Contents

1 Starting the System and Basic Erlang

Temperature Conversion

temp.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: temp.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2012 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(temp).

-export([f2c/1,c2f/1,convert/1]).

-spec f2c(number()) -> number().

f2c(F) ->
    5 * (F - 32) / 9.

-spec c2f(number()) -> number().

c2f(C) ->
    9 * C / 5 + 32.

-spec convert({'c',number()}) -> {'f',number()};
             ({'f',number()}) -> {'c',number()}.

convert({c,C}) ->
    {f,c2f(C)};
convert({f,F}) ->
    {c,f2c(F)}.
```

Simple Pattern Matching

boolean.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: boolean.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(boolean).
-export([b_not/1, b_and/2, b_or/2, b_or2/2]).

-spec b_not(boolean()) -> boolean().
b_not(false)-> true;
b_not(true )-> false.

-spec b_and(boolean(), boolean()) -> boolean().
b_and(true,   true  )-> true ;
b_and(_Bool1, _Bool2)-> false.
```

```
-spec b_or(boolean(), boolean()) -> boolean().
b_or(true,  _Bool)-> true;
b_or(_Bool, true )-> true;
b_or(false, false)-> false.

% other solution
-spec b_or2(boolean(), boolean()) -> boolean().
b_or2(false, false) -> false;
b_or2(_, _) -> true.
```

2 Sequential Programming

Evaluating Expressions

sums.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: sums.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(sums).
-export([sum/1, sum_interval/2]).

%% @doc Adds the integers between 1 and N.
-spec sum(non_neg_integer()) -> integer().
sum(0) ->
    0;
sum(N) ->
    N + sum(N-1).

%% @doc Adds the integers between N and M
-spec sum_interval(integer(), integer()) -> integer().
sum_interval(Max, Max) ->
    Max;
sum_interval(Min, Max) when Min =< Max ->
    Min + sum_interval(Min + 1, Max).
```

Creating Lists

create.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: create.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(create).
-export([create/1, reverse_create/1]).

%% @doc Creates a list with integers [1,..,N]
-spec create(integer()) -> [integer(), ...].
create(N) ->
    create(1, N).

create(M,M) ->
    [M];
create(M,N) ->
    [M | create(M+1, N)].

%% @doc Creates a list with integers [N,..,1]
-spec reverse_create(non_neg_integer()) -> [non_neg_integer(), ...].
reverse_create(1) ->
```

```
    [1];
reverse_create(N) ->
    [N | reverse_create(N-1)].
```

Side Effects

effects.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: effects.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(effects).
-export([print/1, even_print/1]).

%% @doc Prints the integers between 1 and N.
-spec print(non_neg_integer()) -> ok.
print(0) ->
    ok;
print(N) ->
    print(N-1),
    io:format("Number:~p~n",[N]).

%% @doc Prints the even integers between 1 and N.
-spec even_print(non_neg_integer()) -> ok.
even_print(0) ->
    ok;
even_print(N) when N rem 2 == 0 ->
    even_print(N-1),
    io:format("Number:~p~n",[N]);
even_print(N) ->
    even_print(N-1).
```

Database Handling Using Lists

db.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: db.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(db).
-export([new/0, write/3, delete/2, read/2, match/2, destroy/1]).
-export_type([db/0]).
-type db() :: list().

%% @doc Create a new database
-spec new() -> db().
new() ->
    [].
```

```
%% @doc Insert a new element in the database
-spec write(Key::term(), Val::term(), db()) -> db().
write(Key, Element, []) ->
    [{Key, Element}];
write(Key, Element, [{Key, _} | Db]) ->
    [{Key, Element}|Db];
write(Key, Element, [Current | Db]) ->
    [Current | write(Key, Element, Db)].

%% @doc Remove an element from the database
-spec delete(Key::term(), db()) -> db().
delete(Key, [{Key, _Element}|Db]) ->
    Db;
delete(Key, [Tuple|Db]) ->
    [Tuple|delete(Key, Db)];
delete(_Key, []) ->
    [].

%% @doc Retrieve the first element in the database with a matching key
-spec read(Key::term(), db()) -> {ok, term()} | {error, instance}.
read(Key, [{Key, Element}|_Db]) ->
    {ok, Element};
read(Key, [_Tuple|Db]) ->
    read(Key, Db);
read(_Key, []) ->
    {error, instance}.

%% @doc Return all the keys whose values match the given element.
-spec match(Val::term(), db()) -> [term()].
match(Element, [{Key, Element}|Db]) ->
    [Key|match(Element, Db)];
match(Element, [_Tuple|Db]) ->
    match(Element, Db);
match(_Key, []) ->
    [].

%% @doc Deletes the database.
-spec destroy(db()) -> ok.
destroy(_Db) ->
    ok.
```

ADVANCED: Manipulating Lists

manipulating.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: manipulating.erl
%%% @author trainers@erlang-solutions.com
                                        %%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(manipulating).
-export([filter/2, concatenate/1, reverse/1, flatten/1]).

%% @doc Given an integer list, returns a new list where the elements
%% come from List and are smaller than the Key.
```

```
-spec filter([integer()], integer()) -> [integer()].
filter([H|T], Key) when H =< Key ->
    [H|filter(T, Key)];
filter([_|T], Key) ->
    filter(T, Key);
filter([], _Key) ->
    [].

%% @doc Given a list of lists, returns a new list containing the
%% elements of the lists.
-spec concatenate([list()]) -> list().
concatenate([]) -> [];
concatenate([H|T]) ->
    concatenate1(H, T).

concatenate1([H|T], Lists) ->
    [H|concatenate1(T, Lists)];
concatenate1([], Lists) ->
    concatenate(Lists).

%% @doc Will reverse the list order.
-spec reverse(list()) -> list().
reverse(List) ->
    reverse(List, []).
reverse([], Buffer) ->
    Buffer;
reverse([H|T], Buffer) ->
    reverse(T, [H|Buffer]).

%% @doc Takes a list and recursively flattens it.
-spec flatten(list()) -> list().
flatten([H|T]) when is_list(H) ->
    concatenate([flatten(H), flatten(T)]);
flatten([H|T]) ->
    [H|flatten(T)];
flatten([]) ->
    [].
```

3 Concurrent Programming

An Echo Server

echo.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File      : echo.erl
%%% Author    : trainers@erlang-solutions.com
%%% Copyright : 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(echo).
-export([start/0, stop/0, listen/0, print/1]).

-spec start() -> ok.
start()->
    register(echo, spawn(echo, listen, [])),
    ok.

%% Prints a term passed as an argument.
-spec print(term()) -> ok.
print(Message)->
    echo ! {print,  Message},
    ok.

%% Stops the echo server.
-spec stop() -> ok.
stop()->
    echo ! stop,
    ok.

%% The echo server loop
-spec listen() -> true.
listen()->
    receive
        {print, Message} ->
            io:format("~p~n",[Message]),
            listen();
        stop  ->
            true
    end.
```

The Process Ring

ring.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: ring.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
-module(ring).
%% Client Functions
-export([start/3]).

%% Internal Exports
-export([master/3, loop/2]).

%% @doc Starts the master process which in turn spawns off the
%% individual processes which will receive a message.
-spec start(non_neg_integer(), non_neg_integer(), term()) -> pid().
start(ProcNum, MsgNum, Message)->
    spawn(ring, master, [ProcNum, MsgNum, Message]).

%% @private This function starts the slave pids and then gets into
%% the loop which will send the Message MsgNum times to
%% the slaves.
-spec master(non_neg_integer(), non_neg_integer(), term()) -> stop | no_return().
master(ProcNum, MsgNum, Message)->
    Pid = start_slaves(ProcNum,self()),
    master_loop(MsgNum, Message, Pid).


%% Will start ProcNum slave processes
-spec start_slaves(non_neg_integer(), pid()) -> pid().
start_slaves(1, Pid)->
    Pid;
start_slaves(ProcNum, Pid)->
    NewPid = spawn(ring, loop, [ProcNum, Pid]),
    start_slaves(ProcNum - 1, NewPid).

%% The master loop will loop MsgNum times sending a message to
%% Pid. It will iterate every time it receives the Message it is
%% sent to the next process in the ring.
-spec master_loop(non_neg_integer(), term(), pid()) -> stop | no_return().
master_loop(0, _Message, Pid)->
    io:format("Process:1 terminating~n"),
    Pid ! stop;
master_loop(MsgNum, Message, Pid) ->
    Pid ! Message,
    receive
        Message ->
            io:format("Process:1 received:~p~n",[Message]),
            master_loop(MsgNum - 1, Message, Pid)
    end.

%% @private This is the slave loop, where upon receiving a message, the
%% process forwards it to the next process in the ring. Upon
%% receiving stop, it sends the stop message on and terminates.
-spec loop(non_neg_integer(), pid()) -> stop | no_return().
loop(ProcNum, Pid)->
    receive
        stop ->
            io:format("Process:~p terminating~n",[ProcNum]),
            Pid ! stop;
        Message ->
            io:format("Process:~p received: ~p~n", [ProcNum, Message]),
            Pid!Message,
            loop(ProcNum, Pid)
```

```
    end.
```

ADVANCED: The Process Crossring

crossring.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: crossring.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(crossring).

%% Client Functions
-export([start/3]).
%% Internal Exports
-export([master/3, loop/2]).

%% @doc Starts the master process which in turn spawns off the
%% individual processes which will receive a message.
-spec start(pos_integer(), non_neg_integer(), term()) -> pid().
start(ProcNum, MsgNum, Message)->
    spawn(crossring, master, [ProcNum, MsgNum, Message]).

%% @private This function starts the slave pids and then gets into
%% the loop which will send the Message MsgNum times to
%% the slaves.
-spec master(pos_integer(), non_neg_integer(), term()) -> stop | no_return().
master(ProcNum, MsgNum, Message)->
    ProcLim = round(ProcNum / 2),
    {MidPid, FirstPid} = start_slaves(ProcNum, ProcLim, self()),
    master_loop(MsgNum, {first_half, Message}, FirstPid, MidPid).


%% Will start ProcNum slave processes
-spec start_slaves(pos_integer(), non_neg_integer(), pid()) -> pid() | {pid(),
pid()}.
start_slaves(1, _, Pid)->
    Pid;
%% We cross when we're on the midpoint process + 1.
start_slaves(ProcNum, ProcLim, Pid) when ProcNum =:= ProcLim + 1->
    %% We spawn the process the first one will send messages to
    MidPid = spawn(crossring, loop, [ProcNum, Pid]),
    %% We return it in a tuple, and keep starting the other processes
    %% after the first (middle) one. The Last spawned Pid (or the second
    %% element of the crossring) is returned as the second tuple element
    {MidPid, start_slaves(ProcNum - 1, ProcLim, self())};
start_slaves(ProcNum, ProcLim, Pid)->
    NewPid = spawn(crossring, loop, [ProcNum, Pid]),
    start_slaves(ProcNum - 1, ProcLim, NewPid).

%% The master loop will loop MsgNum times sending a message to
%% Pid. It will iterate every time it receives the Message it is
%% sent to the next process in the ring.
-spec master_loop(non_neg_integer(), term(), pid(), pid()) -> stop | no_return().
```

```
master_loop(0, _Message, FirstPid, MidPid)->
    io:format("Process: 1 terminating~n"),
    MidPid ! FirstPid ! stop;
%% Handling the messages on the first half of the crossring
master_loop(MsgNum, {first_half, Message}, FirstPid, MidPid) ->
    FirstPid ! {first_half, Message},
    receive
        {first_half, Message} ->
            io:format("Process: 1 received: ~p halfway through~n",[Message]),
            master_loop(MsgNum, {second_half, Message}, FirstPid, MidPid)
    end;
%% Handling the messages on the second half of the crossring
master_loop(MsgNum, {second_half, Message}, FirstPid, MidPid) ->
    MidPid ! {second_half, Message},
    receive
        {second_half, Message} ->
            io:format("Process: 1 received: ~p~n",[Message]),
            master_loop(MsgNum - 1, {first_half, Message}, FirstPid, MidPid)
    end.

%% @private This is the slave loop, where upon receiving a message, the
%% process forwards it to the next process in the ring. Upon
%% receiving stop, it sends the stop message on and terminates.
-spec loop(pos_integer(), pid()) -> stop | no_return().
loop(ProcNum, Pid)->
    receive
        stop ->
            io:format("Process: ~p terminating~n",[ProcNum]),
            Pid ! stop;
        {Part, Message} ->
            io:format("Process: ~p received: ~p~n", [ProcNum, Message]),
            Pid ! {Part, Message},
            loop(ProcNum, Pid)
    end.
```

4 Process Design Patterns

A Database Server

my_db.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: my_db.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2015 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(my_db).

%% Internal Exports
-export([init/0]).

%%External Exports
-export([start/0,stop/0, write/2, delete/1, read/1, match/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Client Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @doc Starts the database server
-spec start() -> ok.
start() ->
    register(db, spawn(my_db, init, [])),
    ok.

%% @doc Stops the database server
-spec stop() -> ok.
stop() ->
    async_call(stop).

%% @doc Inserts an element in the database server
-spec write(Key::term(), Val::term()) -> ok.
write(Key, Element) ->
    async_call({write, Key, Element}).

%% @doc Removes an element from the database. Will succeed even
%% If the element does not exist.
-spec delete(Key::term()) -> ok.
delete(Key) ->
    async_call({delete, Key}).

%% @doc Will retrieve an element from the database.
-spec read(Key::term()) -> {ok, term()} | {error, instance}.
read(Key) ->
    sync_call({read, Key}).

%% @doc Will return a list of keys which match to the element.
-spec match(Val::term()) -> [Key::term()].
match(Element) ->
    sync_call({match, Element}).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Communication Help Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Will make a synchronous call to the server and return the
%% reply received.
-spec sync_call(term()) -> term().
sync_call(Msg) ->
    db ! {request, self(), Msg},
    receive
        {reply, Reply} -> Reply
    end.

%% Will send an asynchronous reques to the server.
-spec async_call(term()) -> ok.
async_call(Msg) ->
    db ! {request, Msg},
    ok.

%% Sends a reply back to the client's synchronous call.
-spec reply(pid(), term()) -> {reply, term()}.
reply(Pid, Reply) ->
    Pid ! {reply, Reply}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Database Server Loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @private Initialises the database and enters the server loop
-spec init() -> no_return().
init() ->
    loop(db:new()).

%% loop(list())
%% The database server loop which will iterate and handle
%% all requests.
-spec loop(db:db()) -> no_return().
loop(Db) ->
    receive
        {request, {write, Key, Element}} ->
            NewDb = db:write(Key, Element, Db),
            loop(NewDb);
        {request, {delete, Key}} ->
            NewDb = db:delete(Key, Db),
            loop(NewDb);
        {request, Pid, {read, Key}} ->
            reply(Pid, db:read(Key, Db)),
            loop(Db);
        {request, Pid, {match, Element}} ->
            reply(Pid, db:match(Element, Db)),
            loop(Db);
        {request, stop} ->
            db:destroy(Db)
    end.
```

A Turnstile

turnstile.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: turnstile.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 2020 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(turnstile).

%% External Exports.
-export([start/0, insert_coin/2, enter/1]).

%% Internal Exports
-export([init/0]).

-define(PRICE, 100).                            %The price to enter
-define(TIMEOUT, 5000).                         %Timeout

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Client Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @doc Will start the turnstile.
-spec start() -> pid().
start() ->
    {ok, spawn(turnstile, init, [])}.

%% @doc Will insert coin into the turnstile.
-spec insert_coin(pid(), integer()) -> ok.
insert_coin(Pid, Coin) ->
    sync_call(Pid, {insert, Coin}).

%% @doc Enter the turnstile.
-spec enter(pid()) -> ok.
enter(Pid) ->
    sync_call(Pid, enter).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Communication Help Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Will make a synchronous call to the turnstile and return the
%% reply received.
-spec sync_call(pid(), term()) -> term().
sync_call(Pid, Msg) ->
    Pid ! {request, self(), Msg},
    receive
        {reply, Reply} -> Reply
    end.

%% Sends a reply back to the client's synchronous call.
-spec reply(pid(), term()) -> {reply, term()}.
reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Finite State Machine
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @doc Initializes the state machine.
-spec init() -> no_return().
init() ->
    closed(0).

%% @doc The state where the turnstile is closed.
closed(Balance) ->
    receive
        {request, Pid, {insert, Coin}} ->
            reply(Pid, ok),
            NewBalance = Balance + Coin,
            if NewBalance >= ?PRICE ->
                    open(NewBalance - ?PRICE);
               true ->
                    closed(NewBalance)
            end;
        {request, Pid, enter} ->
            reply(Pid, {error, access_denied}),
            closed(Balance)
    end.

%% @doc The state where the turnstile is open.
open(Balance) ->
    receive
        {request, Pid, {insert, Coin}} ->
            reply(Pid, ok),
            closed(Balance + Coin);
        {request, Pid, enter} ->
            reply(Pid, ok),
            closed(Balance)
    after ?TIMEOUT ->
            closed(Balance)
    end.
```

ADVANCED: A Database Server with transactions

my_db_trans.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: my_db_trans.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2015 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(my_db_trans).

%% Internal Exports
-export([init/0]).

%%External Exports
-export([start/0, stop/0]).
-export([write/2, delete/1, read/1, match/1, lock/0, unlock/0]).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Client Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @doc Starts the database server
-spec start() -> ok.
start() ->
    register(db, spawn(my_db_trans, init, [])),
    ok.

%% @doc Stops the database server
-spec stop() -> ok.
stop() ->
    sync_call(stop).

%% @doc Inserts an element in the database server
-spec write(Key::term(), Val::term()) -> ok.
write(Key, Element) ->
    sync_call({write, Key, Element}).

%% @doc Removes an element from the database. Will succeed even
%% If the element does not exist.
-spec delete(Key::term()) -> ok.
delete(Key) ->
    sync_call({delete, Key}).

%% @doc Will retrieve an element from the database.
-spec read(Key::term()) -> {ok, term()} | {error, instance}.
read(Key) ->
    sync_call({read, Key}).

%% @doc Will return a list of keys which match to the element.
-spec match(Val::term()) -> [Key::term()].
match(Element) ->
    sync_call({match, Element}).

%% @doc Will lock the database for the current caller.
-spec lock() -> ok.
lock() ->
    sync_call(lock).

%% @doc Will unlock the database.
-spec unlock() -> ok.
unlock() ->
    sync_call(unlock).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Communication Help Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Will make a synchronous call to the server and return the
%% reply received.
-spec sync_call(term()) -> term().
sync_call(Msg) ->
    db ! {request, self(), Msg},
    receive
        {reply, Reply} -> Reply
```

```
    end.

%% Sends a reply back to the client's synchronous call.
-spec reply(pid(), term()) -> {reply, term()}.
reply(Pid, Reply) ->
    Pid ! {reply, Reply}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Database Server Loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @private Initialises the database and enters the server loop
-spec init() -> no_return().
init() ->
    loop(db:new()).

%% loop(list())
%% The database server loop which will iterate and handle
%% all requests.
-spec loop(db:db()) -> no_return().
loop(Db) ->
    receive
        {request, Pid, {write, Key, Element}} ->
            NewDb = db:write(Key, Element, Db),
            reply(Pid, ok),
            loop(NewDb);
        {request, Pid, {delete, Key}} ->
            NewDb = db:delete(Key, Db),
            reply(Pid, ok),
            loop(NewDb);
        {request, Pid, {read, Key}} ->
            reply(Pid, db:read(Key, Db)),
            loop(Db);
        {request, Pid, {match, Element}} ->
            reply(Pid, db:match(Element, Db)),
            loop(Db);
        {request, Pid, stop} ->
            reply(Pid, ok),
            db:destroy(Db);
        {request, Pid, lock} ->
            reply(Pid, ok),
            locked_loop(Db, Pid)
    end.

%% locked_loop(list(), pid())
%% The database server loop which will iterate and handle
%% all requests from the locking client.
-spec locked_loop(db:db(), pid()) -> no_return().
locked_loop(Db, Locker) ->
    receive
        {request, Locker, {write, Key, Element}} ->
            NewDb = db:write(Key, Element, Db),
            reply(Locker, ok),
            locked_loop(NewDb, Locker);
        {request, Locker, {delete, Key}} ->
            NewDb = db:delete(Key, Db),
            reply(Locker, ok),
            locked_loop(NewDb, Locker);
```

```
    {request, Locker, {read, Key}} ->
        reply(Locker, db:read(Key, Db)),
        locked_loop(Db, Locker);
    {request, Locker, {match, Element}} ->
        reply(Locker, db:match(Element, Db)),
        locked_loop(Db, Locker);
    {request, Locker, stop} ->
        reply(Locker, ok),
        db:destroy(Db);
    {request, Locker, unlock} ->
        reply(Locker, ok),
        loop(Db)
end.
```

5 Process Error Handling

The Linked Ping Pong Server

pingpong.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File      : pingpong.erl
%%% Author    : trainers@erlang-solutions.com
%%% Copyright : 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(pingpong).

%% Interface
-export([start/0, stop/0, send/1]).
%% Internal Exports
-export([init_a/0, init_b/0]).

start() ->
    register(a, spawn(pingpong, init_a, [])),
    register(b, spawn(pingpong, init_b, [])),
    ok.

stop() ->
    exit(whereis(a), non_normal_exit).

send(N) ->
    a ! {msg, message, N},
    ok.

init_a() ->
    loop_a().

init_b() ->
    link(whereis(a)),
    loop_b().

loop_a() ->
    receive
        {msg, _Msg, 0} ->
            loop_a();
        {msg, Msg, N} ->
            io:format("ping...~n"),
            timer:sleep(500),
            b ! {msg, Msg, N -1},
            loop_a()
    after
        15000 ->
            io:format("Ping got bored, exiting.~n"),
            exit(timeout)
    end.

loop_b() ->
    receive
        {msg, _Msg, 0} ->
```

```
            loop_b();
        {msg, Msg, N} ->
            io:format("pong!~n"),
            timer:sleep(500),
            a ! {msg, Msg, N -1},
            loop_b()
    after
        15000 ->
            io:format("Pong got bored, exiting.~n"),
            exit(timeout)
    end.
```

A Reliable Mutex Semaphore

mutex.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: mutex.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(mutex).
%% Client Exports
-export([start/0, signal/0, wait/0]).
%% Internal Exports
-export([init/0]).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Client Functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @doc Will start the mutex semaphore
-spec start() -> true.
start() ->
    register(mutex, spawn(mutex, init, [])).

%% @doc Initializes the state machine.
-spec init() -> no_return().
init() ->
    process_flag(trap_exit, true),
    free().

%% @doc Will free the semaphore currently held by the process
-spec signal() -> ok.
signal() ->
    mutex ! {signal, self()},
    ok.

%% @doc Will keep the process busy until the semaphore is available.
-spec wait() -> ok.
wait() ->
    mutex ! {wait, self()},
    receive
        ok -> ok
    end.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Finite State Machine
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% @doc The state where the semaphore is available
-spec free() -> no_return().
free() ->
    receive
        {'EXIT', Pid, _Reason} ->
            free();
        {wait, Pid} ->
            link(Pid),
            Pid ! ok,
            busy(Pid)
    end.

%% @doc The semaphore is taken by Pid. Pid is the only process which
%% may release it.
-spec busy(pid()) -> no_return().
busy(Pid) ->
    receive
        {'EXIT', Pid, _Reason} ->
            free();
        {signal, Pid} ->
            unlink(Pid),
            free()
    end.
```

ADVANCED: A Supervisor Process

sup.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: sup.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(sup).

%% Client Functions
-export([start/1, stop/1, start_child/4]).

%% Internal Exports
-export([init/0]).

%%% @doc Starts an Erlang Process Supervisor
-spec start(atom()) -> {ok, pid()}.
start(Name) ->
    Pid = spawn(sup, init, []),
    register(Name, Pid),
    {ok, Pid}.

%%% @doc Stops an Erlang supervisor, killing all the monitored children
-spec stop(pid() | atom()) -> ok.
stop(Name) ->
```

```
    Name ! stop,
    ok.

%%% @doc Given a module, function and arguments, will start a child
%%% and monitor it. If it terminates abnormally, the child is
%%% restarted.
-spec start_child(atom(), atom(), atom(), [term()]) -> {ok, pid()}.
start_child(Name, Module, Function, Args) ->
    Name ! {start_child, self(), Module, Function, Args},
    receive
        {ok, Pid} -> {ok, Pid}
    end.

%%% @doc Initialises the supervisor state
-spec init() -> ok.
init() ->
    process_flag(trap_exit, true),
    loop([]).

%%% loop([child()]) -> ok.
%%% child() = {pid(), restar_count(), mod(), func(), [args()]}.
%%% restart_count() = integer(). number of times the child has restarted
%%% mod() = atom(). the module where the spawned function is located
%%% func() = atom(). the function spawned
%%% args() = term(). the arguments passed to the function
%%% The supervisor loop which handles the incoming client requests
%%% and EXIT signals from supervised children.
-type child() :: {pid(), non_neg_integer(), atom(), atom(), [term()]}.
-spec loop([child()]) -> ok.
loop(Children) ->
    receive
        {start_child, ClientPid, Mod, Func, Args} ->
            Pid = spawn_link(Mod, Func, Args),
            ClientPid ! {ok, Pid},
            loop([{Pid, 1, Mod, Func, Args}|Children]);
        {'EXIT', Pid, normal} ->
            NewChildren = lists:keydelete(Pid, 1, Children),
            loop(NewChildren);
        {'EXIT', Pid, Reason} ->
            NewChildren = lists:keydelete(Pid, 1, Children),
            {value, Child} = lists:keysearch(Pid, 1, Children),
            {Pid, Count, Mod, Func, Args} = Child,
            error_message(Pid, Count, Reason, Mod, Func, Args),
            NewPid = spawn_link(Mod, Func, Args),
            loop([{NewPid, Count + 1, Mod, Func, Args}|NewChildren]);
        stop ->
            kill_children(Children)
    end.

%%% Kills all the children in the supervision tree.
-spec kill_children([child()]) -> ok.
kill_children([{Pid, _Count, _Mod, _Func, _Args}|Children]) ->
    exit(Pid, kill),
    kill_children(Children);
kill_children([]) ->
    ok.

%%% Prints an error message for the child which died.
```

```
-spec error_message(pid(), non_neg_integer(), term(), atom(), atom(), [term()])
                -> ok.
error_message(Pid, Count, Reason, Mod, Func, Args) ->
    io:format("~50c~n",[$-]),
    io:format("Error: Process ~p Terminated ~p time(s)~n",[Pid, Count]),
    io:format("       Reason for termination:~p~n",[Reason]),
    io:format("       Restarting with ~p:~p/~p~n",[Mod,Func,length(Args)]),
    io:format("~50c~n",[$-]).
```

6 Functional Programming

Higher Order Functions

funs.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File      : funs.erl
%%% Author    : trainers@erlang-solutions.com
%%% Copyright : 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(funs).
-export([print/1, smaller/2, print_even/1, concatenate/1, sum/1]).


print(N) ->
    List = lists:seq(1,N),
    Print = fun(X) -> io:format("~p~n",[X]) end,
    lists:foreach(Print, List).

smaller(List, Size) ->
    Filter = fun(X) when X > Size -> false;
                (_X)              -> true
             end,
    lists:filter(Filter, List).

print_even(N) ->
    Filter = fun(X) -> X rem 2 == 0 end,
    List = lists:seq(1,N),
    FilteredList = lists:filter(Filter, List),
    Print = fun(X) -> io:format("~p~n",[X]) end,
    lists:foreach(Print, FilteredList).

concatenate(ListOfLists) ->
    Concatenate = fun(X, Buffer) -> Buffer ++ X end,
    lists:foldl(Concatenate, [], ListOfLists).

sum(ListOfInts) ->
    Sum = fun(Integer, Buffer) -> Buffer + Integer end,
    lists:foldl(Sum, 0, ListOfInts).
```

List Comprehensions

lc.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File      : lc.erl
%%% Author    : trainers@erlang-solutions.com
%%% Copyright : 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(lc).
```

```
-export([three/0, filtersquare/1, intersection/2, disjunction/2]).

three() ->
    [X || X <- lists:seq(1,10), X rem 3 == 0].

filtersquare(List) ->
    [X*X || X <- List, is_integer(X)].

intersection(List1, List2) ->
    [H || H <- List1, lists:member(H, List2)].

disjunction(List1, List2) ->
    Intersection = intersection(List1, List2),
    [H || H <- List1 ++ List2, not lists:member(H, Intersection)].
```

7 Records and Maps

Database Handling Using Records

db.hrl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File      : db.hrl
%%% Author    : trainers@erlang-solutions.com
%%% Copyright : 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-record(data, {key, value}).
```

db.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: db.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright : 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(db).
-export([new/0, write/3, delete/2, read/2, match/2, destroy/1]).
-include("db.hrl").

-export_type([db/0]).
-type db() :: list().

%% @doc Create a new database
-spec new() -> db().
new() -> [].

%% @doc Insert a new element in the database
-spec write(Key::term(), Val::term(), db()) -> db().
write(Key, Element, Db) -> [#data{key = Key, value = Element}|Db].

%% @doc Remove an element from the database
-spec delete(Key::term(), db()) -> db().
delete(Key, [#data{key = Key}|Db]) ->
    Db;
delete(Key, [Record|Db]) ->
    [Record|delete(Key, Db)];
delete(_Key, []) ->
    [].

%% @doc Retrieve the first element in the database with a matching key
-spec read(Key::term(), db()) -> {ok, term()} | {error, instance}.
read(Key, [#data{key = Key, value = Element}|_Db]) ->
    {ok, Element};
read(Key, [_Record|Db]) ->
    read(Key, Db);
read(_Key, []) ->
    {error, instance}.
```

```
%% @doc Return all the keys whose values match the given element.
-spec match(Val::term(), db()) -> [term()].
match(Element, [#data{key = Key, value = Element}|Db]) ->
    [Key|match(Element, Db)];
match(Element, [_Record|Db]) ->
    match(Element, Db);
match(_Key, []) ->
    [].

%% @doc Deletes the database.
-spec destroy(db()) -> ok.
destroy(_Db) ->
    ok.
```

Database Handling using Maps

db.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: db.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright : 1999-2019 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(db).
-export([new/0, write/3, delete/2, read/2, match/2, destroy/1]).
-export_type([db/0]).
-type db() :: map().

%% @doc Create a new database
-spec new() -> db().
new() ->
    #{}.

%% @doc Insert a new element in the database
-spec write(Key::term(), Val::term(), db()) -> db().
write(Key, Element, Db) ->
    maps:put(Key, Element, Db).

%% @doc Remove an element from the database
-spec delete(Key::term(), db()) -> db().
delete(Key, Db) ->
    maps:remove(Key, Db).

%% @doc Retrieve the first element in the database with a matching key
-spec read(Key::term(), db()) -> {ok, term()} | {error, instance}.
read(Key, Db) ->
    case maps:find(Key, Db) of
        {ok, Element} -> {ok, Element};
        error -> {error, instance}
    end.

%% @doc Return all the keys whose values match the given element.
-spec match(Val::term(), db()) -> [term()].
match(Element, Db) ->
    [ Key || {Key, El} <- maps:to_list(Db),
```

```
            El =:= Element].

%% @doc Deletes the database.
-spec destroy(db()) -> ok.
destroy(_Db) ->
    ok.
```

8 Erlang Term Storage

Database Handling using ETS

db.erl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File: db.erl
%%% @author trainers@erlang-solutions.com
%%% @copyright 1999-2011 Erlang Solutions Ltd.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(db).

%% Client Function Exports
-export([new/0, write/3, delete/2, read/2, match/2, destroy/1]).
-include("db.hrl").
-export_type([db/0]).
-type db() :: ETSTableRef::term().

%% @doc Create a new database. Make it protected, unnamed with key position
%% pointing to the record key.
-spec new() -> db().
new() -> ets:new(db, [{keypos, #data.key}]).

%% @doc Insert a new element in the database
-spec write(Key::term(), Val::term(), db()) -> db().
write(Key, Element, Db) ->
    ets:insert(Db, #data{key = Key, value = Element}),
    Db.

%% @doc Remove an element from the database
-spec delete(Key::term(), db()) -> db().
delete(Key, Db) ->
    ets:delete(Db, Key),
    Db.

%% @doc Retrieve the first element in the database with a matching key
-spec read(Key::term(), db()) -> {ok, term()} | {error, instance}.
read(Key, Db) ->
    case ets:lookup(Db, Key) of
        [#data{value = Element}] -> {ok, Element};
        []                       -> {error, instance}
    end.

%% @doc Return all the keys whose values match the given element.
-spec match(Val::term(), db()) -> [term()].
match(Element, Db) ->
```

```
    lists:flatten(ets:match(Db, #data{key = '$1', value = Element})).

%% @doc Deletes the database.
-spec destroy(db()) -> ok.
destroy(Db) ->
    ets:delete(Db),
    ok.
```