FREE eBook

# LEARNING

# Erlang Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#erlang

# Table of Contents

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: erlang-language

It is an unofficial and free Erlang Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Erlang Language.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Chapter 1: Getting started with Erlang Language

## Remarks

"Erlang is a programming language originally developed at the Ericsson Computer Science Laboratory. OTP (Open Telecom Platform) is a collection of middleware and libraries in Erlang. Erlang/OTP has been battle tested in a number of Ericsson products for building robust fault-tolerant distributed applications, for example AXD301 (ATM switch). Erlang/OTP is currently maintained by the Erlang/OTP unit at Ericsson" (erlang.org)

## Start here

For installation instructions see Installation topic.

## Links

1. Official Erlang site: https://www.erlang.org
2. Popular package manager for both Erlang and Elixir: http://hex.pm
3. Erlang Patterns: http://www.erlangpatterns.org/

## Versions

| Version | Release notes | Release Date |
|---|---|---|
| 19.2 | http://erlang.org/download/otp_src_19.2.readme | 2016-12-14 |
| 19.1 | http://erlang.org/download/otp_src_19.1.readme | 2016-09-21 |
| 19.0 | http://erlang.org/download/otp_src_19.0.readme | 2016-06-21 |
| 18.3 | http://erlang.org/download/otp_src_18.3.readme | 2016-03-15 |
| 18.2.1 | http://erlang.org/download/otp_src_18.2.1.readme | 2015-12-18 |
| 18.2 | http://erlang.org/download/otp_src_18.2.readme | 2015-12-16 |
| 18.1 | http://erlang.org/download/otp_src_18.1.readme | 2015-09-22 |
| 18.0 | http://erlang.org/download/otp_src_18.0.readme | 2015-06-24 |
| 17.5 | http://erlang.org/download/otp_src_17.5.readme | 2015-04-01 |
| 17.4 | http://erlang.org/download/otp_src_17.4.readme | 2014-12-10 |

| Version | Release notes | Release Date |
|---|---|---|
| 17.3 | http://erlang.org/download/otp_src_17.3.readme | 2014-09-17 |
| 17.1 | http://erlang.org/download/otp_src_17.1.readme | 2014-06-24 |
| 17.0 | http://erlang.org/download/otp_src_17.0.readme | 2014-04-07 |
| R16B03-1 | http://erlang.org/download/otp_src_R16B03-1.readme | 2014-01-23 |
| R16B03 | http://erlang.org/download/otp_src_R16B03.readme | 2013-12-09 |
| R16B02 | http://erlang.org/download/otp_src_R16B02.readme | 2013-09-17 |
| R16B01 | http://erlang.org/download/otp_src_R16B01.readme | 2013-06-18 |
| R16B | http://erlang.org/download/otp_src_R16B.readme | 2013-02-25 |

# Examples

### Hello World

There are two things you will need to know when writing a "hello world" application in Erlang:

1. The source code is written in the *erlang programming language* using the text editor of your choice
2. The application is then executed in the *erlang virtual machine*. In this example we will interact with the erlang VM thorugh the erlang shell.

# First the application source code:

Create a new file `hello.erl` containing the following:

```
-module(hello).
-export([hello_world/0]).

hello_world() ->
  io:format("Hello, World!~n", []).
```

Let's have a quick look at what this means:

- `-module(hello).` All erlang functions exists inside a *module*. Modules are then used to build applications, which are a collection of modules. This first line is to identify this module, namely *hello*. Modules can be compared to Java's *packages*
- `-export([hello_world/0]).` Tells the compiler which functions to make "public" (when compared to OO languages), and the *arity* of the relevant function. The arity is the number of arguments the function takes. Since in erlang a function with 1 argument is seen as a different function than one with 2 arguments even though the name may be exactly the

same. Ie, `hello_world/0` is a completely different function than `hello_world/1` for example.
- `hello_world()` This is the name of the function. The `->` indicates the transitioning to the implementation (body) of the function. This can be read as "hello_world() is defined as ...". Take note that `hello_world()` (no arguments) is identified by `hello_world/0` in the VM, and `hello_world(Some_Arg)` as `hello_world/1`.
- `io:format("Hello, World!~n", [])` From module `io`, the function `format/2` function is called, which is the function for standard output. `~n` is a format specifier that means print a new line. The `[]` is a list of variables to print indicated by format specifiers in the output string, which is in this case nothing.
- All erlang statements must end with a `.` (dot).

In Erlang, the result of the last statement in a function is returned.

## Now, let's run our application:

Start the erlang shell from same directory as the file `hello.erl` file:

```
$ erl
```

You should get a prompt that looks something like this (your version may be different):

```
Eshell V8.0  (abort with ^G)
1>
```

Now enter the following commands:

```
1> c(hello).
{ok,hello}
2> hello:hello_world().
Hello, World!
ok
```

Let's go through each line one by one:

- `c(hello)` - this command calls the function `c` on an atom `hello`. This effectively tells Erlang to find the file `hello.erl`, compile it into a module (a file named `hello.beam` will be generated in the directory) and load it into the environment.
- `{ok, hello}` - this is the result of calling the function `c` above. It is a tuple containing an atom `ok` and an atom `hello`. Erlang functions usually return either `{ok, Something}` or `{error, Reason}`.
- `hello:hello_world()` - this calls a function `hello_world()` from the module `hello`.
- `Hello, World!` - this is what our function prints.
- `ok` - this is what our function returned. Since Erlang is a functional programming language, every function returns *something*. In our case, even though we didn't return anything in `hello_world()`, the last call in that function was to `io:format(...)` and that function returned `ok`, which is in turn what our function returned.

### Modules

An erlang module is a file with couple of functions grouped together. This file usually has `.erl` extension.

A "Hello World" module with name `hello.erl` is shown below

```
-module(hello).
-export([hello_world/0]).

hello_world() ->
  io:format("Hello, World!~n", []).
```

In the file, it is required to declare the module name. As shown before in line 1. The module name and file name before `.erl` extension must be same.

## Function

Function is a set of instructions, which are grouped together. These grouped instructions together perform certain task. In erlang, all the functions will return a value when they are called.

Below is an example of a function that adds two numbers

```
add(X, Y)-> X + Y.
```

This function performs an add operation with X and Y values and returns the result. Function can be used as below

```
add(2,5).
```

Function declarations can consist of multiple clauses, separated by a semicolon. The Arguments in each of these clauses are evaluated by pattern matching. The following function will return 'tuple' if the Argument is a tuple in the Form: {test, X} where X can be any value. It will return 'list', if the Argument is a list of the length 2 in the form ["test", X], and It will return '{error, "Reason"}' in any other case:

```
function({test, X}) -> tuple;
function(["test", X]) -> list;
function(_) -> {error, "Reason"}.
```

If the argument is not a tuple, the second clause will be evaluated. If the argument is not a list, the third clause will be evaluated.

Function declarations can consist of so called 'Guards' or 'Guard Sequences'. These Guards are expressions that limit the evaluation of a function. A function with Guards is only executed, when all Guard Expressions yield a true value. Multiple Guards can be seperated by a semicolon.

```
function_name(Argument) when Guard1; Guard2; ... GuardN -> (...).
```

The function 'function_name' will only be evaluated, when the Guard Sequence is true. The follwing function will return true only if the argument `X` is in the proper range (0..15):

```
in_range(X) when X>=0; X<16 -> true;
in_range(_) -> false.
```

## List Comprehension

List comprehensions are a syntactic construct to create a list based on existing lists.
In erlang a list comprehension has the form `[Expr || Qualifier1, ..., QualifierN]`.
Where qualifiers are either generators `Pattern <- ListExpr` or filter like `integer(X)` evaluating to either `true` or `false`.

The following example shows a list comprehension with one generator and two filters.

```
[X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
```

The result is a list containing only integers greater than 3.

```
[4,5,6]
```

## Starting and Stopping the Erlang Shell

### Starting the Erlang shell

On a UNIX system you start the Erlang shell from a command prompt with the command `erl`

Example:

```
$ erl
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.0  (abort with ^G)
1>
```

The text that shows when you start the shell tells you information about which version of Erlang you are running as well as other useful information about the erlang system.

To start the shell on Windows you click the Erlang-icon in the windows start menu.

### Stopping the Erlang shell

For a controlled exit of the erlang shell you type:

```
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.0  (abort with ^G)
1> q().
```

You can also exit the Erlang shell by pressing Ctrl+C on UNIX systems or Ctrl+Break on Windows, which brings you to the following prompt:

```
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.0  (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
```

If you then press a (for abort) you will exit the shell directly.

Other ways of exiting the erlang shell are: `init:stop()` which does the same thing as `q()` or `erlang:halt().`

## Pattern Matching

One of the most common operations in erlang is pattern matching. It is used when assigning a value to a variable, in function declarations and in control-flow structures like `case` and `receive` statements. A pattern matching operation needs at least 2 parts: a pattern, and a term against wich the pattern is matched.

A variable assignment in erlang looks like this:

```
X = 2.
```

In most programming language the semantics of this operation is straightforward: Bind a value (`2`) to a name of your choice (the variable -- in this case `X`). Erlang has a slightly different approach: Match the pattern on the left hand side (`X`) to the term on the right hand side (`2`). In this case, the effect is the same: the variable `X` is now bound to the value `2`. However, with pattern matching you are able to perform more structured assignments.

```
{Type, Meta, Doc} = {document, {author, "Alice"}, {text, "Lorem Ipsum"}}.
```

This matching operation is performed, by analyzing the structure of the right hand side term, and applying all variables on the left hand side to the appropriate values of the term, so that the left side equals the right side. In this example `Type` is bound to the term: `document`, `Meta` to `{author, "Alice"}` and `Doc` to `{text, "Lorem Ipsum"}`. In this particular example the variables: `Type`, `Meta` and `Doc` are assumed to be *unbound*, so that each variable can be used.

Pattern matchings can also be built, using bound variables.

```
Identifier = error.
```

The variable `Identifier` is now bound to the value `error`. The following pattern matching operation works, because the structure matches, and the bound variable `Identifier` has the same value like the appropriate right hand side part of the term.

```
{Identifier, Reason} = {error, "Database connection timed out."}.
```

A pattern matching operation fails, when there is a mismatch between right hand side term and left hand side pattern. The following match will fail, because `Identifier` is bound to the value `error`, wich has no appropriate expression on the right hand side term.

```
{Identifier, Reason} = {fail, "Database connection timed out."}.
> ** exception error: no match of right hand side value {fail,"Database ..."}
```

Read Getting started with Erlang Language online: https://riptutorial.com/erlang/topic/825/getting-started-with-erlang-language

# Chapter 2: Behaviours

## Examples

### Using a behaviour

Add a `-behaviour` directive to your module to indicate that it follows a behaviour:

```
-behaviour(gen_server).
```

The American spelling is also accepted:

```
-behavior(gen_server).
```

Now the compiler will give a warning if you've forgotten to implement and export any of the functions required by the behaviour, e.g.:

```
foo.erl:2: Warning: undefined callback function init/1 (behaviour 'gen_server')
```

### Defining a behaviour

You can define your own behaviour by adding `-callback` directives in your module. For example, if modules implementing your behaviour need to have a `foo` function that takes an integer and returns an atom:

```
-module(my_behaviour).
-callback foo(integer()) -> atom().
```

If you use this behaviour in another module, the compiler will warn if it does not export `foo/1`, and Dialyzer will warn if the types are not correct. With this module:

```
-module(bar).
-behaviour(my_behaviour).
-export([foo/1]).

foo([]) ->
    {}.
```

and running `dialyzer --src bar.erl my_behaviour.erl`, you get these warnings:

```
bar.erl:5: The inferred type for the 1st argument of foo/1 ([]) is not a supertype of
integer(), which is expected type for this argument in the callback of the my_behaviour
behaviour
bar.erl:5: The inferred return type of foo/1 ({}) has nothing in common with atom(), which is
the expected return type for the callback of my_behaviour behaviour
```

## Optional callbacks in a custom behaviour

18.0

By default, any function specified in a `-callback` directive in a behaviour module must be exported by a module that implements that behaviour. Otherwise, you'll get a compiler warning.

Sometimes, you want a callback function to be optional: the behaviour would use it if present and exported, and otherwise fall back on a default implementation. To do that, write the `-callback` directive as usual, and then list the callback function in an `-optional_callbacks` directive:

```
-callback bar() -> ok.
-optional_callbacks([bar/0]).
```

If the module exports `bar/0`, Dialyzer will still check the type spec, but if the function is absent, you won't get a compiler warning.

In Erlang/OTP itself, this is done for the `format_status` callback function in the `gen_server`, `gen_fsm` and `gen_event` behaviours.

Read Behaviours online: https://riptutorial.com/erlang/topic/7004/behaviours

# Chapter 3: Bit Syntax: Defaults

## Introduction

The Erlang doc titled Bit Syntax has a section called Defaults that contains an error and the doc is confusing as a whole. I rewrote it.

## Examples

### Defaults explained

> 4.4 Defaults
> ...
> ...
> The default Size depends on the type. For integer it is 8. For float it is 64. For binary it is the size of the specified binary:

```
1> Bin = << 17/integer, 3.2/float, <<97, 98, 99>>/binary >>.
<<17,64,9,153,153,153,153,153,154,97,98,99>>

2> size(Bin). % Returns the number of bytes:
12            % 8 bits + 64 bits + 3*8 bits = 96 bits => 96/8 = 12 bytes
```

> In matching, a binary segment without a Size is only allowed at t

Read Bit Syntax: Defaults online: https://riptutorial.com/erlang/topic/10050/bit-syntax--defaults

# Chapter 4: Bit Syntax: Defaults

## Introduction

Blah blah blah.

## Examples

### Rewrite of the docs.

4.4 Defaults

[Beginning omitted: <<3.14>> isn't even legal syntax.]

The default Size depends on the type. For integer it is 8. For float it is 64. For binary it is the actual size of the specified binary:

```
1> Bin = << 17/integer, 3.2/float, <<97, 98, 99>>/binary >>.
<<17,64,9,153,153,153,153,153,154,97,98,99>>
  ^ |<------------------------->|<------>|
```

```
  |             float=64          binary=24
integer=8
```

```
2> size(Bin). % Returns the number of bytes:
12            % 8 bits + 64 bits + 3*8 bits = 96 bits => 96/8 = 12 bytes
```

In matching, a binary segment without a Size is only allowed at the end of the pattern, and the default Size is the rest of the binary on the right hand side of the match:

```
25> Bin = <<97, 98, 99>>.
<<"abc">>

26> << X/integer, Rest/binary >> = Bin.
<<"abc">>

27> X.
97

28> Rest.
<<"bc">>
```

All other segments with type binary in a pattern must specify a Size:

```
12> Bin = <<97, 98, 99, 100>>.
<<"abcd">>

13> << B:1/binary, X/integer, Rest/binary >> = Bin. %'unit' defaults to 8 for
```

```
<<"abcd">>                    %binary type, total segment size is Size * unit

14> B.
<<"a">>

15> X.
98

16> Rest.
<<"cd">>

17> << B2/binary, X2/integer, Rest2/binary >> = Bin.
* 1: a binary field without size is only allowed at the end of a binary pattern
```

Read Bit Syntax: Defaults online: https://riptutorial.com/erlang/topic/10051/bit-syntax--defaults

# Chapter 5: Data Types

## Remarks

Every data type in erlang is called Term. It is a generic name that means *any data type*.

## Examples

### Numbers

In Erlang, numbers are either integers or floats. Erlang uses arbitrary-precision for integers (bignums), so their values are limited only by the memory size of your system.

```
1> 11.
11
2> -44.
-44
3> 0.1.
0.1
4> 5.1e-3.
0.0051
5> 5.2e2.
520.0
```

Numbers can be used in various bases:

```
1> 2#101.
5
2> 16#ffff.
65535
```

`$`-prefix notation yields the integer value of any ASCII/Unicode character:

```
3> $a.
97
4> $A.
65
5> $2.
50
6> $🌎.
129302
```

### Atoms

An atom is an object with a name that is identified only by the name itself.

Atoms are defined in Erlang using atom literals which are either

- an unquoted string that starts with a lowercase letter and contains only letters, digits,

underscores or the `@` character, or
- A single quoted string

# Examples

```
1> hello.
hello

2> hello_world.
hello_world

3> world_Hello@.
world_Hello@

4> '1234'.
'1234'

5> '!@#$%% ä'.
'!@#$%% ä'
```

# Atoms that are used in most Erlang programs

There are some atoms that appear in almost every Erlang program, in particular because of their use in the Standard Library.

- `true` and `false` are the used to denote the respective Boolean values
- `ok` is used usually as a return value of a function that is called only for its effect, or as part of a return value, in both cases to signify a successful execution
- In the same way `error` is used to signify an error condition that doesn't warrant an early return from the upper functions
- `undefined` is usually used as a placeholder for an unspecified value

# Use as tags

`ok` and `error` are quite often used as part of a tuple, in which the first element of the tuple signals success while further elements contain the actual return value or error condition:

```
func(Input) ->
    case Input of
        magic_value ->
            {ok, got_it};
        _ ->
            {error, wrong_one}
    end.

{ok, _} = func(SomeValue).
```

# Storage

One thing to keep in mind when using atoms is that they are stored in their own global table in memory and this table is not garbage collected, so dynamically creating atoms, in particular when a user can influence the atom name is heavily discouraged.

## Binaries and Bitstrings

A binary is a sequence of unsigned 8-bit bytes.

```
1> <<1,2,3,255>>.
<<1,2,3,255>>
2> <<256,257,258>>.
<<0,1,2>>
3> <<"hello","world">>.
<<"helloworld">>
```

A bitstring is a generalized binary whose length in bits isn't necessarily a multiple of 8.

```
1> <<1:1, 0:2, 1:1>>.
<<9:4>> % 4 bits bitstring
```

## Tuples

A tuple is a fixed length ordered sequence of other Erlang terms. Each element in the tuple can be any type of term (any data type).

```
1> {1, 2, 3}.
{1,2,3}
2> {one, two, three}.
{one,two,three}
3> {mix, atom, 123, {<<1,2>>, [list]}}.
{mix,atom,123,{<<1,2>>,[list]}}
```

## Lists

A list in Erlang is a sequence of zero or more Erlang terms, implemented as a singly linked list. Each element in the list can be any type of term (any data type).

```
1> [1,2,3].
[1,2,3]
2> [wow,1,{a,b}].
[wow,1,{a,b}]
```

The list's head is the first element of the list.

The list's tail is the remainder of the list (without the head). It is also a list.
You can use `hd/1` and `tl/1` or match against `[H|T]` to get the head and tail of the list.

```
3> hd([1,2,3]).
1
4> tl([1,2,3]).
[2,3]
5> [H|T] = [1,2,3].
[1,2,3]
6> H.
1
7> T.
[2,3]
```

# Prepending an element to a list

```
8> [new | [1,2,3]].
[new,1,2,3]
```

# Concatenating lists

```
9> [concat,this] ++ [to,this].
[concat,this,to,this]
```

# Strings

In Erlang, strings are not a separate data type: they're just lists of integers representing ASCII or Unicode code points:

```
> [97,98,99].
"abc"
> [97,98,99] =:= "abc".
true
> hd("ABC").
65
```

When the Erlang shell is going to print a list, it tries to guess whether you actually meant it to be a string. You can turn that behaviour off by calling `shell:strings(false)`:

```
> [8].
"\b"
> shell:strings(false).
true
> [8].
[8]
```

In the above example, the integer 8 is interpreted as the ASCII control character for backspace, which the shell considers to be a "valid" character in a string.

### Processes Identifiers (Pid)

Each process in erlang has a process identifier (`Pid`) in this format `<x.x.x>`, `x` being a natural number. Below is an example of a `Pid`

```
<0.1252.0>
```

`Pid` can be used to send messages to the process using 'bang' (`!`), also `Pid` can be bounded to a variable, both are shown below

```
MyProcessId = self().
MyProcessId ! {"Say Hello"}.
```

Read more about creating processes and more in general about processes in erlang

## Funs

Erlang is a functional programming language. One of the features in a function programming language is handling functions as data (functional objects).

- Pass a function as an argument to another function.
- Return function as a result of a function.
- Hold functions in some data structure.

In Erlang those functions are called funs. Funs are anonymous functions.

```
1> Fun = fun(X) -> X*X end.
#Fun<erl_eval.6.52032458>
2> Fun(5).
25
```

Funs may also have several clauses.

```
3> AddOrMult = fun(add,X) -> X+X;
3>                 (mul,X) -> X*X
3> end.
#Fun<erl_eval.12.52032458>
4> AddOrMult(mul,5).
25
5> AddOrMult(add,5).
10
```

You may also use module functions as funs with the syntax: `fun Module:Function/Arity`.
For example, lets take the function `max` from `lists` module, which has arity 1.

```
6> Max = fun lists:max/1.
#Fun<lists.max.1>
7> Max([1,3,5,9,2]).
9
```

## Maps

A map is an associative array or dictionary composed of (key, value) pairs.

```
1> M0 = #{}.
#{}
2> M1 = #{ "name" => "john", "age" => "28" }.
#{"age" => "28","name" => "john"}
3> M2 = #{ a => {M0, M1} }.
#{a => {#{},#{"age" => "28","name" => "john"}}}
```

To update a map :

```
1> M = #{ 1 => x }.
2> M#{ 1 => c }.
#{1 => c}
3> M.
#{1 => x}
```

Only update some existing key:

```
1> M = #{ 1 => a, 2 => b}.
2> M#{ 1 := c, 2:= d }.
#{1 => c,2 => d}
3> M#{ 3 := c }.
** exception error: {badkey,3}
```

Pattern matching:

```
1> M = #{ name => "john", age => 28 }.
2> #{ name := Name, age := Age } = M.
3> Name.
"john"
4> Age.
28
```

## Bit Syntax: Defaults

Clarification of Erlang doc on Bit Syntax:

> 4.4 Defaults
>
> [Beginning omitted: <<3.14>> isn't even legal syntax.]
>
> The default Size depends on the type. For integer it is 8. For float it is 64. For binary it is the actual size of the specified binary:

```
   1> Bin = << 17/integer, 3.2/float, <<97, 98, 99>>/binary >>.
   <<17,64,9,153,153,153,153,153,154,97,98,99>>
     ^ |<-------------------------->|<------>|
     |            float=64           binary=24
 integer=8


   2> size(Bin). % Returns the number of bytes:
```

```
    12            % 8 bits + 64 bits + 3*8 bits = 96 bits => 96/8 = 12 bytes
```

In matching, a binary segment without a Size is only allowed at the end of the pattern, and the default Size is the rest of the binary on the right hand side of the match:

```
25> Bin = <<97, 98, 99>>.
<<"abc">>

26> << X/integer, Rest/binary >> = Bin.
<<"abc">>

27> X.
97

28> Rest.
<<"bc">>
```

All other segments with type binary in a pattern must specify a Size:

```
12> Bin = <<97, 98, 99, 100>>.
<<"abcd">>

13> << B:1/binary, X/integer, Rest/binary >> = Bin. %'unit' defaults to 8 for
<<"abcd">>                    %binary type, total segment size is Size * unit

14> B.
<<"a">>

15> X.
98

16> Rest.
<<"cd">>

17> << B2/binary, X2/integer, Rest2/binary >> = Bin.
* 1: a binary field without size is only allowed at the end of a binary pattern
```

Read Data Types online: https://riptutorial.com/erlang/topic/1128/data-types

# Chapter 6: director

## Introduction

Flexible, fast and powerful supervisor library for Erlang processes.

## Remarks

## Warnings

- Do not use `'count'=>infinity` and element `restart` in your plan.
  like:

```
Childspec = #{id => foo
              ,start => {bar, baz, [arg1, arg2]}
              ,plan => [restart]
              ,count => infinity}.
```

If your process did not start after crash, **director** will lock and retries to restart your process `infinity` times ! If you are using `infinity` for `'count'`, always use `{restart, MiliSeconds}` in `'plan'` instead of `restart`.

- If you have plans like:

```
Childspec1 = #{id => foo
               ,start => {bar, baz}
               ,plan => [restart,restart,delete,wait,wait, {restart, 4000}]
               ,count => infinity}.

Childspec2 = #{id => foo
               ,start => {bar, baz}
               ,plan => [restart,restart,stop,wait, {restart, 20000}, restart]
               ,count => infinity}.

Childspec3 = #{id => foo
               ,start => {bar, baz}
               ,plan => [restart,restart,stop,wait, {restart, 20000}, restart]
               ,count => 0}.

Childspec4 = #{id => foo
               ,start => {bar, baz}
               ,plan => []
               ,count => infinity}.
```

The rest of `delete` element in `Childspec1` and the rest of `stop` element in `Childspec2` will never evaluate!
In `Childspec3` you want to run your plan 0 times!
In `ChildSpec4` you have not any plan to run `infinity` times!

- When you upgrade a release using `release_handler`, `release_handler` calls `supervisor:get_callback_module/1` for fetching its callback module.
  In OTP<19 `get_callback_module/1` uses supervisor internal state record for giving its callback module. Our **director** does not know about supervisor internal state record, then `supervisor:get_callback_module/1` does not work with **director**s.
  Good news is that in OTP>=19 `supervisor:get_callback_module/1` works perfectly with **director**s :).

```
1> foo:start_link().
{ok,<0.105.0>}

2> supervisor:get_callback_module(foo_sup).
foo

3>
```

# Examples

## Download

```
Pouriya@Jahanbakhsh ~ $ git clone https://github.com/Pouriya-Jahanbakhsh/director.git
```

## Compile

Note that **OTP>=19** required (if you want to upgrade it using `release_handler`).
Go to `director` and use `rebar` or `rebar3`.

```
Pouriya@Jahanbakhsh ~ $ cd director
```

rebar

```
Pouriya@Jahanbakhsh ~/director $ rebar compile
==> director_test (compile)
Compiled src/director.erl
Pouriya@Jahanbakhsh ~/director $
```

rebar3

```
Pouriya@Jahanbakhsh ~/director $ rebar3 compile
===> Verifying dependencies...
===> Compiling director
Pouriya@Jahanbakhsh ~/director $
```

## How it works

**director** needs a callback module (like OTP supervisor).
In callback module you should export function `init/1`.
What `init/1` should return? wait, i'll explain step by step.

```
-module(foo).
-export([init/1]).

init(_InitArg) ->
    {ok, []}.
```

Save above code in `foo.erl` in **director** directory and go to the Erlang shell.
Use `erl -pa ./ebin` if you used `rebar` to compile it and use `rebar3 shell` if you used `rebar3`.

```
Erlang/OTP 19 [erts-8.3] [source-d5c06c6] [64-bit] [smp:8:8] [async-threads:0] [hipe] [kernel-
poll:false]

Eshell V8.3  (abort with ^G)
1> c(foo).
{ok,foo}

2> Mod = foo.
foo

3> InitArg = undefined. %% i don't need it yet.
undefined

4> {ok, Pid} = director:start_link(Mod, InitArg).
{ok,<0.112.0>}

5>
```

Now we have a supervisor without children.
Good news is that **director** comes with full OTP/supervisor API and it has its advanced features and specific approach too.

```
5> director:which_children(Pid). %% You can use supervisor:which_children(Pid) too :)
[]

6> director:count_children(Pid). %% You can use supervisor:count_children(Pid) too :)
[{specs,0},{active,0},{supervisors,0},{workers,0}]

7> director:get_pids(Pid). %% You can NOT use supervisor:get_pids(Pid) because it hasn't :D
[]
```

OK, I'll make simple `gen_server` and give it to our **director**.

```
-module(bar).
-behaviour(gen_server).
-export([start_link/0
        ,init/1
        ,terminate/2]). %% i am not going to use handle_call, handle_cast ,etc.


start_link() ->
    gen_server:start_link(?MODULE, null, []).

init(_GenServerInitArg) ->
    {ok, state}.

terminate(_Reason, _State) ->
```

```
    ok.
```

Save above code in `bar.erl` and got back to the Shell.

```
8> c(bar).
bar.erl:2: Warning: undefined callback function code_change/3 (behaviour 'gen_server')
bar.erl:2: Warning: undefined callback function handle_call/3 (behaviour 'gen_server')
bar.erl:2: Warning: undefined callback function handle_cast/2 (behaviour 'gen_server')
bar.erl:2: Warning: undefined callback function handle_info/2 (behaviour 'gen_server')
{ok,bar}

%% You should define unique id for your process.
9> Id = bar_id.
bar_id

%% You should tell diector about start module and function for your process.
%% Should be tuple {Module, Function, Args}.
%% If your start function doesn't need arguments (like our example)
%% just use {Module, function}.
10> start = {bar, start_link}.
{bar,start_link}

%% What is your plan for your process?
%% I asked you some questions at the first of this README file.
%% Plan should be an empty list or list with n elemenst.
%% Every element can be one of
%% 'restart'
%% 'delete'
%% 'stop'
%% {'stop', Reason::term()}
%% {'restart', Time::pos_integer()}
%% for example my plan is:
%% [restart, {restart, 5000}, delete]
%% In first crash director will restart my process,
%% after next crash director will restart it after 5000 mili-seconds
%% and after third crash director will not restart it and will delete it
11> Plan = [restart, {restart, 5000}, delete].
[restart,{restart,5000},delete]

%% What if i want to restart my process 500 times?
%% Do i need a list with 500 'restart's?
%% No, you just need a list with one element, I'll explain it later.

12> Childspec = #{id => Id
                  ,start => Start
                  ,plan => Plan}.
#{id => bar_id,
  plan => [restart,{restart,5000},delete],
  start => {bar,start_link}}

13> director:start_child(Pid, Childspec). %% You can use supervisor:start_child(Pid,
ChildSpec) too :)
{ok,<0.160.0>}

14>
```

Lets check it

```
14> director:which_children(Pid).
```

```
[{bar_id,<0.160.0>,worker,[bar]}]

15> director:count_children(Pid).
[{specs,1},{active,1},{supervisors,0},{workers,1}]

%% What was get_pids/1?
%% It will returns all RUNNING ids with their pids.
16> director:get_pids(Pid).
[{bar_id,<0.160.0>}]

%% We can get Pid for specific RUNNING id too
17> {ok, BarPid1} = director:get_pid(Pid, bar_id).
{ok,<0.160.0>}


%% I want to kill that process
18> erlang:exit(BarPid1, kill).
true

%% Check all running pids again
19> director:get_pids(Pid).
[{bar_id,<0.174.0>}] %% changed (restarted)

%% I want to kill that process again
%% and i will check children before spending time
20> {ok, BarPid2} = director:get_pid(Pid, bar_id), erlang:exit(BarPid2, kill).
true

21> director:get_pids(Pid).
[]

22> director:which_children(Pid).
[{bar_id,restarting,worker,[bar]}] %% restarting

23> director:get_pid(Pid, bare_id).
{error,not_found}

%% after 5000 ms
24> director:get_pids(Pid).
[{bar_id,<0.181.0>}]

25> %% Yoooohooooooo
```

I mentioned **advanced features**, what are they? Lets see other acceptable keys for `Childspec` map.

```
-type childspec() :: #{'id' => id()
                       ,'start' => start()
                       ,'plan' => plan()
                       ,'count' => count()
                       ,'terminate_timeout' => terminate_timeout()
                       ,'type' => type()
                       ,'modules' => modules()
                       ,'append' => append()}.

%% 'id' is mandatory and can be any Erlang term
-type  id() :: term().

%% Sometimes 'start' is optional ! just wait and read carefully
-type  start() :: {module(), function()} % default Args is []
```

```
                | mfa().

%% I explained 'restart', 'delete' and {'restart', MiliSeconds}
%% 'stop': director will crash with reason {stop, [info about process crash]}.
%% {'stop', Reason}: director exactly will crash with reason Reason.
%% 'wait': director will not restart process,
%%  but you can restart it using director:restart_child/2 and you can use
supervisor:restart_child/2 too.
%% fun/2: director will execute fun with 2 arguments.
%%  First argument is crash reason for process and second argument is restart count for
process.
%%  Fun should return terms like other plan elements.
%% Default plan is:
%% [fun
%%      (normal, _RestartCount) ->
%%          delete;
%%      (shutdown, _RestartCount) ->
%%          delete;
%%      ({shutdown, _Reason}, _RestartCount) ->
%%          delete;
%%      (_Reason, _RestartCount) ->
%%          restart
%%  end]
-type  plan() :: [plan_element()] | [].
-type   plan_element() :: 'restart'
                        | {'restart', pos_integer()}
                        | 'wait'
                        | 'stop'
                        | {'stop', Reason::term()}
                        | fun((Reason::term()
                              ,RestartCount::pos_integer()) ->
                                  'restart'
                                | {'restart', pos_integer()}
                                | 'wait'
                                | 'stop'
                                | {'stop', Reason::term()}).

%% How much time you want to run plan?
%% Default value of 'count' is 1.
%% Again, What if i want to restart my process 500 times?
%%  Do i need a list with 500 'restart's?
%%  You just need plan ['restart'] and 'count' 500 :)
-type  count() :: 'infinity' | non_neg_integer().

%% How much time director should wait for process termination?
%% 0 means brutal kill and director will kill your process using erlang:exit(YourProcess,
kill).
%% For workers default value is 1000 mili-seconds and for supervisors default value is
'infinity'.
-type  terminate_timeout() :: 'infinity' | non_neg_integer().

%% default is 'worker'
-type  type() :: 'worker' | 'supervisor'.

%% Default is first element of 'start' (process start module)
-type  modules() :: [module()] | 'dynamic'.

%% :)
%% Default value is 'false'
%% I'll explan it
-type  append() :: boolean().
```

Edit `foo` module:

```
-module(foo).
-export([start_link/0
        ,init/1]).

start_link() ->
    director:start_link({local, foo_sup}, ?MODULE, null).

init(_InitArg) ->
    Childspec = #{id => bar_id
                  ,plan => [wait]
                  ,start => {bar,start_link}
                  ,count => 1
                  ,terminate_timeout => 2000},
    {ok, [Childspec]}.
```

Go to the Erlang shell again:

```
1> c(foo).
{ok,foo}

2> foo:start_link().
{ok,<0.121.0>}

3> director:get_childspec(foo_sup, bar_id).
{ok,#{append => false,count => 1,id => bar_id,
      modules => [bar],
      plan => [wait],
      start => {bar,start_link,[]},
      terminate_timeout => 2000,type => worker}}

4> {ok, Pid} = director:get_pid(foo_sup, bar_id), erlang:exit(Pid, kill).
true

5> director:which_children(foo_sup).
[{bar_id,undefined,worker,[bar]}] %% undefined

6> director:count_children(foo_sup).
[{specs,1},{active,0},{supervisors,0},{workers,1}]

7> director:get_plan(foo_sup, bar_id).
{ok,[wait]}

%% I can change process plan
%% I killed process one time.
%% If i kill it again, entire supervisor will crash with reason {reached_max_restart_plan...
because 'count' is 1
%% But after changing plan, its counter will restart from 0.
8> director:change_plan(foo_sup, bar_id, [restart]).
ok

9> director:get_childspec(foo_sup, bar_id).
{ok,#{append => false,count => 1,id => bar_id,
      modules => [bar],
      plan => [restart], %% here
      start => {bar,start_link,[]},
      terminate_timeout => 2000,type => worker}}
```

```
10> director:get_pids(foo_sup).
[]

11> director:restart_child(foo_sup, bar_id).
{ok,<0.111.0>}

12> {ok, Pid2} = director:get_pid(foo_sup, bar_id), erlang:exit(Pid2, kill).
true

13> director:get_pid(foo_sup, bar_id).
{ok,<0.113.0>}

14> %% Hold on
```

Finally what the `append` key is?

actually always we have one `DefaultChildspec`.

```
14> director:get_default_childspec(foo_sup).
{ok,#{count => 0,modules => [],plan => [],terminate_timeout => 0}}

15>
```

`DefaultChildspec` is like normal childspecs except that it can't accept `id` and `append` keys.
If i change `append` value to `true` in my `Childspec`:
My `terminate_timeout` will be added to `terminate_timeout` of `DefaultChildspec`.
My `count` will be added to `count` of `DefaultChildspec`.
My `modules` will be added to `modules` of `DefaultChildspec`.
My `plan` will be added to `plan` of `DefaultChildspec`.
And if i have `start` key with value `{ModX, FuncX, ArgsX}` in `DefaultChildspec` and `start` key with value `{ModY, FunY, ArgsY}` in `Childspec`, final value will be `{ModY, FuncY, ArgsX ++ ArgsY}`.
And finally if i have `start` key with value `{Mod, Func, Args}` in `DefaultChildspec`, `start` key in `Childspec` is optional for me.
You can return your own `DefaultChildspec` as third element of tuple in `init/1`.
Edit `foo.erl`:

```
-module(foo).
-behaviour(director). %% Yes, this is a behaviour
-export([start_link/0
        ,init/1]).

start_link() ->
    director:start_link({local, foo_sup}, ?MODULE, null).

init(_InitArg) ->
    Childspec = #{id => bar_id
                 ,plan => [wait]
                 ,start => {bar,start_link}
                 ,count => 1
                 ,terminate_timeout => 2000},
    DefaultChildspec = #{start => {bar, start_link}
                         ,terminate_timeout => 1000
                         ,plan => [restart]
                         ,count => 5},
    {ok, [Childspec], DefaultChildspec}.
```

Restart the shell:

```
1> c(foo).
{ok,foo}

2> foo:start_link().
{ok,<0.111.0>}

3> director:get_pids(foo_sup).
[{bar_id,<0.112.0>}]

4> director:get_default_childspec(foo_sup).
{ok,#{count => 5,
      plan => [restart],
      start => {bar,start_link,[]},
      terminate_timeout => 1000}}

5> Childspec1 = #{id => 1, append => true},
%% Default 'plan' is [Fun], so 'plan' will be [restart] ++ [Fun] or [restart, Fun].
%% Default 'count' is 1, so 'count' will be 1 + 5 or 6.
%% Args in above Childspec is [], so Args will be [] ++ [] or [].
%% Default 'terminate_timeout' is 1000, so 'terminate_timeout' will be 1000 + 1000 or 2000.
%% Default 'modules' is [bar], so 'modules' will be [bar] ++ [] or [bar].
5> director:start_child(foo_sup, Childspec1).
{ok,<0.116.0>}

%% Test
6> director:get_childspec(foo_sup, 1).
{ok,#{append => true,
      count => 6,
      id => 1,
      modules => [bar],
      plan => [restart,#Fun<director.default_plan_element_fun.2>],
      start => {bar,start_link,[]},
      terminate_timeout => 2000,
      type => worker}}

7> director:get_pids(foo_sup).
[{bar_id,<0.112.0>},{1,<0.116.0>}]

%% I want to have 9 more children like that
8> [director:start_child(foo_sup
                         ,#{id => Count, append => true})
   || Count <- lists:seq(2, 10)].
[{ok,<0.126.0>},
 {ok,<0.127.0>},
 {ok,<0.128.0>},
 {ok,<0.129.0>},
 {ok,<0.130.0>},
 {ok,<0.131.0>},
 {ok,<0.132.0>},
 {ok,<0.133.0>},
 {ok,<0.134.0>}]

10> director:count_children(foo_sup).
[{specs,11},{active,11},{supervisors,0},{workers,11}]

11>
```

You can change `defaultChildspec` dynamically using `change_default_childspec/2` !

And you can change `Childspec` of children dynamically too and set their `append` to `true` !
But with changing them in different parts of code, you will make **spaghetti code**

## Can i debug director?

Yessssss, **diorector** has its own debug and accepts standard `sys:dbg_opt/0`.
**director** sends valid logs to `sasl` and `error_logger` in different states too.

```
1> Name = {local, dname},
   Mod = foo,
   InitArg = undefined,
   DbgOpts = [trace],
   Opts = [{debug, DbgOpts}].
[{debug,[trace]}]

2> director:start_link(Name, Mod, InitArg, Opts).
{ok,<0.106.0>}
3>
3> director:count_children(dname).
*DBG* director "dname" got request "count_children" from "<0.102.0>"
*DBG* director "dname" sent "[{specs,1},
                              {active,1},
                              {supervisors,0},
                              {workers,1}]" to "<0.102.0>"
[{specs,1},{active,1},{supervisors,0},{workers,1}]

4> director:change_plan(dname, bar_id, [{restart, 5000}]).
*DBG* director "dname" got request "{change_plan,bar_id,[{restart,5000}]}" from "<0.102.0>"
*DBG* director "dname" sent "ok" to "<0.102.0>"
ok

5> {ok, Pid} = director:get_pid(dname, bar_id).
*DBG* director "dname" got request "{get_pid,bar_id}" from "<0.102.0>"
*DBG* director "dname" sent "{ok,<0.107.0>}" to "<0.102.0>"
{ok,<0.107.0>}

%% Start SASL
6> application:start(sasl).
ok
... %% Log about starting SASL

7> erlang:exit(Pid, kill).
*DBG* director "dname" got exit signal for pid "<0.107.0>" with reason "killed"
true

=SUPERVISOR REPORT==== 4-May-2017::12:37:41 ===
     Supervisor: dname
     Context:    child_terminated
     Reason:     killed
     Offender:   [{id,bar_id},
                  {pid,<0.107.0>},
                  {plan,[{restart,5000}]},
                  {count,1},
                  {count2,0},
                  {restart_count,0},
                  {mfargs,{bar,start_link,[]}},
                  {plan_element_index,1},
                  {plan_length,1},
                  {timer_reference,undefined},
```

```
                  {terminate_timeout,2000},
                  {extra,undefined},
                  {modules,[bar]},
                  {type,worker},
                  {append,false}]
8>

%% After 5000 mili-seconds
*DBG* director "dname" got timer event for child-id "bar_id" with timer reference
"#Ref<0.0.1.176>"

=PROGRESS REPORT==== 4-May-2017::12:37:46 ===
          supervisor: dname
             started: [{id,bar_id},
                       {pid,<0.122.0>},
                       {plan,[{restart,5000}]},
                       {count,1},
                       {count2,1},
                       {restart_count,1},
                       {mfargs,{bar,start_link,[]}},
                       {plan_element_index,1},
                       {plan_length,1},
                       {timer_reference,#Ref<0.0.1.176>},
                       {terminate_timeout,2000},
                       {extra,undefined},
                       {modules,[bar]},
                       {type,worker},
                       {append,false}]
8>
```

## Generate API documentation

rebar:

```
Pouriya@Jahanbakhsh ~/director $ rebar doc
```

rebar3:

```
Pouriya@Jahanbakhsh ~/director $ rebar3 edoc
```

erl

```
Pouriya@Jahanbakhsh ~/director $ mkdir -p doc &&
                                 erl -noshell\
                                     -eval "edoc:file(\"./src/director.erl\", [{dir,
\"./doc\"}]),init:stop()."
```

After running one of the above commands, HTML documentation should be in `doc` directory.

Read director online: https://riptutorial.com/erlang/topic/9878/director

# Chapter 7: External Term Format

## Introduction

External Term Format is a binary format used to communicate to outside world. You can use it with any language through ports, drivers or NIF. BERT (Binary ERlang Term) can be used in other languages.

## Examples

### Using ETF with Erlang

```
binary_to_term().
```

### Using ETF with C

# Initializing data structure

```
#include <stdio.h>
#include <string.h>
#include <ei.h>
```

# Encoding number

```
#include <stdio.h>
#include <ei.h>

int
main() {
}
```

# Encoding atom

# Encoding tuple

# Encoding list

# Encoding map

Read External Term Format online: https://riptutorial.com/erlang/topic/10731/external-term-format

# Chapter 8: File I/O

## Examples

### Reading from a file

Let's assume you have a file **lyrics.txt** which contains the following data:

```
summer has come and passed
the innocent can never last
wake me up when september ends
```

# Read the entire file at a time

By using `file:read_file(File)`, you can read the entire file. It's an atomic operation:

```
1> file:read_file("lyrics.txt").
{ok,<<"summer has come and passed\r\nthe innocent can never last\r\nWake me up w
hen september ends\r\n">>}
```

# Read one line at a time

`io:get_line` reads the text until the newline or the end of file.

```
1> {ok, S} = file:open("lyrics.txt", read).
{ok,<0.57.0>}
2> io:get_line(S, '').
"summer has come and passed\n"
3> io:get_line(S, '').
"the innocent can never last\n"
4> io:get_line(S, '').
"wake me up when september ends\n"
5> io:get_line(S, '').
eof
6> file:close(S).
ok
```

# Read with the random access

`file:pread(IoDevice, Start, Len)` reads from `Start` as much as `Len` from `IoDevice`.

```
1> {ok, S} = file:open("lyrics.txt", read).
{ok,<0.57.0>}
2> file:pread(S, 0, 6).
{ok,"summer"}
```

```
3> file:pread(S, 7, 3).
{ok,"has"}
```

### Writing to a file

# Write one line at a time

Open a file with `write` mode and use `io:format/2`:

```
1> {ok, S} = file:open("fruit_count.txt", [write]).
{ok,<0.57.0>}
2> io:format(S, "~s~n", ["Mango 5"]).
ok
3> io:format(S, "~s~n", ["Olive 12"]).
ok
4> io:format(S, "~s~n", ["Watermelon 3"]).
ok
5>
```

The result will be a file named **fruit_count.txt** with the following contents:

```
Mango 5
Olive 12
Watermelon 3
```

Note that opening a file in write mode will created it, if not already existent in the file system.

Note also that using the `write` option with `file:open/2` will **truncate** the file (even if you don't write anything into it). To prevent this, open the file in `[read,write]` or `[append]` mode.

# Write the entire file at once

`file:write_file(Filename, IO)` is the simplest function for writing a file at once. If the file already exists, it will overwritten, otherwise it will be created.

```
1> file:write_file("fruit_count.txt", ["Mango 5\nOlive 12\nWatermelon 3\n"
]).
ok
2> file:read_file("fruit_count.txt").
{ok,<<"Mango 5\nOlive 12\nWatermelon 3\n">>}
3>
```

# Write with random access

For random access writing, `file:pwrite(IoDevice, Location, Bytes)` is used. If you want to replace some string in the file, this method is useful.

Let's assume you want to change "Olive 12" to "Apple 15" in the file created above.

```
1> {ok, S} = file:open("fruit_count.txt", [read, write]).
{ok,{file_descriptor,prim_file,{#Port<0.412>,676}}}
2> file:pwrite(S, 8, ["Apple 15\n"]).
ok
3> file:read_file("fruit_count.txt").
{ok,<<"Mango 5\nApple 15\nWatermelon 3">>}
4> file:close(S).
ok
5>
```

Read File I/O online: https://riptutorial.com/erlang/topic/5232/file-i-o

# Chapter 9: Format Strings

## Syntax

- io:format(FormatString, Args) % write to standard output
- io:format(standard_error, FormatString, Args) % write to standard error
- io:format(F, FormatString, Args) % write to open file
- io_lib:format(FormatString, Args) % return an iolist

## Examples

### Common control sequences in format strings

While there are many different control sequences for `io:format` and `io_lib:format`, most of the time you'll use only three different ones: `~s`, `~p` and `~w`.

## ~s

The `~s` is for *strings*.

It prints strings, binaries and atoms. (Anything else will cause a `badarg` error.) It doesn't quote or escape anything; it just prints the string itself:

```
%% Printing a string:
> io:format("~s\n", ["hello world"]).
hello world

%% Printing a binary:
> io:format("~s\n", [<<"hello world">>]).
hello world

%% Printing an atom:
> io:format("~s\n", ['hello world']).
hello world
```

## ~w

The `~w` is for *writing with standard syntax*.

It can can print any Erlang term. The output can be parsed to return the original Erlang term, unless it contained terms that don't have a parsable written representation, i.e. pids, ports and references. It doesn't insert any newlines or indentation, and strings are always interpreted as lists:

```
> io:format("~w\n", ["abc"]).
```

```
[97,98,99]
```

## ~p

The `~p` is for *pretty-printing*.

It can can print any Erlang term. The output differs from `~w` in the following ways:

- Newlines are inserted if the line would otherwise be too long.
- When newlines are inserted, the next line is indented to line up with a previous term on the same level.
- If a list of integers looks like a printable string, it is interpreted as one.

```
> io:format("~p\n",
[{this,is,a,tuple,with,many,elements,'and',a,list,'of',numbers,[97,98,99],that,'end',up,making,the,line

{this,is,a,tuple,with,many,elements,'and',a,list,'of',numbers,"abc",that,
      'end',up,making,the,line,too,long}
```

If you don't want lists of integers to be printed as strings, you can use the `~lp` sequence (insert a lowercase letter L before `p`):

```
> io:format("~lp\n", [[97,98,99]]).
[97,98,99]

> io:format("~lp\n", ["abc"]).
[97,98,99]
```

Read Format Strings online: https://riptutorial.com/erlang/topic/3722/format-strings

# Chapter 10: gen_server behavior

## Remarks

`gen_server` is an important feature of Erlang, and require some prerequisite to understand every aspect of this functionality:

- Loop, recursion and state
- Spawning processes
- Message passing
- OTP principles

A good way to learn more about a feature in Erlang is to directly read the source code from official github repository. `gen_server` behavior embed lot of useful information and interesting structure in its core.

`gen_server` is defined in `gen_server.erl` and its associated documentation can be find in `stdlib` Erlang documentation. `gen_server` is an OTP feature and more information can be also found in OTP Design Principles and User's Guide.

Frank Hebert give you also another good introduction to `gen_server` from its free online book Learn You Some Erlang for great good!

Official documentation for `gen_server` callback:

- `code_change/3`
- `handle_call/3`
- `handle_cast/2`
- `handle_info/2`
- `init/1`
- `terminate/2`

## Examples

### Greeter Service

Here is an example of a service that greets people by the given name, and keeps track of how many users it encountered. See usage below.

```
%% greeter.erl
%% Greets people and counts number of times it did so.
-module(greeter).
-behaviour(gen_server).
%% Export API Functions
-export([start_link/0, greet/1, get_count/0]).
%% Required gen server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

-record(state, {count::integer()}).
```

```
%% Public API
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, {}, []).

greet(Name) ->
    gen_server:cast(?MODULE, {greet, Name}).

get_count() ->
    gen_server:call(?MODULE, {get_count}).

%% Private
init({}) ->
    {ok, #state{count=0}}.

handle_cast({greet, Name}, #state{count = Count} = State) ->
    io:format("Greetings ~s!~n", [Name]),
    {noreply, State#state{count = Count + 1}};

handle_cast(Msg, State) ->
    error_logger:warning_msg("Bad message: ~p~n", [Msg]),
    {noreply, State}.

handle_call({get_count}, _From, State) ->
    {reply, {ok, State#state.count}, State};

handle_call(Request, _From, State) ->
    error_logger:warning_msg("Bad message: ~p~n", [Request]),
    {reply, {error, unknown_call}, State}.

%% Other gen_server callbacks
handle_info(Info, State) ->
    error_logger:warning_msg("Bad message: ~p~n", [Info]),
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Here is a sample usage of this service in the erlang shell:

```
1> c(greeter).
{ok,greeter}
2> greeter:start_link().
{ok,<0.62.0>}
3> greeter:greet("Andriy").
Greetings Andriy!
ok
4> greeter:greet("Mike").
Greetings Mike!
ok
5> greeter:get_count().
{ok,2}
```

## Using gen_server behavior

A `gen_server` is a specific finite state machine working like a server. `gen_server` can handle different type of event:

- synchronous request with `handle_call`
- asynchronous request with `handle_cast`
- other message (not defined in OTP specification) with `handle_info`

Synchronous and asynchronous message are specified in OTP and are simple tagged tuples with any kind of data on it.

A simple `gen_server` is defined like this:

```
-module(simple_gen_server).
-behaviour(gen_server).
-export([start_link/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() ->
    Return = gen_server:start_link({local, ?MODULE}, ?MODULE, [], []),
    io:format("start_link: ~p~n", [Return]),
    Return.

init([]) ->
    State = [],
    Return = {ok, State},
    io:format("init: ~p~n", [State]),
    Return.

handle_call(_Request, _From, State) ->
    Reply = ok,
    Return = {reply, Reply, State},
    io:format("handle_call: ~p~n", [Return]),
    Return.

handle_cast(_Msg, State) ->
    Return = {noreply, State},
    io:format("handle_cast: ~p~n", [Return]),
    Return.

handle_info(_Info, State) ->
    Return = {noreply, State},
    io:format("handle_info: ~p~n", [Return]),
    Return.

terminate(_Reason, _State) ->
    Return = ok,
    io:format("terminate: ~p~n", [Return]),
    ok.

code_change(_OldVsn, State, _Extra) ->
    Return = {ok, State},
    io:format("code_change: ~p~n", [Return]),
    Return.
```

This code is simple: every message received is printed to standard output.

## gen_server behaviour

To define a `gen_server`, you need to explicitly declare it in your source code with `-behaviour(gen_server)`. Note, `behaviour` can be written in US (behavior) or UK (behaviour).

## start_link/0

This function is a simple shortcut to call another function: `gen_server:start_link/3,4`.

## start_link/3,4

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

This function is called when you want to start your server linked to a `supervisor` or another process. `start_link/3,4` can register automatically your process (if you think your process need to be unique) or can simply spawn it like simple process. When called, this function execute `init/1`.

This function can return these define values:

- `{ok,Pid}`
- `ignore`
- `{error,Error}`

## init/1

```
init([]) ->
    State = [],
    {ok, State}.
```

`init/1` is the first executed function when your server will be launched. This one initialize all prerequisite of your application and return state to newly created process.

This function can return only these defined values:

- `{ok,State}`
- `{ok,State,Timeout}`
- `{ok,State,hibernate}`
- `{stop,Reason}`
- `ignore`

`State` variable can be everything, (e.g. list, tuple, proplists, map, record) and remain accessible to all function inside spawned process.

## handle_call/3

```
handle_call(_Request, _From, State) ->
```

```
    Reply = ok,
    {reply, Reply, State}.
```

`gen_server:call/2` execute this callback. The first argument is your message (`_Request`), the second is the origin of the request (`_From`) and the last one is the current state (`State`) of your running gen_server behaviour.

If you want a reply to caller, `handle_call/3` need to return one of these data structure:

- `{reply,Reply,NewState}`
- `{reply,Reply,NewState,Timeout}`
- `{reply,Reply,NewState,hibernate}`

If you want no reply to caller, `handle_call/3` need to return one of these data structure:

- `{noreply,NewState}`
- `{noreply,NewState,Timeout}`
- `{noreply,NewState,hibernate}`

If you want to stop the current execution of your current gen_server, `handle_call/3` need to return one of these data structure:

- `{stop,Reason,Reply,NewState}`
- `{stop,Reason,NewState}`

## handle_cast/2

```
handle_cast(_Msg, State) ->
    {noreply, State}.
```

`gen_server:cast/2` execute this callback. The first argument is your message (`_Msg`), and the second the current state of your running gen_server behaviour.

By default, this function can't data to the caller, so, you have only two choices, continue current execution:

- `{noreply,NewState}`
- `{noreply,NewState,Timeout}`
- `{noreply,NewState,hibernate}`

Or stop your current `gen_server` process:

- `{stop,Reason,NewState}`

## handle_info/2

```
handle_info(_Info, State) ->
    {noreply, State}.
```

`handle_info/2` is executed when non-standard OTP message come from outside world. This one

can't reply and, like `handle_cast/2` can do only 2 actions, continuing current execution:

- `{noreply,NewState}`
- `{noreply,NewState,Timeout}`
- `{noreply,NewState,hibernate}`

Or stop the current running `gen_server` process:

- `{stop,Reason,NewState}`

## terminate/2

```
terminate(_Reason, _State) ->
    ok.
```

`terminate/2` is called when an error occur or when you want to shutdown your `gen_server` process.

## code_change/3

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

`code_change/3` function is called when you want to upgrade your running code.

This function can return only these defined values:

- `{ok, NewState}`
- `{error, Reason}`

# Starting This process

You can compile your code and start `simple_gen_server`:

```
simple_gen_server:start_link().
```

If you want to send message to your server, you can use these functions:

```
% will use handle_call as callback and print:
%   handle_call: mymessage
gen_server:call(simple_gen_server, mymessage).

% will use handle_cast as callback and print:
%   handle_cast: mymessage
gen_server:cast(simple_gen_server, mymessage).

% will use handle_info as callback and print:
%   handle_info: mymessage
erlang:send(whereis(simple_gen_server), mymessage).
```

## Simple Key/Value Database

This source code create a simple key/value store service based on `map` Erlang datastructure. Firstly, we need to define all information concerning our `gen_server`:

```
-module(cache).
-behaviour(gen_server).

% our API
-export([start_link/0]).
-export([get/1, put/2, state/0, delete/1, stop/0]).

% our handlers
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

% Defining our function to start `cache` process:

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

%%%%%%%%%%%%%%
% API

% Key/Value database is a simple store, value indexed by an unique key.
% This implementation is based on map, this datastructure is like hash
# in Perl or dictionaries in Python.

% put/2
% put a value indexed by a key. We assume the link is stable
% and the data will be written, so, we use an asynchronous call with
% gen_server:cast/2.

put(Key, Value) ->
    gen_server:cast(?MODULE, {put, {Key, Value}}).

% get/1
% take one argument, a key and will a return the value indexed
% by this same key. We use a synchronous call with gen_server:call/2.

get(Key) ->
    gen_server:call(?MODULE, {get, Key}).

% delete/1
% like `put/1`, we assume the data will be removed. So, we use an
% asynchronous call with gen_server:cast/2.

delete(Key) ->
    gen_server:cast(?MODULE, {delete, Key}).

% state/0
% This function will return the current state (here the map who contain all
% indexed values), we need a synchronous call.

state() ->
    gen_server:call(?MODULE, {get_state}).

% stop/0
% This function stop cache server process.
```

```
stop() ->
    gen_server:stop(?MODULE).

%%%%%%%%%%%%%%%
% Handlers

% init/1
% Here init/1 will initialize state with simple empty map datastructure.

init([]) ->
    {ok, #{}}.

% handle_call/3
% Now, we need to define our handle. In a cache server we need to get our
% value from a key, this feature need to be synchronous, so, using
% handle_call seems a good choice:

handle_call({get, Key}, From, State) ->
    Response = maps:get(Key, State, undefined),
    {reply, Response, State};

% We need to check our current state, like get_fea

handle_call({get_state}, From, State) ->
    Response = {current_state, State},
    {reply, Response, State};

% All other messages will be dropped here.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

% handle_cast/2
% put/2 will execute this function.

handle_cast({put, {Key, Value}}, State) ->
    NewState = maps:put(Key, Value, State),
    {noreply, NewState};

% delete/1 will execute this function.

handle_cast({delete, Key}, State) ->
    NewState = maps:remove(Key, State),
    {noreply, NewState};

% All other messages are dropped here.

handle_cast(_Msg, State) ->
    {noreply, State}.

%%%%%%%%%%%%%%%
% other handlers

% We don't need other features, other handlers do nothing.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
```

```
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

## Using our cache server

We can now compile our code and start using it with `erl`.

```
% compile cache
c(cache).

% starting cache server
cache:start_link().

% get current store
% will return:
%   #{}
cache:state().

% put some data
cache:put(1, one).
cache:put(hello, bonjour).
cache:put(list, []).

% get current store
% will return:
%   #{1 => one,hello => bonjour,list => []}
cache:state().

% delete a value
cache:delete(1).
cache:state().
%   #{1 => one,hello => bonjour,list => []}

% stopping cache server
cache:stop().
```

Read gen_server behavior online: https://riptutorial.com/erlang/topic/7481/gen-server-behavior

# Chapter 11: Installation

## Examples

### Build and Install Erlang/OTP on Ubuntu

The following examples show two main methods for installing Erlang/OTP on Ubuntu.

# Method 1 - Pre-built Binary Package

Simply run this command and it will download and install the latest stable Erlang release from Erlang Solutions.

```
$ sudo apt-get install erlang
```

# Method 2 - Build and Install from Source

Download the tar file:

```
$ wget http://erlang.org/download/otp_src_19.0.tar.gz
```

Extract the tar file:

```
$ tar -zxf otp_src_19.0.tar.gz
```

Enter the extracted directory and set `ERL_TOP` to be the current path:

```
$ cd otp_src_19.0
$ export ERL_TOP=`pwd`
```

Now before configuring the build, you want to make sure you have all the dependencies you need in order to install Erlang:

**Basic dependencies:**

```
$ sudo apt-get install autoconf libncurses-dev build-essential
```

**Other applications dependencies**

| Application | Dependency install |
| --- | --- |
| HiPE | `$ sudo apt-get install m4` |

| Application | Dependency install |
|---|---|
| ODBC | `$ sudo apt-get install unixodbc-dev` |
| OpenSSL | `$ sudo apt-get install libssl-dev` |
| wxWidgets | `$ sudo apt-get install libwxgtk3.0-dev libglu-dev` |
| Documentation | `$ sudo apt-get install fop xsltproc` |
| Orber and other C++ projects | `$ sudo apt-get install g++` |
| jinterface | `$ sudo apt-get install default-jdk` |

Configure and build:
You can set your own options, or leave it blank to run the default configuration. Advanced configuration and build for Erlang/OTP.

```
$ ./configure [ options ]
$ make
```

Testing the build:

```
$ make release_tests
$ cd release/tests/test_server
$ $ERL_TOP/bin/erl -s ts install -s ts smoke_test batch -s init stop
```

After running these commands, open `$ERL_TOP/release/tests/test_server/index.html` with your web browser and check that you don't have any fails. If all tests passed we're ok to continue to the installation.

Installing:

```
$ make install
```

### Build and Install Erlang/OTP on FreeBSD

The following examples show 3 main methods for installing Erlang/OTP on FreeBSD.

## Method 1 - Pre-built Binary Package

Use pkg to install the pre-built binary package:

```
$ pkg install erlang
```

Test your new installation:

```
$ erl
Erlang/OTP 18 [erts-7.3.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.3.1  (abort with ^G)
```

# Method 2 - Build and install using the port collection (recommended)

Build and install the port as usual:

```
$ make -C /usr/ports/lang/erlang install clean
```

Test your new installation:

```
$ erl
Erlang/OTP 18 [erts-7.3.1] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.3.1  (abort with ^G)
```

This will fetch the release tarball from the official website, apply a few patches if needed, build the release and install it. Obviously, it will take some time.

# Method 3 - Build and install from the release tarball

Note: building the release manually works, but using the two above methods should be preferred, since the port collection embeds patches that render the release more FreeBSD friendly.

Download the release file:

```
$ fetch 'http://erlang.org/download/otp_src_18.3.tar.gz'
```

Check that its MD5 sum is correct:

```
$ fetch 'http://erlang.org/download/MD5'
MD5                                           100% of   24 kB  266 kBps 00m00s

$ grep otp_src_18.3.tar.gz MD5
MD5(otp_src_18.3.tar.gz)= 7e4ff32f97c36fb3dab736f8d481830b

$ md5 otp_src_18.3.tar.gz
MD5 (otp_src_18.3.tar.gz) = 7e4ff32f97c36fb3dab736f8d481830b
```

Extract the tarball:

```
$ tar xzf otp_src_18.3.tar.gz
```

Configure:

```
$ ./configure --disable-hipe
```

If you want to build Erlang with HiPe, you will need to apply the patches from the port collection.

Build:

```
$ gmake
```

Install:

```
$ gmake install
```

Test your new installation:

```
$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:2:2] [async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
```

# Build and install using kerl

kerl is a tool that helps you build and install Erlang/OTP releases.

Install curl:

```
$ make -C /usr/ports/ftp/curl install clean
```

Download kerl:

```
$ fetch 'https://raw.githubusercontent.com/kerl/kerl/master/kerl'
$ chmod +x kerl
```

Update the list of available releases:

```
$ ./kerl update releases
The available releases are:
R10B-0 R10B-10 R10B-1a R10B-2 R10B-3 R10B-4 R10B-5 R10B-6 R10B-7 R10B-8 R10B-9 R11B-0 R11B-1
R11B-2 R11B-3 R11B-4 R11B-5 R12B-0 R12B-1 R12B-2 R12B-3 R12B-4 R12B-5 R13A R13B01 R13B02-1
R13B02 R13B03 R13B04 R13B R14A R14B01 R14B02 R14B03 R14B04 R14B R14B_erts-5.8.1.1 R15B01
R15B02 R15B02_with_MSVCR100_installer_fix R15B03-1 R15B03 R15B R16A_RELEASE_CANDIDATE R16B01
R16B02 R16B03-1 R16B03 R16B 17.0-rc1 17.0-rc2 17.0 17.1 17.3 17.4 17.5 18.0 18.1 18.2 18.2.1
18.3 19.0
```

Build the required release:

```
$ ./kerl build 18.3 erlang-18.3
```

Check that the build is present in the build list:

```
$ ./kerl list builds
18.3,erlang-18.3
```

Install the build somewhere:

```
$ ./kerl install erlang-18.3 ./erlang-18.3
```

Source the `activate` file if you're running bash or the fish shell. If you're running a cshell, add the build bin directory to the PATH:

```
$ setenv PATH "/some/where/erlang-18.3/bin/:$PATH"
```

Test your new installation:

```
$ which erl
/some/where/erlang-18.3/bin//erl

$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:2:2] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V7.3  (abort with ^G)
```

# Other releases

If you want to build another version of Erlang/OTP, look for the other ports in the collection:

- lang/erlang-runtime15
- lang/erlang-runtime16
- lang/erlang-runtime17
- lang/erlang-runtime18

# Reference

- FreeBSD Handbook -> Chapter 4. Installing Applications: Packages and Ports
- Erlang on FreshPorts
- Kerl documentation on GitHub

### Build and Install Erlang/OTP on OpenBSD

Erlang on OpenBSD is currently broken on `alpha`, `sparc` and `hppa` architectures.

# Method 1 - Pre-built Binary Package

OpenBSD let you choose desired version you want to install on your system:

```
##############################################################################
# free-choice:
##############################################################################
$ pkg_add erlang
# a       0: <None>
#   1: erlang-16b.03p10v0
#   2: erlang-17.5p6v0
#   3: erlang-18.1p1v0
#   4: erlang-19.0v0

##############################################################################
# manual-choice:
##############################################################################
pkg_add erlang%${version}
# example:
pkg_add erlang%19
```

OpenBSD can support multiple version of Erlang. To make thinks easier to use, each binaries are installed Erlang version in its name. So, if you have installed `erlang-19.0v0`, your `erl` binary will be `erl19`.

If you want to use `erl`, you can create a symlink:

```
ln -s /usr/local/bin/erl19 /usr/local/bin/erl
```

or create an alias in your shell configuration file or in `.profile` file:

```
echo 'alias erl="erl19"' >> ~/.profile
# or
echo 'alias erl="erl19"' >> ~/.shrc
```

You can now run `erl`:

```
erl19
# or if you have an alias or symlink
erl
# Erlang/OTP 19 [erts-8.0] [source] [async-threads:10] [kernel-poll:false]
#
# Eshell V8.0  (abort with ^G)
```

# Method 2 - Build and install using ports

```
RELEASE=OPENBSD_$(uname -r  | sed 's/\./_/g')
cd /usr
cvs -qz3 -danoncvs@anoncvs.openbsd.org:/cvs co -r${RELEASE}
cd /usr/ports/lang/erlang
```

```
ls -p
# 16/ 17/ 18/ 19/  CVS/  Makefile  Makefile.inc  erlang.port.mk
cd 19
make && make install
```

# Method 3 - Build from source

Build from source require additional packages:

- `git`
- `gmake`
- `autoconf-2.59`

```
pkg_add git gmake autoconf%2.59
git clone https://github.com/erlang/otp.git
cd otp
AUTOCONF_VERSION="2.59" ./build_build all
```

# References

- http://openports.se/lang/erlang
- http://cvsweb.openbsd.org/cgi-bin/cvsweb/ports/lang/erlang/
- https://www.openbsd.org/faq/faq15.html
- http://man.openbsd.org/OpenBSD-current/man1/pkg_add.1

Read Installation online: https://riptutorial.com/erlang/topic/4483/installation

# Chapter 12: iolists

## Introduction

While an Erlang string is a list of integers, an "iolist" is a list whose elements are either integers, binaries or other iolists, e.g. `["foo", $b, $a, $r, <<"baz">>]`. That iolist represents the string `"foobarbaz"`.

While you can convert an iolist to a binary with `iolist_to_binary/1`, you often don't need to, since Erlang library functions such as `file:write_file/2` and `gen_tcp:send/2` accept iolists as well as strings and binaries.

## Syntax

- -type iolist() :: maybe_improper_list(byte() | binary() | iolist(), binary() | []).

## Remarks

### What is an iolist?

*It's any binary. Or any list containing integers between 0 and 255. Or any arbitrarily nested list containing either of those two things.*

#### Original article

Use deeply nested lists of integers and binaries to represent IO data to avoid copying when concatenating strings or binaries.

They are efficient even when combining large amounts of data. For example combining two fifty kilobyte binaries using binary syntax `<<B1/binary, B2/binary>>` would typically require reallocating both into a new 100kb binary. Using IO lists `[B1, B2]` only allocates the list, in this case three words. A list uses one word and another word per element, see here for more information.

Using the `++` operator would have created a whole new list, instead of just a new two element list. Recreating lists to add elements to the end can become expensive when the list is long.

In cases where the binary data is small, allocating IO lists can be greater than appending the binaries. If the binary data can either be small or large it is often better to accept the consistent cost of IO lists.

Note that appending binaries is optimised as described here. In short, a binary can have extra, hidden space allocated. This will be filled if another binary is appended to it that fits in the free space. This means that not every binary append will cause a full copy of both binaries.

## Examples

### IO lists are typically used to build output to a port e.g. a file or network socket.

```
file:write_file("myfile.txt", ["Hi " [<<"there">>], $\n]).
```

### Add the allowed data types to the front of an IO list, creating a new one.

```
["Guten Tag " | [<<"Hello">>]].
[<<"Guten Tag ">> | [<<"Hello">>]].
[$G, $u, $t, $e, $n , $T, $a, $g | [<<"Hello">>]].
[71,117,116,101,110,84,97,103,<<"Hello">>].
```

### IO data can be efficiently added to the end of a list.

```
Data_1 = [<<"Hello">>].
Data_2 = [Data_1,<<" Guten Tag ">>].
```

### Be careful with improper lists

```
["Guten tag " | <<"Hello">>].
```

In the shell this will be printed as `["Guten tag "|<<"Hello">>]` instead of `["Guten tag ",<<"Hello">>]`. The pipe operator will create an improper list if the last element on the right is not a list. While an improper list whose "tail" is a binary is still a valid iolist, improper lists can cause issues because many recursive functions expect an empty list to be the last element, and not, as in this case a binary.

### Get IO list size

```
Data = ["Guten tag ",<<"Hello">>],
Len = iolist_size(Data),
[<<Len:32>> | Data].
```

The size of an iolist can be calculated using the `iolist_size/1`. This snippet calculates the size of a message and creates and appends it to the front as a four byte binary. This is a typical operation in messaging protocols.

### IO list can be converted to a binary

```
<<"Guten tag, Hello">> = iolist_to_binary(["Guten tag, ",<<"Hello">>]).
```

An IO list can be converted to a binary using the `iolist_to_binary/1` function. If the data is going to be stored for a long period or sent as a message to other processes then it may make sense to convert it to a binary. The one off cost of converting to a binary can be cheaper than copying the IO list many times, in garbage collection of a single process or in message passing to others.

Read iolists online: https://riptutorial.com/erlang/topic/5677/iolists

# Chapter 13: Loop and Recursion

## Syntax

- function ( list | iolist | tuple ) -> function( tail ).

## Remarks

# Why recursive functions?

Erlang is a functional programming language and don't any kind of loop structure. Everything in functional programming is based on data, type and functions. If you want a loop, you need to create a function who call itself.

Traditional `while` or `for` loop in imperative and object oriented language can be represented like that in Erlang:

```
loop() ->
  % do something here
  loop().
```

Good method to understand this concept is to expand all function calls. We'll see that on other examples.

## Examples

### List

Here the simplest recursive function over list type. This function only navigate into a list from its start to its end and do nothing more.

```
-spec loop(list()) -> ok.
loop([]) ->
  ok;
loop([H|T]) ->
  loop(T).
```

You can call it like this:

```
loop([1,2,3]). % will return ok.
```

Here the recursive function expansion:

```
loop([1|2,3]) ->
```

```
  loop([2|3]) ->
    loop([3|]) ->
      loop([]) ->
        ok.
```

# Recursive loop with IO actions

Previous code do nothing, and is pretty useless. So, we will create now a recursive function who execute some actions. This code is similar to `lists:foreach/2`.

```
-spec loop(list(), fun()) -> ok.
loop([], _) ->
  ok;
loop([H|T], Fun)
  when is_function(Fun) ->
    Fun(H),
    loop(T, Fun).
```

You can call it in like this:

```
Fun = fun(X) -> io:format("~p", [X]) end.
loop([1,2,3]).
```

Here the recursive function expansion:

```
loop([1|2,3], Fun(1)) ->
  loop([2|3], Fun(2)) ->
    loop([3|], Fun(3)) ->
      loop([], _) ->
        ok.
```

You can compare with `lists:foreach/2` output:

```
lists:foreach(Fun, [1,2,3]).
```

# Recursive loop over list returning modified list

Another useful example, similar to `lists:map/2`. This function will take one list and one anonymous function. Each time a value in list is matched, we apply function on it.

```
-spec loop(A :: list(), fun()) -> list().
loop(List, Fun)
  when is_list(List), is_function(Fun) ->
    loop(List, Fun, []).

-spec loop(list(), fun(), list()) -> list() | {error, list()}.
```

```
loop([], _, Buffer)
  when is_list(Buffer) ->
    lists:reverse(Buffer);
loop([H|T], Fun, Buffer)
  when is_function(Fun), is_list(Buffer) ->
    BufferReturn = [Fun(H)] ++ Buffer,
    loop(T, Fun, BufferReturn).
```

You can call it like this:

```
Fun(X) -> X+1 end.
loop([1,2,3], Fun).
```

Here the recursive function expansion:

```
loop([1|2,3], Fun(1), [2]) ->
  loop([2|3], Fun(2), [3,2]) ->
    loop([3|], Fun(3), [4,3,2]) ->
      loop([], _, [4,3,2]) ->
        list:reverse([4,3,2]) ->
          [2,3,4].
```

This function is also called "tail recursive function", because we use a variable like an accumulator to pass modified data over multiple execution context.

### Iolist and Bitstring

Like list, simplest function over iolist and bitstring is:

```
-spec loop(iolist()) -> ok | {ok, iolist} .
loop(<<>>) ->
  ok;
loop(<<Head, Tail/bitstring>>) ->
  loop(Tail);
loop(<<Rest/bitstring>>) ->
  {ok, Rest}
```

You can call it like this:

```
loop(<<"abc">>).
```

Here the recursive function expansion:

```
loop(<<"a"/bitstring, "bc"/bitstring>>) ->
  loop(<<"b"/bitstring, "c"/bitstring>>) ->
    loop(<<"c"/bitstring>>) ->
      loop(<<>>) ->
        ok.
```

## Recursive function over variable binary size

This code take bitstring and dynamically define binary size of it. So if, if we set a size of `4`, every `4` bits, a data will be matched. This loop do nothing interesting, its just our pillar.

```
loop(Bitstring, Size)
  when is_bitstring(Bitstring), is_integer(Size) ->
    case Bitstring of
      <<>> ->
        ok;
      <<Head:Size/bitstring,Tail/bitstring>> ->
        loop(Tail, Size);
      <<Rest/bitstring>> ->
        {ok, Rest}
    end.
```

You can call it like this:

```
loop(<<"abc">>, 4).
```

Here the recursive function expansion:

```
loop(<<6:4/bitstring, 22, 38, 3:4>>, 4) ->
  loop(<<1:4/bitstring, "bc">>, 4) ->
    loop(<<6:4/bitstring, 38,3:4>>, 4) ->
      loop(<<2:4/bitstring, "c">>, 4) ->
        loop(<<6:4/bitstring, 3:4>>, 4) ->
          loop(<<3:4/bitstring>>, 4) ->
            loop(<<>>, 4) ->
              ok.
```

Our bitstring is splitted over 7 patterns. Why? Because by default, Erlang use binary size of `8` bits, if we split it in two, we have `4` bits. Our string is `8*3=24` bits. `24/4=6` patterns. Last pattern is `<<>>`. `loop/2` function is called 7 times.

# recursive function over variable binary size with actions

Now, we can do more interesting thing. This function take one more argument, an anonymous function. Everytime we match a pattern, this one will be passed to it.

```
-spec loop(iolist(), integer(), function()) -> ok.
loop(Bitstring, Size, Fun) ->
  when is_bitstring(Bitstring), is_integer(Size), is_function(Fun) ->
        case Bitstring of
      <<>> ->
        ok;
      <<Head:Size/bitstring,Tail/bitstring>> ->
        Fun(Head),
        loop(Tail, Size, Fun);
      <<Rest/bitstring>> ->
        Fun(Rest),
        {ok, Rest}
```

```
    end.
```

You can call it like this:

```
Fun = fun(X) -> io:format("~p~n", [X]) end.
loop(<<"abc">>, 4, Fun).
```

Here the recursive function expansion:

```
loop(<<6:4/bitstring, 22, 38, 3:4>>, 4, Fun(<<6:4>>) ->
  loop(<<1:4/bitstring, "bc">>, 4, Fun(<<1:4>>)) ->
    loop(<<6:4/bitstring, 38,3:4>>, 4, Fun(<<6:4>>)) ->
      loop(<<2:4/bitstring, "c">>, 4, Fun(<<2:4>>)) ->
        loop(<<6:4/bitstring, 3:4>>, 4, Fun(<<6:4>>) ->
          loop(<<3:4/bitstring>>, 4, Fun(<<3:4>>) ->
            loop(<<>>, 4) ->
              ok.
```

# Recursive function over bitstring returning modified bitstring

This one is similar to `lists:map/2` but for bitstring and iolist.

```
% public function (interface).
-spec loop(iolist(), fun()) -> iolist() | {iolist(), iolist()}.
loop(Bitstring, Fun) ->
  loop(Bitstring, 8, Fun).

% public function (interface).
-spec loop(iolist(), integer(), fun()) -> iolist() | {iolist(), iolist()}.
loop(Bitstring, Size, Fun) ->
  loop(Bitstring, Size, Fun, <<>>)

% private function.
-spec loop(iolist(), integer(), fun(), iolist()) -> iolist() | {iolist(), iolist()}.
loop(<<>>, _, _, Buffer) ->
  Buffer;
loop(Bitstring, Size, Fun, Buffer) ->
  when is_bitstring(Bitstring), is_integer(Size), is_function(Fun) ->
    case Bitstring of
      <<>> ->
        Buffer;
      <<Head:Size/bitstring,Tail/bitstring>> ->
        Data = Fun(Head),
        BufferReturn = <<Buffer/bitstring, Data/bitstring>>,
        loop(Tail, Size, Fun, BufferReturn);
      <<Rest/bitstring>> ->
        {Buffer, Rest}
    end.
```

This code seems more complexe. Two functions were added: `loop/2` and `loop/3`. These two functions are simple interface to `loop/4`.

You can execute it like this:

```
Fun = fun(<<X>>) -> << (X+1) >> end.
loop(<<"abc">>, Fun).
% will return <<"bcd">>

Fun = fun(<<X:4>>) -> << (X+1) >> end.
loop(<<"abc">>, 4, Fun).
% will return <<7,2,7,3,7,4>>

loop(<<"abc">>, 4, Fun, <<>>).
% will return <<7,2,7,3,7,4>>
```

## Map

Map in Erlang is equivalent of hashes in Perl or dictionaries in Python, its a key/value store. To list every value stored in, you can list every key, and return key/value pair. This first loop give you an idea:

```
loop(Map) when is_map(Map) ->
  Keys = maps:keys(Map),
  loop(Map, Keys).

loop(_ , []) ->
  ok;
loop(Map, [Head|Tail]) ->
  Value = maps:get(Head, Map),
  io:format("~p: ~p~n", [Head, Value]),
  loop(Map, Tail).
```

You can execute it like that:

```
Map = #{1 => "one", 2 => "two", 3 => "three"}.
loop(Map).
% will return:
% 1: "one"
% 2: "two"
% 3: "three"
```

## Managing State

Recursive function use their states to loop. When you spawn new process, this process will be simply a loop with some defined state.

## Anonymous function

Here 2 examples of recursive anonymous functions based on previous example. Firstly, simple infinite loop:

```
InfiniteLoop = fun
  R() ->
    R() end.
```

Secondly, anonymous function doing loop over list:

```
LoopOverList = fun
  R([]) -> ok;
  R([H|T]) ->
    R(T) end.
```

These two functions could be rewritten as:

```
InfiniteLoop = fun loop/0.
```

In this case, `loop/0` is a reference to `loop/0` from remarks. Secondly, with little more complex:

```
LoopOverLlist = fun loop/2.
```

Here, `loop/2` is a reference to `loop/2` from list example. These two notations are syntactic sugar.

Read Loop and Recursion online: https://riptutorial.com/erlang/topic/10720/loop-and-recursion

# Chapter 14: NIFs

## Examples

### Definition

Official documentation: http://erlang.org/doc/tutorial/nif.html

NIFs were introduced in Erlang/OTP R13B03 as an experimental feature. The purpose is to allow calling C-code from inside Erlang code.

NIFs are implemented in C instead of Erlang, but they appear as any other functions in the scope of Erlang code as they belong to the module where the include happened. NIF libraries are linked on compilation and loaded in runtime.

Because NIF libraries are dynamically linked into the emulator process, they are fast, but also dangerous, because crashing in a NIF brings the emulator down too.

### Example: current UNIX time

Here is a very simple example to illustrate how to write a NIF.

Directory structure:

```
nif_test
├── c_src
│   ├── Makefile
│   └── nif_test.c
├── rebar.config
└── src
    ├── nif_test.app.src
    └── nif_test.erl
```

This structure can be easily initialized using Rebar3:

```
$ rebar3 new lib nif_test && cd nif_test && rebar3 new cmake
```

Contents of `nif_test.c`:

```
#include "erl_nif.h"
#include "time.h"

static ERL_NIF_TERM now(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
        return enif_make_int(env, time(0));
}

static ErlNifFunc nif_funcs[] = {
        {"now", 0, now}
};
```

```
ERL_NIF_INIT(nif_test,nif_funcs,NULL,NULL,NULL,NULL);
```

Contents of `nif_test.erl`:

```
-module(nif_test).
-on_load(init/0).
-export([now/0]).

-define(APPNAME, nif_test).
-define(LIBNAME, nif_test).

%%======================================================================
%% API functions
%%======================================================================

now() -> nif_not_loaded.

%%======================================================================
%% Internal functions
%%======================================================================

init() ->
    SoName = case code:priv_dir(?APPNAME) of
        {error, bad_name} ->
            case filelib:is_dir(filename:join(["..", priv])) of
                true -> filename:join(["..", priv, ?LIBNAME]);
                _ -> filename:join([priv, ?LIBNAME])
            end;
        Dir -> filename:join(Dir, ?LIBNAME)
    end,
    erlang:load_nif(SoName, 0).
```

Contents of `rebar.config`:

```
{erl_opts, [debug_info]}.
{deps, []}.

{pre_hooks, [
  {"(linux|darwin|solaris)", compile, "make -C c_src"},
  {"(freebsd)", compile, "gmake -C c_src"}
]}.
{post_hooks, [
  {"(linux|darwin|solaris)", clean, "make -C c_src clean"},
  {"(freebsd)", clean, "gmake -C c_src clean"}
]}.
```

Now you can run the example:

```
$ rebar3 shell

===> Verifying dependencies...
===> Compiling nif_test
make: Entering directory '/home/vschroeder/Projects/nif_test/c_src'
cc -O3 -std=c99 -finline-functions -Wall -Wmissing-prototypes -fPIC -I
/usr/local/lib/erlang/erts-7.3.1/include/ -I /usr/local/lib/erlang/lib/erl_interface-
3.8.2/include  -c -o /home/vschroeder/Projects/nif_test/c_src/nif_test.o
/home/vschroeder/Projects/nif_test/c_src/nif_test.c
```

```
cc /home/vschroeder/Projects/nif_test/c_src/nif_test.o -shared -L
/usr/local/lib/erlang/lib/erl_interface-3.8.2/lib -lerl_interface -lei -o
/home/vschroeder/Projects/nif_test/c_src/../priv/nif_test.so
make: Leaving directory '/home/vschroeder/Projects/nif_test/c_src'
Erlang/OTP 18 [erts-7.3.1] [source] [64-bit] [smp:4:4] [async-threads:0] [hipe] [kernel-
poll:false]

Eshell V7.3.1  (abort with ^G)
1> nif_test:now().
1469732239
2> nif_test:now().
1469732264
3>
```

## Erlang C API (C to Erlang)

**Official documentation**: http://erlang.org/doc/man/erl_nif.html

The most important structs, types and macros of the Erlang C API are the following:

- `ERL_NIF_TERM`: the type for Erlang terms. This is the return type that NIF functions must follow.
- `ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, reload, upgrade, unload)`: This is the macro that actually creates the NIFs defined in a certain C file. It must be evaluated in the global scope. Normally it will be the last line in the C file.
- `ErlNifFunc`: the type with which each NIF is passed to `ERL_NIF_INIT` to be exported. This struct is composed of name, arity, a poiter to the C function and flags. An array of this type with all NIF definitions should be created to be passed to `ERL_NIF_INIT`.
- `ErlNifEnv`: the Erlang environment where the NIF is being executed. It's mandatory to pass the environment as the first argument for every NIF. This type is opaque and can only be manipulated using the functions that the Erlang C API offers.

Read NIFs online: https://riptutorial.com/erlang/topic/5274/nifs

# Chapter 15: Processes

## Examples

### Creating Processes

We create a new concurrent process by calling the `spawn` function. The `spawn` function will get as parameter a function `Fun` that the process will evaluate. The return value of the `spawn` function is the created process identifier (pid).

```
1> Fun = fun() -> 2+2 end.
#Fun<erl_eval.20.52032458>
2> Pid = spawn(Fun).
<0.60.0>
```

You can also use `spawn/3` to start a process that will execute a specific function from a module: `spawn(Module, Function, Args).`
Or use `spawn/2` or `spawn/4` similarly to start a process in a different node: `spawn(Node, Fun)` or `spawn(Node, Module, Function, Args).`

### Message Passing

Two erlang processes can communicate with each other, wich is also known as *message passing*. This procedure is *asynchronous* in the form that the sending process will not halt after sending the message.

## Sending Messages

This can be achieved with the construct `Pid ! Message`, where `Pid` is a valid process identifier (pid) and `Message` is a value of any data type.

Each process has a "mailbox" that contains the received messages in the received order. This "mailbox" can be emptied with the build in `flush/0` function.

If a message is send to a non existing process, the message will be discarded without any error!

An example might look like the following, where `self/0` returns the pid of the current process and `pid/3` creates a pid.

```
1> Pidsh = self().
<0.32.0>
2> Pidsh ! hello.
hello
3> flush().
Shell got hello
ok
4> <0.32.0> ! hello.
* 1: syntax error before: '<'
```

```
5> Pidsh2 = pid(0,32,0).
<0.32.0>
6> Pidsh2 ! hello2.
hello2
7> flush().
Shell got hello2
ok
```

It is also possible to send a message to multiple processes at once, with `Pid3!Pid2!Pid1!Msg`.

## Receiving Messages

Received messages can be processed with the `receive` construct.

```
receive
  Pattern1            -> exp11, .., exp1n1;
  Pattern2 when Guard -> exp21, .., exp2n2;
  ...
  Other               -> exp31, .., exp3n3;
  ...
  after Timeout       -> exp41, .., exp4n4
end
```

The `Pattern` will be compared to the messages in the "mailbox" starting with the first and oldest message.
If a pattern matches, the matched message is removed from the "mailbox" and the clause body is evaluated.

It is also possible to define timeouts with the `after` construct.
A `Timeout` is either the waiting time in milliseconds or the atom `infinity`.

The return value of `receive` is the last evaluated clause body.

## Example (Counter)

A (very) simple counter with message passing might look like in the following.

```
-module(counter0).
-export([start/0,loop/1]).

% Creating the counter process.
start() ->
    spawn(counter0, loop, [0]).

% The counter loop.
loop(Val) ->
    receive
        increment ->
           loop(Val + 1)
    end.
```

Interaction with the counter.

```
1> C0 = counter0:start().
<0.39.0>
2> C0!increment.
increment
3> C0!increment.
increment
```

## Register Processes

It is possible to register a process (pid) to a global alias.
This can be achieved with the build in `register(Alias, Pid)` function, where `Alias` is the atom to access the process as and `Pid` is the process id.

The alias will be globally available!
It is very easy to create shared state, wich is usually not preferable. (See also here)

It is possible to unregister a process with `unregister(Pid)` and receive the pid from an alias with `whereis(Alias)`.

Use `registered()` for a list of all registered aliases.

The example registers the Atom foo to the pid of the current process and sends a message using the registered Atom.

```
1> register(foo, self()).
true
2> foo ! 'hello world'.
'hello world'
```

Read Processes online: https://riptutorial.com/erlang/topic/2285/processes

# Chapter 16: Rebar3

## Examples

### Definition

**Official page**: https://www.rebar3.org/

**Source code**: https://github.com/erlang/rebar3

Rebar3 is mainly a dependency manager for Erlang and Elixir projects, but it also offers several other features, like bootstrapping projects (according to several templates, following the OTP principles), task executor, build tool, test runner and is extensible by using plugins.

### Installing Rebar3

Rebar3 is written in Erlang, so you need Erlang to run it. It is available as a binary that you can download and run. Just fetch the nightly build and give it execution permissions:

```
$ wget https://s3.amazonaws.com/rebar3/rebar3 && chmod +x rebar3
```

Place this binary in a convenient place and add it to your path. For example, in a bin directory in your home:

```
$ mkdir ~/bin && mv rebar3 ~/bin
$ export PATH=~/bin:$PATH
```

This last line should be put in your `.bashrc`. As an alternative, one can also link the binary to `/usr/local/bin` directory, making it available as a normal command.

```
$ sudo ln -s /path/to/your/rebar3 /usr/local/bin
```

### Installing from Source Code

As Rebar3 is free, open source and written in Erlang, it's possible to simply clone and build it from the source code.

```
$ git clone https://github.com/erlang/rebar3.git
$ cd rebar3
$ ./bootstrap
```

This will create the rebar3 script, which you can put on your PATH or link to `/usr/local/bin` as explained in the section "Installing Rebar3" above.

### Bootstrapping a new Erlang project

To bootstrap a new Erlang project, simply choose the template you want to use from the list. The available templates can be retrieved by the following command:

```
$ rebar3 new

app (built-in): Complete OTP Application structure
cmake (built-in): Standalone Makefile for building C/C++ in c_src
escript (built-in): Complete escriptized application structure
lib (built-in): Complete OTP Library application (no processes) structure
plugin (built-in): Rebar3 plugin project structure
release (built-in): OTP Release structure for executable programs
```

Once you have chosen the appropriate template, bootstrap it with the following command (rebar3 will create a new directory for your project):

```
$ rebar3 new lib libname

===> Writing libname/src/libname.erl
===> Writing libname/src/libname.app.src
===> Writing libname/rebar.config
===> Writing libname/.gitignore
===> Writing libname/LICENSE
===> Writing libname/README.md
```

OBS: Although you *can* run `rebar3 new <template> .` to create the new project in the current directory, this is not recommended, because the bootstrapped files will use `.` (dot) as application and module names and also in the `rebar.config`, which will cause you syntax problems.

Read Rebar3 online: https://riptutorial.com/erlang/topic/4480/rebar3

# Chapter 17: Supervisors

## Examples

### Basic supervisor with one worker process

This example uses the map format introduced in Erlang/OTP 18.0.

```
%% A module implementing a supervisor usually has a name ending with `_sup`.
-module(my_sup).

-behaviour(supervisor).

%% API exports
-export([start_link/0]).

%% Behaviour exports
-export([init/1]).

start_link() ->
    %% If needed, we can pass an argument to the init callback function.
    Args = [],
    supervisor:start_link({local, ?MODULE}, ?MODULE, Args).

%% The init callback function is called by the 'supervisor' module.
init(_Args) ->
    %% Configuration options common to all children.
    %% If a child process crashes, restart only that one (one_for_one).
    %% If there is more than 1 crash ('intensity') in
    %% 5 seconds ('period'), the entire supervisor crashes
    %% with all its children.
    SupFlags = #{strategy => one_for_one,
                 intensity => 1,
                 period => 5},

    %% Specify a child process, including a start function.
    %% Normally the module my_worker would be a gen_server
    %% or a gen_fsm.
    Child = #{id => my_worker,
              start => {my_worker, start_link, []}},

    %% In this case, there is only one child.
    Children = [Child],

    %% Return the supervisor flags and the child specifications
    %% to the 'supervisor' module.
    {ok, {SupFlags, Children}}.
```

Read Supervisors online: https://riptutorial.com/erlang/topic/6996/supervisors

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with Erlang Language | A. Sarid, CodeWarrior, Community, drozzy, Efraim Weiss, evnu, Gokul, legoscia, Limmen, maze-le, YsenGrimm, ZeWaren |
| 2 | Behaviours | legoscia |
| 3 | Bit Syntax: Defaults | 7stud |
| 4 | Data Types | 7stud, A. Sarid, Ali Sabil, Atomic_alarm, bigo, drozzy, Eddie Antonio Santos, Evgeny Levenets, filmor, gabriel14, Gokul, Gootik, halfelf, legoscia |
| 5 | director | Pouriya |
| 6 | External Term Format | M. Kerjouan |
| 7 | File I/O | CodeDreamer, drozzy, Victor Schröder |
| 8 | Format Strings | drozzy, legoscia |
| 9 | gen_server behavior | drozzy, legoscia, M. Kerjouan, Steve Pallen |
| 10 | Installation | A. Sarid, drozzy, M. Kerjouan, ZeWaren |
| 11 | iolists | Dennis Y. Parygin, legoscia |
| 12 | Loop and Recursion | M. Kerjouan |
| 13 | NIFs | Victor Schröder |
| 14 | Processes | A. Sarid, YsenGrimm |
| 15 | Rebar3 | drozzy, Victor Schröder |
| 16 | Supervisors | legoscia |