Erlang
An introduction

Paolo Baldan
Linguaggi e Modelli per il Global Computing
AA 2018/2019
Erlang, in a slogan

Declarative (functional) language for concurrent and distributed fault-tolerant systems

Erlang

= Functions + Concurrency + Messages
Basics

- **Dynamic typing**
- **Light-weight processes**
- Total separation between processes (**no sharing**, naturally enforced by **functional style**)
- (Fast) **Message passing**
- **Transparent distribution**
Where does it come from?

- Old language with modern design
- Created in ’86 at Ericsson
- Open sourced in ’98
- “Programming with Erlang” published in ’07
- Getting more and more popular … also in different incarnations (cfr. Elixir)
Intended domain

• Highly concurrent and distributed (hundreds of thousands of parallel activities)
• (Soft) real time
• Complex software (million of lines of code)
• High Availability (down times of minutes/year – never down)
• Continuous operation (years)
• Continuous evolution / In service upgrade
Principles

- Concurrent programming
- Functional programming
- Concurrency Oriented programming
- Fault tolerance
- Multicore
- Scalability
Fault tolerance

• To make a system **fault tolerant** you need at least ...

• **two** computers (and some form of coordination)

• If one crashes, the other takes over
Fault tolerance

• To make a system very fault tolerant you need (at least) …

• many computers

• Which also addresses scalability
• Concurrency
• Distribution
• Fault tolerance
• Scalability

faces of the same medal (inseparable)
Models of concurrency

- Shared memory
  - Threads
  - Mutexes and locks

- Message passing
  - Processes
  - Messages
Shared Memory

• **Problems:**

  • What if a thread **fails** in the critical section?

  • Where do we (physically) locate the shared memory for distributed systems?
Message passing
Concurrency

- **No sharing** (share by communicating)
- **No locks, mutexes** etc
- (Lots of) processes (fault tolerant, scalable) communicating via pure message passing
Concurrency oriented programming

• The world is parallel and distributed

• The observation of the concurrency patterns and message channels as a way of designing an application

• Concurrency seen as a structuring paradigm (without being shy at creating processes)

Concurrency oriented programming (COP)
Message from …

"My first message is that concurrency is best regarded as a program structuring principle"

Sir Tony Hoare
Structured Oriented Programming
Transparent distribution

- Abstract from physical locations
Functional programming

- **Programs** are **expressions**, **computation** is **evaluation**
  
P1 $\rightarrow$ P2 $\rightarrow$ P3 .... $\rightarrow$ Value

- **No mutable state**
  - copy, not modify
  - essentially no side effects

- Nothing to lock and **automatic thread safety** when parallellized
Multicore (& co.) era

- Paradigm shift in CPU architecture
  - **Multi core**
    (easily up to 8 cores)
  - **GPU** - Graphical Processing Unit
  - NOC - Network on chip
    (up to 80 and more cores)
Hope

- Language and programming style exploiting parallelism

- **Ideally**: Make my program run \textit{N times faster} on an \textit{N core} CPU with
  - no changes to the program
  - no pain and suffering

- Can we have this? Somehow …
Get into the fight
Erlang

- Functional
- dynamic(ally typed)
- garbage collected
- eager
- compiled to Erlang runtime (BEAM instance)
Shell

- Can play most tricks in the **shell**!
Expression

- Terminated with a period evaluate to a value

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1&gt;</td>
<td>2 + 15.</td>
<td>17</td>
</tr>
<tr>
<td>2&gt;</td>
<td>15 div 2.</td>
<td>7</td>
</tr>
<tr>
<td>3&gt;</td>
<td>2#101010.</td>
<td>42</td>
</tr>
<tr>
<td>4&gt;</td>
<td>16#AE.</td>
<td>174</td>
</tr>
</tbody>
</table>
Variables

- Start with **capital letter**
- Once assigned, a variable is **immutable**
- “=” is **pattern matching**
  Compare (and possibly instantiate vars in the lhs)

```
1> Two = 2.
2
2> Two = 2.
2
3> Two = 3.
** exception error: no match of right hand side value 3
```
Modules

- Programs are organised in **modules**

```
-module(myMath).
-export([fac/1]).

fac(0) ->
    1;
fac(N) ->
    N * fac(N-1).
```

```
fac(5) → 5*fac(5-1) → 5*fac(4) → 5*4*fac(4-1) → ...
```
Modules

- Some functions are exported, some others are not

-module(myMath).
-export([fac/1]).

add(X,0) -> X;
add(X,Y) -> add(X,Y-1)+1.

mul(X,0) -> 0;
mul(X,Y) -> add(mul(X,Y-1),X).

fac(0) -> 1;
fac(N) -> mul(N, fac(N-1)).
Compilation

• A module can be compiled (and loaded)

1> c(myMath).
   {ok,myMath}

• And used ...

2> myMath:fac(5).
   120
25> myMath:fac(7).
   5024
Besides integers

- **Atoms**: constants with their own name for value

  1> atom.
  atom
  2> new_atom.
  new_atom

- **Booleans**

  1> true and false.
  false
  2> false or true.
  true
  3> true xor false.
  true
  4> not false.
  true
  5> not (true and true).
  false
Tuples

• Syntax \{\text{comp1, comp2, comp3}\}

1> \text{X} = 10, \text{Y} = 4.
\text{X} = 10, \text{Y} = 4

2> \text{Point} = \{\text{X,Y}\}.
\{\text{X,Y}\} = \{10,4\}

3> \{\text{First,\_}\} = \text{Point}.
\{\text{First,\_}\} = \{10,4\}

4> \text{First}.
\text{First} = 10.
Tagged Tuples

- Tuples can be tagged for identifying their structure

```plaintext
1> P = {point,{10,5}}.
   {point,{10,5}}
2> CP = {colpoint, {{10,5},red}}.
   {colpoint, {{10,5},red}}
3> {colpoint, Val} = P
   ** exception error: no match of right hand side value {point,{10,5}}
4> {colpoint, Val} = CP
   {colpoint,{{10,5},red}}
5> Val.
   {{10,5},red}
```
Temperature converter

- Temperatures denoted by values {Unit, Value} where Unit can be c(elsius), or f(ahrenheit)

- `module(conv).
- `export([convert/1]).

  `convert({c, X}) ->
      {f, 1.8 * X + 32};

  `convert({f, X}) ->
      {c, (X-32)/ 1.8}. 
Temperature converter

2> conv:convert({f,100}).
   {c,37.8}

3> conv:convert({c,100}).
   {f,212.0}

4> conv:convert({k,2}).
   ** exception error: no function clause matching ...
Temperature converter, Reprise

-module(conv).
-export([convert/1]).

call({c, X}) ->
  {f, 1.8 * X + 32};

call({f, X}) ->
  {c, (X-32)/ 1.8}

call(_) ->
  error.
Lists

- **Syntax**  
  
  \[ \text{elem1, elem2, elem3, ...} \]

- Any type of element

- Head and tail

```
1> [1, 2, 3, {numbers, [4, 5, 6]}, 5.34, atom].
 [1, 2, 3, {numbers, [4, 5, 6]}, 5.34, atom]

11> hd([1, 2, 3, 4]).
 1
12> tl([1, 2, 3, 4]).
  [2, 3, 4]
```
Head/Tail, with matching

15> [ Head | Tail ] = [1,2,3,4].
[1,2,3,4]

16> Head.
1

17> Tail.
[2,3,4]

[2,3,4]

19> NewHead.
2
Length

```plaintext
len([]) ->
  0;
len([_|T]) ->
  1+len(T).
```

- With **tail recursion**

```plaintext
lentr(L) ->
  lentr(L,0).

lentr([],N) ->
  N;
lentr([_|T],N) ->
  lentr(T,N+1).
```
More list ops

- Concatenation, subtraction

5> [1,2,3] ++ [4,5].
   [1,2,3,4,5]

6> [1,2,3,4,5] -- [1,2,3].
   [4,5]

7> [2,4,2] -- [2,4].
   [2]

8> [2,4,2] -- [2,4,2].
   []
Comprehension

• Doubling

1> [2*N || N <- [1,2,3,4]].
[2,4,6,8]

• Get the even

2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 ==: 0].
[2,4,6,8,10]

• Sum

5> [X+Y || X <- [1,2], Y <- [2,3]].
[3,4,4,5]
-module(quicksoort).
-export([qsort/1, triqsort/1]).

qsort([]) ->
    [];

qsort([Pivot|Rest]) ->
    qsort([ X || X <- Rest, X < Pivot])
    ++ [Pivot]
    ++ qsort([ Y || Y <- Rest, Y >= Pivot]).
void QuickSort(int list[], int beg, int end)
{
    int piv; int tmp;
    int l,r,p;

    while (beg < end)
    {
        l = beg; p = (beg + end) / 2; r = end;
        piv = list[p];
        while (l)
        {
            while ((l <= r) && ((list[l] - piv) <= 0)) l++;
            while ((l <= r) && ((list[r] - piv) > 0)) r--;
            if (l > r) break;
            tmp = list[l]; list[l] = list[r]; list[r] = tmp;
            if (p==r) p=1;
            l++; r--;
        }
        list[p] = list[r]; list[r] = piv;
        r--;
        if (r - beg < (end - l))
        {
            QuickSort(list, beg, r);
        beg = l;
        }
    else
    {
        QuickSort(list, l, end);
        end = r;
    }
    }
}
If

• Sugar for (conditional) pattern matching

```
test(X,Y) ->
  if
    X < Y   -> -1;
    X == Y   -> 0;
    X > Y   -> 1
  end.

test(X,Y) when X < Y  ->
  -1;
test(X,X)  ->
  0;
test(X,Y) when X > Y  ->
  1.
```
Case

- Sugar for (conditional) pattern matching

```prolog
insert(X,Set) ->
  case lists:member(X,Set) of
    true  -> Set;
    false -> [X|Set]
  end.
```
Types?

- Dynamically typed
- Types inferred runtime (type errors are possible)

- **Type test** functions
  - `is_atom/1`, `is_binary/1`, `is_bitstring/1`, `is_boolean/1`

- **Type conversion** functions
  - `atom_to_list/1`, `list_to_atom/1`, `integer_to_list/1` ...
Higher-order

• Functions are first class values

```
1> Double = fun(X) -> X * 2 end.
#Fun<erl_eval.6.54118792>
2> Double(3).
6.
```

• Profitably used as function arguments
Map, filter …

- Apply to all elements of a list

```prolog
map(Fun, [First | Rest]) -> [Fun(First) | map(Fun, Rest)];
map(Fun, []) -> [].
```

- Filter only elements satisfying a predicate

```prolog
filter(Pred, L)

filter(_, []) -> [];
filter(Pred, [H | T]) ->
    case Pred(H) of
        true -> [H | filter(Pred, T)];
        false -> filter(Pred, T)
    end.
```
Example

- Convert a list of temperatures

```
1> Temps = [{"Milan", {c,10}},{"Turin", {c,12}}, ... ]
2> Fun   = fun(X) ->
   {City,Temp} = X,
   {City,conv:convert(Temp)} end.
3> map(Fun, Temps)
```

```
4> [{"Milan",{f,50.0}},{"Turin",{f,53.6}}, ... ]
```
Example

- Keep only warm temperatures

```prolog
1> Temps = [{“Milan”, {c,10}}, {“Turin”, {c,12}}, ... ]
2> Pred = fun(X) ->
   {City, {c, Temp}} = X,
   Temp >= 12 end.

3> filter(Fun, Temps)

4> [{“Turin”, {c,12}}, ...]
```
Concurrency
Processes

• Basic **structuring concept** for concurrency (everything is a process)

• Execute a **function** on some **parameters**

• Identified by an **identifier** (id or name), that can be passed (and cannot be forged)

• Strongly **isolated** (no sharing)
Messages

• Processes communicate through **asynchronous message passing** (with known companions)

• Messages are atomic (delivered or not)

• Messages are sent to a process and kept in a **message queue** (the **mailbox**)

• A process can be informed about the status of other processes (detect a **failure**)
General structure

• Processes typically sit in an **infinite loop**
  • **get** a message
  • **process** the message
  • start over

• The mailbox can be accessed **selectively**
Actor model

- **Everything is an actor** and actors execute **concurrently**

- Actors can
  - **send** messages to other actors, **asynchronously** *(mailing)*;
  - designate the **behaviour** for the messages received
  - **create** new actors;

- An actor can **communicate** only with actors whose **address is known**, and **addresses can be passed**
Creating processes

\[
\text{spawn(Module, Exported\_Function, Arg\_List)}
\]

- Create a new \textbf{process} executing
  - a \textbf{function}
  - \textbf{exported} by some module
  - on a \textbf{list of arguments}
- Returns a \textbf{pid}, uniquely identifying the process
-module(tick).
-export([[start/0, tick/2]]).

tick(Msg, 0) ->
    done;

tick(Msg, N) ->
    io:format("Here is tick saying ~p~B times~n", [Msg,N]),
    tick(Msg, N - 1).

start() ->
    spawn(tick, tick, [yup, 3]),
    spawn(tick, tick, [yap, 2]).
Tick tock … run

7> tick:start().
Here is tick saying "yup" 3 times
Here is tick saying "yap" 2 times
Here is tick saying "yup" 2 times
Here is tick saying "yap" 1 times
Here is tick saying "yup" 1 times

8>
Fast spawning

- Lightweight (not 1-1 with system threads)
Communication

• Asynchronous message passing

• **Messages** are valid **Erlang terms** (lists, tuples, integers, atoms, **pids**, …)

• Each process has a **message queue**

• A message can be **sent** to a process (non blocking)

• A process can **selectively receive** messages on its queue (blocking)
Send and receive

- **Send**
  
  \[ \text{pid} ! \text{msg} \]

- **Receive**
  
  ```
  receive
  
  msg\_pattern1 ->
  
  action1;

  msg\_pattern2 ->
  
  action2;

  ...

  end
  ```
Multiplier: server

-module(mulServer).
-export([start/0, mul_server/0]).

mul_server() ->
    receive
        {X, Y, Pid} ->
            Pid ! X*Y,
            mul_server();
        stop ->
            io:format("Server stopping ... ", [])
    end.
Multiplier: server (concurrent)

```
-module(mulServerConc).
-export([start/0, mul_server/0]).

mul_server() ->
    receive
        {X, Y, Pid} ->
            spawn(fun() -> Pid ! X*Y end.),
            mul_server(),
        end
    stop ->
        io:format("Server stopping ... ", [])
    end.
```
Multiplier: client

```erlang
start() ->
    Server = spawn(proc2, mul_server, []),
    Server ! {2, 2, self()},
    Server ! {2, 4, self()},
    receive
        P1 ->
            io:format("Product 2*2 = ~B~n", [P1])
        end,
        receive
            P2 ->
                io:format("Product 2*4 = ~B~n", [P2])
            end,
    Server!stop.
```
-module(mulAddServer).
% messages are of the kind {Op, X, Y, Pid}
mul_add_server() ->
    receive
        {mul, X, Y, Pid} ->
            Pid ! X*Y,
            mul_add_server();

        {add, X, Y, Pid} ->
            Pid ! X+Y,
            mul_add_server();

        stop ->
            io:format("Server stopping ...", [])
    end.
Careful with the mailbox

• What if the server gets wrongly formatted messages?

# However, as messages not matched by receive are left in the mailbox, it is the programmer's responsibility to make sure that the system does not fill up with such messages.

• Do something with unmatched messages

• Try to avoid unmatched messages offering a communication interface
Process unmatched messages

-module(mulAddServer).
\% messages are of the kind \{Op, X, Y, Pid\}
mul_add_server() ->
  receive
    {mul, X, Y, Pid} -> … ;
    {add, X, Y, Pid} -> … ;
    stop -> … ;
    …
    \( M \rightarrow \) do st. with message \( M \) (e.g., log error)
  end.
Offer an interface

-module(mulAddServer).

mul(Server, X,Y) ->
    Server ! {mul, X, Y, self()}.

add(Server, X,Y) ->
    Server ! {add, X, Y, self()}. 

mul_add_server() ->
    receive
        {mul, X, Y, Pid} -> ... ;
        {add, X, Y, Pid} -> ... ;
        stop -> ... ;
    end.
Registering

• Processes can be registered

\[\text{register}(\text{Pid, Alias})\]

• Useful for \textbf{restarting behaviours} (node visibility)

• Alias can be unregistered (done automatically when aliased process dies)

\[\text{unregister}(\text{Alias})\]
Timing out

• A receive can be exited after some time:

```plaintext
receive
  Msg1  ->
    action1;
  Msg2  ->
    action2
  ...

after Time  ->
    action after timeout
```

• Example
Multiplier, again

- The server (mul_server) is registered (multiplier)
- Accessible to clients knowing the name
- The server can be stopped ‘only by the creator’ (secret = creator pid … not very secret)
- The client sends and gets tagged messages and possibly timeouts if answer takes too long.
Multiplier, again

• The server (mul_server) is registered (as **multiplier**) when started

```start() ->
    Server = spawn(mulServerReg, mul_server, [self()]),
    register(multiplier, Server),
```

• Known as **multiplier** in the node

• Pid of the creator is passed to the server, to be kept in the “server state”
mul_server(Creator) ->
  receive
  \% mul message: provide answer 'signed'
  \% with an id
  {Id, Pid, X, Y} ->
    Pid ! {Id, X*Y},
    mul_server(Creator);
  \% stop message (only by creator)
  {Creator, stop} ->
    io:format("Server stopping ...~n", []);
  \% stop message, not from creator
  {Pid, stop} ->
    io:format("Process \"~w\" not allowed
to stop ...~n", [Pid]),
    mul_server(Creator)
  end.
client() ->
  % first message
  Id1 = crypto:strong_rand_bytes(5),
  Msg1 = {Id1, self(), 2, 2},
  multiplier ! Msg1,

  receive
    {Id1,P1} ->
      out_result(Msg1,P1)
  after
    10 ->
      out_result(Msg1,fail)
  end,

  multiplier ! {self(), stop },
Robustness

- Primitives allows to “link” processes in a way that processes in the same group are notified of abnormal (error) events.

- Abnormal termination is normal: "Let it crash" philosophy.

- The structuring can be hierarchical allowing for layered applications: workers, monitors, supervisors.
Links and monitors

• A process can be linked to or monitor another process

• A process can exit
  • normally
    run out of code or exit(normal)
  • abnormally
    error or exit(Reason)
Links

• (Bidirectional) link between caller and pid

\[
\text{link}(\text{Pid})
\]

• When a process exits, linked processes receive a signal, carrying \text{pid} and \text{exit reason}

• By default
  • \textbf{normal exits} ignored
  • \textbf{abnormal exits} \textit{kill} the receiving process. \textit{propagate} the error signal to the links of the killed process
Links, more control

• A process can become a **supervisor process** (also called **system process**)
  
  ```prolog
  process_flag(trap_exit, true).
  ```

• The **exit signal** is caught as a **message**
  
  ```prolog
  {'EXIT', Pid, Reason}
  ```
Links, more control

• E.g., in the process `start` (see before)

```erlang
start() ->
    process_flag(trap_exit, true),

    Server = spawn(mulServerReg, mul_server, [self()]),
    link(Server),
    register(multiplier, Server),

    ...

receive
    {'EXIT', Server, Reason} ->
        % depending on reason, possibly
        % restart the server
```
Example

- A server that gets messages consisting of a function and its arguments

- Execute the function on the arguments as a “supervised” servant, keeping a list of the unfinished tasks

- For each servant, get the result and provides it to the corresponding client.

- In case of abnormal exit of the servant, retry
Example: servant

% Given a function and some arguments
% - executes the functions on the arguments
% - or randomly fails (75% of the times)
servant(F, Args, Server) ->
    case rand:uniform(4) of

        % regular execution, notify the server
        % providing the result
        1 ->
            Server ! {answ, {self(), F(Args)});

        % failure
        _ ->
            exit(went_wrong)
    end.
% The server keeps in its state
% - Creator: the pid of the creator
% - WaitingList: list of requests being processed of
%   the kind \{Servant,Client,F,Args\} including
% Servant's pid, client's pid, request data

server(Creator, WaitingList) ->

  % Supervisor process: traps the exit signals
  process_flag(trap_exit, true),

  receive
    % (1) client request
    % (2) normal termination from servant
    % (3) error message from servant
    % (4) stop request from creator
    % (5) stop request from non creator

  end.
Example: server

% (1) client request
{req, {Client, F, Args}} ->

% spawn and link at the same time (atomic)
Servant = spawn_link(hierarchy, servant, [F, Args, self()]),
server(Creator, [{Servant,Client,F,Args} | WaitingList]);

% (2) normal termination from servant
{answ, {Servant, Result}} ->

{_,Client,F,Args} = lists:keyfind(Servant,1,WaitingList),
Client ! {answ, {Client, F,Args}, Result},
server(Creator, WaitingList--[{Servant,Client,F,Args}]);
Example: server

% (3) error message from servant
{'EXIT', Servant, went_wong} ->
  {_,Client,F,Args} = lists:keqyfind(Servant,1,WaitingList),
  io:format("Servant ~w went wrong, retrying ...~n", [Servant]),
  NewServant = spawn_link(hierarchy, servant, [F, Args, self()],)
  server(Creator, (WaitingList--[{Servant,Client,F,Args}])
           ++ [{NewServant,Client,F,Args}]) ;

% (4) stop request from creator
{Creator, stop} ->
  io:format("Server stopping ...~n", []),
  exit(normal) ;

% (5) stop request not from creator
{Pid, stop} ->
  io:format("Process \"~w\" not allowed ...~n", [Pid]),
  server(Creator,WaitingList)
Example: creator

```prolog
start() ->

% create and register the server
Server = spawn(hierarchy, server, [self(),[]]),
register(pserver, Server),

% accessible to some client, without getting the pid
spawn(hierarchy, client, []),

% wait a bit and stops the server
timer: sleep(1000),
pserver ! {self(),stop}.
```
client() ->
% first message
Msg1 = {self(), fun([X,Y]) -> X*Y end, [1,2]},
pserver ! {req, Msg1},

% wait for result, possibly timing out
receive
  {answ, Msg1, R1} ->
    out_result(product,R1)
  after
    10 ->
    out_result(product,fail)
end,

...
Exercise

- Modify the system as follows:
  - The server creates a servant for each request
  - In case of normal termination, the servant itself send the result to the client
  - In case of abnormal termination of the servant, the server is notified and a new servant is created
Monitors

- Create a “unidirectional” link:
  current process monitors the process Pid

  monitor(process, Pid)

- On exits the monitor process gets a message

  {'DOWN', MonitorReference, process, Pid, Reason}
Distribution

- **Distributed Erlang**
  - Processes run in various Erlang nodes, same intranode primitives
  - Applications running in a distributed trusted environment (cluster)

- **Socket-based distribution**
  - TCP/IP sockets to communicate in an untrusted environment
  - less flexible, but more secure
Distributed Erlang

• Actors are spread on different nodes

• Node A can communicate with Node B if they share a cookie (magic cookie) and know each other name

• Start a node (with cookie)

```bash
erl -sname name  -setcookie cookie % same host
erl -name name@host -setcookie cookie % across hosts
```
Connections

- Node in Erlang are loosely connected

- **Connecting nodes**
  
  ```erlang
  net_kernel:connect_node(NodeName)
  ```

  Also implicitly established at first connection attempt

- Connections are **transitive**

- If a node goes down, all connections to it are removed.
erl -sname node1 -setcookie "a"
erl -sname node2 -setcookie "a"
erl -sname node3 -setcookie "a"

node1> nodes().
[]
node1> net_kernel:connect_node('node2@host').
pong

node1> nodes().
['node2@host']

node2> net_kernel:connect_node('node3@host').
pong

node1> nodes().
['node2@host', 'node3@host']
Distributing

- Lifting to the cluster level works reasonably smoothly

- primitives like spawn, link, monitor etc. has additional node parameter, e.g.
  
  ```
  spawn(Node, Module, Exported_Function, Arg List)
  ```

- registered names are local to nodes, hence pid must be used (or see `global` module)

- when spawning a process at a node, the code must be available at that node (care with passing functional arguments)
Example

• The previous example, of a server getting a list of tasks to execute modified as follows:

  • client, server and slaves on different nodes
  
  • the server monitor the slaves, on fail it retries on a (possibly) different node

distributed.erl
Socket-based distribution

- Standard (low level) socket interface (gen_tcp module)
- Server: listen, accept
- Client: connect
- send, recv
Open Telecom Platform (OTP)

- A set of design principles
- A set of libraries
- Developed and used by Ericsson to build large-scale, fault-tolerant, distributed applications with pre-designed skeletons and patterns (server, fsm, event ...
gen_server

- Need to implement a number of callbacks
  - **init** (set up, initialise the state)
  - **handle_cast** (asynchronous call without a reply, determining a state change)
  - **handle_call** (synchronous call with a reply)
  - **terminate**
  - ...
Example

• Multiplier realised with `gen_server`
-module(mulGenServer).

% declare that the gen_server behaviour is implemented
-behaviour(gen_server).

-export([go/0, client/0]).
-export([start/0, mul/2, stop/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2]).
%%% INTERFACE

% Create the server, registered locally as multiplier, calling init
% with parameter self() (the pid of the creator)
start() ->
    gen_server:start({local, multiplier}, ?MODULE, [self()], []).

% multiplication: synchronous call
mul(X,Y) ->
    gen_server:call(multiplier, {mul, X, Y}).

% stop request, asynchronous call passing the pid of the caller
% (better implemented as terminate message, just to have an example of
% cast)
stop() ->
    gen_server:cast(multiplier, {stop, self()}).
%% CALLBACKS

%%% initialization: establish the initial state
\texttt{init([Creator]) ->
  \{ok, [Creator]\}.
}

%%% multiplication handle
%%% IN: message, sender, server state
%%% OUT: reply atom, reply content, new state
\texttt{handle\_call({mul, X, Y}, \_From, [Creator]) ->
  \{reply, X*Y, [Creator]\}.
}

%%% stop handle
\texttt{handle\_cast({stop, From}, [Creator]) ->
  if From == Creator ->
    \{stop, normal, [Creator]\};
    From /= Creator ->
    \texttt{io:format("Invalid shutdown req (pid ~w)\n", [From]),}
    \{noreply, [Creator]\}
  end.
}
% handling termination
terminate(normal, [Creator]) ->
    io:format("Server created by: ~w properly terminated~n", [Creator]).

% other messages
handle_info(Msg, [Creator]) ->
    io:format("Unexpected message: ~p~n", [Msg]),
    {noreply, [Creator]}. 
Dynamic Code Loading

- Built-in in Erlang

- A module can exist in two variants in a system: current and old

- When a module is loaded into the system for the first time, the code becomes `current'.

- If then a new instance is loaded, the previous instance becomes 'old' and the new one 'current'.
Dynamic Code Loading

- Two possible ways of referencing a function
  - Name only: still refers to the old version
    \[ \text{fun(...)} \]
  - Fully qualified: refers to the new version and changes the current code
    \[ \text{module:fun(...)} \]

`hotSwap.erl`

- Module
  - `fun(...)`
  - `module:fun(...)`
  - New module

Example

- **Controller:**
  - **new**: create new **loop** process, return pid
  - Supervises **termination** of loop processes and communicate reason

- **Loop:**
  - **ver**: get version
  - **upd**: update to new version
  - **stop**: stop
Dynamic Code Loading

- Dangerous
- Higher-level abstractions provided in OTP
Concluding ...

- Concurrency Oriented Programming (~ actor model)
- Emphasis on
  - Encapsulation with focus on computing entities (state + reaction to messages)
  - Transparent Distribution
  - Fault tolerance (supervisor trees and let it crash philosophy)
  - Scalability (multiple instances on multiple nodes)
  - Continuous Operation (hot-swapping)
Not perfect
(as everything in the world)

• A bit oldish/low level syntax and design choices …
  alternatives Elixir, Clojure, …

• Untyped … (Scala, Akka)

• Identifying communication channels with computing
  entities possibly cumbersome (see message tagging)

• Primitive security model (restricting access to a node /
  process capabilities)

• Message passing only is good, but can be heavy when
  supporting the sharing of large data sets
Still you can make cool apps