceptionally(Function fn)` - allows to recover from an exception in the pipeline
- `handle(BiFunction fn)` - the method will be executed regardless of whether an exception occurs or not, allowing you to recover or throw an exception downstream
- `whenComplete(BiFunction fn)` - it's used at the end of the pipeline to understand if all the steps completed.

DEVELHOPE

#codeforimpact

Async Programming