Contents

1 Introduction
   • Motivation

2 Erlang Programming Language
   • Sequential Erlang
   • Concurrent Erlang

3 Further reading
Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system.

The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing.

It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system.

The first version was developed by Joe Armstrong in 1986. It was originally a proprietary language within Ericsson, but was released as open source in 1998.
Motivation

Companies and projects using Erlang:

- Amazon
- Yahoo!
- Facebook
- T-Mobile
- Motorola
- Ericsson
- Tuenti
- GitHub
- Goldman Sachs
- WhatsApp
- Battlestar Galactica
- Call of Duty
Sequential Erlang

Numbers:

```
1> 2 + 15.
17
2> 49 * 100.
4900
3> 1892 - 1472.
420
4> 5 / 2.
2.5
5> 5 div 2.
2
6> 5 rem 2.
1
7> (50 * 100) - 4999.
1
8> -(50 * 100 - 4999).
-1
9> -50 * (100 - 4999).
244950
```
Sequential Erlang

Variables & Pattern-Matching:

```
1> One.
   \* 1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
   ** exception error: no match of right hand side value 3
7> two = 2.
   ** exception error: no match of right hand side value 2
8> 47 = 45 + 2.
47
9> 47 = 45 + 3.
   ** exception error: no match of right hand side value 48
10> _ = 14+3.
17
11> _.
   \* 1: variable '_' is unbound
```
Sequential Erlang

Atoms:

```
1> atom.  
atom  
2> atoms_rule.  
atoms_rule  
3> atoms_rule@erlang.  
atoms_rule@erlang  
4> 'Atoms can be cheated!'.  
'Atoms can be cheated!'  
5> atom = 'atom'.  
atom  
```
Sequential Erlang

Boolean Algebra & Comparison Operators:

```
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
6> 5 =:= 5.
true
7> 1 =:= 0.
false
8> 1 =/= 0.
true
9> 5 =:= 5.0.
false
10> 5 == 5.0.
true
11> 5 /= 5.0.
false
```
Sequential Erlang

Boolean Algebra & Comparison Operators:

```
12> 1 < 2.
true
13> 1 < 1.
false
14> 1 >= 1.
true
15> 1 =< 1.
true
```

```
13> 5 == true.
false
14> 0 == false.
false
15> 1 < false.
true
```
Sequential Erlang

Tuples:

```
1> X = 10, Y = 4.
4
2> Point = {X,Y}.
{10,4}

>f().  %clear all variable names

3> Point = {4,5}.
{4,5}
4> {X,Y} = Point.
{4,5}
5> X.
4
6> {X,} = Point.
{4,5}
7> {_,} = {4,5}.
{4,5}
8> {_,} = {4,5,6}.
** exception error: no match of right hand side value {4,5,6}

10> PreciseTemperature = {celsius, 23.213}.
{celsius,23.213}
11> {kelvin, T} = PreciseTemperature.
** exception error: no match of right hand side value {celsius,23.213}
12> {point, {X,Y}}.
{point,{4,5}}
```
Sequential Erlang

Lists:

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

2> [97, 98, 99].
"abc"

3> [97,98,99,4,5,6].
[97,98,99,4,5,6]

4> [233].
"é"

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].
[2]

9> [1,2,3] -- [1,2] -- [3].
[3]

10> [1,2,3] -- [1,2] -- [2].
[2,3]
Sequential Erlang

Lists:

```
11> hd([1,2,3,4]).
1
12> tl([1,2,3,4]).
[2,3,4]
13> List = [2,3,4].
[2,3,4]
14> NewList = [1|List].
[1,2,3,4]
15> [Head|Tail] = NewList.
[1,2,3,4]
16> Head.
1
17> Tail.
[2,3,4]
[2,3,4]
19> NewHead.
2
20> [ 1  |  []].
[1]
21> [ 2  |  [1  |  []]].
[2,1]
[3,2,1]
```
Sequential Erlang

List comprehensions:

\[
\{2 \times n \mid n \in L\} \text{ where } L = \{1, 2, 3, 4\}
\]

\[
\{x \mid x \in L \land x \mod 2 = 0\} \text{ where } L = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}
\]

Prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.
Sequential Erlang

List comprehensions:

\[ \{ 2 \times n \mid n \in L \} \text{ where } L = \{1, 2, 3, 4\} \]

\[ \{ x \mid x \in L \land x \mod 2 = 0 \} \text{ where } L = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

Prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.
Sequential Erlang

List comprehensions:

\[
\{ 2 \times n \mid n \in L \} \text{ where } L = \{1, 2, 3, 4\}
\]

\[
\{ x \mid x \in L \land x \mod 2 = 0 \} \text{ where } L = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}
\]

Prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.
Sequential Erlang

List comprehensions:

\[ \{ 2 \times n \mid n \in L \} \text{ where } L = \{1, 2, 3, 4\} \]

\[ \{ x \mid x \in L \land x \mod 2 = 0 \} \text{ where } L = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

Prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.
Sequential Erlang

List comprehensions:

\[ \{ 2 \times n \mid n \in L \} \text{ where } L = \{1, 2, 3, 4\} \]

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

\[ \{ x \mid x \in L \land x \text{ mod 2} = 0 \} \text{ where } L = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

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

Prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.
Sequential Erlang

List comprehensions:

\[ \{ 2 \times n \mid n \in L \} \text{ where } L = \{1, 2, 3, 4\} \]

\[
1> \{2*N \mid \mathbf{N} <- [1,2,3,4]\}. \n\]
\[
[2,4,6,8] \n\]

\[ \{ x \mid x \in L \land x \mod 2 = 0 \} \text{ where } L = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \]

\[
2> \{ x \mid \mathbf{x} <- [1,2,3,4,5,6,7,8,9,10], x \mod 2 =: 0\}. \n\]
\[
[2,4,6,8,10] \n\]

Prices of all the items costing between $3 and $10 with taxes (say 7%) counted in afterwards.

\[
3> \text{RestaurantMenu} = \{\{\text{steak}, 5.99\}, \{\text{beer}, 3.99\}, \{\text{poutine}, 3.50\}, \{\text{kitten}, 20.99\}, \{\text{water}, 0.00\}\}.
\]

\[
4> \{\{\text{Item}, \mathbf{Price} \times 1.07\} \mid \{\text{Item}, \mathbf{Price}\} <- \text{RestaurantMenu}, \mathbf{Price} \geq 3, \mathbf{Price} \leq 10\}. \n\]
\[
\{\{\text{steak}, 6.4093\}\}, \{\text{beer}, 4.2693\}, \{\text{poutine}, 3.745\}\} \n\]
Sequential Erlang

List comprehensions:

\[ \{ x + y \mid x \in L_x \land y \in L_y \} \text{ where } L_x = \{1, 2\} \text{ and } L_y = \{2, 3\} \]

Get all the places where it is foggy
Sequential Erlang

List comprehensions:

\[ \{ x + y \mid x \in L_x \land y \in L_y \} \text{ where } L_x = \{1, 2\} \text{ and } L_y = \{2, 3\} \]

Get all the places where it is foggy

\[ 5> \ [X+Y \ || \ X <- [1,2], Y <- [2,3]]. \]
\[ [3,4,4,5] \]
Sequential Erlang

List comprehensions:

\[ \{ x + y \mid x \in L_x \land y \in L_y \} \text{ where } L_x = \{1, 2\} \text{ and } L_y = \{2, 3\} \]

5> [X+Y || X <- [1,2], Y <- [2,3]].
[3,4,4,5]

Get all the places where it is foggy
List comprehensions:

\[ \{ x + y \mid x \in L_x \land y \in L_y \} \text{ where } L_x = \{1, 2\} \text{ and } L_y = \{2, 3\} \]

Get all the places where it is foggy

```
5> [X+Y || X <- [1,2], Y <- [2,3]].
[3,4,4,5]
6> Weather = [{toronto, rain}, {montreal, storms}, {london, fog},
7>             {paris, sun}, {boston, fog}, {vancouver, snow}].
```

```
6> Weather = [{toronto, rain},
             {montreal, storms},
             {london, fog},
             {paris, sun},
             {boston, fog},
             {vancouver, snow}]
7> FoggyPlaces = [X || {X, fog} <- Weather].
[London, boston]
```
Sequential Erlang

Functions:

Suppose you want to write this function in Erlang:

```erlang
function greet(Gender, Name)  
  if Gender == male then
    print("Hello, Mr. %s!", Name)
  else if Gender == female then
    print("Hello, Mrs. %s!", Name)
  else
    print("Hello, %s!", Name)
end
```

It would look like this:
Sequential Erlang

Functions:

Suppose you want to write this function in Erlang:

```erlang
function greet(Gender, Name)  
  if Gender == male then
    print("Hello, Mr. %s!", Name)
  else if Gender == female then
    print("Hello, Mrs. %s!", Name)
  else
    print("Hello, %s!", Name)
end
```

It would look like this:

```erlang
greet(male, Name) ->
  io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
  io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
  io:format("Hello, ~s!", [Name]).
```
Sequential Erlang

Functions:

```erlang
head([H|_]) -> H.
second([_|Y|_]) -> Y.

same(X,X) ->
  true;
same(_,_) ->
  false.

valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
io:format("The Date tuple (~p) says today is: ~p/~p/~p,~n",[Date,Y,M,D]),
io:format("The time tuple (~p) indicates: ~p:~p:~p,~n", [Time,H,Min,S]);
valid_time(_) ->
io:format("Stop feeding me wrong data!~n").
```
Sequential Erlang

Functions:

```erlang
head([H|_]) -> H.
second([_,X|_]) -> X.

same(X,X) ->
    true;
same(_,_) ->
    false.

valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
    io:format("The Date tuple (~p) says today is: ~p/~/p/~/p,~n",[Date,Y,M,D]),
    io:format("The time tuple (~p) indicates: ~p:~p:~p.~n", [Time,H,Min,S]);
valid_time(_) ->
    io:format("Stop feeding me wrong data!~n").
```

```erlang
5> functions:valid_time({[2011,09,06],[09,04,43]}).
The Date tuple (2011,9,6) says today is: 2011/9/6,
The time tuple (9,4,43) indicates: 9:4:43.
ok
6> functions:valid_time({[2011,09,06],[09,04]}).
Stop feeding me wrong data!
ok
```
Sequential Erlang

Guards:

old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
Sequential Erlang

Guards:

```erlang
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

```erlang
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```
Sequential Erlang

Guards:

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_)  -> true.
```

```
old_enough(X) when X >= 16 -> true;
old_enough(_)  -> false.
```

```
right_age(X) when X >= 16, X =< 104 ->
    true;
right_age(_)   ->
    false.
```

```
wrong_age(X) when X < 16; X > 104 ->
    true;
wrong_age(_)   ->
    false.
```
Sequential Erlang

case-of expressions:

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

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

case-of expressions:

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

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

```erlang
beachf({celsius, N}) when N >= 20, N <= 45 ->
  'favorable';
...
beachf(_) ->
  'avoid beach'.
```
if expressions:

```erlang
heh_fine() ->
    if 1 ==:= 1 ->
      works
    end,
    if 1 ==:= 2; 1 ==:= 1 ->
      works
    end,
    if 1 ==:= 2, 1 ==:= 1 ->
      fails
    end.
```

```erlang
oh_god(N) ->
    if N ==:= 2 -> might_succeed;
      true -> always_does  %% this is Erlang's if's 'else'
    end.
```

```erlang
help_me(Animal) ->
    Talk = if Animal == cat -> "meow";
             Animal == beef -> "mooo";
             Animal == dog -> "bark";
             Animal == tree -> "bark";
             true -> "fgdudfngna"
     end,
    {Animal, "says " ++ Talk ++ ",!"}.
```
if expressions:

\[
\begin{align*}
\text{if } X > Y & \rightarrow a() \\
; \text{true} & \rightarrow b() \\
\end{align*}
\]

\[
\begin{align*}
\text{if } X > Y & \rightarrow a() \\
; X =< Y & \rightarrow b() \\
\end{align*}
\]

\[
\begin{align*}
\text{if } X > Y & \rightarrow a() \\
; X < Y & \rightarrow b() \\
; \text{true} & \rightarrow c() \\
\end{align*}
\]

\[
\begin{align*}
\text{if } X > Y & \rightarrow a() \\
; X =< Y & \rightarrow b() \\
; X == Y & \rightarrow c() \\
\end{align*}
\]
Sequential Erlang

Modules:

```
1> erlang:element(2, {a,b,c}).
b
2> element(2, {a,b,c}).
b
3> lists:seq(1,4).
[1,2,3,4]
4> seq(1,4).
** exception error: undefined shell command seq/2
```
Sequential Erlang

Modules:

```
1> erlang:element(2, {a,b,c}).
  b
2> element(2, {a,b,c}).
   b
3> lists:seq(1,4).
   [1,2,3,4]
4> seq(1,4).
   ** exception error: undefined shell command seq/2
```

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

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

%%% Shows greetings.
%%% io:format/1 is the standard function used to output text.
hello() ->
    io:format("Hello, world!\n").

greet_and_add_two(X) ->
    hello(),
    add(X,2).
```
Sequential Erlang

Modules:

```erlang
1> cd("/path/to/where/you/saved/the-module/").
"Path Name to the directory you are in"
ok

2> c(useless).
{ok,useless}

3> useless:add(7,2).
9

4> useless:hello().
Hello, world!
ok

5> useless:greet_and_add_two(-3).
Hello, world!
ok

6> useless:not_a_real_function().
** exception error: undefined function useless:not_a_real_function/0
```
Sequential Erlang

Higher order functions:

```erlang
-module(hhfuns).
-compile(export_all).

one() -> 1.
two() -> 2.
add(X,Y) -> X() + Y().
```
Sequential Erlang

Higher order functions:

```erlang
-module(hhfuns).
-compile(export_all).

one() -&gt; 1.
two() -&gt; 2.
add(X,Y) -&gt; X() + Y().
```

```
1> c(hhfuns).
{ok, hhfuns}
2> hhfuns:add(one,two).
** exception error: bad function one
   in function hhfuns:add/2
3> hhfuns:add(1,2).
** exception error: bad function 1
   in function hhfuns:add/2
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```
Sequential Erlang

Higher order functions:

```
map(_, []) -> [];  
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.  
decr(X) -> X - 1.
```

Module lists includes common examples of this abstractions like lists:map/2, lists:filter/2, lists:foldl/3 or lists:foldr/3.
Sequential Erlang

Higher order functions:

```
map(_, []) -> [];  
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.  
decr(X) -> X - 1.
```

```
1> c(hhfuns).
{ok, hhfuns}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hhfuns:map(fun hhfuns:incr/1, L).
[2,3,4,5,6]
4> hhfuns:map(fun hhfuns:decr/1, L).
[0,1,2,3,4]
```

Module lists includes common examples of this abstractions like lists:map/2, lists:filter/2, lists:foldl/3 or lists:foldr/3.
### Sequential Erlang

#### Higher order functions:

```erlang
c(lists:map(fun X -> X + 1 end, [1, 2, 3, 4, 5]).
c(lists:map(fun X -> X - 1 end, [1, 2, 3, 4, 5]).
```

Module `lists` includes common examples of this abstractions like `lists:map/2`, `lists:filter/2`, `lists:foldl/3` or `lists:foldr/3`. 
Sequential Erlang

Anonymous functions:

```erlang
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hhfun:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hhfun:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

```erlang
11> PrepareAlarm = fun(Room) ->
    io:format("Alarm set in ~s~n", [Room]),
    fun() -> io:format("Alarm tripped in ~s! Call Batman!~n", [Room])
end.
11> end.
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```
Sequential Erlang

Anonymous functions:

```erlang
case A of
  0 -> 0;
  _ -> B end.
```

```erlang
16> math:pow(5, 2).
25.0
17> Base = 2.
2
18> PowerOfTwo = fun(X) -> math:pow(Base, X) end.
#Fun<erl_eval.6.13229925>
17> hhfuns:map(PowerOfTwo, [1, 2, 3, 4]).
[1.0, 4.0, 8.0, 16.0]
```

```erlang
fun(Arg1) ->
  Expression1, Exp2, ..., ExpN;
fun(Arg2) ->
  Expression1, Exp2, ..., ExpN;
fun(Arg3) ->
  Expression1, Exp2, ..., ExpN
end
```
Sequential Erlang

Records:

```erlang
-module(records).
-compile(export_all).

-record(robot, {name, type=industrial, hobbies, details=[]}).

first_robot() ->
    #robot{name="Mechatron",
        type=handmade,
        details=["Moved by a small man inside"]).
```
Records:

```erlang
-module(records).
-compile(export_all).

-record(robot, {name, type=industrial, hobbies, details=[]}).

define first robot:
  #robot{name="Mechatron", type=handmade, details=['Moved by a small man inside']}.

1> c(records).
{ok, records}
2> records: first robot().
{robot,"Mechatron", handmade, undefined, ['Moved by a small man inside']}
Sequential Erlang

Records:

```
5> Crusher = #robot{name="Crusher", hobbies=["Crushing people","petting cats"]}.
#robot{name = "Crusher", type = industrial,
    hobbies = ["Crushing people","petting cats"],
    details = []}
6> Crusher#robot.hobbies.
["Crushing people","petting cats"]
```

```
repairman(Rob) ->
    Details = Rob#robot.details,
    NewRob = Rob#robot{details=["Repaired by repairman"|Details]},
    {repaired, NewRob}.
```
Sequential Erlang

Records:

```erlang
5> Crusher = #robot{name="Crusher", hobbies=["Crushing people","petting cats"]}.
   #robot{name = "Crusher", type = industrial,
         hobbies = ["Crushing people","petting cats"],
         details = []}
6> Crusher#robot.hobbies.
   ["Crushing people","petting cats"]
```

```erlang
repairman(Rob) ->
    Details = Rob#robot.details,
    NewRob = Rob#robot{details=["Repaired by repairman"|Details]},
    {repaired, NewRob}.
```

```erlang
17> records:repairman(#robot{name="Ulbert", hobbies=["trying to have feelings"]}).
   {repaired,#robot{name = "Ulbert", type = industrial,
                   hobbies = ["trying to have feelings"],
                   details = ["Repaired by repairman"]}}
```
Sequential Erlang

Records:

```erlang
-record(user, {id, name, group, age}).

%%% use pattern matching to filter
admin_panel(#user{name=Name, group=admin}) ->
    Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
    Name ++ " is not allowed".

%%% can extend user without problem
adult_section(U = #user{}) when U#user.age >= 18 ->
    %%% Show stuff that can't be written in such a text
    %%% allowed;
adult_section(_) ->
    %%% redirect to sesame street site
    forbidden.
```
Sequential Erlang

Other data structures:

**Key-Value Stores**
- Property lists (*proplists*), dictionaries (*dict*), ordered dictionaries (*orddict*) and general balanced trees (*gb_trees*).

**Arrays**
- Module *array*.

**Sets**
- Classification according its internal representation: Not defined (*sets*), ordered lists (*ordsets*) or general balanced trees (*gb_sets*). Additionally, it is provided a module to manipulate set of sets (*sofs*).

**Directed Graphs**
- Two modules: *digraph* and *digraph_utils*.

**Queues**
- A double-ended FIFO queue (*queue*).

**ETS**
- Erlang built-In Term Storage.
try-catch operator:

```erlang
-module(exceptions).
-compile(export_all).

throws(F) ->
    try F() of
    _ -> ok
    catch
      Throw -> {throw, caught, Throw}
    end.
```
## Sequential Erlang

### try-catch operator:

```erlang
-module(exceptions).
-compile(export_all).

throws(F) ->
    try F() of
    _ -> ok
    catch
    Throw -> {throw, caught, Throw}
    end.
```

```erlang
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
```
Sequential Erlang

try-catch operator:

```erlang
errors(F) ->
    try F() of
    _ -> ok
    catch
        error:Error -> {error, caught, Error}
    end.

exits(F) ->
    try F() of
    _ -> ok
    catch
        exit:Exit -> {exit, caught, Exit}
    end.
```
Sequential Erlang

**try-catch operator:**

```erlang
errors(F) ->
    try F() of
    _ -> ok
    catch
        error:Error -> {error, caught, Error}
    end.

exits(F) ->
    try F() of
    _ -> ok
    catch
        exit:Exit -> {exit, caught, Exit}
    end.
```

```
4> c(exceptions).
{ok,exceptions}
5> exceptions:errors(fun() -> erlang:error("Die!") end).
{error,caught,"Die!"}
6> exceptions:exits(fun() -> exit(goodbye) end).
{exit,caught,goodbye}
```
Sequential Erlang

**try-catch operator:**

```
whoa() ->
    try
        talk(),
        _Knight = "None shall Pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    of
        tequila -> "hey this worked!"
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

```
im_impressed() ->
    try
        talk(),
        _Knight = "None shall Pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```
Sequential Erlang

catch operator:

```erlang
1> catch throw(whoa).
   whoa
2> catch exit(die).
   {'EXIT',die}
3> catch 1/0.
   {'EXIT',{badarith,[[erlang,'/',[1,0]],
                   {erl_eval,do_apply,5},
                   {erl_eval,expr,5},
                   {shell,exprs,6},
                   {shell,eval_exprs,6},
                   {shell,eval_loop,3}]}},
4> catch 2+2.
   4
```

catcher(X,Y) ->
case catch X/Y of
   {'EXIT', {badarith,_}} -> "uh oh";
   N -> N
end.
Sequential Erlang

catch operator:

```erlang
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{'EXIT',{badarith,{{erlang,'/',[1,0]},
                   {erl_eval,do_apply,5},
                   {erl_eval,expr,5},
                   {shell,exprs,6},
                   {shell,eval_exprs,6},
                   {shell,eval_loop,3}}}]
4> catch 2+2.
4
```

catcher(X,Y) ->
case catch X/Y of
  {'EXIT', {badarith,_}} -> "uh oh";
  N -> N
end.

```erlang
6> c(exceptions).
{ok,exceptions}
7> exceptions:catcher(3,3).
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```
Concurrent Erlang

Function `spawn/1`:

1> F = fun() -> 2 + 2 end.
#Fun<erl_eval.20.67289768>
2> spawn(F).
<0.44.0>

3> spawn(fun() -> io:format("~p~n", [2 + 2]) end).
4
<0.46.0>

4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.
#Fun<erl_eval.6.13229925>
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].
[<0.273.0>,<0.274.0>,<0.275.0>,<0.276.0>,<0.277.0>,
 <0.278.0>,<0.279.0>,<0.280.0>,<0.281.0>,<0.282.0>]

10
9
8
7
6
5
4
3
2
1
Concurrent Erlang

The message passing operator (!):

9> self() ! hello.
   hello

10> self() ! self() ! double.
   double

11> flush().
   Shell got hello
   Shell got double
   Shell got double
   ok
Concurrent Erlang

The receive statement:

```erlang
-module(dolphins).
-compile(export_all).

dolphin1() ->
    receive
        do_a_flip ->
            io:format("How about no?\n");
        fish ->
            io:format("So long and thanks for all the fish!\n");
        _ ->
            io:format("Heh, we're smarter than you humans.\n")
    end.
```
Concurrent Erlang

The receive statement:

```erlang
-module(dolphins).
-compile(export_all).

dolphin1() ->
    receive
        do_a_flip ->
            io:format("How about no?~n");
        fish ->
            io:format("So long and thanks for all the fish!~n");
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
    end.
```

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin = spawn(dolphins, dolphin1, []).
<0.40.0>
13> Dolphin ! "oh, hello dolphin!".
Heh, we're smarter than you humans.
"oh, hello dolphin!"
14> Dolphin ! fish.
fish
15>
```
Concurrent Erlang

The receive statement:

dolphin2() ->
   receive
       {From, do_a_flip} ->
           From ! "How about no?";
       {From, fish} ->
           From ! "So long and thanks for all the fish!";
       _ ->
       io:format("Heh, we're smarter than you humans.~n")
   end.
Concurrent Erlang

The receive statement:

```
adolphin2() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?";
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
    end.
```

```
11> c(dolphins).
   {ok, dolphins}
12> Dolphin2 = spawn(dolphins, dolphin2, []).
   <0.65.0>
13> Dolphin2 ! {self(), do_a_flip}.
   {<0.32.0>,do_a_flip}
14> flush().
   Shell got "How about no?"
   ok
```
Concurrent Erlang

The receive statement:

```erlang
dolphin3() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?",
            dolphin3();
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.\n"),
            dolphin3()
    end.
```
Concurrent Erlang

The receive statement:

dolphin3() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?",
            dolphin3();
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
    _ ->
        io:format("Heh, we're smarter than you humans.~n"),
        dolphin3()
    end.

15> Dolphin3 = spawn(dolphins, dolphin3, []).
<0.75.0>
16> Dolphin3 ! Dolphin3 ! {self(), do_a_flip}.
<{0.32.0},do_a_flip}
17> flush().
Shell got "How about no?"
Shell got "How about no?"
ok
18> Dolphin3 ! {self(), unknown_message}.
Heh, we're smarter than you humans.
<{0.32.0},unknown_message}
19> Dolphin3 ! Dolphin3 ! {self(), fish}.
<{0.32.0},fish}
20> flush().
Shell got "So long and thanks for all the fish!"
ok
Further reading

http://www.tryerlang.org/
http://learnyouosomeerlang.com/
http://tidier.softlab.ntua.gr/
http://erldocs.com/
http://groups.google.com/group/erlang-programming
Thanks for listening!