Programmeertalen: Erlang

A distributed functional programming language

Jeroen Koops, Ana Oprescu

Universiteit van Amsterdam

3 maart 2020

Programmeertalen: where are we now?

	Programming Language	**Lecture Concepts**[1]	**Lecture Best Practices**[2]	
Week 1	Bash	Tue 4/2	Thu 6/2	
Week 2	Haskell	Tue 11/2	Thu 13/2	
Week 3	Prolog	Tue 18/2	Thu 20/2	
Week 4	Python	Tue 25/2	Thu 27/2	
Week 5	**Erlang**	**Tue 3/3**	**Thu 5/3**	
Week 6	Go	Tue 10/3	Thu 12/3	
Week 7	C++	Tue 17/3	Thu 19/3	
Week 8	Exam	Tue 24/3	09:00-12:00	Sporthal 2

[1] 11:00-13:00, CWI Turingzaal
[2] 13:00-15:00, see DataNose

Today's menu

Erlang

- Developed by Ericsson (see also Open Telecom Platform)
  - Er-lang, but also Agner Erlang
- First developed for use in telecom switches, but also found use in other applications
- First appeared in 1986, released as open source in 1998
- Joe Amstrong (co-creator) gets his PhD from KTH in 2003
  `http://erlang.org/download/armstrong_thesis_2003.pdf`
- Distributed, functional
- 'Pragmatically' functional
- Dynamic and strong typing
- Fault tolerant
  - Let it crash!
- Bytecode runs in a VM

Characteristics

- Functional programming language
- Strict evaluation
- Prolog-like syntax
- **Support for concurrency**
- Joe Amstrong: "Write once, run forever" :)
- Runtime code replacement for nine nines uptime
- Both a language (Erlang) and a platform (OTP)

Learning Erlang – recommended reading[3]

- Programming Erlang, Software for a Concurrent World (Armstrong)
- Learn You some Erlang for Great Good! `http://learnyousomeerlang.com/content`
- An Erlang Course `http://www.erlang.org/course/course.html`
- Erlang/OTP documentation `http://www.erlang.org/doc/`
- Erlang programming guidelines `http://www.erlang.se/doc/programming_rules.shtml`

[3]Some slides and examples in this presentation originate from this material.

Functional vs Imperative

- In a purely functional language, the result of the invocation of a function depends solely on the values of the arguments
- In an imperative language, the result of the invocation of a function may depend on state as well

Functional vs Imperative, example

imperative.pseudo

```
1> stack s = stack_create()
2> stack_push(s, 42)
3> stack_push(s, 2002)
4> stack_pop(s)
2002
5> stack_pop(s)
42
```

Functional vs Imperative, example

imperative.pseudo

```
1> stack s = stack_create()
2> stack_push(s, 42)
3> stack_push(s, 2002)
4> stack_pop(s)
2002
5> stack_pop(s)
42
```

functional.pseudo

```
1> stack s = stack_create()
2> s0 = stack_push(s, 42)
3> s1 = stack_push(s0, 2002)
4> {s2, e0} = stack_pop(s1)
{ Stack@10029004, 2002 }
5> {s3, e1} = stack_pop(s2)
{ Stack@10048006, 42 }
```

Functional vs Imperative, example 2

imperative.pseudo

```
1> get_date()
Tue Mar  3 08:45:44 CET 2020
2> get_date()
Tue Mar  3 08:45:47 CET 2020
3> generate_random()
0.1773876
4> generate_random()
0.9177282
```

Functional vs Imperative, example 2

imperative.pseudo

```
1> get_date()
Tue Mar  3 08:45:44 CET 2020
2> get_date()
Tue Mar  3 08:45:47 CET 2020
3> generate_random()
0.1773876
4> generate_random()
0.9177282
```

functional.pseudo

```
???
```

Erlang is "pragmatically" functional

- It allows some functions with side effects
- For example, sending and receives messages
- There are library functions to do I/O, get the date, etc.
- Developers will spend most of their time writing purely functional code

Strict vs lazy evaluation

- In a functional language with lazy evaluation, there is no telling when a function will be evaluated
- In a functional language with strict evalution, a function is evaluated in order
- Erlang has strict evaluation

Starting Erlang

Open a terminal and start the Erlang runtime system using **erl**

```
$ erl
Erlang/OTP 21 [erts-10.2.3] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe] ...

Eshell V10.2.3  (abort with ^G)
1>
```

Files, modules, comments and Hello World!

hello.erl

```
-module(hello).
-export([hello_world/0]).

% Say hello!
hello_world() -> io:fwrite("Hello World!\n").
```

```
Eshell V6.4  (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:hello_world().
Hello World!
ok
```

Data Types

- Numbers: integers and floats
- Variables:
  - assignment only once "binding", begin with uppercase
  - only in the shell environment, variables may be reset: f(Variable)
- Atoms: begin with lowercase
  - enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @
  - reserved words: after and andalso band begin bnot bor bsl bsr bxor case catch cond div end fun if let not of or orelse query receive rem try when xor
  - atom table is not garbage collected!
  - Functions available to convert from strings to atoms, but be careful with externally supplied strings
- "Boolean"
- Tuples
- Lists
- Binaries

Strings

- No explicit datatype for strings, both lists and binaries can be used as strings.

```
34> L = [ 72, 97, 108, 108, 111 ].
"Hallo"
35> B = << 72, 97, 108, 108, 111 >>.
<<"Hallo">>
36> U = << 195, 165, 108, 108, 111 >>.
<<"éllo"/utf8>>
37> S = "Hallo".
"Hallo"
38> SB = <<"Hallo">>.
<<"Hallo">>
```

Number literals

- Use 16# for hexadecimal numbers
- ... or 8# for octal numbers
- Use $ to get the value of a character

```
50> 16#a9.
169
51> 16#1000.
4096
52> 8#10.
8
53> 2#101010.
42
54> $a.
97
55> $\n.
10
```

(In)variables assignment

- Assignment is actually a comparison with special treatment of unbound variables.

```
7> A=5.
5
8> A=5.0.
** exception error: no match of right hand side value 5.0
9> A=2+3.
5
10> f(A).
ok
11> A=5.0.
5.0
```

- When assigning to a variable that is not used anymore, a compile-time warning is issued.
- This can be suppressed by having the variable name start with an underscore, or use just _ as the variable name.

(In)variables assignment - tuples

```
59> { A, B } = { 23, "Hello" }.
{23,"Hello"}
60> A.
23
61> B.
"Hello"
62> { C, C } = { 23, 23 }.
{23,23}
63> C.
23
64> { D, D } = { 23, 45 }.
** exception error: no match of right hand side value {23,45}
```

- No more "smartness", though, so `{ A, A+1 } = { 2, 3 }.` will not work.

Assignments as assertions

- Function `find_customer_by_name(Name)`.
- Returns `{ ok, CustomerId }` on success
- Or `{ error, not_found }` when not found

```
3> { ok, IdEva } = lecture:find_customer_by_name("Eva").
{ok,42}
4> IdEva.
42
5> { ok, IdLucas } = lecture:find_customer_by_name("Lucas").
** exception error: no match of right hand side value {error,not_found}
```

Arithmetic operations

```
1> 3 + 20.
23
2> 5 rem 4.
1
3> 19 div 3.
6
4> false and true.
false
5> true + false.
** exception error: an error occurred when evaluating an arithmetic expres
     in operator  +/2
        called as true + false
6> not false.
true
7> 3 / 2.
1.5
```

And more, such as `or`, `xor`, `andalso` and `orelse`.

(In)equality

- Test for identical terms: =:= and =/=
- Test (in)equality with int/float conversion: == en /=

```
1> 1 >= 1.
true
2> 1 =< 1.0.
true
12> 4+1 == 2+3.
true
13> 4+1 =:= 2+3.
true
14> 4+1.0 =:= 2+3.
false
15> 4+1.0 == 2+3.
true
19> false < true.
true
6> wrong =:= 'wrong'.
true
```

Atoms and variables

```
3> 5+llama.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as 5 + llama
3> false + true.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as false + true
4> llama = 4.
** exception error: no match of right hand side value 4
5> Llama = 4.
4
6> 5 + Llama.
9
10> 4 == llama.
false
11> 4 == Llama.
true
12> 5 + true.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as 5 + true
```

Atoms and variables

- **Total ordering** is important! which order, not.
- number < atom < reference < fun < port < pid < tuple < list < bit string

```
13> true == llama.
false
12> f(Llama).
ok
16> 5 < true.
true
17> 5 < llama.
true
18> true < llama.
false
20> Llama < true.
* 1: variable 'Llama' is unbound
21> Llama = 4.
4
22> Llama < true.
true
23> Llama < 5.
true
26> ((false < true) and (true < Llama)) and (Llama < 5).
false
```

Atoms and variables

- **Total ordering** is important!

```
7> ((false < true) and (true < Llama)) and (Zebra = 6).
** exception error: bad argument
     in operator  and/2
        called as false and 6
28> ((false < true) and (true < Llama)) and (Zebra == 6).
* 1: variable 'Zebra' is unbound
29> ((false < true) and (true < Llama)) and ((Zebra = 6) == 6).
false
36> ((false < true) and (true < Llama)) andalso ((Lion = 9) == 6).
false
37> Lion.
* 1: variable 'Lion' is unbound
38> ((false < true) and (true < Llama)) and ((Lion = 9) == 6).
false
39> Lion.
9
```

Today's menu

Lists

Collection of elements, which may be of different types.

Lists

Collection of elements, which may be of different types.

```
1> X = [1,2,3,4,5,6].
[1,2,3,4,5,6]
2> [H|T] = X.
[1,2,3,4,5,6]
3> H.
1
4> T.
[2,3,4,5,6]
5> hd(X).
1
6> tl(X).
[2,3,4,5,6]
7> length(X).
6
8> Y = [ 1, { "Newton", "Kepler" }, stop ].
[1,{"Newton","Kepler"},stop]
```

Lists

Two lists can be concatenated using the ++ operator. The -- removes members that are in both lists from the left list. Both operators are *right-associative*.

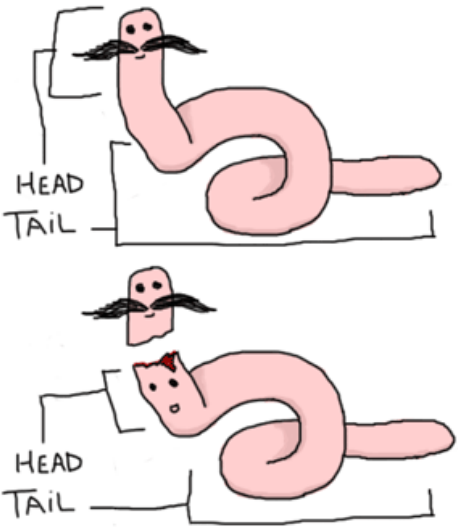
```
1> [1,2,3] ++ "abc" ++ "def".
[1,2,3,97,98,99,100,101,102]

2> [1,2,3] -- [1,2].
[3]

3> [1,2,3] -- [1,3].
[2]

4> [1,2,3,1] -- [1,3].
[2,1]

5> [1,2,3,1] -- [1,3,1].
[2]
```

Lists

Similar to Prolog:

```
1> [1,2,3,4,5,6,7].
[1,2,3,4,5,6,7]
2> [1,2 | [3,4,5,6,7]].
[1,2,3,4,5,6,7]
3> [1,2,3 | [4,5,6,7]].
[1,2,3,4,5,6,7]
4> [1,2 | [[3,4,5,6,7]]].
[1,2,[3,4,5,6,7]]
5> [[1,2] | [3,4,5,6,7]].
[[1,2],3,4,5,6,7]
```

- Consult the documentation for more built-in functions (BIFs)
  `http://www.erlang.org/doc/man/lists.html`
- Watch out for improper lists `[1|2]`.



List comprehensions

Similar to Haskell:
NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN, Condition1, Condition2, ... ConditionM]

```
1> L = lists:seq(1,20).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

3> [ 2 * X || X <- L, X rem 2 =:= 1].
[2,6,10,14,18,22,26,30,34,38]

4> [ { X, Y } || X <- [ 1,2 ], Y <- [ a, b ]].
[{1,a},{1,b},{2,a},{2,b}].

5> Weather = [{toronto, rain}, {montreal, storms}, {london, fog}, {paris,
...
6> RainyPlaces = [X || {X, rain} <- Weather].
[toronto,amsterdam]
```

List comprehensions

Pythagorean triples?

List comprehensions

Pythagorean triples?

```
5> L = lists:seq(1,20).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

6> [ {X,Y,Z} || X <- L, Y <- L, Z <- L, X*X+Y*Y =:= Z*Z, X < Y].
[{3,4,5},{5,12,13},{6,8,10},{8,15,17},{9,12,15},{12,16,20}]
```

Question time!

Define your own version of `lists:reverse()/1`.

```
1> lists:reverse("Hoooi").
"ioooH"
```

Question time!

Define your own version of `lists:reverse()/1`.

toy.erl

```
-module(toy).
-export([tail_reverse/1,myreverse/1]).

tail_reverse(L) ->
            tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).

myreverse([]) -> [];
myreverse([H|T]) -> myreverse(T)++[H].
```

```
2> lecture:tail_reverse("Hoooi").
"ioooH"
```

Question time!

Define your own version of `lists:reverse()/1`.

toy.erl

```
-module(toy).
-export([tail_reverse/1,myreverse/1]).

tail_reverse(L) ->
            tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).

myreverse([]) -> [];
myreverse([H|T]) -> myreverse(T)++[H].
```

```
2> lecture:myreverse("Hoooi").
"ioooH"
```

Question time!

Define your own version of `lists:member/2`.

```
1> lists:member(1,[1,2,3]).
true
2> lists:member(4,[1,2,3]).
false
```

Question time!

Define your own version of `lists:member/2`.

```
1> lists:member(1,[1,2,3]).
true
2> lists:member(4,[1,2,3]).
false
```

lecture.erl

```
member(_, [])    -> false;
member(E, [E|_]) -> true;
member(E, [_|T]) -> member(E, T).
```

```
3> lecture:member(1,[1,2,3]).
true
4> lecture:member(4,[1,2,3]).
false
```

Today's menu

Pattern-matching and unbound-variables

lecture.erl

```
lucky(4) -> lucky;
lucky(6) -> doomed;
lucky(7) -> very_lucky;
lucky(_) -> not_so_lucky.
```

```
1> lecture:lucky(6).
doomed
2> lecture:lucky(7).
very_lucky
3> lecture:lucky(10).
not_so_lucky
```

Functions

- Function is defined by module-name, function-name, and arity
- Functions in same module and with same name but with different arity are completely different functions.
- Erlang does not have default-valued arguments, but this is commonly implemented by using more than one function.

lecture.erl

```
-export([ http_get/1, http_get/2 ]).

http_get(Url) -> http_get(Url, []).

http_get(Url, Options) -> ...
```

Functions - let it crash

- It is common to write a function that accepts only certain terms as arguments.
- When the function is called with a term that doesn't match, this will cause an exception.
- This is a feature, not an error.

http.erl

```
-module(http).

http_do({ get, Url }) -> ...
http_do({ post, Url, ContentType, Data }) -> ...
```

Guards, Guards!

Older than 18?

```
old_enough(0)  -> false;
old_enough(1)  -> false;
old_enough(2)  -> false;
...
old_enough(16) -> false;
old_enough(17) -> false;
old_enough(_)  -> true.
```

Guards, Guards!

Older than 18?

```
old_enough(0)  -> false;
old_enough(1)  -> false;
old_enough(2)  -> false;
...
old_enough(16) -> false;
old_enough(17) -> false;
old_enough(_)  -> true.
```

Using guards:

```
old_enough(X) when X >= 18 -> true;
old_enough(_)              -> false.
```

Guards, Guards! (2)

Older than 18, younger than 104

```
right_age(X) when X >= 18, X =< 104 -> true;
right_age(_)                        -> false.
```

Watch out for , versus ;! similar to `andalso` versus `orelse`. However, only `andalso` and `orelse` can be nested in guard statements.

```
wrong_age(X) when X < 18; X > 104 -> true;
wrong_age(_)                      -> false.
```

If-statements Erlang-style

```
answer_to_life(X) ->
    if
        X =:= 42  -> galaxy;
        X =:= 666 -> lucifer;
        true      -> false  % else-statement Erlang-style
    end.
```

- It is not necessary to handle all cases, it is common to let the function crash with a match error when unexpected input is encountered.

Case

```
lucky_case(X) ->
    case X of
        4 -> lucky;
        6 -> doomed;
        7 -> very_lucky;
        _ -> not_so_lucky
    end.
```

Case with when

- Case with Pattern Matching

```
beach(Temperature) ->
  case Temperature of
    {celsius, N} when N >= 20, N =< 45     -> 'favorable';
    {kelvin, N} when N >= 293, N =< 318    -> 'scientifically favorable';
    {fahrenheit, N} when N >= 68, N =< 113 -> 'favorable in the US';
    _                                      -> 'avoid beach'
  end.
```

- Again: it is not necessary to handle all cases, it is common to let the function crash with a match error when unexpected input is encountered.

Function and case equivalence

- The following two functions are equivalent

test.erl

```
test1(a) -> "Say A";
test1(b) -> "Say B".

test2(S) ->
    case S of
        a -> "Say A";
        b -> "Say B"
    end.
```

Input and output

lecture.erl

```
length_input() ->
   X = io:get_line("Input sentence: "),
   L = length(X),
   io:format("Sentence: ~s, length: ~w~n",[X,L]).
```

- Consult `http://erlang.org/doc/man/io.html` for information regarding functions in the `IO-module`.
- Note that `io:format/2` takes a list as second argument, this can easily be overlooked

Today's menu

Defined functions as arguments

lecture.erl

```
one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

How can we call add/2 such that the outcome is 3?

Defined functions as arguments

lecture.erl

```
one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

How can we call add/2 such that the outcome is 3?

```
1> lecture:add(fun lecture:one/0,fun lecture:two/0).
3
```

- Both `module:function/arity` and `function/arity` are allowed

Anonymous functions

Syntax:

```
fun(Args1) ->
Expression1, Exp2, ..., ExpN;
(Args2) ->
Expression1, Exp2, ..., ExpN;
(Args3) ->
Expression1, Exp2, ..., ExpN
end
```

Example:

```
1> lists:map(fun(A) -> A + 1 end, [1,2,3]).
[2,3,4]

2> lists:filter(fun(A) -> A > 10 end, lists:seq(1,20)).
[11,12,13,14,15,16,17,18,19,20]
```

Anonymous functions with guards

Syntax:

```
fun(Args1) when Guard1, Guard2, ... ->
Exp1, Exp2, Expn;
fun(Args2) when Guard3, Guard4, ... ->
...
end.
```

Anonymous functions, assign to variable

Syntax:

```
F = fun(...) ->
    ...
end.

F(42).
```

Anonymous recursive functions

Example:

```
21> F = fun Factorial(0) -> 1;
21>         Factorial(N) -> N * Factorial(N-1)
21> end.
#Fun<erl_eval.30.127694169>

22> F(6).
720
```

Anonymous functions in fold

Erlang has `foldl` and `foldr`, but with a different syntax than Haskell.

```
1> lists:foldl(fun(X,Y) -> X+2*Y end, 4, [1,2,3]).
43

2> lists:foldr(fun(X,Y) -> X+2*Y end, 4, [1,2,3]).
49
```

- Use exceptions to 'early-out' of the fold without having to process the rest of the list

Today's menu

Spawning processes

Processes can be started using `spawn`. `spawn` returns the Process Identifier (PID) of the new process.

```
1> F = fun() -> io:format("Hello process!~n") end.
#Fun<erl_eval.20.90072148>
2> F().
Hello process!
ok
3> spawn(F).
Hello process!
<0.43.0>
```

Spawning processes (2)

lecture.erl

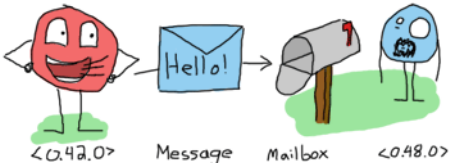
```
hello_process(X) ->
    io:format("Hello ~w~n",[X]).
```

```
1> spawn(lecture, hello_process,[world]).
Hello world
<0.108.0>
```

- When spawning a process with a `fun`, it is not possible to pass arguments, you need the `module, function` version for that
- It is also possible to spawn processes on other nodes (virtual machine), which may or may not be running on a different host.
- There is also a `spawn_link` function, which will send a message to the parent when then child process exits.

Sending messages

Using the !-operator (bang symbol) you can send messages to processes.




```
3> self() ! hello.
hello
4> self() ! world.
world
5> flush().
Shell got hello
Shell got world
ok
```

Receiving messages

lecture.erl

```
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

Receiving messages

lecture.erl

```
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

```
1> P = spawn(lecture,hi_process,[]).
<0.111.0>
2> P ! {print, "Hello!"}.
Message: Hello!
{print,"Hello!"}
3> P ! {write, "Hello!"}.
What?
{write,"Hello!"}
4> P ! {stop}.
What?
```

Receiving messages

lecture.erl

```
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

```
8>  P ! {stop}.
What?
{stop}
9>  P ! stop.
Goodbye!
stop
10>  P ! {stop}.
{stop}
```

Receiving messages

lecture.erl

```
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

- Compiler takes care of tail-recursion optimisation, so each new iteration of `hi_process` will *not* create a new stackframe. This is fundamental to how processes work in Erlang.
- Because `hi_process` is called without its module-name, it is impossible to update the module at runtime.

A functional mindset of concurrency

How can we create ten processes at once?

A functional mindset of concurrency

How can we create ten processes at once?

lecture.erl

```
hi_process(N) ->
    receive
        {print, Message} -> io:format("~w: Message: ~s~n",[N,Message]),
                            hi_process(N);
        stop             -> io:format("Goodbye from ~w!~n",[N]);
        _                -> io:format("~w What?~n",[N]),
                            hi_process(N)
    end.
```

```
1> L = [ spawn(lecture, hi_process, [X]) || X <- lists:seq(1,10) ].
[<0.46.0>,<0.47.0>,<0.48.0>,<0.49.0>,<0.50.0>,<0.51.0>,
 <0.52.0>,<0.53.0>,<0.54.0>,<0.55.0>]
```

A functional mindset of concurrency (2)

How do we send a message to all ten processes?

A functional mindset of concurrency (2)

How do we send a message to all ten processes?

```
2> lists:foreach( fun(P) -> P ! {print, "Hello"} end, L).
1: Message: Hello
2: Message: Hello
3: Message: Hello
4: Message: Hello
5: Message: Hello
6: Message: Hello
8: Message: Hello
9: Message: Hello
7: Message: Hello
10: Message: Hello
ok
```

There is no guarantee that the messages would arrive in this order!

A functional mindset of concurrency (3)

```
3> lists:foreach( fun(P) -> P ! {write, "Hello"} end, L).
2 What?
3 What?
5 What?
7 What?
8 What?
1 What?
10 What?
4 What?
6 What?
9 What?
ok
```

There is no guarantee that the messages would arrive in this order!

A functional mindset of concurrency (4)

```
3> lists:map( fun(P) -> P ! stop end, L).
Goodbye from 8!
Goodbye from 9!
Goodbye from 10!
Goodbye from 1!
Goodbye from 2!
Goodbye from 3!
Goodbye from 4!
Goodbye from 5!
Goodbye from 6!
Goodbye from 7!
[stop,stop,stop,stop,stop,stop,stop,stop,stop,stop]
```

There is no guarantee that the messages would arrive in this order!

Round Robin: the rotate game!

lecture.erl

```
rotate([H|T]) -> T ++ [H].

rotate_game(X) ->
    receive
        stop     -> io:format("~w game is over~n",[X]);
        {25, Ps} -> io:format("~w ends game~n",[X]),
                    lists:foreach( fun(P) -> P ! stop end, Ps);
        {N, Ps } -> io:format("~w increases ~w~n",[X, N]),
                    hd(Ps) ! {N + 1, rotate(Ps)},
                    rotate_game(X)
    end.
```

Round Robin: the rotate game!

lecture.erl

```
rotate([H|T]) -> T ++ [H].

rotate_game(X) ->
    receive
        stop     -> io:format("~w game is over~n",[X]);
        {25, Ps} -> io:format("~w ends game~n",[X]),
                    lists:foreach( fun(P) -> P ! stop end, Ps);
        {N, Ps } -> io:format("~w increases ~w~n",[X, N]),
                    hd(Ps) ! {N + 1, rotate(Ps)},
                    rotate_game(X)
    end.
```

```
1> Ps = [spawn(lecture, rotate_game, [X]) || X <- lists:seq(1,10) ].
2> hd(Ps) ! {0, Ps}.
```

Selective receive

lecture.erl

```
receive
    { print, Message } ->
        io:format("message received: ~p~n", [ Message ]);
    stop ->
        halt(0)
end.
```

Selective receive

lecture.erl

```
receive
    { print, Message } ->
        io:format("message received: ~p~n", [ Message ]);
    stop ->
        halt(0)
end.
```

- You would expect this `receive` to crash with a match error when receiving anything apart from a `{ print, Message }` or a `stop`, but that is *not* the case
- Instead the message remains in the process' message-box, and can be received later on.

Receive with timeout

lecture.erl

```
F = fun Server() ->
        receive
            X -> io:format("Thanks for ~p~n", [ X ])
        after
            5000 -> io:format("Please send me something!")
        end,
        Server()
end.
```

Receive with timeout

lecture.erl

```
F = fun Server() ->
        receive
            X -> io:format("Thanks for ~p~n", [ X ])
        after
            5000 -> io:format("Please send me something!")
        end,
        Server()
end.
```

- This is lots of fun, but in production code, you'll hardly every do things like this.
- Instead, you'll use standard OTP behaviours to do the message passing

Processes are cheap

- Processes in Erlang are cheap with respect to both memory and CPU consumption
- Therefore it is entirely acceptable to start a new process for each
  - ... user if you have thousands of users
  - ... network-connection of you have dozens of connections
  - ... incoming message if you process thousands of messages per second

Processes provide fault isolation

- When sending a message from one process to another, this will always be a copy
- Processes never share any data, they send each other copies
- Therefore, when a process crashes, it can never leave corrupted data behind for other processes
- This is fundamental to how Erlang handles faults

Message passing enables concurrency

- Since processes don't share data, there is never the need to synchronize access
- This means that in a multi-processor machine, or in a cluster consisting of multiple machines, processes can run truly concurrently.

Behaviours

- Behaviours in Erlang are equivalent to interfaces in, say, Java
- One module declares itself to be a behaviour
- Other modules declare themselves to implement that behaviour
- Compile-time warning when not implementing (all of) the callbacks mandated by the behaviour

Behaviours

calculator.erl

```
-module(calculator).
-export([ behaviour_info/1 ]).

behaviour_info(callbacks) ->
    [ { add, 2 },
      { subtract, 2 } ].
```

calculator_impl.erl

```
-module(calculator_impl).
-behaviour(calculator).
-export([ add/2, subtract/2 ]).

add(A, B) -> A + B.

subtract(A, B) -> A - B.
```

Behaviours

calculator.erl

```
-module(calculator).
-export([ behaviour_info/1 ]).
-export([ do_calculation/1 ]).

behaviour_info(callbacks) ->
    [ { add, 2 },
      { subtract, 2 } ].

do_calculation(Implementation) ->
    X = Implementation:add(5, 6),
    Y = Implementation:subtract(X, 1),
    Y.
```

- Note: no runtime check that `Implementation` actually implements all callbacks from the `calculator` behaviour.

OTP Behaviours

Erlang conceptually distinguishes between work and supervision

Hierarchical organization of code to deliver fault-tolerance

OTP Behaviours are formalizations of patterns, i.e., servers, finite-state machines, event handlers

- generic part: behaviour module
- specific part: callback module, supplied by the developer

OTP behaviour example: gen_server

gen_server implements a server-process, typically managing one resource

This resource could be a key/value store, a TCP socket, a user-session

In order to "be" a gen_server, your module must implement the gen_server behaviour:

- `init(Args) -> { ok, State }`
- `handle_call(Call, From, State) -> { reply, Reply, NewState } | {noreply, NewState }`
- `handle_cast(Cast, State) -> { noreply, NewState }`
- `handle_info(Info, State) -> { noreply, NewState }`

gen_server example: Key/Value server (1)

key_value_server.erl

```
-module(key_value_server).

-behaviour(gen_server).

-export([ start/1, write/3, read/2 ]).

-export([ init/1, handle_call/3, handle_cast/2, handle_info/2 ]).

start(DefaultValue) -> gen_server:start(key_value_server, DefaultValue, []).

write(Pid, Key, Value) -> gen_server:cast(Pid, { write, Key, Value }).

read(Pid, Key) -> gen_server:call(Pid, { read, Key }).
```

gen_server example: Key/Value server (2)

key_value_server.erl

```
init(DefaultValue) -> { ok, { DefaultValue, [] } }.

handle_call({ read, Key }, _From, { DefaultValue, KeyValues }=State) ->
    { reply, find(Key, KeyValues, DefaultValue), State }.

handle_cast({ write, Key, Value }, { DefaultValue, KeyValues }) ->
    { noreply, { DefaultValue, [ { Key, Value }|KeyValues ] } }.

handle_info(_, State) ->
    { noreply, State }.
```

gen_server example: Key/Value server (3)

key_value_server.erl

```
find(_Key, [], DefaultValue) -> DefaultValue;
find(Key, [ { Key, Value }|_ ], _DefaultValue) -> Value;
find(Key, [ _|T ], DefaultValue) -> find(Key, T, DefaultValue).
```

- Who can come up with a modification to allow setting the default value to a different value?

Supervisors

- Supervisors are processes that start, monitor and restart other processes
- `supervisor` is an OTP behaviour - the developer implements a callback that defines which processes must be started and what there restart strategy is
- Restart strategies are:
  - Just restart the failed process
  - Terminate all remaining processes, then restart everything
  - Terminate processes after the failed process, then restart the failed process and the processes that came after
- Supervisors can supervise other supervisors
- After too many restart attempts of a child, the supervisor itself terminates

Distributed Erlang

Erlang conceptually maps a VM to a node

Nodes may be named `name@host`

- long names: `host` is the full host name
- short names: `host` is the first part of the host name

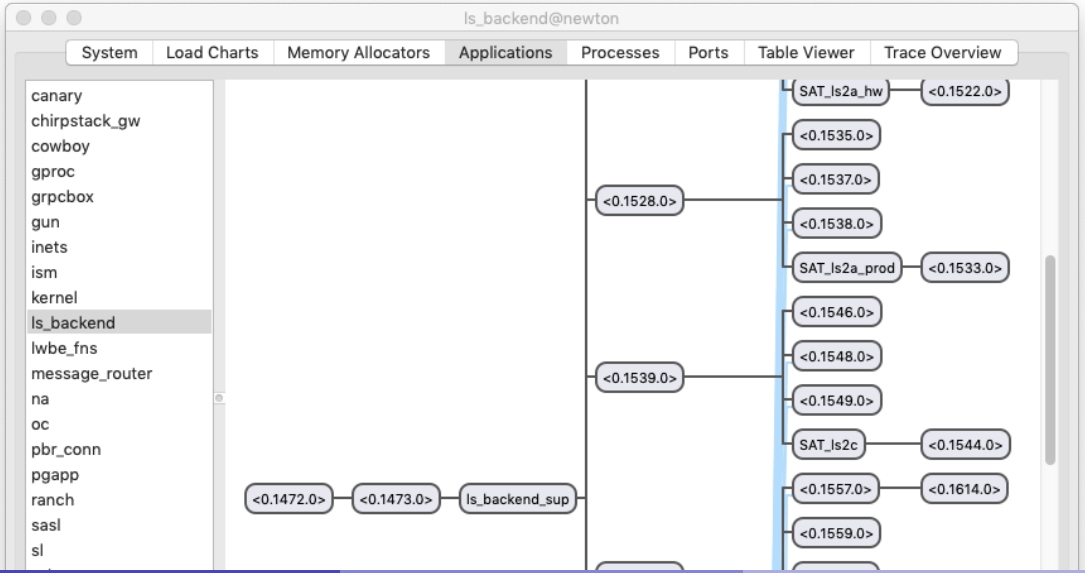
Processes may be spawned on another node

Today's menu

Tips 'n tricks

- Use `halt().` or Ctrl+\to terminate the Erlang interpreter.
- Reset a variable: `f/1`
- Reset all variables: `f/0`
- Server Behaviour: `http://erlang.org/doc/design_principles/gen_server_concepts.html`
- Supervisor Behaviour: `http://erlang.org/doc/design_principles/sup_princ.html`
- make sure the epmd is running before naming a node
- Use rebar3 when using a real Erlang application
- It is possible to provide type information by using `-spec` directives

Example of running Erlang release



Next week

Go!