Concurrent Programming in Erlang

The aim of this exercise is to give you experience of creating processes in Erlang, and how to program communication between them.

We'll do this by writing a simple palindrome checker process, and interacting with it.

Use the comments on this step to share your solutions or questions about the exercise. Remember to 'like' messages that you think are particularly interesting or raise important questions.

There is limited formatting in the comments, so throughout this course you may want to use an external site to host your code, and just post the link. Suggested ways of sharing include:

- a github gist. You'll need a github account to use this.
- a hastebin file. You don't need to create an account to use hastebin.

If you do want to share your solutions without spoiling things for other learners, you could also head any post containing a solution as a "SPOILER".

There is a supporting file, `palin.erl`, available (as a zip file) under 'Downloads' below.

Checking for palindromes

The simplest check for palindromes reverses a list and checks to see whether the result is equal to the list we started with:

```
pal_check(String) -> String==lists:reverse(String).
```

but we can make it more sophisticated by removing punctuation, and transforming capital letters into small letters first, using:

```
rem_punct(String) -> lists:filter(fun (Ch) ->
                                      not(lists:member(Ch,"\"\'\t\n
"))
                                    end,
                                    String).

to_small(String) -> lists:map(fun(Ch) ->
                                  case ($A =< Ch andalso Ch =< $Z)
of
                                      true -> Ch+32;
                                      false -> Ch
                                   end
                                 end,
                                 String).

palindrome_check(String) ->
    Normalise = to_small(rem_punct(String)),
    lists:reverse(Normalise) == Normalise.
```

A palindrome checking server

Define a function `server/1` that accepts messages of the form

```
{check,"Madam I\'m
Adam"}
```

and returns results like

```
{result,"\"Madam I\'m Adam\" is a
palindrome"}
```

If it is sent any other format of message, such as

```
stop
```

the server should stop, by terminating its operation. The argument to `server/1` should be the `Pid` of the process to which results are to be returned.

Running the server

You can run the server from the Erlang shell. Before you spawn it you'll need to get the `Pid` of the shell itself, like this:

```
1> Self = self().
<0.32.0>
```

You can then pass this value to the appropriate call to spawn as argument for the server process.

Going further

You have now covered the key parts of the exercise, and it's OK to move on to the next step now. If, however, you would like to take the exercise a bit further, here are some suggestions. We'll come back to the ideas introduced here in later steps of the course, so you are not missing anything if you skip them now

Modifying your server

Can you modify the definition of your server to take requests from multiple clients? For this to work, each client will need to pass its `Pid` as part of each message, and the server will have to extract that information as the destination of its reply.

You can try this out with two clients: one being you at the Erlang shell, another being a client process that you will need to define and spawn from the shell. What argument(s) will you need to pass to the `client` on startup?

Replicating the server

Suppose that you need to replicate the server because of the volume of traffic. One way of doing this is to put in place a front end which distributes work to the two servers, which then return their results directly to the clients.

Diagnostic information

In order to see how your replicated system is working, it would be useful to instrument your server.

Please add appropriate `io:format/2` statements to the server code to show which requests have gone to which server.
