ERLANG BASIC COURSE

AGENDA

- Main characteristics and brief history
- The Erlang shell
- Basic types and syntax
  - variables
  - garbage collector
  - atoms
  - numbers and math operators
  - boolean algebra and term comparison
  - tuples
  - lists and strings
  - maps
  - references
  - pids
  - pattern matching

AGENDA

- Sequential programming
  - modules and functions
  - BIFs
  - guards
  - conditional evaluation
  - preprocessor, macros and records
  - list comprehensions

AGENDA

- Sequential programming
  - recursion
  - library modules
  - error handling
  - funs and higher order functions
  - bit syntax

AGENDA

- Concurrent programming
  - processes and message passing
  - links and monitors
  - exit signals
  - supervision

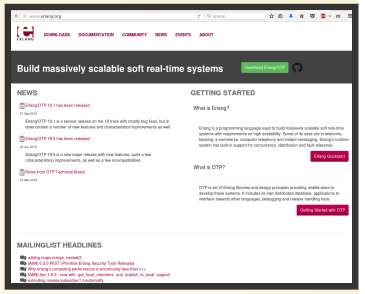
AGENDA

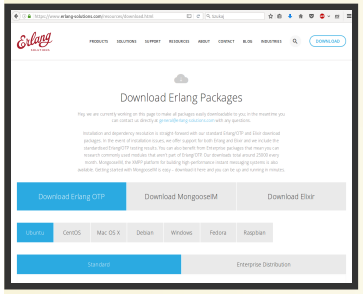
- Robust Systems
  - code loading and upgrade
  - BEAM - the Erlang VM
  - process skeletons
  - introduction to troubleshooting

ONLINE RESOURCES

www.erlang.org

Basic info, Erlang downloads, documentation, tutorials, user guides and much more



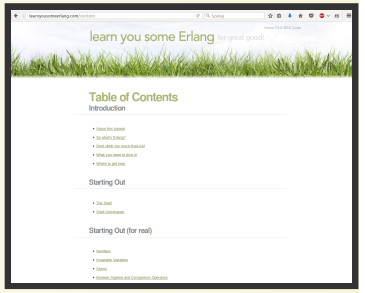


erlang-solutions.com

Erlang packages for many platforms

www.learnyousomeerlang.com

Great online tutorial



BRIEF INTRODUCTION

Learning Erlang/OTP is easy, as it is a very compact language and framework. The difficult part is unlearning other programming language models and getting rid of the bad habits they bring with them.

Francesco Cesarini/Mike Williams
"The route to the successful adoption of non-mainstream programming languages"

MAIN CHARACTERISTICS

- declarative language
- functional (?)
- BEAM virtual machine
- concurrency
- scalability
- robustness (9 nines)
- ...oriented

MAIN CHARACTERISTICS

- strongly and dynamicly typed
- fault tolerant
- soft real-time
- easily distributed
- open source since Dec 1998

HISTORY

- mid-80s: first works on new language in Ericsson Computer Science Laboratory
- 1991: first C-based Erlang VM
- 1995: first Erlang release
- 1996: OTP framework (Open Telecom Platform)
- 1998: Erlang becomes open-source (EPL license)

THE SHELL

How to start and stop the shell

```
matt@liliput:~$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
matt@liliput:~$ erl +Bc
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1> ** exception exit: killed
1> q().
ok
2> matt@liliput:~$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1> init:stop().
ok
2> matt@liliput:~$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1> halt().
matt@liliput:~$
```

THE SHELL

Job Control Mode: CTRL+G

```
matt@liliput:~$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1>
User switch command
 --> j
   1* {shell,start,[init]}
 --> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q                 - quit erlang
  ? | h             - this message
 --> k 1
 --> j
 --> s
 --> j
   2* {shell,start,[]}
 --> c 2
Eshell V7.3  (abort with ^G)
1>
```

THE SHELL

```
1> help().
** shell internal commands **
b()          -- display all variable bindings
e(N)         -- repeat the expression in query <N>
f()          -- forget all variable bindings
f(X)         -- forget the binding of variable X
h()          -- history
history(N)   -- set how many previous commands to keep
results(N)   -- set how many previous command results to keep
catch_exception(B) -- how exceptions are handled
v(N)         -- use the value of query <N>
rd(R,D)      -- define a record
rf()         -- remove all record information
rf(R)        -- remove record information about R
rl()         -- display all record information
rl(R)        -- display record information about R
rp(Term)     -- display Term using the shell's record information
rr(File)     -- read record information from File (wildcards allowed)
rr(F,R)      -- read selected record information from file(s)
rr(F,R,O)    -- read selected record information with options
```

THE SHELL

```
** commands in module c **
bt(Pid)     -- stack backtrace for a process
c(File)     -- compile and load code in <File>
cd(Dir)     -- change working directory
flush()     -- flush any messages sent to the shell
help()      -- help info
i()         -- information about the system
ni()        -- information about the networked system
i(X,Y,Z)    -- information about pid <X,Y,Z>
l(Module)   -- load or reload module
lc([File]) -- compile a list of Erlang modules
ls()        -- list files in the current directory
ls(Dir)     -- list files in directory <Dir>
m()         -- which modules are loaded
m(Mod)      -- information about module <Mod>
memory()    -- memory allocation information
memory(T)   -- memory allocation information of type <T>
nc(File)    -- compile and load code in <File> on all nodes
nl(Module) -- load module on all nodes
pid(X,Y,Z) -- convert X,Y,Z to a Pid
pwd()       -- print working directory
q()         -- quit - shorthand for init:stop()
regs()      -- information about registered processes
nregs()     -- information about all registered processes
uptime()    -- print node uptime
xm(M)       -- cross reference check a module
y(File)     -- generate a Yecc parser
** commands in module i (interpreter interface) **
ih()        -- print help for the i module
```

THE SHELL

```
1> 2 + 2.
4
2> 2.098e8.
2.098e8
3> v(-1) +5.0e7.
2.598e8
4> T={4,b,[4,alpha,0.7,{key1, 45}],{beta, 88},0,0,[]}.
{4,b,[4,alpha,0.7,{key1,45}],{beta,88},0,0,[]}
5> erlang:element(3,T).
[4,alpha,0.7,{key1,45}]
6> erlang:element(7,T).
[]
7> erlang:element(8,T).
** exception error: bad argument
in function  element/2
called as element(8,{4,b,[4,alpha,0.7,{key1,45}],{beta,88},0,0,[]})
```

BASIC TYPES

VARIABLES

- variable is an expression!
- name starts with a capital letter or '_'
  and can contain letters, digits, '_' and '@'
- have 2 states: unbound and bound to a value
- store any type of data
- are not declared
- get values by pattern matching
- are immutable - do not change the value once bound
- scope is the function clause where defined
- special variable: '_' - has no bound state and matches with every term

```
1> A = (1+2)*3.
9
2> A.
9
3> A + A.
18
4> B.
* 1: variable 'B' is unbound
5> B = A + 7.
16
6> C = B + A - 0.88.
24.12
7> C.
24.12
8> A = A + 3.
** exception error: no match of right hand side value 12
9> f(A).
ok
10> A.
* 1: variable 'A' is unbound
```

```
11> B.
16
12> b().
B = 16
C = 24.12
ok
13> _ = 7.
7
14> _.
* 1: variable '_' is unbound
15> _T = 8 + B.
24
16> T.
* 1: variable 'T' is unbound
17> _T.
24
```

GARBAGE COLLECTOR

- separate per process
- copying
- generational
- triggered by
  - memory limit exceeded (after GC the limit is increased)
  - `erlang:garbage_collect/[0-2]` invoked

Read more:
erlang-garbage-collection-details-and-why-it-matters.html

ATOMS

- literal constants with a name
- name starts with lowercase letter
  and can contain letters, digits, `'@'`, `'_'` and `'.'`
- any character is allowed if the atom is encapsulated within single quotes (`' '`)
- atoms are used as tags or enumerators
- the only allowed operation on atoms is comparison
- atoms are stored in a separate heap **which is not garbage collected**

ATOMS

```
1> atom.
atom
2> 'Atom'.
'Atom'
3> teddy_bear@home.
teddy_bear@home
4> is_atom(v(-1)).
true
5> '*#$!  __+IoJe""'.
'*#$!  __+IoJe""'
6> is_atom(v(-1)).
true
7> '1
7> 2
7> 3
7> 4
7> '.
'1\n2\n3\n4\n'
8> '\n'.
'\n'
```

NUMBERS: INTEGERS AND FLOATS

Integers

- positive/negative, size not limited by Erlang VM
- denoted with bases from 2 to 36
  for base =/= 10 the *Base#Value* notation is used
- ASCII character values: *$Char*

Floats

- represent real numbers
- double precision numbers(64-bit IEEE 754-1985)
- written with decimal point, also in scientific notation

```
9> 11.
11
10> 0.0000058.
5.8e-6
11> 16#88fe0a.
8977930
12> 36#zzi88.
60443144
13> 37#e088l.
* 1: illegal base '37'
13> $W.
87
14> $x.
120
15> $\n.
10
16> 1.234E-10.
1.234e-10
```

BOOLEAN ALGEBRA

There is no in-built boolean() type in Erlang.
Atoms 'true' and 'false' serve this purpose.
A few examples:

```
17> 8 > 7.
true
18> 8.0 < 7.
false
19> atom == 'atom'.
true
20> 8 == 8.0.
true
21> false and (8 > 7).
false
22> is_boolean(true).
true
23> is_boolean(false).
true
```

Logical operators:

Operator	Description
and	returns 'true' if both arguments are 'true'
andalso	short-circuit of 'and'; lazy evaluation of arguments
or	returns 'true' if any arguments is 'true'
orelse	short-circuit of 'or'; lazy evaluation of arguments
xor	returns 'true' if one argument is 'true' and the other is 'false'
not	unary negation operator

Comparison operators:

Operator	Description
==	equal to
/=	not equal to
=:=	identical (exactly equal) to
=/=	not identical (exactly not equal) to
=<	less than or equal to
<	less than
>=	greater than or equal to
>	greater than

Non numeric types have the following order:

```
number < atom < ref < fun < port < pid < tuple < map < nil < list < bitstring
```

```
26> false and true.
false
27> false andalso true.
false
28> true xor true.
false
29> 2 < 9.
true
30> 4 == 4.0.
true
31> 4 =:= 4.0.
false
32> 1 < false.
true
33> 0 == false.
false
34> 1 =/= 0.
true
```

TUPLES

- compound data type of **fixed** number of terms:

```
{Elem1,Elem2,...,ElemN}
```

- constant access time to every element
- cannot be modified once created

```
35> P = {adam, 24, {july, 29}}.
{adam,24,{july,29}}
36> element(1, P).
adam
37> P2 = setelement(2, P, 25).
{adam,25,{july,29}}
38> tuple_size(P).
3
39> X = {}.
{}
40> tuple_size(X).
0
```

LISTS AND STRINGS

- compound data type of **variable** number of terms:

```
[Elem1, Elem2, ..., ElemN]
```

- linear access time (sequential access)
- dynamically sized
- created as linked lists (element + pointer to the remainder)
- strings (text) represented as **char lists**

LISTS AND STRINGS

- **head** is the first element of non-empty list
- **tail** is the remainder (always a list*)
- list operators:

`|`	(cons) add or extract elements (recommended, efficient and fast)
`++`	append 2 lists (uses copying, inefficient), alias for `lists:append/2`
`--`	subtract elements (very slow), alias for `lists:subtract/2`

- `lists` module contains a set of useful functions for operations on lists

* if **tail** is not a list, e.g. `[a | b]`, this term is called 'improper list'

```
42> [john, {1,2}, [a, b, 12], 77.9].
[john,{1,2},[a,b,12],77.9]
43> [].
[]
44> [1 | [delta, 8 | [9]]].
[1,delta,8,9]
45> "jklo".
"jklo"
46> [$a, $T, $M].
"aTM"
47> [97, 98, 99].
"abc"
48> [one, two, 3] ++ [8, nine, ten].
[one,two,3,8,nine,ten]
49> "ala" ++ " ma " ++ "kota.".
"ala ma kota."
50> [1, 2, 4, 1, 1, 4] -- [1, 4, 4].
[2,1,1]
51> [1,2,3,4,5] -- [1,2] -- [2].
[2,3,4,5]
```

```
52> hd([34, 56, 89]).
34
53> tl([34, 56, 89]).
"8Y"
54> tl([34, 56, 890]).
[56,890]
55> [H | T] = [1, 2, 3].
[1,2,3]
56> H.
1
57> T.
[2,3]
58> [improper | list].
[improper|list]
59> lists:seq(1,10,2).
[1,3,5,7,9]
60> lists:reverse([1,3,5,6,ala,[1,2]]).
[[1,2],ala,6,5,3,1]
61> lists:sort([8,0,78,atom, 4, 3]).
[0,3,4,8,78,atom]
62> lists:flatten([5, [3, 4, [9, []], a, [b]]]).
[5,3,4,9,a,b]
```

MAPS

- compound type of data with variable number of Key/Value pairs

```
#{Key1 => Value1, Key2 => Value2, ..., KeyN => ValueN}
```

- stored as ordered collection (sorted by Key values)

```
1> #{c => 3, b => 4, a => 0}.
#{a => 0,b => 4,c => 3}
```

- keys and values can be any valid Erlang terms

```
2> #{1 => a, a => <<"hello">>, {e, 3} => [6, 7, 8]}.
#{1 => a,a => <<"hello">>,{e,3} => [6,7,8]}
```

- if duplicate keys are declared, the latter one takes precedence

```
3> #{1  => a, 1 => b}.
#{1 => b}
```

- bound variables are allowed as keys and values

- map operators:

  `=>` sets the new Key/Value pair or updates the Value of existing Key
  (the operation always succeeds)

  `:=` updates the Value of existing Key
  (the operation fails if the updated map does not contain the Key)

- `maps` module contains a set of useful functions for operations on maps

```
1> #{}.
#{}
2> M = #{ k1 => a, "k2" => b}.
#{k1 => a,"k2" => b}
3> M2 = M#{k3 => c}.
#{k1 => a,k3 => c,"k2" => b}
4> M3 = M2#{"k2" := c}.
#{k1 => a,k3 => c,"k2" => c}
5> M4 = M3#{k4 := c}.
** exception error: {badkey,k4}
     in function  maps:update/3
        called as maps:update(k4,c,#{k1 => a,k3 => c,"k2" => c})
     in call from erl_eval:'-expr/5-fun-0-'/2 (erl_eval.erl, line 255)
     in call from lists:foldl/3 (lists.erl, line 1263)
6> M4.
* 1: variable 'M4' is unbound
7> maps:keys(M3).
[k1,k3,"k2"]
8> maps:size(M3).
3
9> maps:from_list([{"a",ignored},{1337, "two"},{42,three}]).
#{42 => three,1337 => "two","a" => ignored}
10> V = "77".
"77"
11> #{V => k3}.
#{"77" => k3}
```

REFERENCES

- **Ref** is a term, unique in Erlang runtime system, created by BIF

```
make_ref()
```

- useful for tagging and creating unique names for
  - processes
  - messages sent to other processes (to identify possible reply)
  - stored data (e.g. related to user authorization)

```
13> MyRef = make_ref().
#Ref<0.0.4.173>
14> MyOtherRef = make_ref().
#Ref<0.0.4.178>
```

PIDS

Process ID (Pid) identifies a process. It can only be obtained:

- by calling process spawning BIFs:

```
spawn/1-4, spawn_link/1-4, spawn_opt/4
```

- by retrieving Pid of existing process:

```
self/0, whereis/1, group_leader/0,
erlang:port_info/2, processes/0
```

- by converting a string to Pid (debug only):

```
list_to_pid/1
pid/1 % only in erlang shell
```

More info and examples in Concurrent Programming chapter

PATTERN MATCHING

Pattern matching in Erlang is used to:

- Assign values to variables
- Control the execution flow of programs
- Extract values from compound data types

PATTERN MATCHING - BASIC FORM

```
Pattern = Expression|Term
```

where

```
Term
```

The simplest form of expression, i.e. number, atom, string, list, map or tuple; return value is the term itself

```
Pattern
```

The same structure as term but can contain bound and unbound variables

```
Expression
```

Consists of data structures, bound variables, mathematical operations and function calls. It may not contain unbound variables.

PATTERN MATCHING - EXECUTION

First, **Expression** is evaluated. If succeeds, the returned value is matched with **Pattern** - type, size, and elements values are evaluated one by one. If matching succeeds (and only then!), unbound variables in **Pattern** are bound to values from **Expression**. The value of **Expression** is returned.

'=' operator is not the only case when pattern matching occurs. It happens also:

- at function clause evaluation
- in *case*, *receive* and *try* expressions

PATTERN MATCHING - EXAMPLES

```
1> A.
* 1: variable 'A' is unbound
2> A = {x, 23}.
{x,23}
3> A.
{x,23}
4> {B, 23} = A.
{x,23}
5> B.
x
6> {C, 22} = A.
** exception error: no match of right hand side value {x,23}
7> C.
* 1: variable 'C' is unbound
8> f(), [A | Tail] = [8, halo, 0.4e2, {a, b, 99}].
[8,halo,40.0,{a,b,99}]
9> A.
8
10> Tail.
[halo,40.0,{a,b,99}]
```

PATTERN MATCHING - EXAMPLES

```
11> {_Int, Int} = {2, 3}.
{2,3}
12> _Int.
2
13> Int.
3
14> [X | Y] = [].
** exception error: no match of right hand side value []
15> X.
* 1: variable 'X' is unbound
16> f(), [1, A | B] = [1, 2, 3].
[1,2,3]
17> A.
2
18> B.
[3]
19> {_, _} = {{5,6},[a, b, 34, B]}.
{{5,6},[a,b,34,[3]]}
```

SEQUENTIAL PROGRAMMING

MODULES

Modules are containers for our code. They are written as files with the same name as the module's name and consist of functions and attributes. Function namespace is local to the module, i.e. the same function name can be reused in another module. Attributes are given as module directives.

ATTRIBUTES

Attribute syntax:

```
-Tag(Value).
```

Tag = atom(), Value = *literal* term().
If `Value` is form of `Name/Arity`, it is translated as `{Name, Arity}`

ATTRIBUTES

Pre-defined attributes:

```
-module(Module).
-export(Functions).
-import(Module,Functions).
-compile(Options).
-vsn(Vsn).
-on_load(Function).
```

Behaviour module attribute:

```
-behaviour(Behaviour).
-callback Name(Arguments) -> Result.
```

Record definitions:

```
-record(Record, Fields).
```

ATTRIBUTES

Preprocessor:

```
-include("SomeFile.hrl").
-define(Macro,Replacement).
```

Setting file and line:

```
-file(File, Line). %%to replace default values of ?FILE and ?LINE
```

Types and function specifications

```
-type my_type() :: atom() | integer().
-spec my_function(integer()) -> integer().
```

example.erl:

```
%% Directives (here only 'module' and 'export'):
-module(example).
-export([calc/3, hello/0]).

%% Exported functions:
calc(Arg1, Arg2, sum) ->
    sum(Arg1, Arg2);
calc(Arg1, Arg2, sub) ->
    substr(Arg1, Arg2).
hello() -> io:format("Hello world!~n").

%% Private functions:
sum(Arg1, Arg2) -> Arg1 + Arg2.
substr(Arg1, Arg2) -> sum(Arg1, -Arg2).
```

MODULES

Let's try it in the shell:

```
1> c(example).
{ok,example}
2> example:hello().
Hello world!
ok
3> example:calc(1,2, sum).
3
4> example:calc(1,2, sub).
-1
```

FUNCTIONS

Function declaration:
a sequence of **function clauses** separated by `';'` and terminated by `'.'`
Function clause - a clause **head** and a clause **body**, separated by `'->'`
Function **head** - a function **name** and an **argument list**, followed by an optional **guard sequence**:

```
Name(Pattern11,...,Pattern1N) [when GuardSeq1] ->
    Body1;
...;
Name(PatternK1,...,PatternKN) [when GuardSeqK] ->
    BodyK.
```

FUNCTIONS

Function **body** - a sequence of expressions separated by ' , ':

```
Expr1,
...,
ExprN
```

Example:

```
fact(N) when N>0 ->  % first clause head
    N * fact(N-1);   % first clause body

fact(0) ->           % second clause head
    1.               % second clause body
```

FUNCTION EVALUATION

Calling a function `m:f(Args)` requires that this function must be exported.
Otherwise (or if no definition is found) `undef` runtime error occurs.
Every clause is checked sequentially until the following conditions are fulfilled:

- all patterns in the head match given arguments
- the guard sequence (if any) returns *true*

If no matching clause is found, `function_clause` runtime error occurs.

FUNCTION EVALUATION

Matching clause's body is evaluated sequentially and **the value of the last expression** is returned.

```
1> m:fact(1).
```

This matches the following clause:

```
N * fact(N-1) => (N is bound to 1)
1 * fact(0)
```

Now the pattern `N` is matched against `0` and the second clause is executed:

```
1 * fact(0) =>
1 * 1 =>
1
```

FUNCTION CALLS

```
ExprF(Expr1,...,ExprN)
ExprM:ExprF(Expr1,...,ExprN)
```

1st call by **implicitly qualified function name**
2nd call by **fully qualified function name**, referred to as **remote** or **external** call.

ExprF is an atom or expression evaluated to an atom.

NON-FULLY QUALIFIED FUNCTION CALLS

```
ExprF(Expr1,...,ExprN)
```

- ExprF is locally defined function or
- ExprF is imported from M(odule)
  (then M:ExprF(Expr1,...,ExprN) is called) or
- ExprF is auto-imported BIF

If the locally defined function name is the same as the name of auto-imported BIF, local function call has precedence.

BUILT-IN FUNCTIONS

- functions implemented in C in the runtime system
- do things difficult or impossible to implement in Erlang
- most BIFs are auto-imported from `erlang` module (no need to prefix them while called)
- some functions from other modules than `erlang` became BIFs for performance reasons (e.g. from `lists` and `ets` modules)

BUILT-IN FUNCTIONS

BIFs that deal with built-in types:

- `hd/1`
- `tl/1`
- `length/1`
- `tuple_size/1`
- `element/2`
- `setelement/3`

BUILT-IN FUNCTIONS

```
1> List = [8, x, {key, 90}, four].
[8,x,{key,90},four]
2> hd(List).
8
3> tl(List).
[x,{key,90},four]
4> length(List).
4
5> hd(tl(List)).
x
6> Tuple = {1, e, 8, 0.4}.
{1,e,8,0.4}
7> tuple_size(Tuple).
4
8> element(4, Tuple).
0.4
9> setelement(3, Tuple, "in").
{1,e,"in",0.4}
10> erlang:append_element(Tuple, elem@five).
{1,e,8,0.4,elem@five}
```

BUILT-IN FUNCTIONS

Type conversion BIFs:

- `atom_to_list/1, list_to_atom/1`
- `list_to_tuple/1, tuple_to_list/1`
- `float/1, list_to_float/1`
- `round/1, trunc/1, list_to_integer/1`

BUILT-IN FUNCTIONS

```
11> atom_to_list(person).
"person"
12> list_to_atom("thursday").
thursday
13> list_to_atom([1,2,3]).
'\001\002\003'
14> list_to_tuple([1,2,3]).
{1,2,3}
15> list_to_float("2.2017764e+0").
2.2017764
16> float(4).
4.0
17> round(6.993).
7
18> trunc(6.993).
6
```

BUILT-IN FUNCTIONS

Some other BIFs:

- `apply/3` allows function calls where MFA are not known at compile time
- `put/2, get/[0-1], get_keys/[0-1]` - related to process dictionary
- `erlang:system_time/[0-1]` returns current time
- `spawn/[1,3], link/1, monitor/1` - process handling

See the documentation on erlang.org or erlang man pages for the complete list and usage guide.

GUARDS AND GUARDS SEQUENCES

A **guard sequence** is a sequence of guards, separated by **';'**. The guard sequence is true if at least one of the guards is true. (The remaining guards, if any, are not evaluated).

```
Guard1;...;GuardK
```

A **guard** is a sequence of guard expressions, separated by **','**. The guard is true if all guard expressions evaluate to true.

```
GuardExpr1,...,GuardExprN
```

GUARDS AND GUARDS SEQUENCES

Guards are used as additional restrictions for pattern matching and can be used in function clause head, 'case' or 'receive' clause. They are preceeded by 'when' keyword and must be an expression free of side effects. That's why only restricted set of expressions are allowed. User defined functions are forbidden.

GUARDS AND GUARDS SEQUENCES

Some examples of valid guard expressions:

- The atom true
- Other constants (terms and bound variables), all regarded as false
- Calls to the BIFs specified in table Type Test BIFs
- Term comparisons
- Arithmetic expressions
- Boolean expressions
- Short-circuit expressions (`andalso/orelse`)

Type test BIFs	Other valid BIFs
is_atom/1	abs(Number)
is_binary/1	bitsize(Bitstring)
is_bitstring/1	element(N, Tuple)
is_boolean/1	float(Term)
is_float/1	hd(List)
is_function/1	length(List)
is_list/1	map_size(Map)
is_integer/1	node()
etc...	

Single condition in guard:

```
factorial(N) when N > 0 ->
    N * factorial(N - 1);
    factorial(0) -> 1.
```

More complex guard:

```
guard(X,Y) when not(((X>Y) or not(is_atom(X)) )
            and (is_atom(Y) or (X==3.4))) ->
    X+Y.
```

Guard sequence examples:

```
guard2(X,Y) when not(X>Y) , is_atom(X) ; not(is_atom(Y)) , X=/=3.4 ->
    X+Y.
```

```
guard3(X,Y) when not(X>Y) andalso is_atom(X) orelse not(is_atom(Y))
            andalso X=/=3.4 ->
    X+Y.
```

THE CASE CONSTRUCT

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

Alternative syntax for pattern matching arguments in function head.

```
is_valid_signal(Signal) ->
    case Signal of
        {signal, _What, _From, _To} ->
            true;
        {signal, _What, _To} ->
            true;
        _Else ->
            false
    end.
```

THE CASE CONSTRUCT

Defensive programming:

```
convert(Day) ->
    case Day of
      monday    -> 1;
      tuesday   -> 2;
      wednesday -> 3;
      thursday  -> 4;
      friday    -> 5;
      saturday  -> 6;
      sunday    -> 7;
      _Other    -> {error, unknown_day}
end.
```

Do not take this approach in your Erlang programs!

THE `IF` CONSTRUCT

```
if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN
end
```

`if`, like `case` can replace different function clauses, but here instead of arguments, guards are evaluated.

```
if
    X < 1 -> ok;
    X > 1 -> nok;
    true -> dunno
end
```

`true` here means *catch-all* branch (*else* in other languages), this is sometimes required to avoid compiler complain about 'no true branch' - remember that every expression in Erlang **must** return something.

PREPROCESSOR

Preprocessor in Erlang allows conditional compiling, including header files, adding macros and records.

Header files can contain only macros and records and are included to the module as follows:

```
-include(File).
-include_lib(File).
```

For example:

```
-include("my_records.hrl").
-include("incdir/my_records.hrl").
-include("/home/user/proj/my_records.hrl").
-include("$PROJ_ROOT/my_records.hrl").
-include_lib("kernel/include/file.hrl").
```

MACROS

Like in other languages, macros are a way of defining constants and are simple text replacement during compilation:

```
-define(Const, Replacement).
```

For example:

```
-define(TIMEOUT, 200).
```

Usage:

```
call(Request) ->
    server:call(refserver, Request, ?TIMEOUT).
```

which is expanded to:

```
call(Request) ->
server:call(refserver, Request, 200).
```

MACROS

Macros can be parametrized, behaving like functions:

```
-define(Func(Var1,...,VarN), Replacement).
```

For example:

```
-define(MACRO1(X, Y), {a, X, b, Y}).
```

Usage:

```
bar(X) ->
    ?MACRO1(a, b),
    ?MACRO1(X, 123)
```

which is expanded to:

```
bar(X) ->
    {a,a,b,b},
    {a,X,b,123}.
```

PREDEFINED MACROS

`?MODULE` - The name of the current module.

`?MODULE_STRING` - The name of the current module, as a string.

`?FILE` - The file name of the current module.

`?LINE` - The current line number.

`?MACHINE` - The machine name, 'BEAM'.

`?FUNCTION_NAME` - The name of the current function.

`?FUNCTION_ARITY` - The arity (number of arguments) for the current function.

RECORDS

Data structures with fixed number of fields, accessed by name. They are similar to structures in C but in Erlang are just syntactic sugar on top of tuples. Preprocessor uses a record definition and translates it to a tuple using record name as a first element and the values from all record content.

DEFINING RECORDS

In a module or header file:

```
-record(Name, {Field1 [= Default1],
               ...
               FieldN [= DefaultN]}).
```

In the shell:

```
>rd(Name, {Field1 [= Default1],...}).
```

A record definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the record. If a record is used in several modules, it is recommended that the record definition is placed in an include file.

CREATING RECORDS

The following syntax applies for creating a record instance:

```
#Name{Field1=Expr1,...,FieldK=ExprK}
```

Ommited fields take either default values from record definition, or atom 'undefined'. To overwrite this behaviour, '_' variable can be used:

```
#Name{Field1=Expr1,...,FieldK=ExprK, '_' = ExprL}
```

Example:

```
-record(person, {name, phone, address}).
...
lookup(Name, Tab) ->
    ets:match_object(Tab, #person{name=Name, _='_'}).
```

ACCESSING RECORD FIELDS

```
Expr#Name.Field
```

Example:

```
-record(person, {name, phone, address}).
...
lookup(Name, List) ->
    lists:keysearch(Name, #person.name, List).
```

UPDATING RECORDS

```
Expr#Name{Field1=Expr1,...,FieldK=ExprK}
```

Expr is to evaluate to a Name record. A copy of this record is returned, with the value of each specified field FieldI changed to the value of evaluating the corresponding expression ExprI. All other fields retain their old values. Example:

```
-record(person, {name,age=0,phone=""}).
...
    Person = #person{name="Fred"}, % age=0, phone=""
    NewPerson = Person#person{phone="999-999",age=37},
...
```

PATTERN MATCHING OVER RECORDS

```
joesBirthday(#person{age=Age,name="Joe"} = P) ->
    P#person{age=Age+1}.
```

`'Age'` and `'P'` are unbound variables in the pattern, and get values when a record with proper name (i.e. `#person{}`) and `'name'` field value (i.e. `"Joe"`) is given in the function call.

INTERNAL REPRESENTATION OF RECORDS

Record expressions are translated to tuple expressions during compilation. A record defined as:

```
-record(Name, {Field1,...,FieldN}).
```

is internally represented by the tuple:

```
{Name,Value1,...,ValueN}
```

Additionally

```
#Name.FieldN
```

returns the **index** in the tuple representation of 'FieldN' of the record 'Name'.

LIST COMPREHENSIONS

Expression used for creating and manipulating lists with concise and powerful manner. General syntax:

```
[Expr || Qualifier1,...,QualifierN]
```

where 'Qualifier' can be either a 'Generator' or a 'Filter'

Generator syntax:

```
Pattern <- ListExpr
```

A **Filter** is an expression, which evaluates to true or false (in fact, any other value than 'true' is interpreted as 'false')

Examples:

```
1> [X*2 || X <- [1,2,3]].
[2,4,6]
2> [X || X <- [1,2,a,3,4,b,5,6], is_integer(X), X > 3].
[4,5,6]
3> [{X, Y} || X <- [1,2,3], Y <- [a,b]].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
4> [ X*2 || X <- [1,5,3,0], X>2].
[10,6]
5> [ {X, Y} || X <- lists:seq(1,3),
               Y <- string:sub_string("Hello World!!!", 1, 2)].
[{1,72},{1,101},{2,72},{2,101},{3,72},{3,101}]
6> [ "Hello world" || is_integer(2)].
["Hello world"]
7> [ x || is_integer(x)].
[]
8> X = 5.
5
9> [X || is_number(X)].
[5]
```

In list comprehensions variables are inherited from external scope, but not vice versa. If the variable name is the same inside, it will shadow the inherited variable.

Examples:

```
1> B = 20.
20
2> L = [ X || X <- lists:seq(1, B)].
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
3> X.
* 2: variable 'X' is unbound
4> X = 3.
3
5> L = [ X || X <- lists:seq(1, B)].
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
6> X.
3
```

RECURSION

Erlang uses recursive function calls to tackle problems which require repetitive or iterative approach.
Calling the function itself using new values of arguments gives flexibility and allows splitting the big problem to solve into smaller parts which can be handled sequentially.

```
print_interval(X, Y) when X > Y ->
    io:format("done.~n");
print_interval(X, Y) ->
    io:format("~p,",[X]),
    print_interval(X + 1, Y).
```

This gives:

```
1> recursion:print_interval(3,8).
3,4,5,6,7,8,done.
```

RECURSION

Another example with list processing:
'bump' function takes a list of integers and increments every element with 1.

```
bump([]) -> [];
bump([Head | Tail]) -> [Head + 1 | bump(Tail)].
```

Let's call the function on the list [1,2,3]:

```
bump([1, 2, 3] => [1 + 1 | bump([2, 3])
    1 + 1 => 2
    bump([2, 3]) => [2 + 1 | bump([3])
        2 + 1 => 3
        bump([3]) => [3 + 1 | bump([])
            3 + 1 => 4
            bump([]) => []
            [4 | []] => [4]
        [4] <=
        [3 | [4]] => [3, 4]
    [3, 4] <=
    [2 | [3, 4]] => [2, 3, 4]
[2, 3, 4] <=
```

The function returning the length of a list:

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

The function returning the sum of all numbers from a list:

```
sum([]) -> 0;
sum([Head | Tail]) -> Head + sum(Tail).
```

These can be combined to form a function which calculates average of all elements in the list:

```
average(List) -> sum(List) / len(List).
```

TAIL AND NON-TAIL RECURSION

Recursion defined as in previous examples:

```
sum([]) -> 0;
sum([Head | Tail]) -> Head + sum(Tail).
```

is called 'direct' or 'non-tail'recursive function call.

In such cases, the function cannot return its value until all 'iterations' done internally are complete.
All results are put on a call stack and wait for the last iteration return.

TAIL AND NON-TAIL RECURSION

If the last epression of a function body is a (recursive) function call, then it is called 'tail-recursive' call.
No call stack is consumed - all variables can be cleared before the next iteration.
Servers (working in infinite loop) cannot then use non-tail recursive calls.

```
loop(N) ->
    io:format("~w~n", [N]),
    loop(N+1).
```

TAIL-RECURSION AND LIST PROCESSING

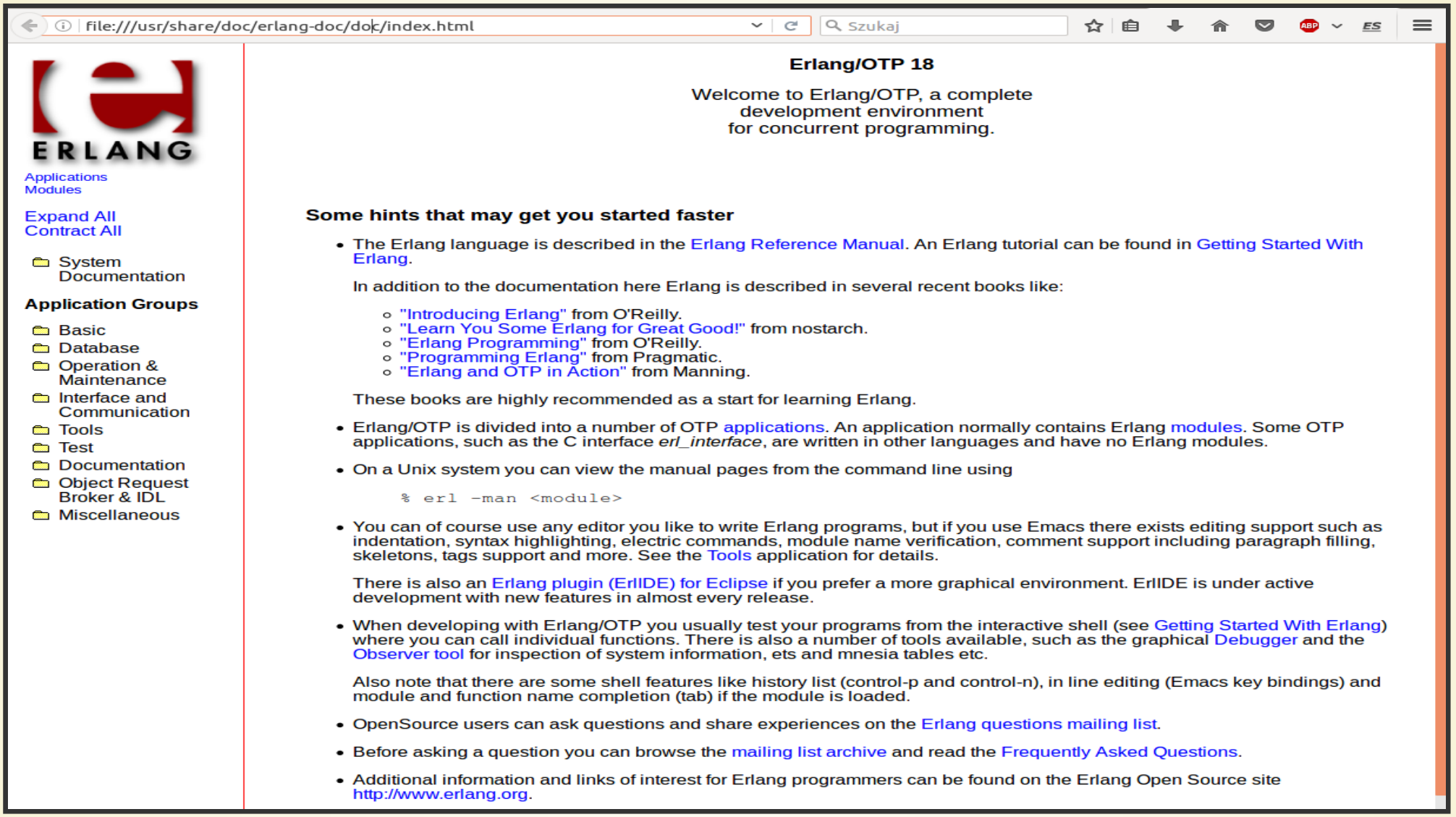
Common practice in list processing functions is using tail-recursive loops with additional parameter holding the partly calculated values - accumulator.

```
sum(List) -> sum_acc(List,0). %%interface function

sum_acc([],Sum) -> Sum; %%worker function, private
sum_acc([Head|Tail], Sum) -> sum_acc(Tail, Head+Sum).
```

```
bump(List) -> bump_acc(List, []).

bump_acc([], Acc)
-> reverse(Acc);
bump_acc([Head | Tail], Acc) -> bump_acc(Tail, [Head + 1 | Acc]).
```

LIBRARY MODULES

Documentation for library modules and user guides are available on www.erlang.org or locally as html or man pages (only unix-like systems: *erl -man Module*)


file:///usr/share/doc/erlang-doc/doc/index.html
Szukaj
ERLANG
Applications
Modules
Expand All
Contract All
System Documentation
Application Groups
Basic
Database
Operation & Maintenance
Interface and Communication
Tools
Test
Documentation
Object Request Broker & IDL
Miscellaneous
Erlang/OTP 18
Welcome to Erlang/OTP, a complete development environment for concurrent programming.
Some hints that may get you started faster
The Erlang language is described in the Erlang Reference Manual. An Erlang tutorial can be found in Getting Started With Erlang.
In addition to the documentation here Erlang is described in several recent books like:
"Introducing Erlang" from O'Reilly.
"Learn You Some Erlang for Great Good!" from nostarch.
"Erlang Programming" from O'Reilly.
"Programming Erlang" from Pragmatic.
"Erlang and OTP in Action" from Manning.
These books are highly recommended as a start for learning Erlang.
Erlang/OTP is divided into a number of OTP applications. An application normally contains Erlang modules. Some OTP applications, such as the C interface erl_interface, are written in other languages and have no Erlang modules.
On a Unix system you can view the manual pages from the command line using
% erl -man <module>
You can of course use any editor you like to write Erlang programs, but if you use Emacs there exists editing support such as indentation, syntax highlighting, electric commands, module name verification, comment support including paragraph filling, skeletons, tags support and more. See the Tools application for details.
There is also an Erlang plugin (ErlIDE) for Eclipse if you prefer a more graphical environment. ErlIDE is under active development with new features in almost every release.
When developing with Erlang/OTP you usually test your programs from the interactive shell (see Getting Started With Erlang) where you can call individual functions. There is also a number of tools available, such as the graphical Debugger and the Observer tool for inspection of system information, ets and mnesia tables etc.
Also note that there are some shell features like history list (control-p and control-n), in line editing (Emacs key bindings) and module and function name completion (tab) if the module is loaded.
OpenSource users can ask questions and share experiences on the Erlang questions mailing list.
Before asking a question you can browse the mailing list archive and read the Frequently Asked Questions.
Additional information and links of interest for Erlang programmers can be found on the Erlang Open Source site http://www.erlang.org.

LIBRARY MODULES

```
lists(3erl)                                  Erlang Module Definition                                  lists(3erl)

NAME
       lists - List Processing Functions

DESCRIPTION
       This module contains functions for list processing.

       Unless otherwise stated, all functions assume that position numbering starts at 1. That is, the first element of a list is at position 1.

       Two terms T1 and T2 compare equal if T1 == T2 evaluates to true. They match if T1 =:= T2 evaluates to true.

       Whenever an ordering function F is expected as argument, it is assumed that the following properties hold of F for all x, y and z:

         * if x F y and y F x then x = y (F is antisymmetric);

         * if x F y and y F z then x F z (F is transitive);

         * x F y or y F x (F is total).

       An example of a typical ordering function is less than or equal to, =</2.

EXPORTS
       all(Pred, List) -> boolean()

              Types:

                 Pred = fun((Elem :: T) -> boolean())
                 List = [T]
                 T = term()

              Returns true if Pred(Elem) returns true for all elements Elem in List, otherwise false.

       any(Pred, List) -> boolean()

              Types:

                 Pred = fun((Elem :: T) -> boolean())
                 List = [T]
                 T = term()

              Returns true if Pred(Elem) returns true for at least one element Elem in List.
```

LIBRARY MODULES

The most important and useful modules:

`calendar`
Local and universal time, day-of-the-week, date and time conversions.

`dict`
Key-Value Dictionary

`erlang`
The Erlang BIFs.

`file`
File Interface Module

`filename`
Filename Manipulation Functions

`io`
standard I/O server manipulation functions

LIBRARY MODULES

lists

List Processing Functions

math

Mathematical Functions

queue

Abstract Data Type for FIFO Queues

rand

Pseudo random number generation

string

String Processing Functions

timer

Timer Functions

ERROR HANDLING

Errors can be divided into four different types:

- Compile-time errors
- Logical errors
- Run-time errors
- Generated errors (exceptions)

Compile-time errors are reported by the compiler.
Logical errors - the program does not work as expected but does not crash.
Run-time errors are exceptions thrown by the run-time system. Can be emulated by calling `erlang:error/[1-2]`
Generated errors are exceptions of class `exit` and `throw` invoked by the code itself.

Run-time error, examples:

```
1> 1 + b.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as 1 + b
2> length(helloWorld).
** exception error: bad argument
     in function  length/1
        called as length(helloWorld)
3> test:hello().
** exception error: undefined function test:hello/0
4> {1,2} = [1,2].
** exception error: no match of right hand side value [1,2]
5> lists:sort(zxyp).
** exception error: no function clause matching lists:sort(zxyp)
(lists.erl, line 479)
6> if 1 > 2 -> ok;
6> is_list({1,2,3}) -> also_ok
6> end.
** exception error: no true branch found when evaluating an if expression
```

EXCEPTIONS

Class	Reason
`error`	Run-time error, for example, 1+a, or the process called erlang:error/1,2
`exit`	The process called `exit/1`
`throw`	The process called `throw/1`

Exception contains `class:Reason` where `class` is one the above and `Reason` is a term describing error type.

CATCH

Error stack is a stack of function calls when the error occured, given as list of {Module, Name, Arity} or {Module, Name, [Arg]} (the most recent call on top), with src file and line.
`catch` expression prevents exceptions from terminating the process. If exceptions of class `error` occurs, error stack is returned.

```
30> catch 1+2.
3
31> catch 1+c.
{'EXIT',{badarith,[{erlang,'+',[1,c],[]},
                   {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,674}]},
                   {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,431}]},
...
```

TRY...CATCH

Erlang has another construct used for trapping exceptions:

```
try Exprs of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
after
    AfterBody
end
```

TRY...CATCH

```
1> X = 5.
5
2> X.
5
3> try (X=4) of
3> Value -> {ok, Value}
3> catch
3> error:Reason -> {error, Reason}
3> end.
{error,{badmatch,4}}
```

From inside `try...catch` the error stack can be retrieved by `erlang:get_stackstrace/0`:

```
5> try (X = 9) of
5> Value -> {ok, Value}
5> catch
5> _:Reason -> {Reason, erlang:get_stacktrace()}
5> end.
{{badmatch,9},[{erl_eval,expr,3,[]}]}
```

FUNS AND HIGHER ORDER FUNCTIONS

Fun is a functional object, used to create anonymous functions and pass the function as argument to some other function.

```
fun
    [Name](Pattern11,...,Pattern1N) [when GuardSeq1] ->
              Body1;
    ...;
    [Name](PatternK1,...,PatternKN) [when GuardSeqK] ->
              BodyK
end
```

Examples:

```
6> Fun1 = fun(X) -> X + 1 end.
#Fun<erl_eval.6.50752066>
7> Fun1(3).
4
8> fun(X,Y) -> X+Y end(5,6).
11
```

FUNS AND HIGHER ORDER FUNCTIONS

Alternative fun syntax with reference to existing function:

```
F = fun FunctionName/Arity

F = fun Module:FunctionName/Arity
```

```
16> fun codecs:int_2_bcd/1.
#Fun‹codecs.int_2_bcd.1›
17> fun codecs:int_2_bcd/1(45).
<<"E">>
```

FUNS AND HIGHER ORDER FUNCTIONS

Funs allow creating abstractions (higher order functions):

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

```
foreach(fun(H) -> io:format(S, "~p~n",[H]) end, L)
```

```
9> lists:foreach(fun (X) -> io:format("~p ", [X+1]) end, [1,2,3,5]).
2 3 4 6 ok
```

FUNS AND HIGHER ORDER FUNCTIONS

Funs can also be returned from functions:

```
20> Adder = fun(X) -> fun (Y) -> X + Y end end.
#Fun‹erl_eval.6.50752066›
21> Add6 = Adder(6).
#Fun‹erl_eval.6.50752066›
22> Add6(9).
15
23> (Adder(6))(9).
15
```

FUNS AND HIGHER ORDER FUNCTIONS

Own abstractions can be produced using funs, e.g. for loop:

funs.erl:

```
for(Max, Max, F) ->
    [F(Max)];
for(I, Max, F) ->
    [F(I) | for(I + 1, Max, F)].
```

```
26> funs:for(1,3,fun(I) -> I end).
[1,2,3]
27> funs:for(1,8,fun(I) -> I*I end).
[1,4,9,16,25,36,49,64]
```

BIT SYNTAX

A Bin is a low-level sequence of bits or bytes. The purpose of a Bin is to enable construction of binaries:

```
Bin = << E1, E2, ... En >>
```

Creating a binary - examples:

```
1> Bin = << 14, 89, 808 >> .
<<14,89,40>>
2> {ok, Bin2} = file:read_file("net_packet.pcap").
{ok,<<212,195,178,161,2,0,4,0,0,0,0,0,0,0,0,0,0,0,4,0,1,
      0,0,0,44,82,76,...>>}
3> term_to_binary({halo, 35, test, [1,2,3]}).
<<131,104,4,100,0,4,104,97,108,111,97,35,100,0,4,116,101,
  115,116,107,0,3,1,2,3>>
```

BIT SYNTAX

Pattern matching of binaries:

```
<< E1, E2, ...,En >> = Bin
```

```
4> A = 1, B = 17, C = 42.
42
5> Bin2 = << A, B, C:16 >>.
<<1,17,0,42>>
6> << D:16, E, F/binary >> = Bin2.
<<1,17,0,42>>
7> D.
273
8> E.
0
9> F.
<<"*">>
```

BIT SYNTAX

```
9> Frame = <<1,3,0,0,1,0,0,0>>.
<<1,3,0,0,1,0,0,0>>
10> << Type, Size, Bin:Size/binary-unit:8, _/binary >> = Frame.
<<1,3,0,0,1,0,0,0>>
11> Type.
1
12> Size.
3
13> Bin.
<<0,0,1>>
```

BIT SYNTAX

Each segment has the following syntax:

```
Value:Size/TypeSpecifierList
```

`Size` and `TypeSpecifierList` are optional:

```
Value
Value:Size
Value/TypeSpecifierList
```

BIT SYNTAX

`Size` is the number of units per segment.

`TypeSpecifierList` is a list of type specifiers separated by '-':

- **Type** - integer, float, or binary (default: integer)
- **Signedness** - signed/unsigned (default: unsigned)
- **Endianess** - big, little, or native (default: big)
- **Unit** - given as `unit:Integer`, number of bits per unit (1-256) (default: for int, float, bitstring: 1, for binary: 8)

```
X:4/little-signed-integer-unit:8
```

BIT SYNTAX

It is possible to traverse the Bin like lists, using recursion:

```
reverse_bits(Bits) ->
    reverse_bits(Bits,<<>>).
reverse_bits(<< X:1,Rest/bits >>,Acc) ->
    reverse_bits(Rest,<< X:1, Acc/bits >>);
reverse_bits(<<>>,Acc) ->
    Acc.
```

or bitstring comprehensions:

```
1> << << (X*2) >> || << X >> <= << 1,2,3 >> >>.
<<2,4,6>>
```

Text oriented applications (like web apps) use more convenient strings than classic charlist, i.e. binary strings:

```
<<"this is a binary string!\n">>
```

CONCURRENT PROGRAMMING

SEQUENTIAL INTO CONCURRENT

PROCESSES

Erlang is designed for massive concurrency. Concurrent activities are called **processes**.

Philosophy behind Erlang concurrency model by Joe Armstrong:

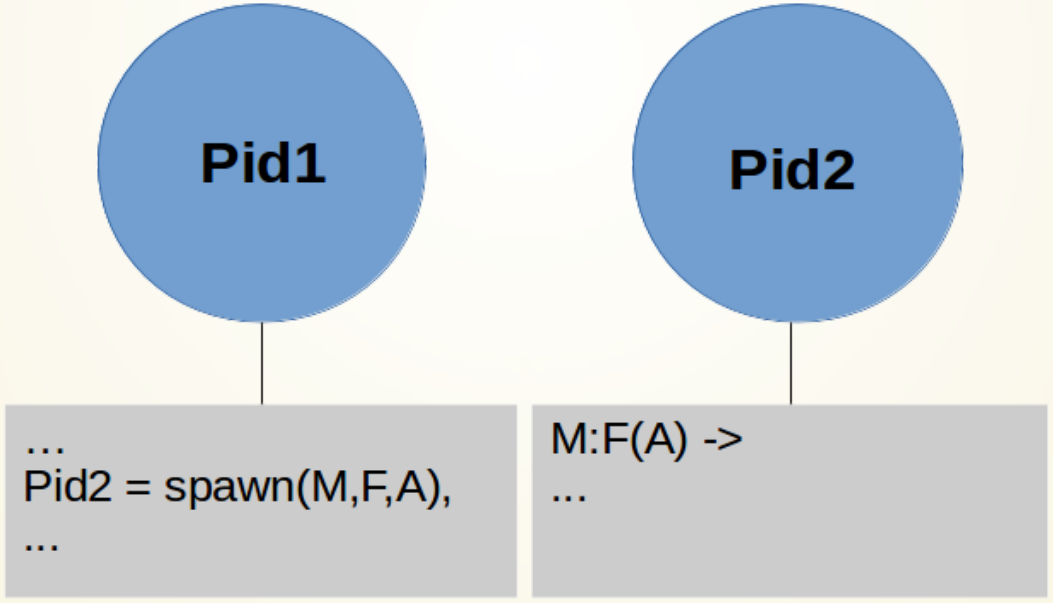
- the world is concurrent
- things in the world don't share data
- things communicated with messages
- things fail

PROCESSES

A process is created by calling `spawn/3`:

```
spawn(Module, Name, [Arg1, ..., ArgN]) -> pid()
```

The new process starts executing in `Module:Name(Arg1, ..., ArgN)`, `spawn/3` returns Process Id



PROCESSES

Processes:

- are independent from OS threads
- are equal
- do not share memory
- communicate with each other by message passing

PROCESS TERMINATION

Process always terminates with an `exit reason` which can be any term.

- normal termination (`exit reason == normal`) - no more code to execute.
- abnormal termination (`exit reason == {Reason, Stack}`) - a run-time error occurs.
- termination triggered by other process by sending `exit` signal (with `Reason =/= normal`).

```
1> spawn(lists,seq,[1,4]).
<0.131.0>
2> spawn(no_module, no_function, [foo]).
<0.133.0>
3>
=ERROR REPORT==== 11-Dec-2016::13:15:56 ===
Error in process <0.133.0> with exit value:
{undef,[{no_module,no_function,[foo],[]}]}
4> Pid = spawn(fun () -> io:format("~p: hello world~n", [self()]) end).
<0.144.0>: hello world
<0.144.0>
5> is_process_alive(Pid).
false
```

MESSAGE PASSING

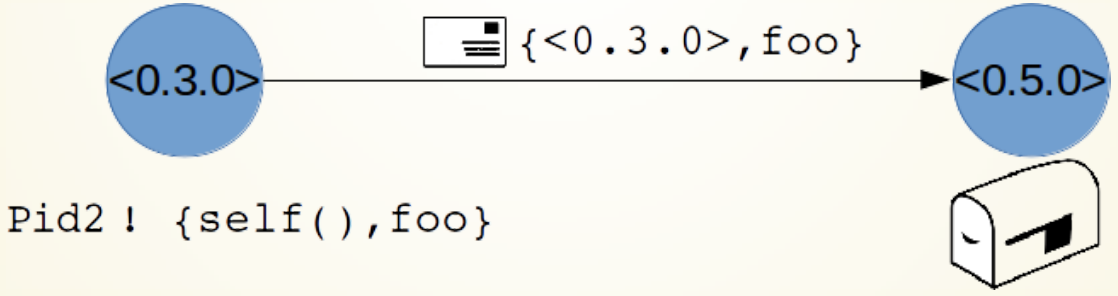
Message is sent using send operator:

```
Pid ! Msg
```

an alias to

```
erlang:send/2
```

This is asynchronous and safe operation - the message is delivered to destination process as long as it is alive.



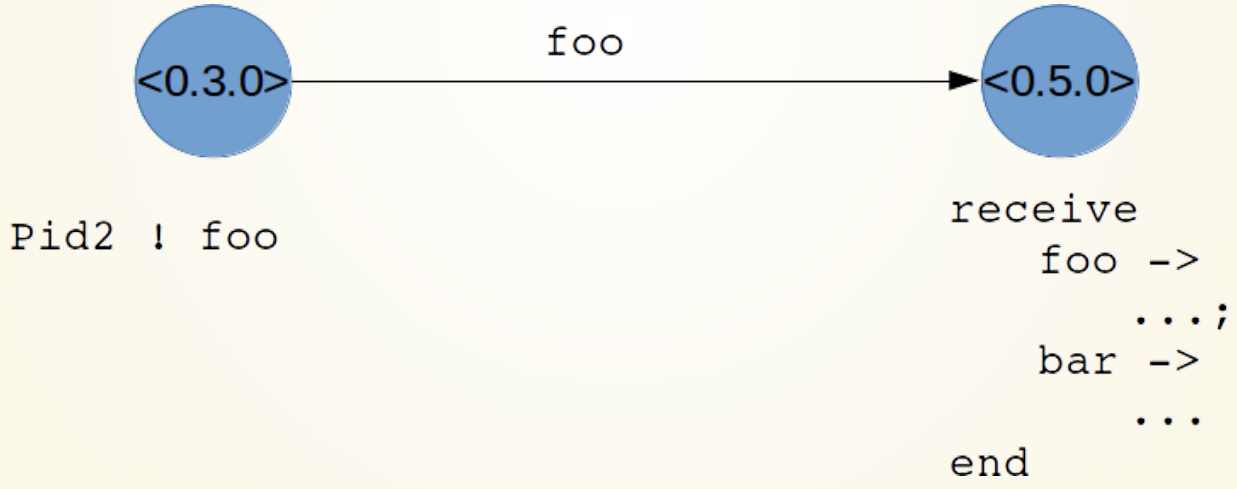
MESSAGE PASSING

```
1> self().
<0.136.0>
2> <0.136.0> ! hello.
* 1: syntax error before: '<'
3> pid(0,136,0) ! hello.
hello
4> flush().
Shell got hello
ok
5> Pid = self().
<0.136.0>
6> Pid ! hello_again.
hello_again
7> Pid ! hello_again.
hello_again
8> Pid ! hello_again.
hello_again
9> flush().
Shell got hello_again
Shell got hello_again
Shell got hello_again
ok
```

MESSAGE PASSING

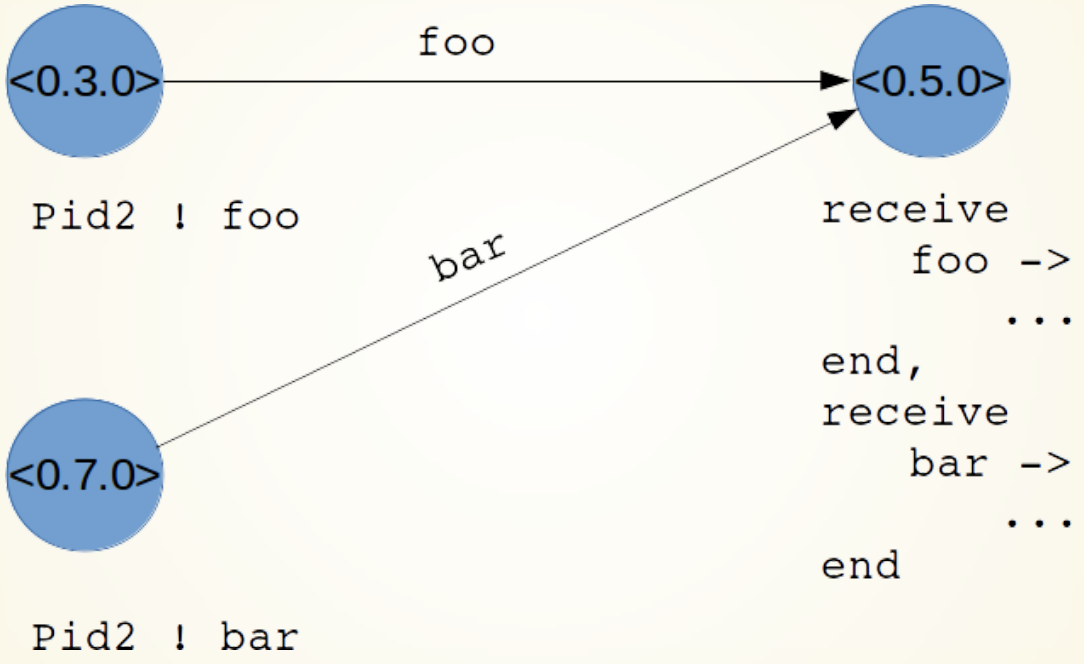
Messages are handled using `receive` expression:

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```



SELECTIVE RECEIVE

To avoid hazard if many messages arrive to the process, they can be handled selectively:



```
-module(echo).
-export([start/0, loop/0]).

start() ->
    Pid = spawn(?MODULE, loop, []),
    Pid ! {self(), echo},
    receive
        {Pid, Msg} ->
            io:format("Got ~p from ~p~n", [Msg, Pid])
    end,
    Pid ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            ok
    end.
```

REGISTERED PROCESSES

It is possible to register process Pid under a unique name and refer to it by this name (atom) while sending messages.
`register/2` BIF associates the name with the Pid. `registered/0` returns a list of registered names and `whereis/1` returns the Pid which has been registered under the given name.

Echo example again:

```
-module(echo).
-export([start/0, loop/0]).

start() ->
    register(echo, spawn(?MODULE, loop, [])),
    echo ! {self(), echo},
    receive
        {Pid, Msg} ->
            io:format("Got ~p from ~p~n", [Msg, Pid])
    end,
    echo ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            ok
    end.
```

TIMEOUTS

If no message match any pattern in `receive` expression, the process gets suspended.
It is possible to wait for a message a certain time, and if no message appears, give up and execute apropriate part of code.

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
after
    ExprT ->
        BodyT
end
```

TIMEOUTS

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    after
        60000 ->
            disconnect(),
            error()
end.
```

2 special values of Timeout exist:

- `0` - timeout occurs immediately if process mailbox is empty
- `infinity` - process waits indefinitely for a matching message (like no timeout implemented)

It is possible to use `receive...after` with no branches:

```
receive
after
    ExprT ->
        BodyT
end
```

This can be used to create simple timers - echo.erl

```
timer() ->
    spawn(?MODULE, timer, [self()]).

timer(Pid) ->
    receive
    after
        5000 ->
            Pid ! timeout
    end.
```

```
98> echo:timer().
<0.180.0>
99> flush().
Shell got timeout
ok
```

LINKS AND MONITORS

Linking - built-in mechanism for error detection and handling among processes
`link/1` creates bi-directional connection between processes. When any of them terminates abnormally, all the linked neighbours get exit signals.
`spawn_link/3` combines spawning and linking into one atomic operation.
`unlink/1` removes the link.

LINKS AND MONITORS

`monitor/2` is an alternative to `link/1` - it creates unidirectional connection, meaning that if the monitored process dies, monitor 'owner' receives a 'DOWN' message instead of exit signal:

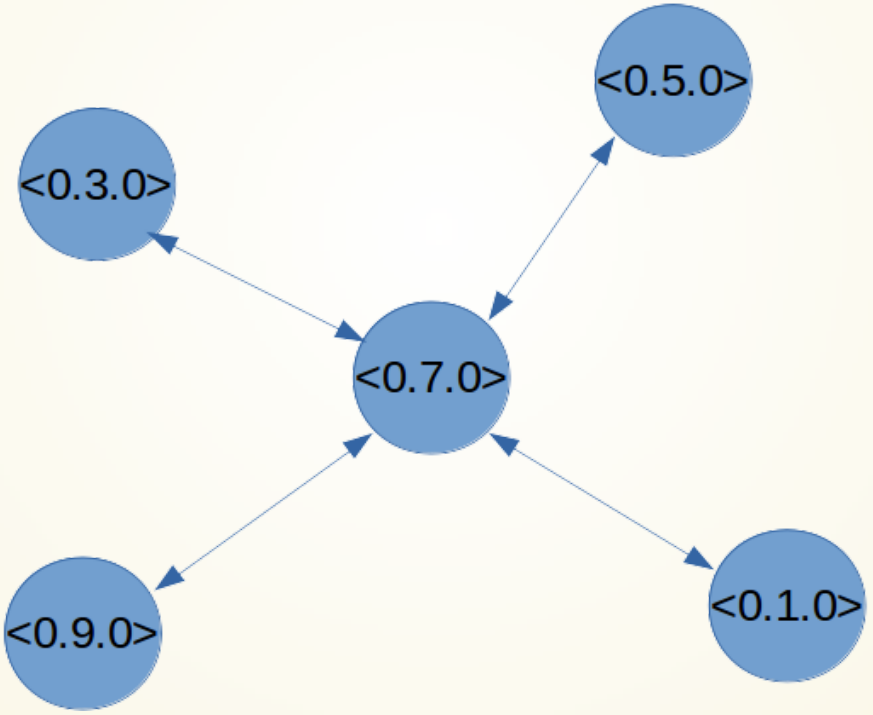
```
{'DOWN', Ref, process, Pid2, Reason}
```

and can react accordingly.

`Ref` is the reference returned from `monitor/2` and can be used to match with the incoming message.

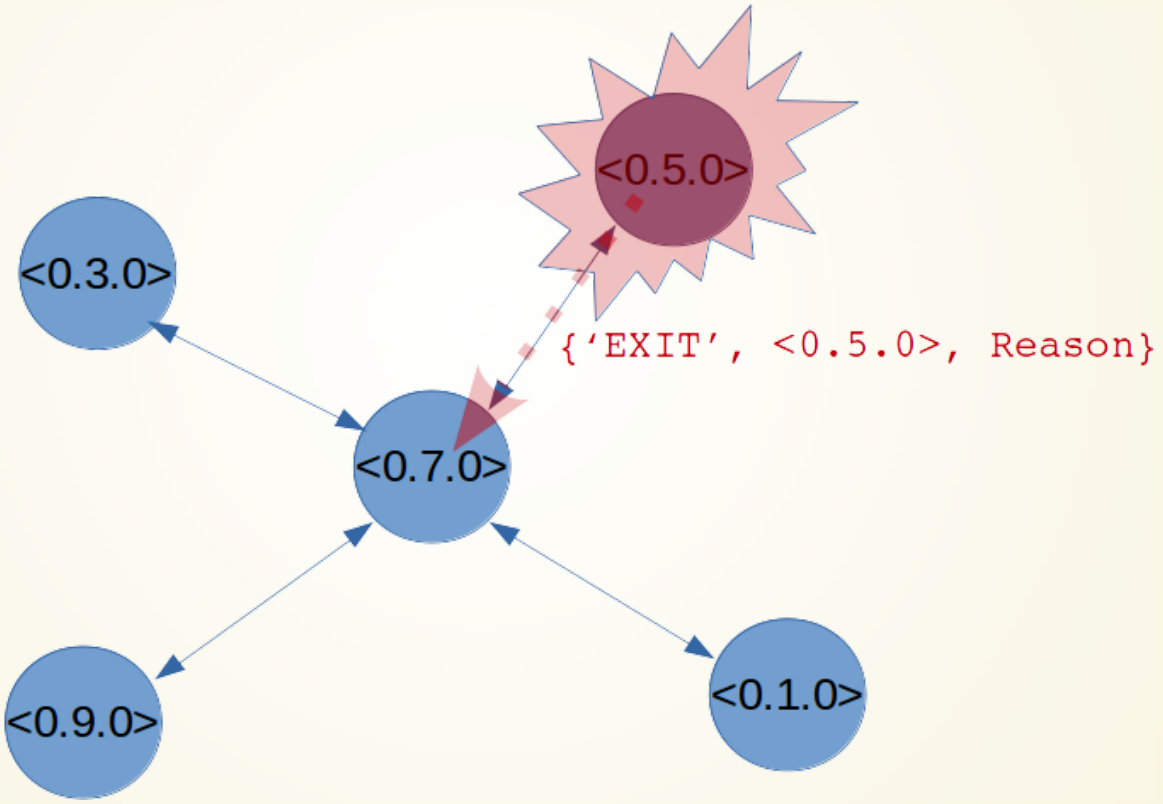
EXIT SIGNALS

Process terminates with **exit reason** which is sent to all linked processes in an **exit signal**.



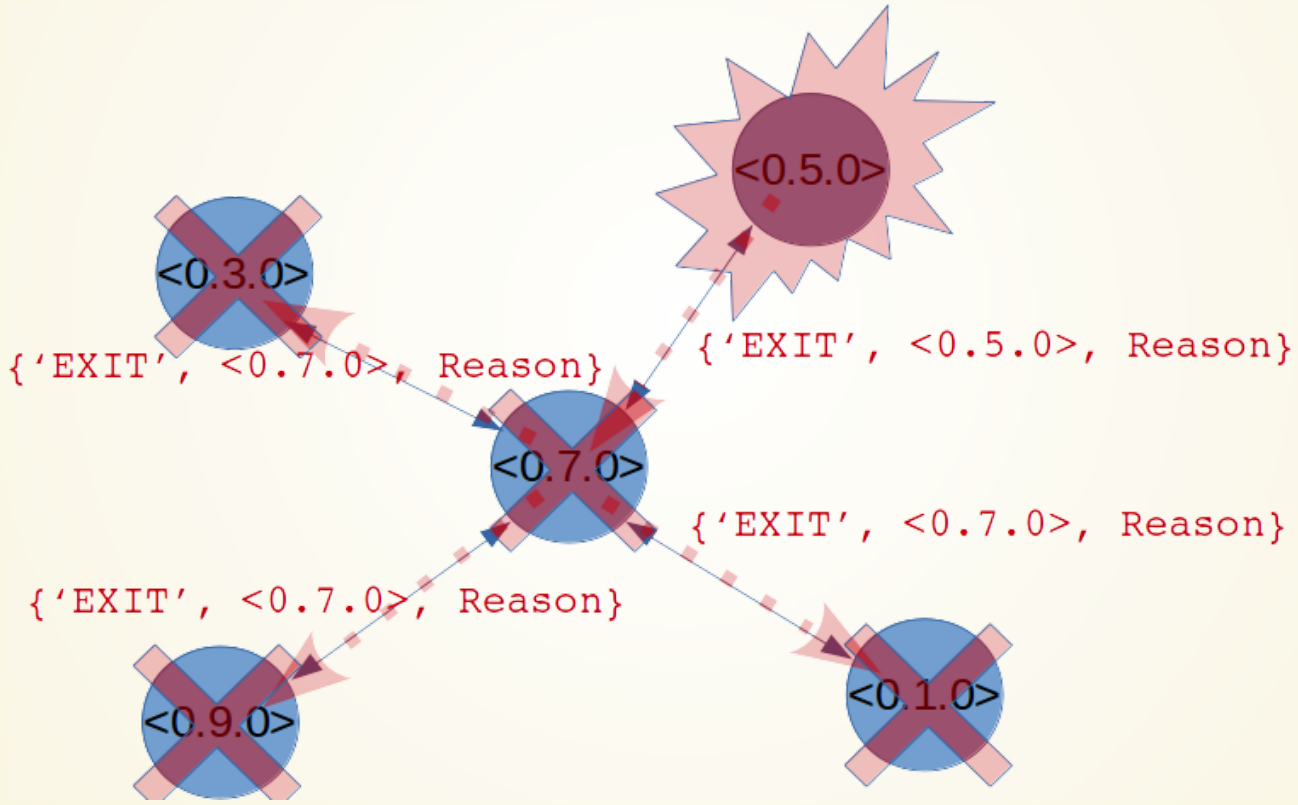
EXIT SIGNALS

Process terminates with **exit reason** which is sent to all linked processes in an **exit signal**.



EXIT SIGNALS

Process terminates with **exit reason** which is sent to all linked processes in an **exit signal**.



RECEIVING EXIT SIGNALS

Default behaviour when received an exit signal with reason other than *normal* is to terminate and propagate the signal to other linked processes. Exit signal with reason *normal* is ignored.

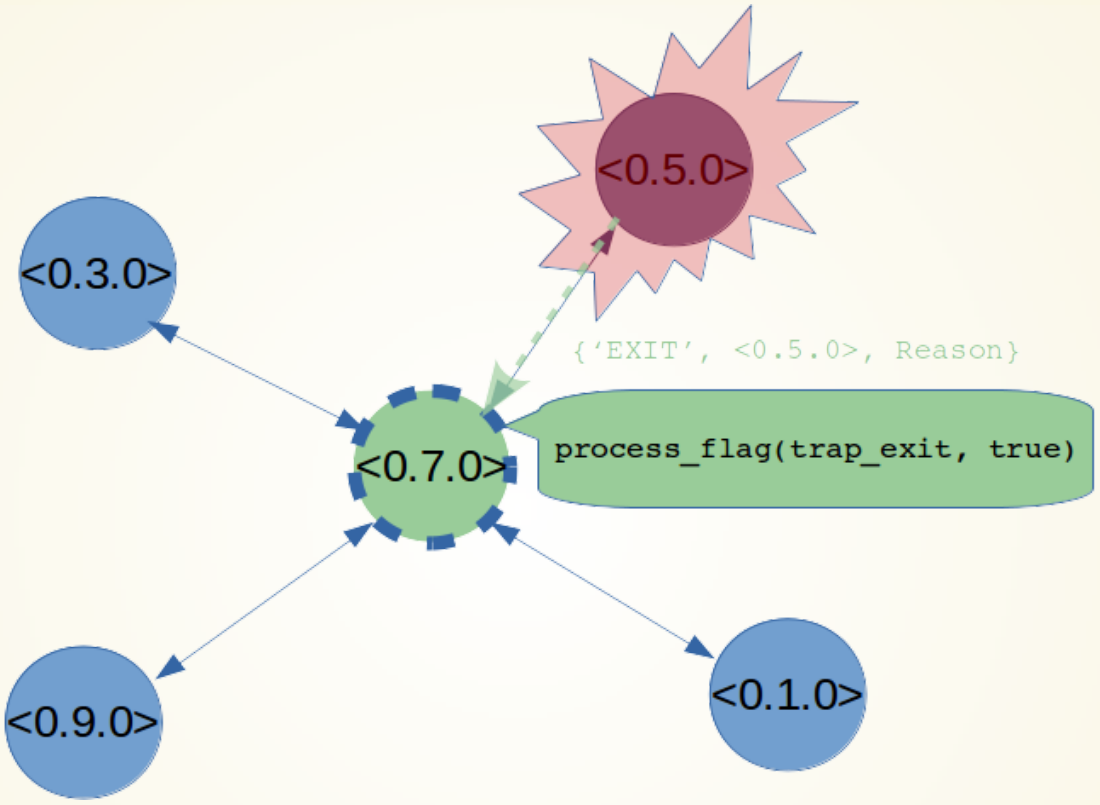
```
process_flag(trap_exit, true)
```

This expression protects the process receving exit signals from termination, the signal is converted to regular message:

```
{'EXIT', FromPid, Reason}
```

which is put into process mailbox. Exit signal is not further propagated.

RECEIVING EXIT SIGNALS

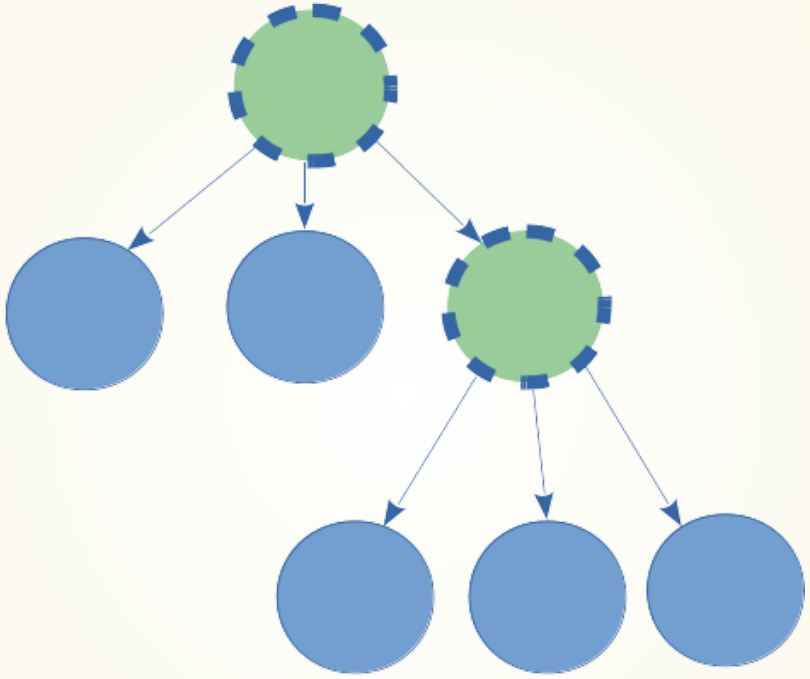


PROPAGATION SEMANTICS

Reason	`trap_exit=true`	`trap_exit=false`
normal	receives `{'EXIT', Pid, normal}`	nothing happens
kill	terminates with reason `killed`	terminates with reason `killed`
Other	receives `{'EXIT, Pid, Other}`	terminates with reason `Other`

SUPERVISION

Supervisor is a process which start children processes and monitors them.



SUPERVISION

Typical supervisor provides functions:
`start_link/2` - spawns the supervisor process (usually invoking `init/1`) and registers its Pid
`init/1` - sets `process_flag(trap_exit, true)` and starts the main loop
`start_children/1` - calls M:F(A) for every child process and adds its Pid to the list of children, this function is invoked for every static child given in `start_link/2`
`restart_child/2` - deletes the child Pid from the list of children, calls M:F(A) in the new process and stores the new Pid in the children list
`loop/1` - waits for messages from the children processes, especially `{'EXIT, Pid, Reason}` (causing restart child) or {stop, From}
`stop/1` - sends the message {stop, self()} to the supervisor process
`terminate/1` - kills the supervisor process and all children that are linked to it

The full example: simple_supervisor.erl

ROBUST SYSTEMS

COMPILATION AND LOADING

Before loading, the code must be compiled to bytecode.
Erlang compiler is located in module `compile`

```
compile:file(Module, Options)
```

Shell understands `c(Module)` - compiles and loads Module
From OS prompt compiler is run either as `erl` flag, or separate program `erlc`

```
$erl -compile Module1 ... ModuleN
$erl -make
$erlc < flags > File1.erl ... FileN.erl
```

COMPILATION AND LOADING

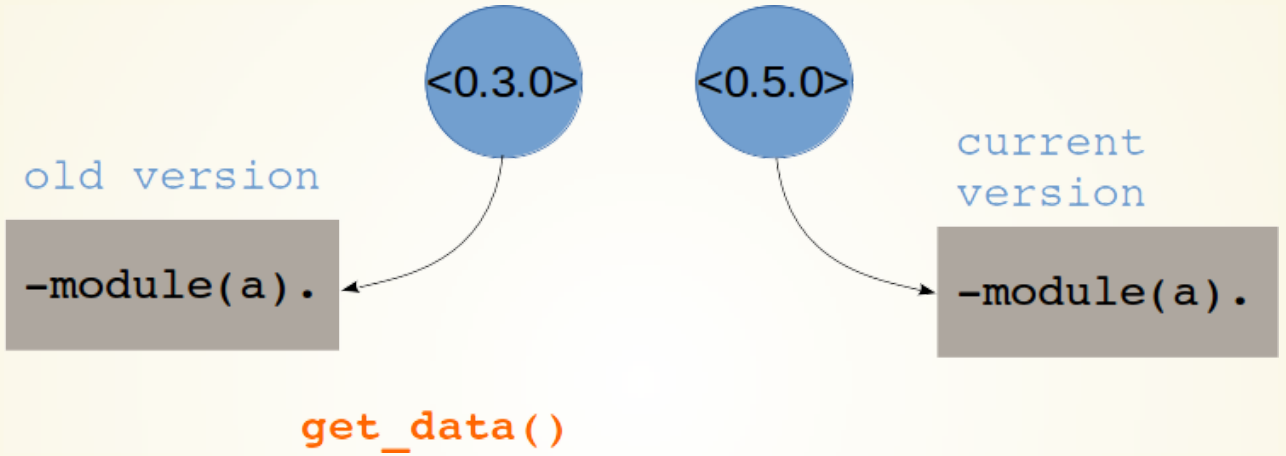
Loading object code to ERTS is done by **code server**.
Loading occurs when:

- function call to non-loaded module
- `c(Module)` or `l(Module)` typed in Erlang shell
- `code:load_file(File)` called

In **embedded mode** code is loaded at start-up according to a **boot script**.

DYNAMIC SOFTWARE UPGRADE

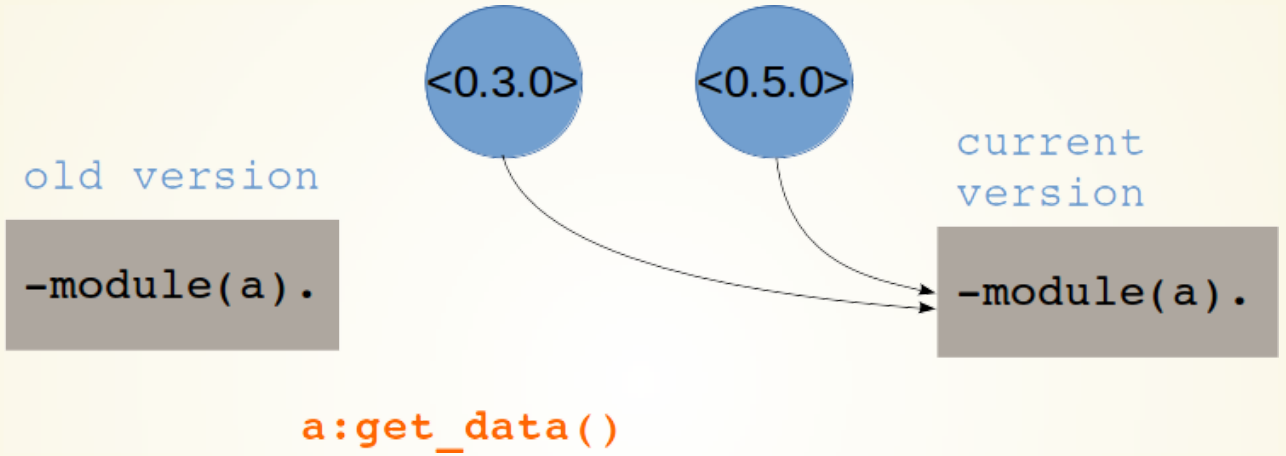
Code replacement can be done on running system, on module level.
Erlang VM always keeps maximum 2 versions of code.



implicit function call - process works on "old" version

DYNAMIC SOFTWARE UPGRADE

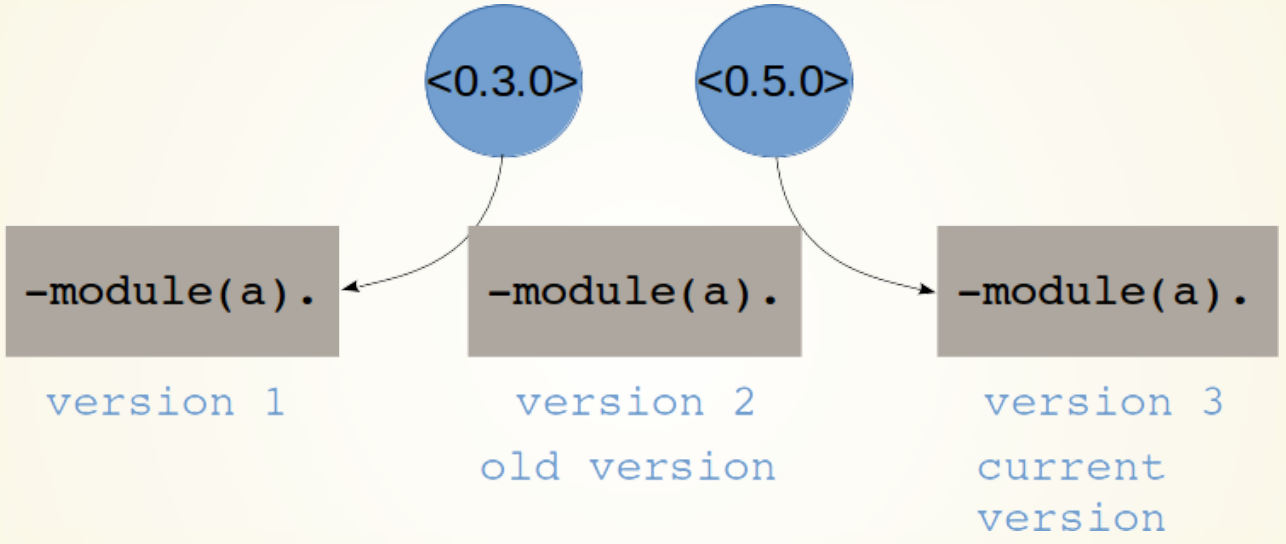
Code replacement can be done on running system, on module level.
Erlang VM always keeps maximum 2 versions of code.



fully qualified function call - process switches to "current" version

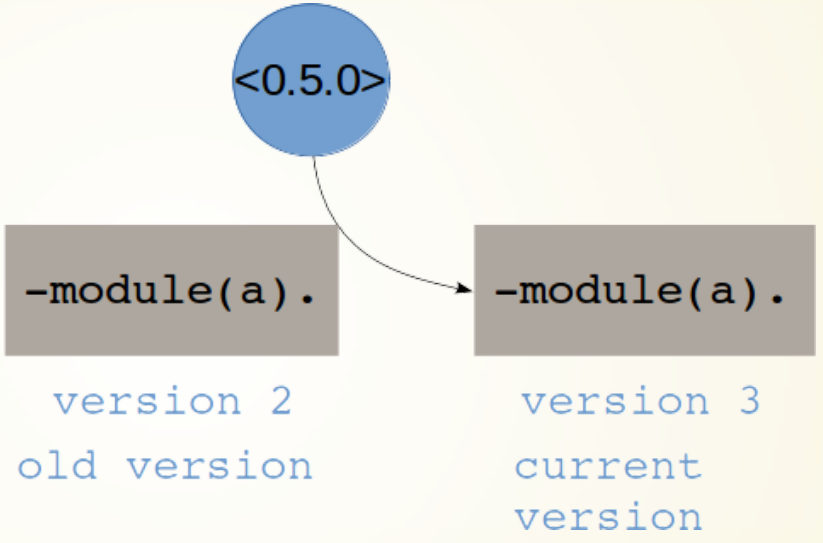
DYNAMIC SOFTWARE UPGRADE

Loading 3rd version makes this version "current", 2nd becomes "old" and 1st is purged from memory.
Consequently, all processes running on this version are killed.



DYNAMIC SOFTWARE UPGRADE

Loading 3rd version makes this version "current", 2nd becomes "old" and 1st is purged from memory.
Consequently, all processes running on this version are killed.



DYNAMIC SOFTWARE UPGRADE

More controlled way of removing old versions is provided by **code server**:

```
code:purge(Module)
```

removes the "old" version and kills all processes using it

```
code:soft_purge(Module)
```

removes the "old" version only if no process is using it