02 Behaviours

Frequency

Setup

All commands should be run either in an operating system shell (commands beginning with `$`) or in `iex`, the Elixir shell (commands beginning with `>`).

This and the following exercises assume that the current working directory is set to either the `beam2024` repository or one of its subdirectories. We'll start in `02-behaviours/frequency`.

How to start

To start the project and load all relevant modules run:

```
$ iex -S mix
```

Frequency module

The Frequency module implements a *process skeleton* described in the presentation. This skeleton is used on multiple occasions when working with OTP components in Elixir or Erlang. The process can be started by calling:

```
> Frequency.start()
```

Then we can use the provided `allocate/0` and `deallocate/1` functions to interact with the process. To stop the process call:

```
> Frequency.stop()
```

Question What will happen if you try to start Frequency one more time?

Frequency2 and Server modules

`Server` module implements a common behaviour of our process skeleton. It contains functions like `start/2`, `stop/1`, `init/2`, `loop/2` and `call/2`. In this module two callbacks (`init/1` and `handle/2`) are also defined, those functions will be implemented in `Frequency2` module (or any other module using `Server` as its behaviour). This decouples the generic part (e.g. process management, listening for messages etc.) from our specific part - allocating and deallocating frequencies.

Since the logic to handle process and it's information is moved into a separate module, interacting with `Frequency2` module is slightly different than in the first example. Namely, process management, so things like starting or stopping, will be done by the `Serve` module. We can still use `Frequency2` module to call the `allocate/0` and `deallocate/1` functions.

To start the process use:

```
> {:ok, pid} = Server.start(Frequency2, [])
```

Allocating a frequency:

```
> Frequency2.allocate()
```

Deallocating a frequency:

```
> Frequency2.deallocate(10)
```

Finally, to stop the process:

```
> Server.stop(pid)
```

Other uses

In this section of the lecture and exercises, we are using a process skeleton to demonstrate how to apply behaviors.

However, this is not always the primary use case. Another common use case is to establish a form of contract between different modules that implement the same functionality.

A typical example is modules that handle communication with third-party APIs. Often, we do not need or want a live API implementation when running tests, or even during development.

Your task is to create a module named `*_behaviour.ex` with 2 or 3 callbacks. Optionally, you can also implement default function specifications using `defmacro` and `__using__`.

In the next step create two modules that implement the behavior defined in the previous step.