DEALING WITH EXCEPTIONS

I've already mentioned quite a few times that throws, errors and exits can be handled. The way to do this is by using a `try ... catch` expression.

A `try ... catch` is a way to evaluate an expression while letting you handle the successful case as well as the errors encountered. The general syntax for such an expression is:

```
try Expression of
SuccessfulPattern1 [Guards] ->
Expression1;
SuccessfulPattern2 [Guards] ->
Expression2
catch
TypeOfError:ExceptionPattern1 ->
Expression3;
TypeOfError:ExceptionPattern2 ->
Expression4
end.
```

The *`Expression`* in between `try` and `of` is said to be *protected*. This means that any kind of exception happening within that call will be caught. The patterns and expressions in between the `try ... of` and `catch` behave in exactly the same manner as a `case ... of`. Finally, the `catch` part: here, you can replace *`TypeOfError`* by either`error`, `throw` or `exit`, for each respective type we've seen in this chapter. If no type is provided, a `throw` is assumed. So let's put this in practice.

First of all, let's start a module named `exceptions`. We're going for simple here:

```
-module(exceptions).
-compile(export_all).

throws(F) ->
try F() of
_ -> ok
catch
Throw -> {throw, caught, Throw}
end.
```

We can compile it and try it with different kinds of exceptions:

```
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
```

As you can see, this `try ... catch` is only receiving throws. As stated earlier, this is because when no type is mentioned, a throw is assumed. Then we have functions with catch clauses of each type:

```
errors(F) ->
try F() of
_ -> ok
catch
error:Error -> {error, caught, Error}
end.

exits(F) ->
try F() of
_ -> ok
catch
exit:Exit -> {exit, caught, Exit}
end.
```

And to try them:

```
4> c(exceptions).
{ok,exceptions}
5> exceptions:errors(fun() -> erlang:error("Die!") end).
{error,caught,"Die!"}
6> exceptions:exits(fun() -> exit(goodbye) end).
{exit,caught,goodbye}
```

The next example on the menu shows how to combine all the types of exceptions in a single `try ... catch`. We'll first declare a function to generate all the exceptions we need:

```
sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
try Attack() of
_ -> "None shall pass."
catch
throw:slice -> "It is but a scratch.";
error:cut_arm -> "I've had worse.";
exit:cut_leg -> "Come on you pansy!";
_:_ -> "Just a flesh wound."
end.
```

Here `is_function/2` is a BIF which makes sure the variable *`Attack`* is a function of arity 0. Then we add this one for good measure:

```
talk() -> "blah blah".
```

And now for something completely different:

```
7> c(exceptions).
{ok,exceptions}
8> exceptions:talk().
"blah blah"
9> exceptions:black_knight(fun exceptions:talk/0).
"None shall pass."
10> exceptions:black_knight(fun() ->
> exceptions:sword(1) end).
"It is but a scratch."
11> exceptions:black_knight(fun() ->
> exceptions:sword(2) end).
"I've had worse."
12> exceptions:black_knight(fun() ->
> exceptions:sword(3) end).
"Come on you pansy!"
13> exceptions:black_knight(fun() ->
> exceptions:sword(4) end).
"Just a flesh wound."
14> exceptions:black_knight(fun() ->
> exceptions:sword(5) end).
"Just a flesh wound."
```

The expression on line 9 demonstrates normal behavior for the black knight, when function execution happens normally. Each line that follows that one demonstrates pattern matching on exceptions according to their class (throw, error, exit) and the reason associated with them (`slice`, `cut_arm`, `cut_leg`).

One thing shown here on expressions 13 and 14 is a catch-all clause for exceptions. The `_:_` pattern is what you need to use to make sure to catch any exception of any type. In practice, you should be careful when using the catch-all patterns: try to protect your code from what you can handle, but not any more than that. Erlang has other facilities in place to take care of the rest.

There's also an additional clause that can be added after a `try ... catch` that will always be executed. This is equivalent to the 'finally' block in many other languages:

```
try Expr of
Pattern -> Expr1
catch
Type:Exception -> Expr2
after % this always gets executed
Expr3
end
```

No matter if there are errors or not, the expressions inside the `after` part are guaranteed to run. However, you can not get any return value out of the `after` construct. Therefore, `after` is mostly used to run code with side effects. The canonical use of this is when you want to make sure a file you were reading gets closed whether exceptions are raised or not.

We now know how to handle the 3 classes of exceptions in Erlang with catch blocks. However, I've hidden information from you: it's actually possible to have more than one expression between the `try` and the `of`!

```
whoa() ->
try
talk(),
_Knight = "None shall Pass!",
_Doubles = [N*2 || N <- lists:seq(1,100)],
throw(up),
_WillReturnThis = tequila
of
tequila -> "hey this worked!"
catch
Exception:Reason -> {caught, Exception, Reason}
end.
```

By calling `exceptions:whoa()`, we'll get the obvious `{caught, throw, up}`, because of `throw(up)`. So yeah, it's possible to have more than one expression between `try` and `of`...

What I just highlighted in `exceptions:whoa/0` and that you might have not noticed is that when we use many expressions in that manner, we might not always care about what the return value is. The `of` part thus becomes a bit useless. Well good news, you can just give it up:

```
im_impressed() ->
try
talk(),
_Knight = "None shall Pass!",
_Doubles = [N*2 || N <- lists:seq(1,100)],
throw(up),
_WillReturnThis = tequila
catch
Exception:Reason -> {caught, Exception, Reason}
end.
```

And now it's a bit leaner!

Note: It is important to know that the protected part of an exception can't be tail recursive. The VM must always keep a reference there in case there's an exception popping up. Because the `try ... catch` construct without the `of` part has nothing but a protected part, calling a recursive function from there might be dangerous for programs supposed to run for a long time (which is Erlang's niche). After enough iterations, you'll go out of memory or your program will get slower without really knowing why. By putting your recursive calls between

the`of` and `catch`, you are not in a protected part and you will benefit from Last Call Optimisation.
Some people use `try ... of ... catch` rather than `try ... catch` by default to avoid unexpected errors of that kind, except for obviously non-recursive code with results that won't be used by anything. You're most likely able to make your own decision on what to do!

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