# NUMBERS AND INVARIABLE VARIABLES

## Numbers

In the Erlang shell, **expressions have to be terminated with a period followed by whitespace** (line break, a space etc.), otherwise they won't be executed. You can separate expressions with commas, but only the result of the last one will be shown (the others are still executed). This is certainly unusual syntax for most people and it comes from the days Erlang was implemented directly in Prolog, a logic programming language.
Open the Erlang shell as described in the previous chapters and let's type them things!

```
1> 2 + 15.
17
2> 49 * 100.
4900
3> 1892 - 1472.
420
4> 5 / 2.
2.5
5> 5 div 2.
2
6> 5 rem 2.
1
```

You should have noticed Erlang doesn't care if you enter floating point numbers or integers: both types are supported when dealing with arithmetic.


Integers and floating values are pretty much the only types of data Erlang's mathematical operators will handle transparently for you. However, if you want to have the integer-to-integer division, use `div`, and to have the modulo operator, use `rem`(remainder).
Note that we can use several operators in a single expression, and mathematical operations obey the normal precedence rules.

```
7> (50 * 100) - 4999.
1
8> -(50 * 100 - 4999).
-1
```

```
9> -50 * (100 - 4999).
244950
```

If you want to express integers in other bases than base 10, just enter the number as `Base#Value` (given Base is in the range 2..36):

```
10> 2#101010.
42
11> 8#0677.
447
12> 16#AE.
174
```

Awesome! Erlang has the power of the calculator you have on the corner of your desk with a weird syntax on top of it! Absolutely exciting!

## Invariable Variables

Doing arithmetic is alright, but you won't go far without being able to store results somewhere. For that, we'll use variables. If you have read the intro to this book, you'll know that variables can't be variable in functional programming. The basic behavior of variables can be demonstrated with these 7 expressions (note that variables begin with an uppercase letter):

```
1> One.
* 1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
7> two = 2.
** exception error: no match of right hand side value 2
```

The first thing these commands tell us is that you can assign a value to a variable exactly once; then you can 'pretend' to assign a value to a variable if it's the same value it already has. If it's different, Erlang will complain. It's a correct observation, but the explanation is a bit more complex and depends on the `=` operator.

The `=` operator (not the variables) has the role of comparing values and complaining if they're different. If they're the same, it returns the value:

```
8> 47 = 45 + 2.
47
9> 47 = 45 + 3.
```

```
** exception error: no match of right hand side value 48
```

What this operator does when mixed with variables is that if the left-hand side term is a variable and it is unbound (has no value associated to it), Erlang will automatically bind the right-hand side value to the variable on the left-hand side. The comparison will consequently succeed and the variable will keep the value in memory.

This behavior of the `=` operator is the basis of something called 'Pattern matching', which many functional programming languages have, although Erlang's way of doing things is usually regarded as more flexible and complete than alternatives. We'll see pattern matching with more detail when we visit the tuple and list types in this very chapter, and also with functions in the following chapters.
The other thing the commands 1-7 told us is that variable names must begin with a capital letter. Command 7 failed because the word *two* had a lowercase letter to begin with. Technically, variables can start with an underscore ('_') too, but by convention their use is restricted to values you do not care about, yet you felt it was necessary to document what it contains.
You can also have variables that are only an underscore:

```
10> _ = 14+3.
17
11> _.
* 1: variable '_' is unbound
```

Unlike any other kind of variable, it won't ever store any value. Totally useless for now, but you'll know it exists when we need it.

**Note:** If you're testing in the shell and save the wrong value to a variable, it is possible to 'erase' that variable by using the function `f(Variable).`. If you wish to clear all variable names, do `f().`.
These functions are there only to help you when testing and only work in the shell. When writing real programs, we won't be able to destroy values that way. Being able to do it only in the shell makes sense if you acknowledge Erlang being usable in industrial scenarios: it is wholly possible to have a shell being active for years without interruption... Let's bet that the variable *X* would be used more than once in that time period.

Source : http://learnyousomeerlang.com/starting-out-for-real#numbers