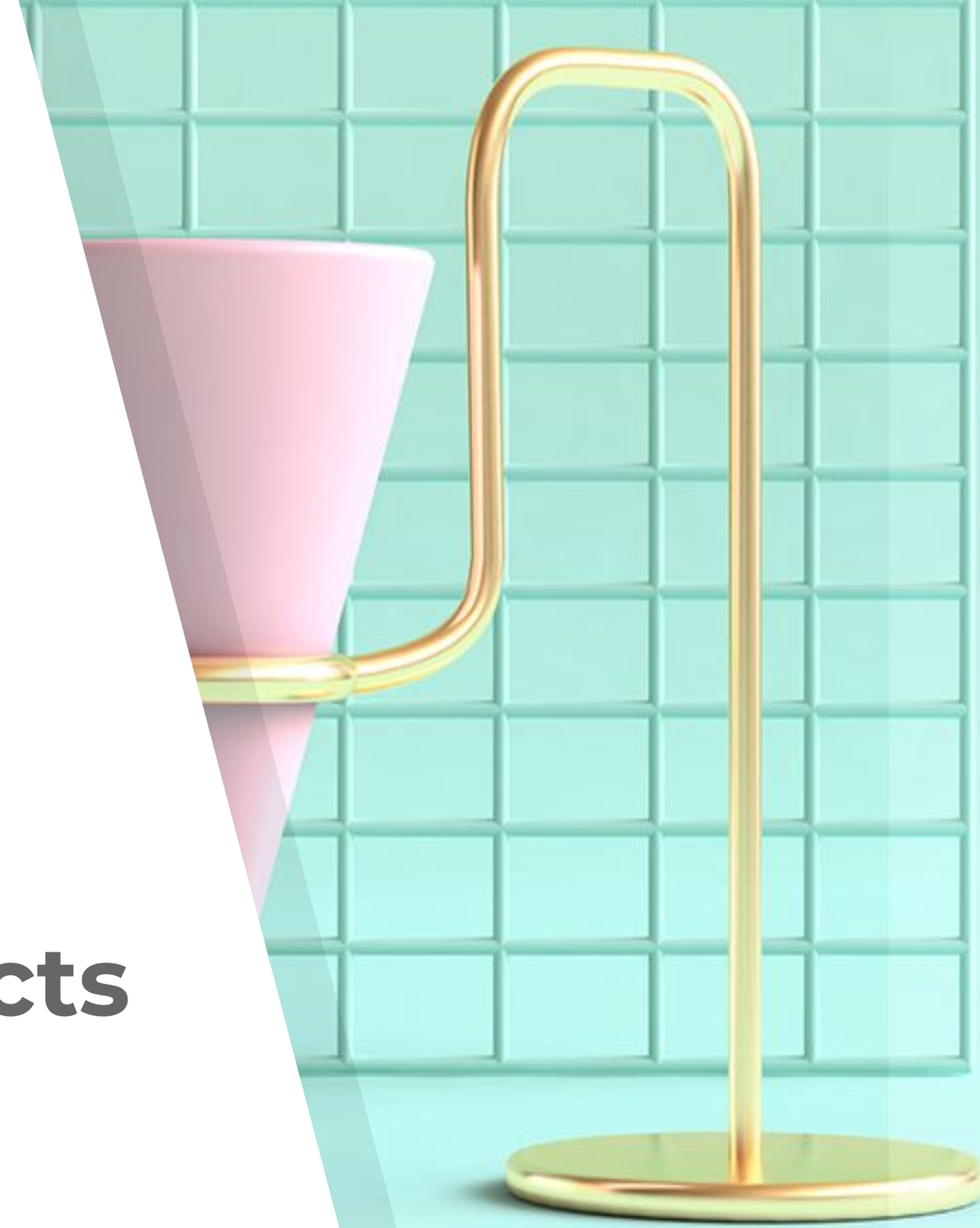
Types & Constructs



Erlang SOLUTIONS

Overview: Types & Constructs

- The Shell
- Data Types
- Variables
- Complex Data Structures
- Pattern Matching
- BIFs
- Function Calls
- Modules

Erlang Shell

```
$ erl
Erlang/OTP 22 [erts-10.5] [source] [64-bit] [smp:12:12] [ds:12:12:10]
[async-threads:1]

Eshell V10.5  (abort with ^G)
1> help().
** shell internal commands **
b()        -- display all variable bindings
e(N)       -- repeat the expression in query <N>
f()        -- forget all variable bindings
f(X)       -- forget the binding of variable X
h()        -- history
history(N) -- set how many previous commands to keep
results(N) -- set how many previous command results to keep
catch_exception(B) -- how exceptions are handled
...
```

Data Types: floats

```
17.368
-56.654
12.34E-10
```

- Not efficiently implemented
- Stored as a double
  - 64-bit representation
- Follows the IEEE 754 standard

Data Types: integers

```
0
10
100000000
-234
16#AB10F
2#1010
$a
$A
$\n
```

- **B#Val** is used to store numbers in base **B**
- **$Char** is used for ascii values
  - $A is equivalent to 65
- Large integers are converted to bignums
- Max size depends on physical constraints:
  - RAM
  - Paging memory

Data Types: atoms

```
january
fooBar
alfa21
start_with_lower_case
node@ramone
true
false

'January'
'a space'
'Anything inside quotes{}#@ \n\012'
'node@ramone.erlang.org'
```

- Atoms are constant literals
- Start with a lower case letter or are encapsulated by ''
- Any character code is allowed within an atom if using ''
- Letters, integers and _ are allowed if the atom starts with a lowercase letter

Data Types: booleans

```
true
false
1 == 2
1 /= 2
1 == 1.0
1 =:= 1
1 =/= 1.0
1 < 2
a > z
less < more
is_boolean(9+6)
is_boolean(true)
not((1 < 3) and (2 == 2))
not((1 < 3) or (2 == 2))
not((1 < 3) xor (2 == 2))
```

- No separate type for booleans: atoms **true** and **false** are used instead.
- Operators (and, andalso, or, orelse, xor, not) accept **true** and **false** as if they actually were boolean types.

Data Types: tuples

```
{123, bcd}
{123, def, abc}
{abc, {def, 123}, ghi}
{}
{person, 'Joe', 'Armstrong'}
{person, 'Mike', 'Williams'}
```

Tuples are used to denote data-types with a fixed number of items

Tuples of any size are allowed

Contain valid Erlang expressions

Data Types: lists

```
[1, 2, 3, 4, 5, 6, 7, eight, nine]
```

Lists are written beginning with a [and ending with a]

Elements are separated by commas

Used to store a variable number of items

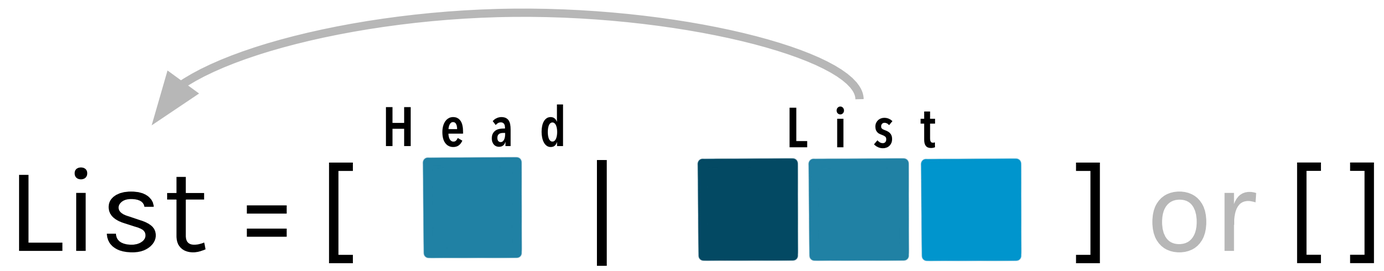
Lists are dynamically sized

Strings in Erlang are lists of ASCII values

Data Types: lists

```
[january, february, march]
[123, def, abc]
[a,[b,[c,d,e],f],g]
[]
[{person, 'Joe', 'Armstrong'},
 {person, 'Robert', 'Virding'},
 {person, 'Mike', 'Williams'}]
[72,101,108,108,111,32,87,111,114,108,100]
[$H,$e,$l,$l,$o,$ ,$W,$o,$r,$l,$d]
"Hello World"
```

Data Types: lists



A recursive list definition consists of a head and a tail

Lists whose last tail term is [] are called:

- **proper** lists or
- **well formed** lists

The tail can be any valid Erlang data type

Most Erlang programs manipulate proper lists

Data Types: lists

```
[one, two, three, four]
[one, two, three, four | []]
[one, two|[three, four]]
[one, two|[three|[four|[]]]]
[one|[two|[three|[four|[]]]]]
```

Complex Data Structures

```
[{{person, "Joe", "Armstrong"},
  [{telephone_number, [3,5,9,7]},
   {shoe_size, 42},
   {pets, [{cat, tubby}, {cat, tiger}]},
   {children, [{thomas, 5}, {claire, 1}]}]
 },
 {{person, "Mike", "Williams"},
  [{shoe_size, 41},
   {likes, [boats, beer]}]
 }
].
```

Complex Data Structures

- Arbitrary complex data structures can be created by nesting other data structures
- Data structures may contain bound variables
- Data structures are created by writing them down
- No explicit memory allocation or deallocation is needed
  - Allocated automatically
  - Deallocated by the **garbage collector** when no longer referenced.

Variables

```
A_long_variable_name
Flag
Name2
DbgFlag


_another_variable
_
```

- Variables can start with an uppercase letter or _
- They may not contain any 'funny characters'
- _ alone is a don't care variable
  - Its values are ignored and never bound

Variables

Variables can only be bound once!

- Variables are used to store values of data structures
- The value of a variable can not be changed once it has been bound
- There is no need to declare them. Just use them!
- Erlang does not have a static type system
- Types are determined at run time

Pattern Matching

```
Pattern = Expression
```

```
Pattern matching is used for:
  Assigning values to variables
  Controlling the execution flow of programs (if, case, function heads)
  Extracting values from compound data types
The pattern can contain variables which are bound when the
matching succeeds
The expression may not contain unbound variables
```

Pattern Matching: assigning

A = 10

Succeeds, binds **A** to **10**

{B, C, D} = {10, foo, bar}

Succeeds, binds **B** to **10**, **C** to **foo** and **D** to **bar**.

{E, E, foo} = {abc, abc, foo}

Succeeds, binds **E** to **abc**.

[H|T] = [1,2,3]

Succeeds, binds **H** to **1**, **T** to **[2,3]**.

Pattern Matching: controlling

A match must either succeed or fail

{A, A, B} = {abc, def, 123}

fails

[A,B,C,D] = [1,2,3]

fails

[A,B|C] = [1,2,3,4,5,6,7]

succeeds, A = 1, B = 2, C = [3,4,5,6,7]

[H|T] = []

fails

Pattern Matching: extracting

{A, _, [B|_], {B}} = {abc, 23, [22, x], {22}}

Succeeds, A = abc, B = 22

C = 10,
{C, C, 13, D, _} = {10, 10, 13, 12, 15}

Succeeds, D = 12, C = 10

Var = {person, 'Francesco', 'Cesarini'},
{person, Name, Surname} = Var

Succeeds, Name = 'Francesco', Surname = 'Cesarini'

[Head|Tail] = [1,2,3,4]

Succeeds, Head = 1, Tail = [2,3,4]

Built-in Functions

```
date()

time()

length(List)

size(Tuple)

atom_to_list(Atom)

list_to_tuple(List)

integer_to_list(2235)

tuple_to_list(Tuple)
```

- Do what you cannot do (or is difficult to do) in Erlang
- Mostly written in C for fast execution
- BIFs are by convention regarded as being in the **erlang** module.

Built-in Functions

- There are BIFs for:
  - Process and port handling
  - Object access and examination
  - Meta programming
  - Type conversion
  - System information
  - Distribution
  - Others
- For a complete list, see the manual page for the erlang module.

Built-in Functions: examples

```
1> date().
{2010,9,25}
2> atom_to_list(abcd).
"abcd"
3> tuple_to_list(list_to_tuple([1,2,3,4])).
[1,2,3,4]
4> length([1,2,3,4,5]).
5
5> processes().
[<0.0.0>,<0.1.0>,<0.2.0>,<0.3.0>,<0.4.0>,<0.5.0>,<0.6.0>,
 <0.9.0>,<0.41.0>,<0.43.0>,<0.45.0>,<0.46.0>,<0.48.0>,
 <0.49.0>,<0.50.0>,<0.51.0>,<0.52.0>,<0.53.0>,<0.54.0>,
 <0.55.0>,<0.56.0>,<0.57.0>,<0.58.0>,<0.59.0>,<0.60.0>,
 <0.61.0>,<0.62.0>,<0.63.0>,<0.64.0>|...]
6> length(processes()).
44
7>
```

Functions: calls

```
module:function(Arg1, Arg2, ..., ArgN)
function(Arg1, Arg2, ..., ArgN)
```

- Erlang programs consist of functions that call each other
- Functions are defined within modules
- Function names and module names must be atoms
- The **arity** of a function is its number of arguments.

Functions: syntax

```
circumference(R) ->
    2 * math:pi() * R.


product(X, Y) -> X * Y.

product(X, Y, Z) -> X * Y * Z.
```

Functions: syntax

clause head

clause separator

```
area({square, Side}) ->
    Side * Side;
area({circle, Radius}) ->
    math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
    S = (A + B + C) / 2,
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

function clauses

clause body

expression separator

function terminator

Functions: syntax

```
Func(Pattern1, Pattern2, ...) ->
   <expression 1>,
   <expression 2>,
   ...
   <expression n>;
Func(Pattern1, Pattern2, ...) ->
   <expression 1>,
   <expression 2>,
   ...
   <expression n>;
...
Func(Pattern1, Pattern2, ...) ->
   <expression 1>,
   <expression 2>,
   ...
   <expression n>.
```

- A function is defined as a collection of clauses
- Variables are pattern matched in the function clause head
- If pattern matching fails on a clause, the next one is tested
- The first clause matched is used
- The last expression executed in the clause body is returned

Functions: examples

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


> factorial(3).
   (matches N = 3 in clause 2)
   == 3 * factorial(3-1)
   (matches clause 2)
   == 3 * 2 * factorial(2-1)
   (matches clause 2)
   == 3 * 2 * 1 * factorial(1-1)
   (matches clause 1)
   == 3 * 2 * 1 * 1
   == 6
```

- Pattern matching occurs in the function head
  - Unbound variables get bound after a successful pattern match
- Variables are local to each clause
- Variables are allocated and deallocated automatically

Modules

module name

module directives

```
-module(demo).
-export([double/1]).

% This is a comment.
% Everything after '%' is ignored.

double(X) ->
    times(X, 2).

times(X, N) ->
    X * N.
```

arity

exported functions

functions

Modules

- Modules are stored in files with the **.erl** suffix
- The module and file names must be the same
  - You store the module **demo** in **demo.erl**
- Modules are named with the **-module(Name).** directive
- Exported functions can be called from outside the module

Modules

- Use **-export([Func/Arity, Func/Arity, ...]).**
- Local functions may only be called within the module
- Prefix function calls with the module name when making a call
  from outside the module
  - **Module:Fun(Arg1, ...)**
  - This is a **fully qualified call**

Modules: meta calls

```
apply(Module, Function, Arguments)
M:function(Arg1, ...)
M:F(Arg1, ...)
```

- ▶️ **apply/3** is a BIF used to dynamically evaluate functions
  - The function must be exported
  - The arguments can possibly be an empty list
  - All the arguments can be established at runtime
  - Extremely powerful when implementing generic code

Modules: meta calls

```
1> Module = io.
io
2> Function = format.
format
3> Arguments = ["Hello World ", []].
["Hello World ",[]]
4> apply(Module, Function, Arguments).
Hello World ok
5> io:Function("Hello World ", []).
Hello World ok
6> Module:Function("Hello World ", []).
Hello World ok
```

The arguments to **apply** could have been evaluated during runtime

The arities of the **M:func(Arg1, ...)** and **M:F(Arg1, ...)** forms are static

Modules: compiling

```
2> c(demo).
{ok,demo}
3> demo:double(25).
50
4> demo:times(2, 25).
** exception error: undefined function demo:times/2
5>
```

Modules: Erlang Shell

pwd().

Shows the path of the current directory.

cd(Dir).

Change directory. The directory name is a quoted string.

c(Module).

Compiles the file Module.erl and loads it into the shell.

ls(). - ls(Dir).

Lists the files in the current directory or in directory **Dir**.

q().

Quit from the shell. Shorthand for **init:stop().**

Editors

Emacs

Vim

ErlIDE

Sublime

IntelliJ

Visual Studio

TextMate

Notepad++

SciTE

Atom

Overview: Types & Constructs

- The Shell
- Data Types
- Variables
- Complex Data Structures
- Pattern Matching
- BIFs
- Function Calls
- Modules