

# Introduction to Erlang and the Erlang shell

## Module 8 - Erlang tutorial 1

**Data types**

**Single assignment  
variables**

**Pattern matching**

**The Erlang shell**

**Arithmetic  
expressions**

**Comments**

**Boolean expressions**

**Variable binding**

**Expression sequences**

**Atoms**

**Lists**

**Characters**

**Strings**

**Tuples**

**Operating systems and process oriented programming 2020**

**1DT096**

# Erlang from the far distance (top down)



# Erlang (programming language) (1)

From Wikipedia, the free encyclopedia

**Erlang** is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing. For concurrency it follows the Actor model. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system.<sup>[1]</sup>

Source: [http://en.wikipedia.org/wiki/Erlang\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Erlang_(programming_language))

2012-03-18

## Evaluation strategies

In computer programming, strict evaluation, eager evaluation or greedy evaluation is the evaluation strategy in most traditional programming languages.

In **eager evaluation** an expression is evaluated as soon as it gets bound to a variable.

# Erlang from the distance (top down)



# Small syntax

The Erlang language is small but expressive.

- ★ Small set of built in data types (a.k.a terms)
- ★ Pattern matching
- ★ Variables (but not really)
- ★ Function clauses
- ★ Modules

# Data

# types

# Data types

A small set of built in **immutable** data types.

★ **Number**

★ **Atom**

★ **Bit strings and binaries**

★ **Reference**

★ **Function identifier**

★ **Port identifier**

★ **Pid**

★ **List**

★ **Tuple**

★ **Map**

Among the built in data types we will initially only study and use numbers, atoms, function identifiers, pids, lists and tuples.

# Booleans

There is no special built in boolean data type in Erlang.

In Erlang the boolean values are represented by the two atoms **true** and **false**.

An **atom** is a literal, a constant with name.

# Strings

There is no special built in string data type in Erlang.

In Erlang **strings** are represented by **lists** of integers.

The string “Hello World!” is represented by the following list of ASCII values:

```
[72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
```

There exist syntactic sugar that allows you to write "Hello World!" instead of the list of ASCII values.

# Custom data types

To represent a custom data type you typically use a **tagged tuple**.

To store data about muppets a three element tuple may be used with the first element being the atom muppet, the second element the name (string) of the muppet and the third element the species (string) of the muppet.

```
{muppet, "Kermit", "frog"}
```

```
{muppet, "Fozzie", "bear"}
```

Each tuple is **tagged** with the **atom** muppet. This makes it possible to distinguish these muppet tuples from other three element tuples.

# variables

# Variables

Erlang uses **immutable single assignment** variables.

Variables in Erlangs are not so variable.

Variables can only be bound once.

**Pattern  
matching**

# Pattern matching

Variables are bound to values through the pattern matching mechanism.

In a pattern matching, a left-hand side **pattern** is matched against a right-hand side **term**.

If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs.

`A = 127.`      Variable A is bound to the number 127

`B = A + 3.`      Variable B is bound to the number 130

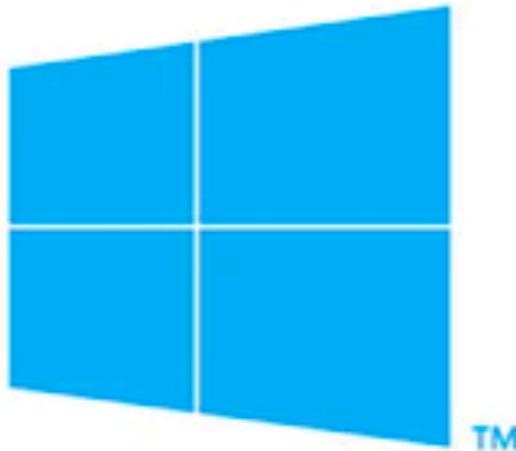
`A = A + 1.`      This is an invalid pattern match (error)





install

Erlang



# Windows®

**<https://www.erlang.org/downloads>**

At the top of the page you find Windows downloads.

On Windows Erlang is distributed as an exe file:

- [OTP 22.3 Windows 32-bit Binary File \(92703609\)](#)
- [OTP 22.3 Windows 64-bit Binary File \(95003088\)](#)



## Check if Erlang is installed on Linux or macOS

From the terminal, check if Erlang is already installed by typing **erl** at the shell prompt and press enter.

```
$> erl
```

If Erlang is not installed, you will see something similar to this:

```
$> erl  
command not found: erl  
$>
```



## Check if Erlang is installed on Linux or macOS

From the terminal, check if Erlang is already installed by typing `erl` at the shell prompt and press enter.

```
$> erl
```

If Erlang is installed, you will see something similar to this:

```
$> erl
Erlang/OTP 22 [erts-10.7] [source] [64-bit] [smp:12:12] [d
s:12:12:10] [async-threads:1] [hipe] [dtrace]

Eshel V10.7 (abort with ^G)
1>
```

<https://www.erlang.org/downloads>

At the bottom of the page you find instructions for how to install Erlang on Linux and macOS.

## Install Erlang on Linux

- For [Ubuntu](#) and [Debian](#): apt-get install erlang
- For [Fedora](#): yum install erlang
- For [FreeBSD](#): pkg install erlang



## Install Erlang on macOS

- For [Homebrew](#) on OS X: brew install erlang
- For [MacPorts](#) on OS X: port install erlang

# Erlang - a closer look (bottom up)



# **The Erlang shell**

# Try yourself

This is a short introduction to Erlang and the Erlang interactive shell.

You are strongly recommended to type in the examples given in this introduction yourself to the Erlang shell.



You may use SSH to log in to the department Linux system to start the Erlang shell.

If you prefer, you can install Erlang on your private computer and then start the Erlang shell.

# The Erlang shell

A good way starting to learn Erlang is by playing around in the interactive Erlang emulator, the Erlang shell.

To start the shell, open a terminal and then type in `erl` and press enter.

```
$> erl
```

Here `$>` is used to denote the **Linux shell prompt**. You may see a different prompt depending on the system you are using.

If you have Erlang installed, the Erlang shell will start.

```
$> erl
Erlang R16B (erts-5.10.1) [source] [smp:8:8]
[async-threads:10] [hipe] [kernel-
poll:false]
```

```
Eshell v5.10.1 (abort with ^G)
1>
```



This is the **Erlang shell prompt**

# **Arithmetic expressions**

# Arithmetic expression

An arithmetic expression is an expression that results in a numeric value.

At the Erlang shell prompt we can enter and evaluate arithmetic expressions.

```
1> 4+1 .
```

**Note:** every expression must end with a period.

```
1> 4+1.
```

If we press enter the shell will respond with the result of evaluating the expression.

```
1> 4+1.
```

```
5
```

```
2>
```

The result of adding 4 and 1, 5, is printed on the next line.

**Note:** the prompt changed from **1>** to **2>**.

The prompt counts the number of evaluated expressions!

Lets try some more **arithmetic expressions**.

```
1> 4+1.  
5  
2> 4-1.  
3  
3> 4*2.  
8  
4> 4/2.  
2.0  
5> 4/3.  
1.333333333333333  
6> 17 rem 5.  
2
```

**Note:** an expression must end with a period.

X rem Y gives the integer remainder of X/Y.

```
1> 4+99 - 7*18
```

If we forget the ending period and press enter ...

```
1>
```

... the Erlang shell responds with the same prompt again, here the prompt is still **1>**. We can now provide the missing period and press enter.

```
1> .  
-23  
2>
```

Finally the value of the expression is printed and we get a new prompt **2>**.

# Comments

A comment begins with the character %, continues up to, but does not include the next end-of-line, and has no effect.

# Boolean expressions

A boolean expression is either **true** or **false**.

op	Description
<code>==</code>	Equal to
<code>/=</code>	Not equal to
<code>=&lt;</code>	Less than or equal to
<code>&lt;</code>	Less than
<code>&gt;=</code>	Greater than or equal to
<code>&gt;</code>	Greater than
<code>=:=</code>	Exactly equal to
<code>=/=</code>	Exactly not equal to

```
1> 1 == 1.      %% equal
true
2> 1 == 1.0.
true
3> 1 == 2.
false
4> 1 := 1.      %% exactly equal
true
5> 1 := 1.0.
false
6> 1 := 2.
False
7> 1 /= 1.      %% not equal
false
8> 1 /= 1.0.
false
9> 1 /= 2.
true
10> 1 /= 1.     %% exactly not equal
false
11> 1 /= 1.0.
true
12> 1 /= 2.
true
13>
```

```
14> 1 < 2.      %% less than
true
15> 1 > 2.      %% greater than
false
16> 2 <= 1.     %% less than or equal
false
17> 1 <= 2.
true
18> 1 >= 2.     %% greater than or equal
false
19> 2 >= 2.
true
20> 3 >= 2.
true
21>
```

**variable**

**binding**

# Pattern matching

Variables are bound to values through the pattern matching mechanism.

In a pattern matching, a left-hand side **pattern** is matched against a right-hand side **term**.

If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs.

A = 127.      Variable A is bound to the number 127

B = A + 3.      Variable B is bound to the number 130

A = A + 1.      This is an invalid pattern match (error)

# Binding values to named variables

```
7> A = 39.  
39  
8> A+3.  
42  
9> A*A.  
1521  
10> Beta = A*A - 99.  
1422  
11> Beta.  
1422
```

Variable names must begin with an upper case letter.

The equal sign is used to bind a value to a variable.

1

2

```
7> A = 39.
```

```
39
```

```
8>
```

1

The variable binding is also an expression.

All expression must have a value.

2

On success, the value of the variable binding expression is the value of the right hand side.

# What if the left hand side already is bound?

```
13> A.
```

```
39
```

```
14> Beta.
```

```
1422
```

```
15>
```

```
16> A = A.
```

```
39
```

```
17> Beta = Beta.
```

```
1422
```

```
18>
```

```
12> A = Beta.
```

Lets try this. What do you think will happen?

```
12> A = Beta.
```

```
** exception error: no match of  
right hand side value 1422
```

**Unbound  
variables**

```
18> x.
```

Lets try this. What do you think will happen?

```
18> x.
```

```
* 1: variable 'x' is unbound
```

We haven't bound the name X to any value yet!

As before, we can bind the name X to a value.

```
19> x = Beta + 1.
```

```
1423
```

**variables  
and pattern  
matching**

# Variables and pattern matching

```
19> X = Beta + 1.
```

```
1423
```

X and Beta are called variables in Erlang.

- A variable is an expression.
- If a variable is bound to a value, this value is the value of the expression.
- Unbound variables are only allowed in something called patterns (more about patterns later).
- In the above pattern match expression, X is unbound and on the left-hand side of the match operator =. The result of the pattern match expression is for the name X to be bound to the value of the right-hand side.

# Variables and single assignment

Variables must start with an uppercase letter or underscore \_ and may contain alphanumeric characters, underscore and @.

Erlang uses **single assignment**, a variable can only be bound once.

Examples of valid variable names

x

Name1

PhoneNumber

Phone\_number

\_Height

# **expression**

# **sequence**

# Sequences of expressions

You can sequence expression by separating them by commas (,).

```
10> A = 33, B = 2*A.  
66
```

The value of a sequence of expressions is the value of the last expression, in the above example the value of B (66).

```
11> A.  
33  
12> B.  
66
```

Both A and B where bound to values when evaluating the expression sequence.

# Atoms

# Atoms

An atom is a literal, a constant with a name.

An atom should be enclosed in single quotes ('') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (\_), or @.

**Examples of valid atoms**

hello  
phone\_number  
'Monday'  
'phone number'

# Atoms and pattern matching

```
20> cat = dog.  
** exception error: no match of  
right hand side value dog  
21> cat = 42.  
** exception error: no match of  
right hand side value 42
```

An atom can only match itself.

```
22> cat = cat.  
cat
```

Using pattern matching, an unbound variable can be bound to an atom.

```
23> Pet = cat.
```

```
cat
```

```
24> Food = fish.
```

```
fish
```

The following pattern match fails.

```
25> Pet = Food.
```

```
** exception error: no match of  
right hand side value fish
```

The following pattern matches are valid.

```
26> fish = Food.
```

```
fish
```

```
27> Food = fish.
```

```
fish
```

```
28> cat = Pet.
```

```
cat
```

```
29> Pet = cat.
```

```
cat
```

# Lists

# Lists

Compound data type with a **variable number of terms**. The elements of a list can be any valid Erlang term.

[Term<sub>1</sub>, . . . , Term<sub>N</sub>]

Examples of lists.

```
[]  
[1,2]  
[1,3,fish, 4, cat]  
[[1,2], a, [b,c], 77, z]  
[[1,2,[88,cat]],[44,88],0]
```

A list can contain other lists ... which also contains lists ...

# Anatomy of a list

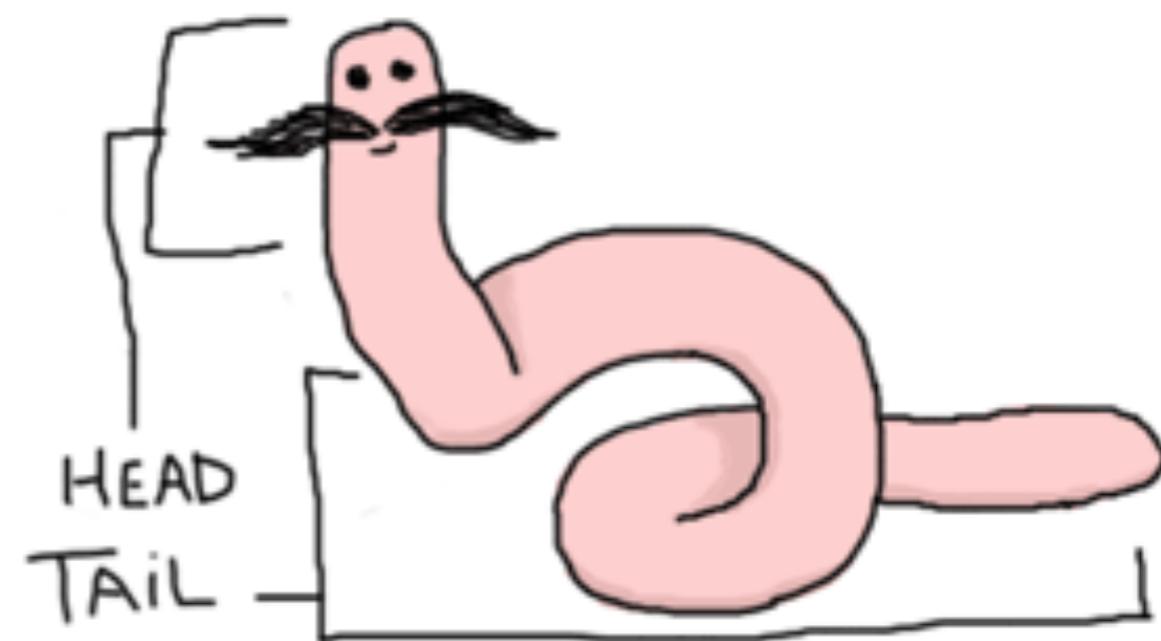
**Head** - the first element of a list.

**Tail** - the list that remains if we cut off the head.

```
1> L = [1,2,3,4].  
[1,2,3,4]
```

Head of L: 1

Tail of L: [2,3,4] head.



# Adding elements to the head of a list

The list construction operator | can be used to add one ore more elements to the head of a list.

```
1> L = [1,2,3,4].  
[1,2,3,4]
```

Add 0 to the head of L.

```
2> [0|L].  
[0,1,2,3,4]
```

# Extracting the head and tail of a list

Using the list constructor operator together with pattern matching, the head and tail of a list can be extracted.

```
1> L = [1,2,3,4].  
[1,2,3,4]  
2> [Head | Tail] = L.  
[1,2,3,4]  
3> Head.  
1  
4> Tail.  
[2,3,4]
```

# Length of a list

The built in function (BIF) **length** returns the number of elements in a list.

```
1> L = [1,2,3,4].  
[1,2,3,4]  
4> length(L).  
4  
5> length([]).  
0  
6> length([1,2, [a,b,c], 99]).  
4
```

**Strings are  
lists**

# Special lists

Lets type in this list and press enter.

```
15> [97 ,98 ,99] .
```

The Erlang shell responds with "abc".

```
15> [97 ,98 ,99] .  
"abc"
```

# ASCII

The American Standard Code for Information Interchange (ASCII).

Character	Code
A	65
B	66
C	67
↓	
z	90
[	91
↓	
a	97
b	98
c	99
↓	
z	122

```
15> [97,98,99].  
"abc"
```

All elements of this list are valid ASCII codes. The Erlang shell therefore chooses to show us the corresponding characters enclosed between quotes as a human readable string.

# Characters

# Character codes in Erlang

Character	Code
A	65
B	66
C	67
↓	
Z	90
[	91
↓	
a	97
b	98
c	99
↓	
z	122

To get the character code for a given character, precede the character with a dollar sign.

```
1> $a .  
97  
2> $A .  
65  
3> $[ .  
91  
28> [$a , $b , $c] .  
"abc"
```

# Strings

In Erlang, a list of valid ASCII characters is called a string. Strings are usually denoted by enclosing a sequence of characters between double quotes.

Examples of strings.

""

" "

"a"

"ab"

"Hello!"

"This is a longer string."

# Strings are lists

All operations on lists are valid on strings since strings are lists.

```
2> [H | T] = "ABC".  
"ABC"  
3> H.  
65  
4> T.  
"BC"  
5> [$a | "ABC"] .  
"aABC"  
6> length("") .  
0  
7> length("Hello") .  
5
```

# Tuples

# Tuples

A tuple is a compound data type with a **fixed number of terms**. The different terms can have different data types.

```
{Term1, ..., TermN}
```

Examples of tuples of various sizes.

```
{}  
{cat}  
{137}  
{"Hello"}  
{cat, 5}  
{cat, 5, "Kitty"}
```

# Tuples and pattern matching

Using pattern matching we can extract individual values from a tuple.

```
28> Pet = {cat, 5, "Kitty"} .  
{cat,5,"Kitty"}  
29> {Species, Age, Name} = Pet.  
{cat,5,"Kitty"}  
30> Species.  
cat  
31> Age.  
5  
32> Name.  
"Kitty"
```

Tuples are often used to collect related information.

# The anonymous variable `_`

The anonymous variable is denoted by underscore `_` and can be used when a variable is required but we want to ignore its value.

```
1> [H | _] = [1,2,3].  
[1,2,3]  
2> H.  
1  
3> Pet = {cat, 5, "Kitty"}.  
{cat,5,"Kitty"}  
4> {Species, _, _} = Pet.  
{cat,5,"Kitty"}  
5> Species.  
cat
```

# Pattern matching twice

Sometimes it is useful to pattern match twice in order to bind the whole match to one variable and parts of the match to other variables.

```
1> T = {A, B} = {a, tuple}.
{a, tuple}
2> T.
{a, tuple}
3> A.
a
4> B.
tuple
```

Here **T** is bound to the tuple **{a, tuple}** at the same time as **A** is bound to the atom **a** and **B** to the atom **tuple**.

**Exit the  
Erlang shell**

# Exit from the Erlang shell

To exit from the Erlang shell, type **q()**. and press enter.

```
1> 1 + 4.
```

```
5
```

```
2> q().
```

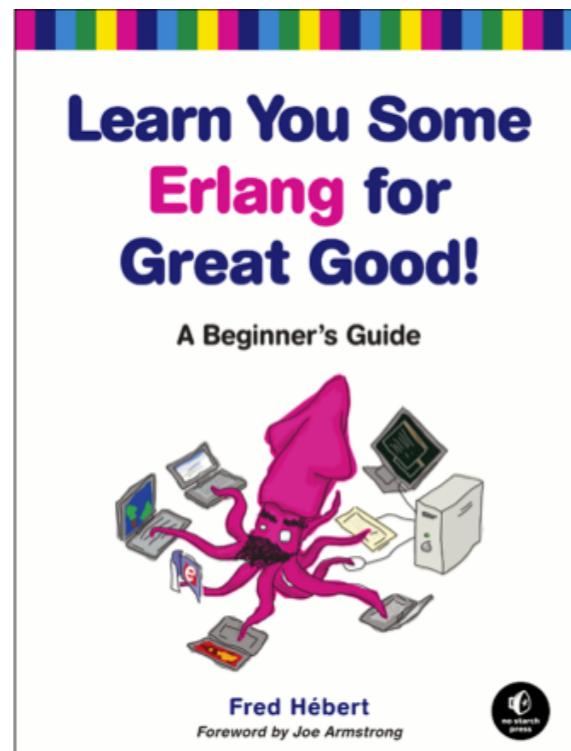
```
ok
```

```
$>
```

# Recommended reading

## **Learn you some Erlang for Great Good! A Beginner's Guide**

Fred Hébert



Available for free online  
<http://learnyousomeerlang.com>

## **Erlang Programming A Concurrent Approach to Software Development**

Cesarini & Thompson



You might also like this book as  
a complement to online  
resources.