AN EVENT MODULE

Get into the `src/` directory and start an event.erl module, which will implement the x, y and z events in the earlier drawings. I'm starting with this module because it's the one with the fewest dependencies: we'll be able to try to run it without needing to implement the event server or client functions.

Before really writing code, I have to mention that the protocol is incomplete. It helps represent what data will be sent from process to process, but not the intricacies of it: how the addressing works, whether we use references or names, etc. Most messages will be wrapped under the form `{Pid, Ref, Message}`, where *`Pid`* is the sender and *`Ref`*is a unique message identifier to help know what reply came from who. If we were to send many messages before looking for replies, we would not know what reply went with what message without a reference.

So here we go. The core of the processes that will run `event.erl`'s code will be the function `loop/1`, which will look a bit like the following skeleton if you remember the protocol:

```
loop(State) ->
receive
{Server, Ref, cancel} ->
...
after Delay ->
...
end.
```

This shows the timeout we have to support to announce an event has come to term and the way a server can call for the cancellation of an event. You'll notice a variable *`State`* in the loop. The *`State`* variable will have to contain data such as the timeout value (in seconds) and the name of the event (in order to send the message `{done, Id}`.) It will also need to know the event server's pid in order to send it notifications.

This is all stuff that's fit to be held in the loop's state. So let's declare a `state` record on the top of the file:

```
-module(event).
-compile(export_all).
-record(state, {server,
name="",
to_go=0}).
```

With this state defined, it should be possible to refine the loop a bit more:

```
loop(S = #state{server=Server}) ->
receive
{Server, Ref, cancel} ->
Server ! {Ref, ok}
after S#state.to_go*1000 ->
Server ! {done, S#state.name}
end.
```

Here, the multiplication by a thousand is to change the `to_go` value from seconds to milliseconds.

Don't drink too much Kool-Aid:
Language wart ahead! The reason why I bind the variable 'Server' in the function head is because it's used in pattern matching in the receive section. Remember, records are hacks!The expression `S#state.server` is secretly expanded to `element(2, S)`, which isn't a valid pattern to match on.
This still works fine for `S#state.to_go` after the `after` part, because that one can be an expression left to be evaluated later.

Now to test the loop:

```
6> c(event).
{ok,event}
7> rr(event, state).
[state]
8> spawn(event, loop,
[#state{server=self(), name="test", to_go=5}]).
<0.60.0>
9> flush().
ok
10> flush().
Shell got {done,"test"}
ok
11> Pid = spawn(event, loop,
[#state{server=self(), name="test", to_go=500}]).
<0.64.0>
12> ReplyRef = make_ref().
#Ref<0.0.0.210>
13> Pid ! {self(), ReplyRef, cancel}.
{<0.50.0>,#Ref<0.0.0.210>,cancel}
14> flush().
Shell got {#Ref<0.0.0.210>,ok}
ok
```

Lots of stuff to see here. Well first of all, we import the record from the event module with `rr(Mod)`. Then, we spawn the event loop with the shell as the server (`self()`). This event should fire after 5 seconds. The 9th expression was run after 3 seconds, and the 10th one after 6 seconds. You can see we did receive the `{done, "test"}` message on the second try.

Right after that, I try the cancel feature (with an ample 500 seconds to type it). You can see I created the reference, sent the message and got a reply with the same reference so I know the `ok` I received was coming from this process and not any other on the system.
The reason why the cancel message is wrapped with a reference but the `done` message isn't is simply because we don't expect it to come from anywhere specific (any place will do, we won't match on the receive)

nor should we want to reply to it. There's another test I want to do beforehand. What about an event happening next year?

```
15> spawn(event, loop,
[#state{server=self(), name="test", to_go=365*24*60*60}]).
<0.69.0>
16>
=ERROR REPORT==== DD-MM-YYYY::HH:mm:SS ===
Error in process <0.69.0> with exit value:
{timeout_value,[{event,loop,1}]}
```

Ouch. It seems like we hit an implementation limit. It turns out Erlang's timeout value is limited to about 50 days in milliseconds. It might not be significant, but I'm showing this error for three reasons:

1. It bit me in the ass when writing the module and testing it, halfway through the chapter.
2. Erlang is certainly not perfect for every task and what we're seeing here is the consequences of using timers in ways not intended by the implementers.
3. That's not really a problem; let's work around it.

The fix I decided to apply for this one was to write a function that would split the timeout value into many parts if turns out to be too long. This will request some support from the `loop/1` function too. So yeah, the way to split the time is basically divide it in equal parts of 49 days (because the limit is about 50), and then put the remainder with all these equal parts. The sum of the list of seconds should now be the original time:

```
%% Because Erlang is limited to about 49 days
(49*24*60*60*1000) in
%% milliseconds, the following function is used
normalize(N) ->
Limit = 49*24*60*60,
[N rem Limit | lists:duplicate(N div Limit, Limit)].
```

The function `lists:duplicate/2` will take a given expression as a second argument and reproduce it as many times as the value of the first argument (`[a,a,a] = lists:duplicate(3, a)`). If we were to send `normalize/1` the value `98*24*60*60+4`, it would return `[4,4233600,4233600]`.

The `loop/1` function should now look like this to accommodate the new format:

```
%% Loop uses a list for times in order to go around the ~49
days limit
%% on timeouts.
loop(S = #state{server=Server, to_go=[T|Next]}) ->
receive
{Server, Ref, cancel} ->
Server ! {Ref, ok}
after T*1000 ->
if Next =:= [] ->
Server ! {done, S#state.name};
Next =/= [] ->
```

```
loop(S#state{to_go=Next})
end
end.
```

You can try it, it should work as normal, but now support years and years of timeout. How this works is that it takes the first element of the `to_go` list and waits for its whole duration. When this is done, the next element of the timeout list is verified. If it's empty, the timeout is over and the server is notified of it. Otherwise, the loop keeps going with the rest of the list until it's done.

It would be very annoying to have to manually call something like `event:normalize(N)` every time an event process is started, especially since our workaround shouldn't be of concern to programmers using our code. The standard way to do this is to instead have an `init` function handling all initialization of data required for the loop function to work well. While we're at it, we'll add the standard `start` and `start_link` functions:

```
start(EventName, Delay) ->
spawn(?MODULE, init, [self(), EventName, Delay]).

start_link(EventName, Delay) ->
spawn_link(?MODULE, init, [self(), EventName, Delay]).

%%% Event's innards
init(Server, EventName, Delay) ->
loop(#state{server=Server,
name=EventName,
to_go=normalize(Delay)}).
```

The interface is now much cleaner. Before testing, though, it would be nice to have the only message we can send, cancel, also have its own interface function:

```
cancel(Pid) ->
%% Monitor in case the process is already dead
Ref = erlang:monitor(process, Pid),
Pid ! {self(), Ref, cancel},
receive
{Ref, ok} ->
erlang:demonitor(Ref, [flush]),
ok;
{'DOWN', Ref, process, Pid, _Reason} ->
ok
end.
```

Oh! A new trick! Here I'm using a monitor to see if the process is there or not. If the process is already dead, I avoid useless waiting time and return `ok` as specified in the protocol. If the process replies with the reference, then I know it will soon die: I remove the reference to avoid receiving them when I no longer care about them. Note that I also supply the `flush` option, which will purge the `DOWN` message if it was sent before we had the time to demonitor.

Let's test these:

```
17> c(event).
{ok,event}
18> f().
ok
19> event:start("Event", 0).
<0.103.0>
20> flush().
Shell got {done,"Event"}
ok
21> Pid = event:start("Event", 500).
<0.106.0>
22> event:cancel(Pid).
ok
```

And it works! The last thing annoying with the event module is that we have to input the time left in seconds. It would be much better if we could use a standard format such as Erlang's datetime (`{{Year, Month, Day}, {Hour, Minute, Second}}`). Just add the following function that will calculate the difference between the current time on your computer and the delay you inserted:

```
time_to_go(TimeOut={{_,_,_}, {_,_,_}}) ->
Now = calendar:local_time(),
ToGo = calendar:datetime_to_gregorian_seconds(TimeOut) -
calendar:datetime_to_gregorian_seconds(Now),
Secs = if ToGo > 0 -> ToGo;
ToGo =< 0 -> 0
end,
normalize(Secs).
```

Oh, yeah. The calendar module has pretty funky function names. As noted above, this calculates the number of seconds between now and when the event is supposed to fire. If the event is in the past, we instead return `0` so it will notify the server as soon as it can. Now fix the init function to call this one instead of `normalize/1`. You can also rename *`Delay`* variables to say *`DateTime`* if you want the names to be more descriptive:

```
init(Server, EventName, DateTime) ->
loop(#state{server=Server,
name=EventName,
to_go=time_to_go(DateTime)}).
```

Now that this is done, we can take a break. Start a new event, go drink a pint (half-litre) of milk/beer and come back just in time to see the event message coming in.

Source : http://learnyousomeerlang.com/designing-a-concurrent-application