STATE YOUR STATE

The examples shown in the previous chapter were all right for demonstrative purposes, but you won't go far with only that in your toolkit. It's not that the examples were bad, it's mostly that there is not a huge advantage to processes and actors if they're just functions with messages. To fix this, we have to be able to hold state in a process.

Let's first create a function in a new kitchen.erl module that will let a process act like a fridge. The process will allow two operations: storing food in the fridge and taking food from the fridge. It should only be possible to take food that has been stored beforehand. The following function can act as the base for our process:

```
-module(kitchen).
-compile(export_all).

fridge1() ->
receive
{From, {store, _Food}} ->
From ! {self(), ok},
fridge1();
{From, {take, _Food}} ->
%% uh....
From ! {self(), not_found},
fridge1();
terminate ->
ok
end.
```

Something's wrong with it. When we ask to store the food, the process should reply with `ok`, but there is nothing actually storing the food; `fridge1()` is called and then the function starts from scratch, without state. You can also see that when we call the process to take food from the fridge, there is no state to take it from and so the only thing to reply is `not_found`. In order to store and take food items, we'll need to add state to the function.

With the help of recursion, the state to a process can then be held entirely in the parameters of the function. In the case of our fridge process, a possibility would be to store all the food as a list, and then look in that list when someone needs to eat something:

```
fridge2(FoodList) ->
receive
{From, {store, Food}} ->
From ! {self(), ok},
fridge2([Food|FoodList]);
{From, {take, Food}} ->
case lists:member(Food, FoodList) of
```

```
        true ->
            From ! {self(), {ok, Food}},
            fridge2(lists:delete(Food, FoodList));
        false ->
            From ! {self(), not_found},
            fridge2(FoodList)
    end;
terminate ->
    ok
end.
```

The first thing to notice is that `fridge2/1` takes one argument, *`FoodList`*. You can see that when we send a message that matches `{From, {store, Food}}`, the function will add *`Food`* to *`FoodList`* before going. Once that recursive call is made, it will then be possible to retrieve the same item. In fact, I implemented it there. The function uses `lists:member/2` to check whether *`Food`* is part of *`FoodList`* or not. Depending on the result, the item is sent back to the calling process (and removed from *`FoodList`*) or `not_found` is sent back otherwise:

```
1> c(kitchen).
{ok,kitchen}
2> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.51.0>
3> Pid ! {self(), {store, milk}}.
{<0.33.0>,{store,milk}}
4> flush().
Shell got {<0.51.0>,ok}
ok
```

Storing items in the fridge seems to work. We'll try with some more stuff and then try to take it from the fridge.

```
5> Pid ! {self(), {store, bacon}}.
{<0.33.0>,{store,bacon}}
6> Pid ! {self(), {take, bacon}}.
{<0.33.0>,{take,bacon}}
7> Pid ! {self(), {take, turkey}}.
{<0.33.0>,{take,turkey}}
8> flush().
Shell got {<0.51.0>,ok}
Shell got {<0.51.0>,{ok,bacon}}
Shell got {<0.51.0>,not_found}
ok
```

As expected, we can take bacon from the fridge because we have put it in there first (along with the milk and baking soda), but the fridge process has no turkey to find when we request some. This is why we get the last `{<0.51.0>,not_found}` message.

Source : http://learnyousomeerlang.com/more-on-multiprocessing