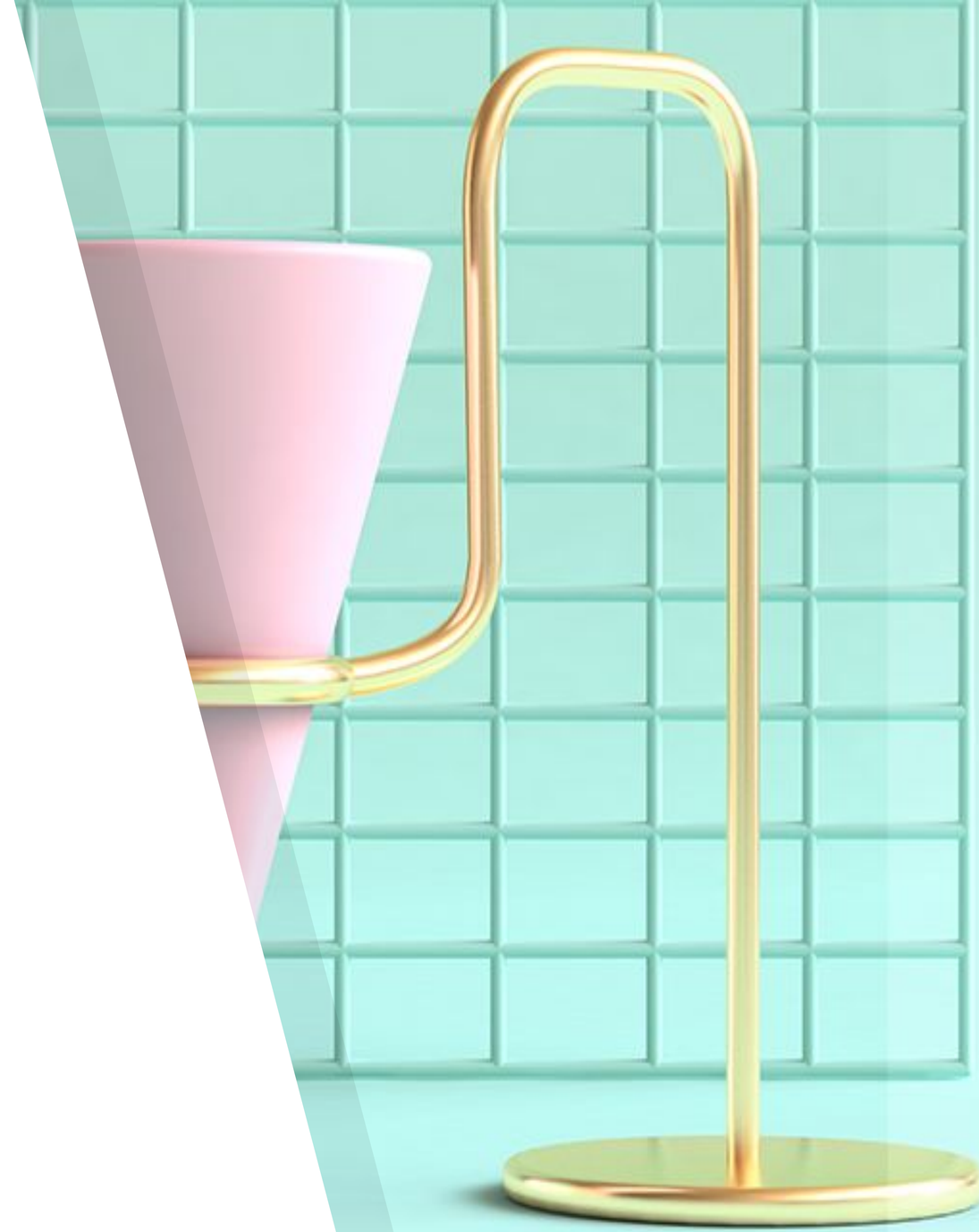
Sequential Error Handling



Erlang SOLUTIONS

Overview: Sequential Error Handling

- Run-time errors
- Try ... catch
- Throw
- Catch

Run Time Errors: match

```
-module(math).
-export([factorial/1]).

factorial(N) when N > 0 ->
    N * factorial(N - 1);
factorial(0) -> 1.
```

- **function_clause** is returned when none of the existing function patterns matches

```
1> math:factorial(-1).
** exception error: no function clause matching
  math:factorial(-1)
```

Run Time Errors: match

```
-module(case_test).
-export([test/1]).

test(N) ->
    case N of
        -1 -> false;
         1 -> true
    end.
```

- **case_clause** is returned when none of the existing function patterns matches

```
1> case_test:test(0).
** exception error: no case clause matching 0
     in function  case_test:test/1
```

Run Time Errors

```
-module(if_test).
-export([test/1]).

test(N) ->
    if
        N < 0 -> false;
        N > 0 -> true
    end.
```

- **if_clause** is returned when none of the existing expressions in the if statement evaluates to **true**

```
1> if_test:test(0).
** exception error: no true branch found when evaluating an if expression
     in function  if_test:test/1
```

Run Time Errors: match

```
1> Tuple = {1, two, 3}.
{1,two,3}
2> {1, two, 3, Four} = Tuple.
** exception error: no match of right hand side value
  {1,two,3}
```

- **badmatch** errors occur in situations when pattern matching fails and there are no other alternative clauses to choose from

Run Time Errors: **others**

```
1> length(helloWorld).
** exception error: bad argument in function length/1
        called as length(helloWorld)
```

- **badarg** is returned when a BIF is called with the wrong type

Run Time Errors: **others**

```
1> test:hello().
  ** exception error: undefined function test:hello/0
```

- **undef** will be returned if the global function being called is not defined or exported

Run Time Errors: others

```
1> 1 + a.
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 1 + a
```

▶️ **badarith** is returned when arithmetical operations are executed with values that are neither integers or floats.

Try ... catch

```
try <expressions> of
  Pattern1 ->
     <expressions 1>;
  Pattern2 ->
     <expressions 2>
catch
  [Class1:]ExceptionPattern1 ->
     <exception expressions 1>;
  [Class2:]ExceptionPattern2 ->
     <exception expressions 2>
end
```

- **try ... catch** provides a mechanism for monitoring the evaluation of an expression
- It will trap exits caused by expected run time errors
- The patterns **Class1:** and **Class2:** can define the type of exception handled
- The **ExceptionPatterns** can restrict the reason why an exception is raised.

Try ... catch

```
1> self().
<0.53.0>
2> X = 2, X = 3.
** exception error: no match of right
hand side value 3
3> self().
<0.57.0>
4> try (X = 3) of
4>   Val -> {normal, Val}
4> catch
4>   Class:Error ->
4>     {Class, Error}
4> end.
{error,{badmatch,3}}
5> self().
<0.57.0>
```

▶️ _:_ allows to match on all errors no matter what they are.

▶️ The error is caught and the process doesn't crash

Try ... catch

```
1> X = 2.
2
2> try (X = 3) of
2>   Val -> {normal, Val}
2> catch
2>   error:Error ->
       {error, Error}
2> end.
{error,{badmatch,3}}
3> try (X = 3) of
3>   Val -> {normal, Val}
3> catch
3>   error:{badmatch, Val} ->
       {error, {badmatch, Val}}
3> end.
{error,{badmatch,3}}
```

- The **error:Error** pattern allows to bind the error reason to a variable and match on it
- **error:{badmatch,_}** allows to match only errors caused by erroneous pattern matching

Throw

```
throw(<expression>)
```

WARNING!!
use with care as it makes the code hard to debug and understand

- **throw** is used for non-local returns in deep recursive function calls.
- The execution flow jumps to the first **catch** in the execution stack
- Useful for handling exceptions in deeply nested code when you do not want to handle possible errors.

Throw

```
-module(example).
-export([add/2]).

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

int(X) when is_integer(X) -> X;
int(X)                    -> throw({error, {non_integer, X}}).
```

```
1> example:add(1, one).
** exception throw:
{error,{non_integer,one}}
2> try example:add(1, one) of
2>   Int -> Int
2> catch
2>   throw:Reason -> {throw, Reason}
2> end.
{throw,{error,{non_integer,one}}}
```

Try ... catch: examples

```
-module(exception).
-export([try_wildcard/1]).

try_wildcard(X) when is_integer(X) ->
  try return_error(X)
  catch
    throw:Error ->
      {throw, Error};
    error:Error:StackTrace ->
      {error,{Error,StackTrace}};
    exit:Exit ->
      {exit, Exit}
end.
```

Try ... catch: examples

```
return_error(X) when X < 0 ->
  throw({badarith,[{exception,return_error,1},
                   {erl_eval,do_apply,6},
                   {shell,exprs,7},
                   {shell,eval_exprs,7},
                   {shell,eval_loop,3}]});
return_error(X) when X == 0 -> 1/X;
return_error(X) when X > 0 ->
  exit({badarith,[{exception,return_error,1},
                  {erl_eval,do_apply,6},
                  {shell,exprs,7},
                  {shell,eval_exprs,7},
                  {shell,eval_loop,3}]}).
```

Try … catch: examples

```
1> exception:try_wildcard(-1).
{throw,{badarith,[{exception,return_error,1},
                  {erl_eval,do_apply,6},
                  ...
                  {shell,eval_loop,3}]}}
2> exception:try_wildcard(0).
{error,{badarith,[{exception,return_error,1,[{file,…},…]},
                  {exception,try_wildcard,1,[{file,…},…]},
                  {erl_eval,do_apply,6,[{file,…},…]},
                  ...
                  {shell,eval_loop,3,[{file,…},…]}]}}
3> exception:try_wildcard(1).
{exit,{badarith,[{exception,return_error,1},
                 {erl_eval,do_apply,5},
                 ...
                 {shell,eval_loop,3}]}}
```

Catch

```
catch <expression>
```

- **catch** provides a mechanism for monitoring the evaluation of an expression
- It will trap exits caused by runtime errors
- A function call resulting in a run time error called in the scope or a catch will return the tuple **{'EXIT', Reason}**
- **Reason** is the runtime error which occurred

Catch

```
1> self().
<0.28.0>
2> 1/0.
** exception error: an error occurred when evaluating an arithmetic
expression
     in operator  '/'/2
        called as 1 / 0
3> self().
<0.33.0>
4> catch 1/0.
{'EXIT',{badarith,[{erlang,'/',[1,0],[]},
                   {erl_eval,do_apply,6,
                             [{file,"erl_eval.erl"},{line,684}]},
                   {erl_eval,expr,5,
                             [{file,"erl_eval.erl"},{line,437}]},
                   ...
                   {shell,eval_loop,3,
                          [{file,"shell.erl"},{line,627}]}]}}
5> self().
<0.33.0>
```

Catch

```
1> catch 1/throw(whoops).
whoops
2> X = catch 1/0.
* 1: syntax error before: 'catch'
2> X = (catch 1/0).
{'EXIT',{badarith,[{erlang,'/',[1,0],[]},
                   {erl_eval,do_apply,6,
                             [{file,"erl_eval.erl"},{line,684}]},
                   {erl_eval,expr,5,
                             [{file,"erl_eval.erl"},{line,437}]},
...
                   {shell,eval_loop,3,
                          [{file,"shell.erl"},{line,627}]}]}}
```

Summary: Sequential Error Handling

- Run-time errors
- Try ... catch
- Throw
- Catch