Essentials

Heads Up

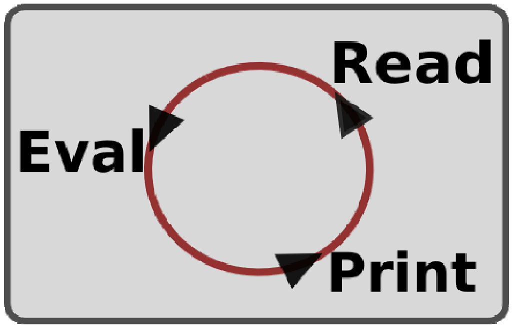
- Data Types
- Pattern Matching
- Modules
- Processes

Services, Data/State

(super important, important, gotchas)

The Shell (super important)

- Single most important tool for Erlang developers!
- Use to experiment, learn, prove/disprove
- Important to know:
  - `erl` starts the shell!
  - `ctrl-c, ctrl-c` stops the shell abruptly!
  - `q().` shuts down cleanly
  - Expressions always end in a period!



A Powerful Calculator

- You can add!

```
1> 1 + 1.
2
```

- You can assign values to variables!

```
2> X = 2.
2
3> X + 1.
3
```

Integers

- Basically what you’d expect
- But, whah???

```
4> Y =
1111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111.
```

- On no he di’int!

```
5> Y + 1.
1111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111112
```

Floats

- Pretty much what you’d expect
- Alas, not freaky (plain ol’ 64 bit double-precision)

```
6> 1.1234567890123456789012345678901234567890.
1.1234567890123457
```

- An unusual conversion from float to int

```
1> X = 1.12345.
1.12345
2> is_float(X).
true
3> Y = erlang:trunc(X).
1
4> is_integer(Y).
true
```

Variable Assignments

```
1> X = 1.
1
2> Y = 2.
2
3> X = Y.
** exception error: no match
   of right hand side value 2
4> X = 3.
** exception error: no match of right hand
   side value 3
5> X = X = 1.
1
```

Variables

- Don’t vary!
- Capitalized!
- Not really an assignment – more of a “truth seeker” (or a “let” operation, or a definition)
- Erlang always makes you not be illogical

Strings

- Pretty much what you'd expect

```
1> Slogan = "Mongo DB is Web Scale".
"Mongo DB is Web Scale"
```

- But, wait, OMG, noooo!

```
2> io_lib:format("Mongo DB is ~s", ["Web Scale"]).
[77,111,110,103,111,32,68,66,32,105,115,32,"Web Scale"]
```

- Yep, strings are just lists!

```
3> is_string(Slogan).
** exception error: undefined
shell command is_string/1
4> is_list(Slogan).
true
```

ErlangCamp Chicago 2010

^G

Replacing Emacs

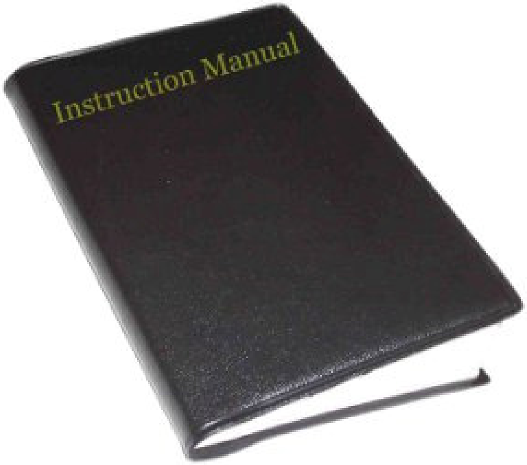
```
1> Note = "Add sharding to /dev/null".
"Add sharding to /dev/null"
2> file:write_file("note.txt", Note).
ok
3> {ok, Notes} = file:read_file("note.txt").
{ok,<<"Add sharding to /dev/null">>}
4> {ok, Notes2} = file:read_file("note2.txt").
** exception error: no match of right hand side
value {error,enoent}
```

Core Modules Are ~~Useful~~ Indispensable!

- Second most important Erlang developer tool: a decent module reference

  http://erlang.org/doc/man_index.html

  http://erldocs.com/
- The `file` module provides an interface to the file system
- We will keep using them – keep an eye out



ErlangCamp Chicago 2010

Atoms

- Basically named integers (symbols)
- Not capitalized

```
this_is_an_atom_that_i_just_made_up_cool_huh
```

- Except when single quoted

```
'This is an atom that I just made up, cool huh?'
```

- Atoms + tuples + pattern matching = very fundamental Erlang idiom

Tuples

- Pretty much what you'd expect
- Values separated by commas, surrounded by curly brackets

```
{This, is, a, "tuple"}
```

- Your basic "struct" data structure in Erlang
- Commonly used to "tag" values

```
{ok, "A good value"}
{error, "A bad value"}
{color, red}
```

Pattern Matching (super important)

```
1> {X, Y} = 1.
** exception error: no match
of right hand side value 1
2> {X, Y} = {1, 2}.
{1,2}
3> {X, Y} = {1, 2, 3}.
** exception error: no match of
right hand side value {1,2,3}
4> {ok, Value} = {ok, 123}.
{ok,123}
4> {ok, Value} = {not_okay, 123}.
** exception error: no match of
right hand side value {not_okay,123}
```

Binaries

- Sometimes used to represent strings

```
<<"This is a binary">>
```

- `is_binary` differentiates from `is_list`
- More typically used to work with actual binary data (e.g. bitstring syntax – not covered in essentials)
- Have protocol, will use binaries/bitstrings!

To Do List

- Let’s make a list!

```
1> ToDo = ["Shard /dev/null",
           "Learn Ruby",
           "Remove Bing from Phone"].
```

- Write to disk (easy to encode Erlang!)

```
2> file:write_file("todo.bin", term_to_binary(ToDo)).
ok
```

- Read from disk (easy to decode Erlang!)

```
3> {ok, Bin} = file:read_file("todo.bin").
{ok,<<131,108,0,0,0,3,107,0,15,83,104,97,114,100,32,47,
      100,101,118,47,110,117,108,108,107,0,10,...>>}
4> binary_to_term(Bin).
["Shard /dev/null","Learn Ruby","Remove Bing from Phone"]
```

Lists (important)

- Comma separated Erlang terms surrounded by brackets

  `[this, is, "a", {List, [of, stuff]}]`

- Used ehhhverywhere
- An important pattern for iterative operations
  - `lists:map/2`
  - list comprehension
- The basis for "associative array" structure in Erlang (`proplist`) `[{foo, "Foo"}, {bar, 123}]`

ErlangCamp Chicago 2010

Heads and Tails

(List “Cons”)

```
1> L1 = [2, 3].
[2,3]
2> L2 = [1|L1].
[1,2,3]
3> [H|T] = L2.
[1,2,3]
4> H.
1
5> T.
[1,2]
```

^G

Doing Something With Lists

- A basic sort

```
5> lists:sort(ToDo).
["Learn Ruby","Remove Bing from Phone","Shard
/dev/null"]
```

- Wait, that’s not what I want!

```
6> ToDo2 = [{2, "Shard /dev/null"},
            {3, "Learn Ruby"},
            {1, "Remove Bing from Phone"}].
7> lists:sort(ToDo2).
[{1,"Remove Bing from Phone"},
 {2,"Shard /dev/null"},
 {3,"Learn Ruby"}]
```

Taking Control Of Sort

- Default sort comparison uses “natural order” of Erlang term
- We don’t need no stinking natural order!

```
8> lists:sort(
      fun({P1, N1}, {P1, N2}) -> N1 < N2 end,
      ToDo2).
[{3,"Learn Ruby"},
 {1,"Remove Bing from Phone"},
 {2,"Shard /dev/null"}
```

Erlang Has Closures!

(well, *anonymous functions*)

- Use fun to create a function from within a function
- Anonymous functions have access to variables visible within their defining block, even when executed elsewhere

```
1> X = 1.
1
2> F = fun(Y) -> X + Y end.
#Fun<erl_eval.6.13229925>
3> F(10).
11
```

Modules!

- You basic module is very basic:

```
-module(fib).
```

- Must save in file named *MODULE* + “.erl”
- It compiles!

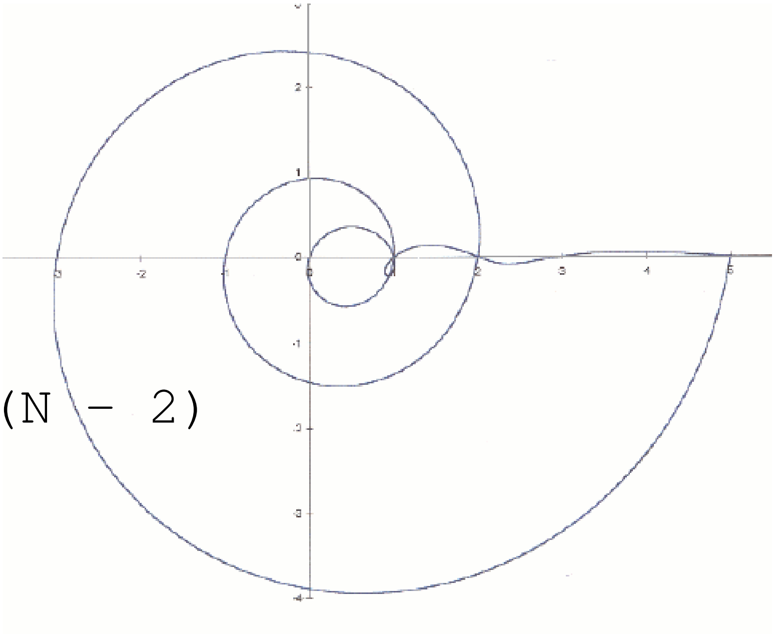
```
$ erlc fib.erl
$ ls
fib.beam  fib.erl
```

Let’s Do Something Useful

- The point of a module is to export functions
- Let’s do some math!

  http://en.wikipedia.org/wiki/Fibonacci_number

```
-export([f/1]).
f(N) ->
    case N of
        0 -> 0;
        1 -> 1;
        _ -> f(N - 1) + f(N - 2)
    end.
```



Case Statements

- Conditional value
- Looks like a switch statement in C, it’s not!

```
case Foo of
    1 -> one;
    2 -> two;
    {foo, F} -> a_foo;
    X -> {not_sure, X};
    _ -> dont_care
end
```

ErlangCamp Chicago 2010

Recursion (important)

- Function directly or indirectly calling itself
- It’s how you iterate!
- Tail optimization enables long running loops (eliminates unbounded stack growth)

Function Headers (important)

- Erlang functions support multiple headers, selected using pattern matching
- Use them when you can!
- Let's:

```
f(0) -> 0;
f(1) -> 1;
f(N) -> f(N - 1) + f(N - 2).
```

Guards

- Augment pattern matching in conditional clauses (case, function headers, etc.)
- Limited to “side effect free” operations

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) when N > 1 ->
  fib(N-1) + fib(N-2).
```

`lists:map/2` and List Comprehension

- Build a list from a list
- Very, very useful

```
1> lists:map(fun(I) -> I + 1 end, [3, 4, 5]).
[4,5,6]
2> [X + 1 || X <- [3, 4, 5]].
[4,5,6]
```

- Let's enhance our module:

```
seq(N) when N > 0 ->
    [f(I) || I <- lists:seq(1, N)].
```

State

- Where is state stored in Erlang anyway?
  - No globals!
  - No objects!
  - No module state!
- Seriously, where the is state stored in Erlang?
- Let’s start with a simple convention…

“Data Structure” Module

```
-module(todo).
-export([new/1, print/1]).

new(Name) ->
  {dont_mess_with_me, Name}.

print({dont_mess_with_me, Name}) ->
   io:format("TODO: ~s~n", [Name]).
```

Data Structure Modules
(Old School Encapsulation)

- Define a *don't mess with me*, private, data structure
- Provide operations that work with that structure
- Keep the interface simple – e.g.

```
-export([eat_and_enjoy/0]
```

ErlangCamp Chicago 2010

Enhancing To Do

```
-module(todo).
-export([new/1, new/2, set_priority/2, get_priority/1, print/1]).

new(Name) ->
  new(Name, 2).

new(Name, Priority) ->
  {todo, Name, Priority}.

set_priority(Priority, {todo, Name, _}) ->
   {todo, Name, Priority}.

get_priority({todo, _, Priority}) ->
   Priority.

print({todo, Name, Priority}) ->
   io:format("TODO: ~s (~b)~n", [Name, Priority]).
```

Data Structure Design

- Use a *tagged* tuple

  `{my_data_structure, "Some Value", 123}`
- Use a `new` function to create a structure
- Use `from_xxx` to create a structure based on another structure (e.g. `from_list`)
- Use appropriately named functions to mutate the structure (e.g. `store`, `insert`, `add`, `set`, `put`)
- Pick a convention and stick with it as much as possible!

```
P1=new()

P2=f(P1)

P3=f(P2)
```

ErlangCamp
Chicago 2010

Wrangling Tuples With Records

```
-record(todo, {name, priority}).

new(Name, Priority) ->
  #todo{name=Name, priority=Priority}.

set_priority(Priority, Todo) ->
  Todo#todo{priority=Priority}.

get_priority(#todo{priority=Priority}) ->
  Priority.

print(Todo) ->
  io:format("TODO: ~s (~p ~b)~n",
            [Todo#todo.name,
             Todo#todo.priority]).
```

Records

- Syntactic ~~sugar~~ sanity for working with tuples
- Try to keep private whenever possible
- Use proplists for public interfaces. E.g.

  ```
  your_mod:do([{what, be_awesome}, fast, light])
  ```

  is (almost always) better than:

  ```
  -include("your_mod.hrl") ...
  your_mod:do(#your_rec{what=be_awesome,
                        fast=true,
                        light=true})
  ```

- When your tuples stop being totally obvious, use records
- If you're using `erlang:element/2` (tuple positional access) somewhere, you probably want a record

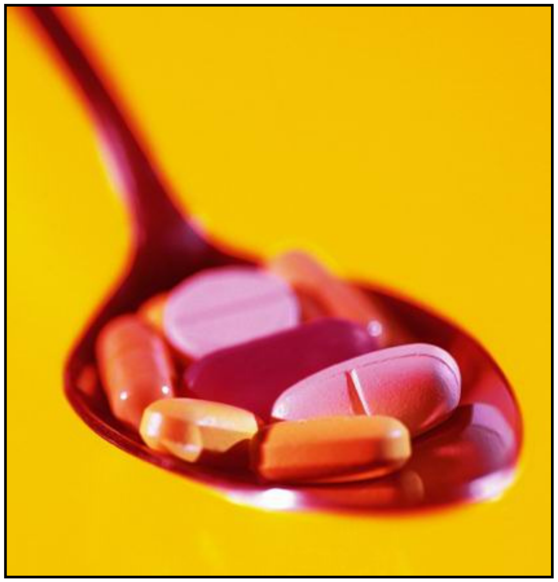
Side Effects

- A function that does something that effects the outside world has a “side effect”
- By contrast, here’s a crusty ol' classically side-effect --free function:

```
-module(seq).
-export([next/1]).
next(N) -> N + 1.
```

- Using it:

```
1> seq:next(0).
1
2> seq:next(1).
2
```



But What If…

- A super simple sequential function!

```
1> seq:next().
1
2> seq:next().
2
```

- That's freakin' me out! How'd you do that??

```
next() ->
    N = case erlang:get(seq_n) of
            undefined -> 0;
            LastN -> LastN
        end,
    Next = N + 1,
    erlang:put(seq_n, Next),
    Next.
```

Side Effects Can Lead To… Surprises

- You just saw a *side effect*
- Side effects are unavoidable in 99.9% of applications
- Printing text to stdout (`io:format/2`) is a side effect!
- Writing to a database is a side effect!
- Recognize a side effect when you use it – understand the implications – *don't be caught by surprise!*

A Real Life “This Could Happen To You” Surprise

```
1> random:uniform().
0.09230089279334841
2> random:uniform().
0.4435846174457203
3> random:uniform().
0.7230402056221108
4>
Eshell V5.7.5  (abort with ^G)
1> random:uniform().
0.09230089279334841
2> random:uniform().
0.4435846174457203
3> random:uniform().
0.7230402056221108
```

^G

Process Dictionary (gotcha)

- Place to store process specific values
- Use is a classic side effect in Erlang
- Use discouraged
- Use frowned upon
- Use is bad
- Don't use it
- Just put it down!

So What's a Process?

```
1> P = fun() -> io:format("~b~n",[seq:next()]) end.
#Fun<erl_eval.20.67289768>
2> F = fun() -> P(), P(), P() end.
#Fun<erl_eval.20.67289768>
3> F().
1
2
3
ok
4> Pid = spawn(F).
1
<0.41.0>
2
3
5> is_process_alive(Pid).
false
```

Processes

- Closest analogy is a posix thread
- No, wait… closest analogy is an OS system process
- Started using `spawn/1`
- Use `process_info` to get info – try it!

```
 6> process_info(spawn(F)).
```

- Use `exit` to send a term signal to a process – try it!

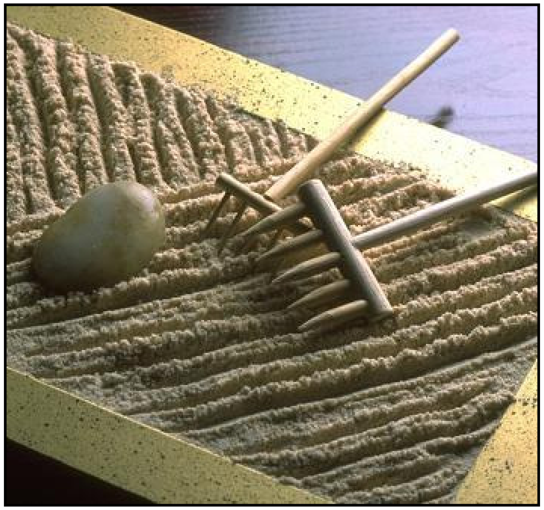
```
 7> exit(spawn(F), kill).
```

Remember Those Side Effects?

```
8> spawn(F), spawn(F), spawn(F).
1
1
2
1
2
<0.89.0>
3
2
3
3
```

Process Zen (super important)

- Yes, a unit of concurrency (like threads)
- But more like OS daemons (services) and batch jobs (scripts)
- Can be started and monitored
- Well-defined exit semantics
- Isolated from other processes
- You can send them stuff
- They can send you stuff, if they know you



Simple Process Communication

```
1> F = fun() -> receive {From, Msg} ->
    From ! {got, Msg, thanks} end end.
#Fun<erl_eval.20.67289768>
2> Pid = spawn(F).
<0.36.0>
3> flush().
ok
4> Pid ! {self(), "hey"}.
{<0.33.0>,"hey"}
5> flush().
Shell got {got, "hey", thanks}
ok
```

^G

More On Processes

- Indentified with a unique ID (Pid)
- Send messages using bang `!` operator

```
Pid ! "hey, what up?"
```

- Receive messages using a `receive` block
- Pattern matching and guards apply
- Keep going using a recursive call
- `self()` returns the Pid of the current process

Fixing Our Sequence Module

```
-module(seq).
-export([start/0, next/0]).

start() ->
  Pid = spawn(fun() -> loop(0) end),
  register(?MODULE, Pid),
  {ok, Pid}.

next() ->
  ?MODULE ! {next, self()},
  receive
    {next, N} -> N
  end.

loop(N) ->
  receive
    {next, From} ->
      Next = N + 1,
      From ! {next, Next},
      loop(Next)
  end.
```

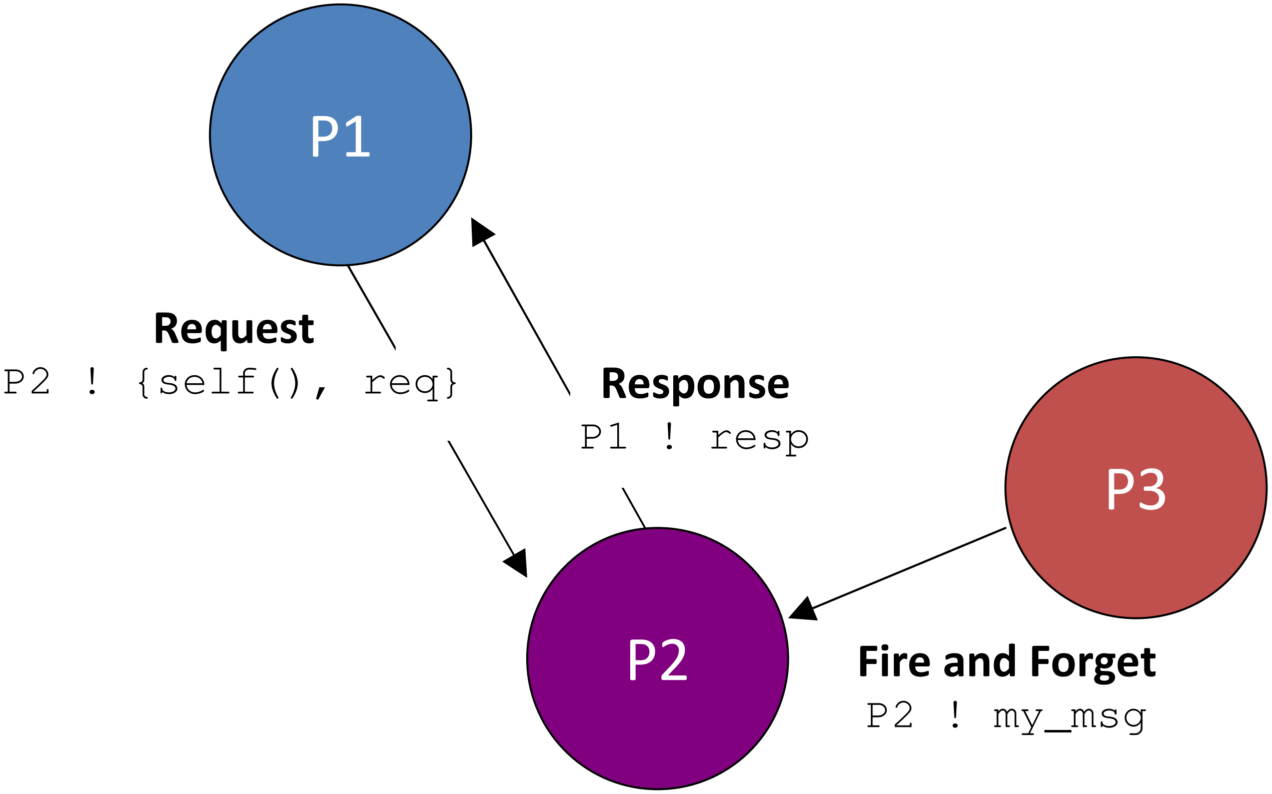
Much Better!

```
1> seq:start().
{ok,<0.35.0>}
2> P = fun() -> io:format("~b~n", [seq:next()]) end.
#Fun<erl_eval.20.67289768>
3> F = fun() -> P(), P(), P() end.
#Fun<erl_eval.20.67289768>
4> spawn(F), spawn(F), spawn(F).
1
2
3
<0.41.0>
4
5
6
7
8
9
```

What Just Happened?

- Used a single process to explicitly manage `seq` operations
- Similar to using an OS daemon (service)
- Process managed its state using a single `loop` function and tail recursion
- What looks like a function call (`seq:next`) is actually a wrapper for sending and receiving messages to a process

Basic Message Patterns



Request

`P2 ! {self(), req}`

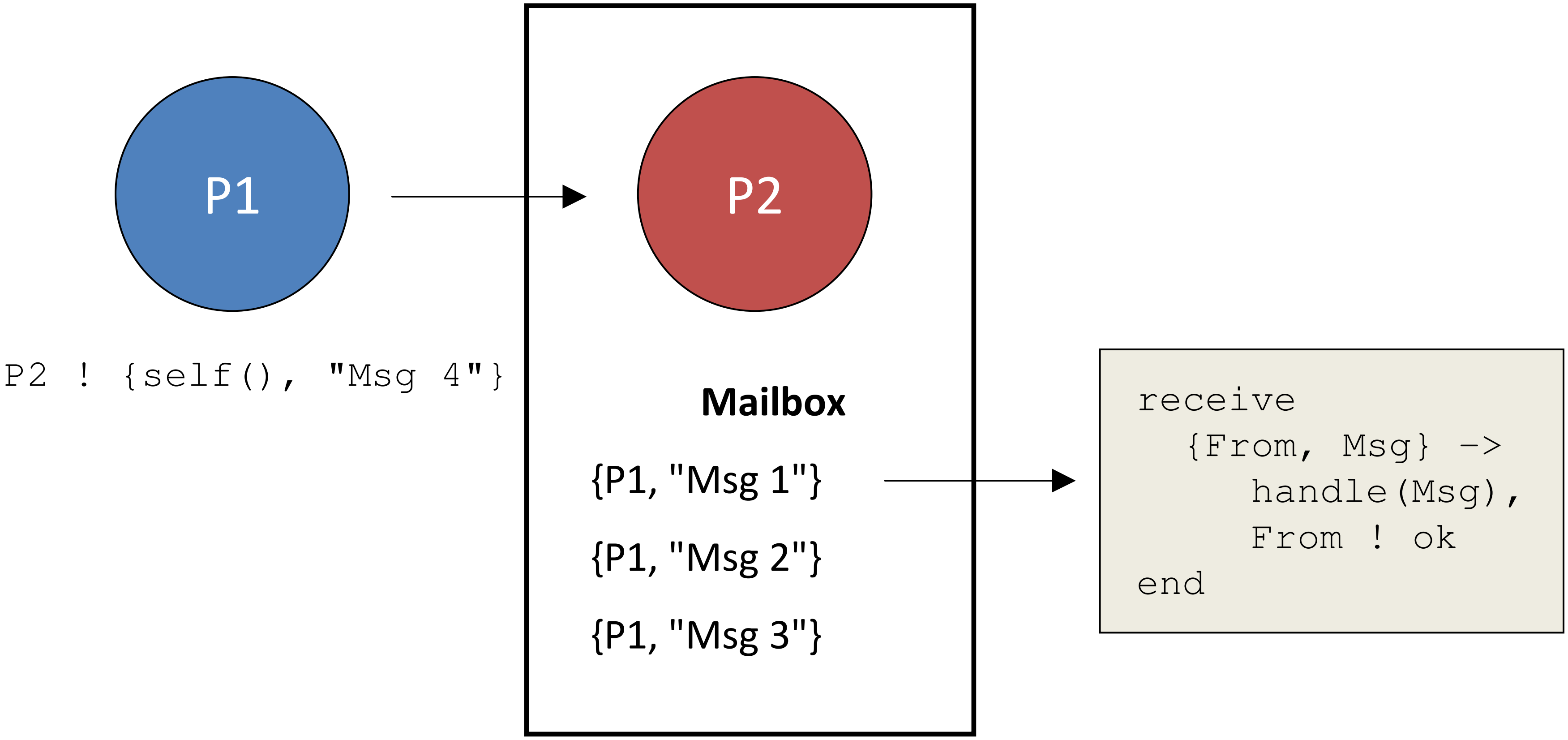
Response

`P1 ! resp`

Fire and Forget

`P2 ! my_msg`

Mailboxes



```
P2 ! {self(), "Msg 4"}
```

Mailbox

{P1, "Msg 1"}

{P1, "Msg 2"}

{P1, "Msg 3"}

```
receive
   {From, Msg} ->
       handle(Msg),
       From ! ok
end
```

Wrapping Up

- Erlang is a *functional* language, not imperative!
- Erlang’s language design encourages concise, clear, side-effect free code
- Modules:
  - Provide functions that perform closely related tasks
  - Can represent data structures
  - Can encapsulate process related services
- Processes are very similar to OS system processes – think of them this way (not as “light weight threads”)

ErlangCamp
Chicago 2010