```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : boolean.erl
%%% Author  : trainers@erlang-solutions.com
%%% Created : 21 Jun 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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


% not/1 -> true | false
b_not(false)-> true;
b_not(true )-> false.

% and/2 -> true | false
b_and(true,   true  )-> true ;
b_and(_Bool1, _Bool2)-> false.

% or/2 -> true | false
b_or(true,  _Bool)-> true;
b_or(_Bool, true )-> true;
b_or(false, false)-> false.

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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : sums.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution of the Sequential Programming
%%%           Exercise 1
%%% Created : 25 Jan 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(sums).
-author('francesco@erlang-solutions.com').

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Exercise 1, Evaluating expressions
%% A: sum(int()) -> int()
%% Adds the integers between 1 and N.
sum(0) ->
    0;
sum(N) ->
    N + sum(N-1).

%% B: sum_interval(int()) -> int()
%% Adds the integers between N and M
sum_interval(Max, Max) ->
    Max;
sum_interval(Min, Max) when Min =< Max ->
    Min + sum_interval(Min + 1, Max).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : create.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution of the Sequential Programming
%%%           Exercise 2
%%% Created : 25 Jan 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(create).
-author('francesco@erlang-solutions.com').

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Exercise 2, Creating lists
%% A: create(int()) -> list()
%% Creates a list with integers [1,..,N]
create(N) ->
    create(1, N).

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

%% B: reverse_create(int()) -> list()
%% Creates a list with integers [N,..,1]
reverse_create(1) ->
    [1];
reverse_create(N) ->
    [N | reverse_create(N-1)].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : effects.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution of the Sequential Programming
%%%           Exercise 3
%%% Created : 25 Jan 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(effects).
-author('francesco@erlang-solutions.com').

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Exercise 3, Side Effects
%% A: print(int()) -> ok.
%% Prints the integers between 1 and N.
print(0) ->
    ok;
print(N) ->
    print(N-1),
    io:format("Number:~p~n",[N]).

%% B: even_print(int()) -> ok
%%  Prints the even integers between 1 and N.
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).
```

```erlang
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : db.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution for Exercise 4, sequential programming
%%%           exercises. A database back end module
%%% Created : 10 Jan 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(db).
-author('francesco@erlang-solutions.com').

-export([new/0, write/3, delete/2, read/2, match/2, destroy/1]).

%% new() -> list().
%% Create a new database
new() ->
    [].

%% insert(term(), term(), list()) -> list()
%% Insert a new element in the database
write(Key, Element, []) ->
    [{Key, Element}];
write(Key, Element, [{Key, _} | Db]) ->
    [{Key, Element}|Db];
write(Key, Element, [Current | Db]) ->
    [Current | write(Key, Element, Db)].

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

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

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


%% destroy(list()) -> ok.
%% Deletes the database.

destroy(_Db) ->
    ok.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : manipulating.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution of the Sequential Programming
%%%           Exercise 5
%%% Created : 25 Jan 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(manipulating).
-author('francesco@erlang-solutions.com').

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Exercise 5, Manipulating Lists
%% A: filter(list(), int()) -> list().
%% Given an integer list, returns a new list where the elements
%% come from List and are < Key.
filter([H|T], Key) when H =< Key ->
    [H|filter(T, Key)];
filter([_|T], Key) ->
    filter(T, Key);
filter([], _Key) ->
    [].

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

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

%% C: reverse(list()) -> list()
%% Will reverse the list order.

reverse(List) ->
    reverse(List, []).
reverse([], Buffer) ->
    Buffer;
reverse([H|T], Buffer) ->
    reverse(T, [H|Buffer]).

%% D: flatten(list()) -> list()
%% Takes a list and recursively flattens it.

flatten([H|T]) when is_list(H) ->
    concatenate([flatten(H), flatten(T)]);
flatten([H|T]) ->
    [H|flatten(T)];
flatten([]) ->
    [].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : sorting.erl
%%% Author  : Fred Hebert <fred.hebert@erlang-solutions.com>
%%% Purpose : Solution of the Sequential Programming Exercise 6
%%% Created : 16 Nov 2010 by Fred Hebert
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(sorting).
-export([quicksort/1, mergesort/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Exercise 6, Sorting lists
%% A: quicksort(list()) -> list()
%% Sorts a list using the quicksort algorithm
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
    {Smaller, Larger} = partition(Pivot, Rest),
    quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger).

%% Partition breaks the list into elements smaller or larger
%% Than the pivot
partition(Pivot, List) -> partition(Pivot, List, {[],[]}).

partition(_Pivot, [], Acc) ->
    Acc;
partition(Pivot, [Smaller|Rest], {S,L}) when Smaller =< Pivot ->
    partition(Pivot, Rest, {[Smaller|S], L});
partition(Pivot, [Larger|Rest], {S,L}) ->
    partition(Pivot, Rest, {S, [Larger|L]}).


%% B: mergesort(list()) -> list()
%% Sorts a list with the mergesort algorithm
mergesort([]) -> [];
mergesort([X]) -> [X];
mergesort(L) when is_list(L) ->
    {Left, Right} = split(length(L) div 2, L),
    merge(mergesort(Left), mergesort(Right)).

%% Splits a list at the point N and returns both parts
split(N, List) -> split(N, List, []).
split(0, List, Acc) -> {Acc, List};
split(N, [H|T], Acc) -> split(N-1, T, [H|Acc]).

%% Merges two sorted lists in a single one, still sorted.
merge([], Right) -> Right;
merge(Left, []) -> Left;
merge(Left = [L|Ls], Right = [R|Rs]) ->
    if L =< R -> [L | merge(Ls, Right)];
       L  > R -> [R | merge(Left, Rs)]
    end.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : echo.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Exercise 1, Concurrent Programming
%%% Created : 21 Jun 2001 by Francesco Cesarini <francesco@erlang-solutions.com>
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

%% start() -> ok
%% Spawns the echo server process.

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

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

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

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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : ring.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Exercise 2, Concurrent Programming
%%% Created : 21 Jun 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(ring).
-author('francesco@erlang-solutions.com').

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

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

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

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


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

%%master_loop(int(), term(), pid())
%% 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.
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.

%%loop(int(), pid())
%% 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.
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.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : crossring.erl
%%% Author  : Fred Hebert <fred.hebert@erlang-solutions.com>
%%% Purpose : Exercise 3, Concurrent Programming
%%% Created : 18 Nov 2010
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(crossring).
-author('trainers@erlang-solutions.com').

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

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

%%start(int(), int(), term()) -> Pid.
%% Starts the master process which in turn spawns off the
%% individual processes which will receive a message.
start(ProcNum, MsgNum, Message)->
    spawn(crossring, master, [ProcNum, MsgNum, Message]).

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


%%start_slaves(int(), int(), pid()) -> {pid(), pid()}
%% Will start ProcNum slave processes
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).

%%master_loop(int(), term(), pid(), pid())
%% 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.
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.

%%loop(int(), pid())
%% 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.
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.
```

```
%%% File    : pingpong.erl
%%% Author  :  <simon@erlang-solutions.com>, <martin@erlang-solutions.com>
%%% Description : Sends a message N times between two processes
%%% Created : Dec 2005 by  Simon Aurell and Martin Carlson

-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.
```

```
%%% File:       fussball.erl
%%% Author:         <code@erlang-solutions.com>
%%% Description:    A simple game of Fussball.

-module(fussball).

%% Interface
-export([start/2, init/2, stop/1, kickoff/1]).

start(MyCountry, OtherCountry) ->
    spawn(?MODULE, init, [MyCountry, OtherCountry]),
    ok.

stop(Country) ->
    Country ! stop.

kickoff(Country) ->
    Country ! kick,
    ok.

init(MyCountry, OtherCountry) ->
    process_flag(trap_exit, true),
    register(MyCountry, self()),
    catch link(whereis(OtherCountry)),
    loop(MyCountry, OtherCountry).

loop(MyCountry, OtherCountry) ->
    receive
    {'EXIT', _Pid, Reason} ->
        io:format("Got exit signal ~p~n", [Reason]),
        loop(MyCountry, OtherCountry);
    stop ->
        ok;
    save ->
        io:format("~p just saved...~n", [OtherCountry]),
        loop(MyCountry, OtherCountry);
    score ->
        io:format("Oh no! ~p just scored!!~n", [OtherCountry]),
        loop(MyCountry, OtherCountry);
    kick ->
        timer:sleep(500),
        case random:uniform(1000) of
        N when N > 950 ->
            io:format("~p SAVES! And what a save!!~n", [MyCountry]),
            OtherCountry ! save,
            OtherCountry ! kick;
        N when N > 800 ->
            io:format("~p SCORES!!~n", [MyCountry]),
            OtherCountry ! score;
        _ ->
            io:format("~p kicks the ball...~n", [MyCountry]),
            OtherCountry ! kick
        end,
        loop(MyCountry, OtherCountry)
    end.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : mutex.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution Exercise 2 Process Error Handling
%%%           A reliable binary semaphore
%%% Created : 25 Apr 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(mutex).
-author('francesco@erlang-solutions.com').

%% Client Exports
-export([start/0, signal/0, wait/0]).

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

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

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

%% init() -> ok.
%% Initializes the state machine.

init() ->
    process_flag(trap_exit, true),
    free().

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

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

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

%% free()
%% The state where the semaphore is available
free() ->
    receive
      {wait, Pid} ->
          case catch link(Pid) of  %%Process terminated while
            {'EXIT', _Reason} -> %%Waiting for the signal
                free();
            true ->              %%Process still alive
                Pid ! ok,
                busy(Pid)
          end
    end.

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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% File    : sup.erl
%%% Author  : Francesco Cesarini <francesco@erlang-solutions.com>
%%% Purpose : Solution Exercise 3 Process Error Handling
%%%           Implements a supervisor process
%%% Created : 26 Apr 2001 by Francesco Cesarini
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(sup).
-author('francesco@erlang-solutions.com').

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

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

%%% start(atom()) -> {ok, pid()}
%%% Starts an Erlang Process Supervisor

start(Name) ->
    Pid = spawn(sup, init, []),
    register(Name, Pid),
    {ok, Pid}.

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

stop(Name) ->
    Name ! stop,
    ok.

%%% start_child(atom(), atom(), atom(), [term()]) -> {ok, Pid}
%%% Given a module, function and arguments, will start a child
%%% and monitor it. If it terminates abnormally, the child is
%%% restarted.

start_child(Name, Module, Function, Args) ->
    Name ! {start_child, self(), Module, Function, Args},
    receive
      {ok, Pid} -> {ok, Pid}
    end.

%%% init() -> ok.
%%% Initialises the supervisor state

init() ->
    process_flag(trap_exit, true),
    loop([]).

%%% loop([child()]) -> ok.
%%% child() -> {pid(), restar_count(), mod(), func(), [args()]}
%%% restart_count() -> number of times the child has restarted
%%% mod(), func() -> atom(), the module & 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.

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.

%%% kill_children([child()]) -> ok
%%% Kills all the children in the supervision tree.

kill_children([{Pid, Count, Mod, Func, Args}|Children]) ->
    exit(Pid, kill),
    kill_children(Children);
kill_children([]) ->
    ok.

%%% error_message(pid(), int(), term(), atom(), atom(), [term()]) -> ok.
%%% Prints an error message for the child which died.

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