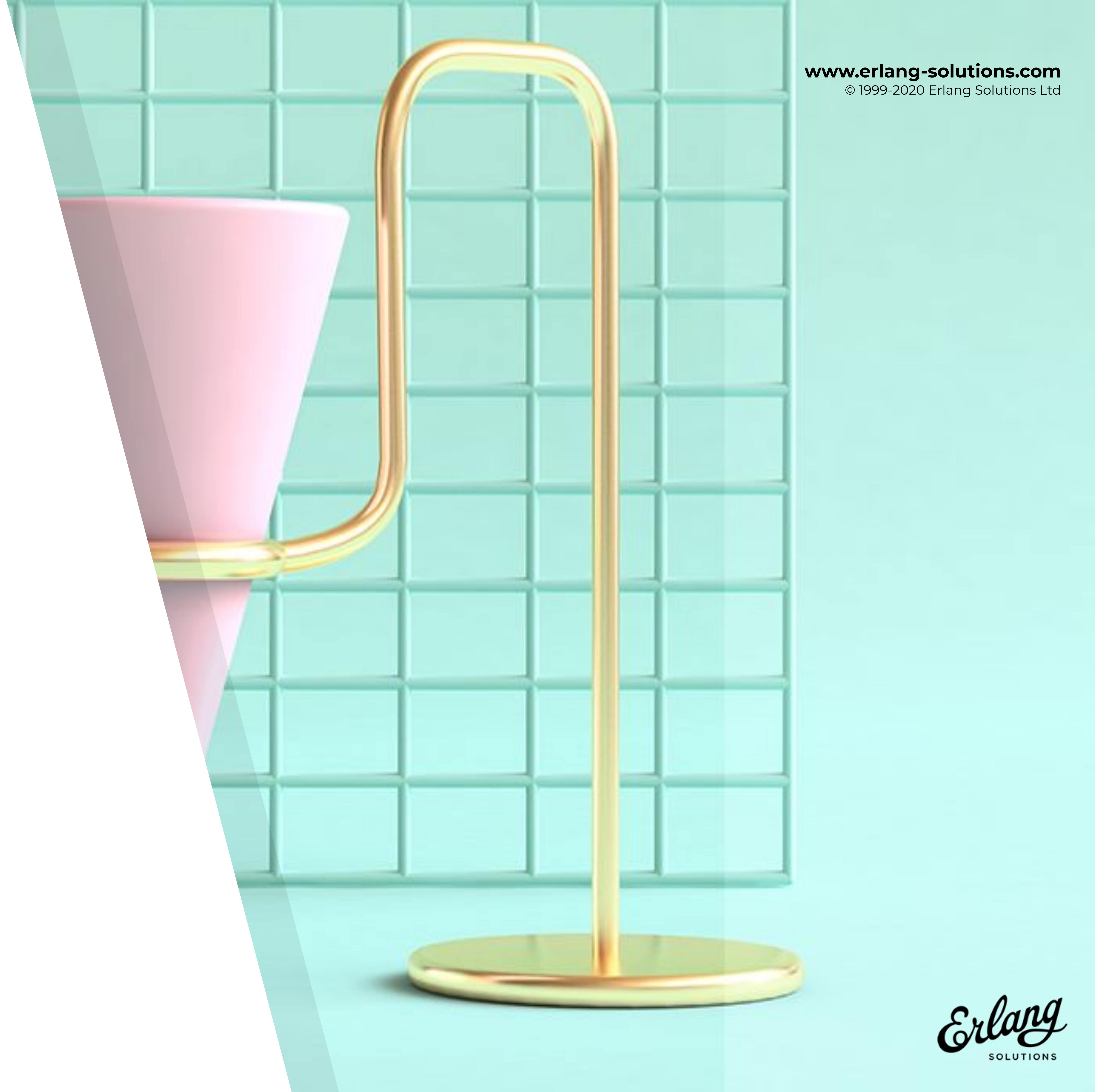
Functional Programming Constructs



Functional Programming Constructs

- Funs
- Higher Order Functions
- List Comprehensions

Funs

```
1> Add = fun(X, Y) -> X+Y end.
#Fun<erl_eval>
2> Add(2,3).
5
```

- Funs are data types encapsulating functional objects
- They can be passed as arguments
- They can be the return value of function calls

Funs: examples

```
1> Bump = fun(X) -> X+1 end.
#Fun<erl_eval.19.120858100>
2> lists:map(Bump, [1,2,3,4,5]).
[2,3,4,5,6]
3> Positive = fun(X) -> X >= 0 end.
#Fun<erl_eval.19.120858100>
4> lists:filter(Positive, [-2,-1,0,1,2]).
[0,1,2]
5> lists:all(Positive, [0,1,2,3,4]).
true
6> lists:all(Positive, [-1,0,1]).
false
```

Funs: higher order functions

lists:all(Predicate, List) -> true | false

Returns **true** if the **Predicate** fun returns **true** for all elements in **List**

lists:filter(Predicate, List) -> NewList

Returns a list with elements for which **Predicate** is true

lists:foreach(Fun, List) -> ok

Applies **Fun** on every element in **List**. Used for side effects

lists:map(Fun, List) -> NewList

Returns a list with the return value of **Fun** applied to all elements in **List**

Funs: higher order functions

- Functions taking funs as arguments are called higher order functions
- They encourage the encapsulation of common design patterns, facilitate the re-usage of these functions
- Improves the clarity of the program
- Hides recursive calls
- The process of abstracting out common patterns in programs is called **procedural abstraction**.

Funs: procedural abstraction

```
double([H|T]) ->
   [H*2 | double(T)];
double([]) ->
   [].

bump([H|T]) ->
   [H+1 | bump(T)];
bump([]) ->
   [].
```

```
map(Fun, [H|T]) ->
   [Fun(H) | map(Fun, T)];
map(_Fun, []) ->
   [].

double(L) ->
   map(fun(X)-> X*2 end, L).

bump(L) ->
   map(fun(X)-> X+1 end, L).
```

Functions: syntax

fun head

```
fun (X, X) ->
        X * 2;
    (X, Y) ->
        X + Y
end
```

clause separator

fun clauses

no separator in last clause

reserved words

fun body

Funs: syntax

```
fun(Var1, ..., VarN) ->
  <Expr1>,
  <Expr2>,
  ...
  <ExprN>;
   (Var1, ..., VarN) ->
  <Expr1>,
  <Expr2>,
   ...
  <ExprN>
end
```

- The syntax is similar to that of functions, only that it starts with the keyword **fun** and ends with the keyword **end**.

Funs: syntax

F = fun Function/Arity

Will bind the local function in the current module to **F**

F = fun Module:Function/Arity

Will bind the function exported in **Module** to **F**.

Funs: scope of variables

- All variables in the head of the Fun are considered fresh, and bound when the fun is first called
- Variables bound before the Fun can be used in the Fun and in guard tests
- No variables may be exported from the Fun
- Variables in the function head shadow already bound variables in the function the Fun is defined in

Funs: scope of variables

```
foo() ->
  X = 2,
  Bump = fun(X) -> X + 1 end,
  Bump(10).
```

X is shadowed in the fun

```
1> funs:foo().
11
```

Funs: scope of variables

```
bar() ->
  X = 10,
  Bump = fun(Y) -> X + Y end,
  Bump(10).
```

X is not shadowed in the fun

```
1> funs:bar().
20
```

Funs: scope of variables

```
1> GreaterThan = fun(X) ->
1>                  fun(Y) -> Y>X end
1>                end.
#Fun<erl_eval.6.13229925>
2> Gt4 = GreaterThan(4).
#Fun<erl_eval.6.13229925>
3> Gt4(3).
false
4> (GreaterThan(4))(5).
true
5> lists:filter(Gt4,
5>          [1,6,8,3,5,0,4,11]).
[6,8,5,11]
```

- It is possible for a Fun to return another Fun.
- This can be used to introduce a new variable in the Fun's scope to 'wrap' the arguments it would usually need.

List Comprehensions

```
[ X || X <-[1,2,3,4], X < 3 ]
```

- The above example should be read as the list of X where X comes from the list [1,2,3,4] and X is less than 3
  - **Pattern <- List** is the **generator**
  - **Filter** is either a boolean expression or a function which returns true or false

List Comprehensions

```
[ Expression || Generator1 , ..., GeneratorN,
                Filter1, ..., FilterN ]
```

- A feature common in functional programming languages
- Analogous to set comprehensions in the Zermelo-Frankel set theory
- A syntactical and semantical notation to generate lists

List Comprehensions: examples

```
1> [Int || Int <- [zero, 1, two, three, 4, 5],
1>          is_integer(Int)].
[1,4,5]
2> [{X,Y} || X <- [1,2,3], Y <- [a,b,c]].
[{1,a},{1,b},{1,c},{2,a},{2,b},{2,c},{3,a},{3,b},{3,c}]
3> [X || X <- [1,2,3,4], Y <- [3,4,5,6], X == Y].
[3,4]
4> [X+1 || X <- [1,2,3,4], X rem 2 == 0].
[3,5]
```

- Filtering, cartesian products, intersections, and selective mapping using list comprehensions

List Comprehensions: examples

```
map(Fun, List) ->
    [Fun(X) || X <- List].
filter(Predicate, List) ->
    [X || X <- List, Predicate(X)].
append(ListOfLists) ->
    [X || List <- ListOfLists, X <- List].
```

- Rewriting lists library functions using list comprehensions

List Comprehensions: examples

```
perm([])  ->
    [[]];
perm(List) ->
    [[H|T] || H <- List, T <- perm(List -- [H])].
```

- perm([c,a,t]) -> [[c,a,t],[c,t,a],[a,c,t],[a,t,c],[t,c,a],[t,a,c]]
- We take **H** from **List** in all possible ways, and append all permutations of **List** with **H** removed to it

List Comprehensions: variables

- All variables in the generator pattern are considered fresh
- Bound variables in the generator and before the LC expression which are used in the filter retain their value
- No variable can be exported from a LC expression
- The compiler gives a warning when you shadow variables

List Comprehensions: variables

```
1> X = 1, Y = 2.
2
2> [{X, Y} || X <- lists:seq(1,3)].
[{1,2},{2,2},{3,2}]
3> List = [{1,one}, {2,two}, {3,three}].
[{1,one},{2,two},{3,three}]
4> [Z || {X1, Z} <- List, X1 == X].
[one]
```

Functional Programming Constructs

- Funs
- Higher Order Functions
- List Comprehensions