RUN-TIME ERRORS

Run-time errors are pretty destructive in the sense that they crash your code. While Erlang has ways to deal with them, recognizing these errors is always helpful. As such, I've made a little list of common run-time errors with an explanation and example code that could generate them.

function_clause

```
1> lists:sort([3,2,1]).
[1,2,3]
2> lists:sort(fffffff).
** exception error: no function clause matching lists:sort(fffffff)
```

All the guard clauses of a function failed, or none of the function clauses' patterns matched.

case_clause

```
3> case "Unexpected Value" of
3>  expected_value -> ok;
3>  other_expected_value -> 'also ok'
3> end.
** exception error: no case clause matching "Unexpected Value"
```

Looks like someone has forgotten a specific pattern in their `case`, sent in the wrong kind of data, or needed a catch-all clause!

if_clause

```
4> if 2 > 4 -> ok;
4>  0 > 1 -> ok
4> end.
** exception error: no true branch found when evaluating an if expression
```

This is pretty similar to `case_clause` errors: it can not find a branch that evaluates to `true`. Ensuring you consider all cases or add the catch-all `true` clause might be what you need.

badmatch

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

Badmatch errors happen whenever pattern matching fails. This most likely means you're trying to do impossible pattern matches (such as above), trying to bind a variable for the second time, or just anything that isn't equal on both sides of the `=` operator (which is pretty much what makes rebinding a variable fail!). Note that this error sometimes happens because the programmer believes that a variable of the form *`_MyVar`* is the same as `_`. Variables with an underscore are normal variables, except the compiler won't complain if they're not used. It is not possible to bind them more than once.

badarg

```
6> erlang:binary_to_list("heh, already a list").
** exception error: bad argument
in function  binary_to_list/1
called as binary_to_list("heh, already a list")
```

This one is really similar to `function_clause` as it's about calling functions with incorrect arguments. The main difference here is that this error is usually triggered by the programmer after validating the arguments from within the function, outside of the guard clauses. I'll show how to throw such errors later in this chapter.

undef

```
7> lists:random([1,2,3]).
** exception error: undefined function lists:random/1
```

This happens when you call a function that doesn't exist. Make sure the function is exported from the module with the right arity (if you're calling it from outside the module) and double check that you did type the name of the function and the name of the module correctly. Another reason to get the message is when the module is not in Erlang's search path. By default, Erlang's search path is set to be in the current directory. You can add paths by using `code:add_patha/1` or `code:add_pathz/1`. If this still doesn't work, make sure you compiled the module to begin with!

badarith

```
8> 5 + llama.
** exception error: bad argument in an arithmetic expression
in operator  +/2
```

```
called as 5 + llama
```

This happens when you try to do arithmetic that doesn't exist, like divisions by zero or between atoms and numbers.

badfun

```
9> hhfuns:add(one,two).
** exception error: bad function one
in function  hhfuns:add/2
```

The most frequent reason why this error occurs is when you use variables as functions, but the variable's value is not a function. In the example above, I'm using the `hhfuns` function from the previous chapter and using two atoms as functions. This doesn't work and `badfun` is thrown.

badarity

```
10> F = fun(_) -> ok end.
#Fun<erl_eval.6.13229925>
11> F(a,b).
** exception error: interpreted function with arity 1 called with two arguments
```

The `badarity` error is a specific case of `badfun`: it happens when you use higher order functions, but you pass them more (or fewer) arguments than they can handle.

system_limit

There are many reasons why a `system_limit` error can be thrown: too many processes (we'll get there), atoms that are too long, too many arguments in a function, number of atoms too large, too many nodes connected, etc. To get a full list in details, read the Erlang Efficiency Guide on system limits. Note that some of these errors are serious enough to crash the whole VM.

Source : http://learnyousomeerlang.com/errors-and-exceptions