KEEPING MESSAGES SECRET

Something annoying with the previous example is that the programmer who's going to use the fridge has to know about the protocol that's been invented for that process. That's a useless burden. A good way to solve this is to abstract messages away with the help of functions dealing with receiving and sending them:

```
store(Pid, Food) ->
Pid ! {self(), {store, Food}},
receive
{Pid, Msg} -> Msg
end.

take(Pid, Food) ->
Pid ! {self(), {take, Food}},
receive
{Pid, Msg} -> Msg
end.
```

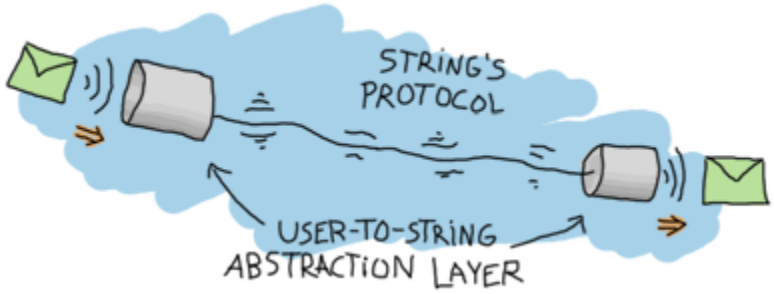
Now the interaction with the process is much cleaner:

```
9> c(kitchen).
{ok,kitchen}
10> f().
ok
11> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.73.0>
12> kitchen:store(Pid, water).
ok
13> kitchen:take(Pid, water).
{ok,water}
14> kitchen:take(Pid, juice).
not_found
```

We don't have to care about how the messages work anymore, if sending `self()` or a precise atom like `take` or`store` is needed: all that's needed is a pid and knowing what functions to call. This hides all of the dirty work and makes it easier to build on the fridge process.

One thing left to do would be to hide that whole part about needing to spawn a process. We dealt with hiding messages, but then we still expect the user to handle the creation of the process. I'll add the following `start/1`function:

```
start(FoodList) ->
spawn(?MODULE, fridge2, [FoodList]).
```



Here, `?MODULE` is a macro returning the current module's name. It doesn't look like there are any advantages to writing such a function, but there really are some. The essential part of it would be consistency with the calls to `take/2` and `store/2`: everything about the fridge process is now handled by the kitchen module. If you were to add logging when the fridge process is started or start a second process (say a freezer), it would be really easy to do inside our `start/1` function. However if the spawning is left for the user to do through `spawn/3`, then every place that starts a fridge now needs to add the new calls. That's prone to errors and errors suck.
Let's see this function put to use:

```
15> f().
ok
16> c(kitchen).
{ok,kitchen}
17> Pid = kitchen:start([rhubarb, dog, hotdog]).
<0.84.0>
18> kitchen:take(Pid, dog).
{ok,dog}
19> kitchen:take(Pid, dog).
not_found
```

Yay! The dog has got out of the fridge and our abstraction is complete!

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