Erlang for Java developers

an introduction to functional programming

This Talk

- for total beginners!
- much longer than intended
- History and “why” of Erlang
- Look at the language’s basic features
- Highlight some differences with Java / imperative
- Ease your learning curve
- Let’s get started...

What is Erlang?

- 20+ year old programming language.
- designed for telephone switches by Ericsson.
- which could have no downtime
- required massive parallelism to scale

These needs shaped the language...

Defining Features

- Process isolation / thread safety:
  - Immutable data structures
  - Pass by value (no references/pointers)
  - Processes communicate via messaging
- Distributed (uses same messaging approach)
- Hot code loading
- Supervision hierarchies and restart strategies
  - Code the happy path and let it fail!

Basic Data Types

Atoms

red

blue

'Really Green'

- Start with a lower case letter
- Single quoted if contain spaces or capitalized
- Used like a Java const
- But, not assigned a value

Variables

```
X = 5

FirstName = "Chris"
```

- Start with an uppercase letter
- Can take any type of value
- Dynamically Typed

"A programming language is said to be dynamically typed, when the majority of its type checking is performed at run-time as opposed to at compile-time. In dynamic typing, values have types but variables do not; that is, a variable can refer to a value of any type."

-Wikipedia

Single Assignment

- Once assigned a value, it cannot change

```
26      X = 5,
27      X = 4.
```

- Runtime "no match" error

Pattern Matching

```
26      X = 5,
27      X = 4.
```

- = in Java is an assignment operator
- In Erlang it matches the left hand side to the right
- Any unassigned variables are bound
- Anything else is compared
- 5 = 4

Complex Data Types

Lists

- Look similar to Java arrays, but function differently
- L = [1, 2, 3]
- Not indexed
- manipulate via Erlang's lists library

Lists

- Pattern matching also works for lists

```
43
44      [A, B, C] = [1, 2, 3],
45
46      io:format("B equals ~p~n", [B]),
47
```

- “B equals 2”

Lists

- The '|' operator separates a list in to a head and a tail.

```
50      [Head | Tail] = [1, 2, 3],
51
52      io:format("Head equals ~p~n", [Head]),
53      io:format("Tail equals ~p~n", [Tail]),
54
```

- “Head equals 1”
- “Tail equals [2, 3]”

Tuples

```
65      X = {1, 2, elephant},
```

- A compound data type with a number of elements
- elements can be of any type

Tuples

```
65      X = {1, 2, elephant},
66
67      Nested = {X, {name, "Chris"}, [1, 2, 3]},
```

- including other tuples
- can be nested arbitrarily deeply

Tuples

```
69      {Y, Z} = {3, 4}.
```

- and of course pattern matching
- works at any depth
- match / bind any number of elements

Binaries

```
74      Binary = <<3,4,5>>,
75
76      AnotherBinary = <<"Chris">>,
77
78      <<A, B, C/binary>> = <<1, 2, "Chris">>.
```

- low-level sequence of bits or bytes
- integers, floats, bitstrings
- pattern matching, of course
- great for processing data! (files, streaming, etc.)

Control Flow

Case statement

```
26      X = "Joe",
27      case X of
28          "Chris" ->
29              io:format("Hey Chris!~n");
30          SomeoneElse when is_list(X) ->
31              io:format("Hey, ~s have you Seen Chris?~n", [SomeoneElse]);
32          _ ->
33              error
34      end.
```

- “Hey, Joe have you seen Chris?”
- builds on the principle of pattern matching for control flow

Code Organization

taking the building blocks we have learned and using them to

put together working programs.

Java

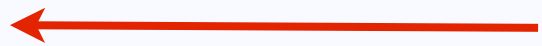
```java
public class FirstExample {

    public void sayHello(String name){

        System.out.printf("hello %s\n", name);
    }

}
```

Erlang

```erlang
-module(first_example).

-export([say_hello/1]).

say_hello(Name) ->
    io:format("hello ~s~n", [Name]).

```

Modules

```
1  -module(first_example).
2
3  -export([say_hello/1]).
4
5  say_hello(Name) ->
6      io:format("hello ~s~n", [Name]).
7
```



- In Erlang the module is the basic unit of code organization.
- Module name must match the file name.
- Modules are not instantiated in to objects

Functions

```
1  -module(first_example).
2
3  -export([say_hello/1]).
4
5  say_hello(Name) ->
6      io:format("hello ~s~n", [Name]).
7
```

- name(Args) ->
- ends with .
- exporting is equivalent of making a method public

Functions

```
47  simple_function() ->
48      X = 5,
49      Y = 4,
50      X + Y.
```

- multiple lines separated by ,
- return value is the value of the last statement: “9”

Functions

```
52  case_function("Chris") ->
53      io:format("Hey Chris!~n");
54
55  case_function(SomeoneElse) when is_list(SomeoneElse) ->
56      io:format("Hey, ~s have you Seen Chris?~n", [SomeoneElse]);
57
58  case_function(_) ->
59      error.
```

- Yes, pattern matching works for function dispatch
- same behavior as the Case example

Functions

```
61  start_print_loop() ->
62      X = [1, 2, 3],
63      print_loop(X).
64
65  print_loop([H | T]) ->
66      io:format("printing ~p~n", [H]),
67      print_loop(T);
68
69  print_loop([]) ->
70      io:format("finished printing.~n").
71
```

- recursion is used in place of iteration in list processing

OO vs Functional Programming

First Class Functions

- Functions can be assigned to variables
- Functions can be passed to other functions
- anonymous functions

Functions

```
24  first_class() ->
25      io:format("Yea, that's how I roll.~n"),
26      4 + 5.
27
28  assign_fun() ->
29      A = fun first_class/0,
30      io:format("notice, first_class hasn't been evaluated yet.~n"),
31      io:format("value: ~p~n", [A()]).
```

- evaluated inside io:format call

Functions

```
34  anon_fun() ->
35      Evens = lists:filter(fun(X) -> X rem 2 == 0 end, [1, 2, 3, 4, 5, 6]),
36      io:format("evens: ~p~n", [Evens]).
```

- “evens: [2, 4, 6]”

Concurrency

Processes

- like a lightweight thread
- started by running a single function
- can loop or call other functions
- dies when the function exits

Processes

```
24  start_simple() ->
25      spawn(fun() -> io:format("Oh hai world!") end).
```

- pass spawn a function to start a process

Process Communication

- each process has a mailbox (an ordered queue)
- function can read the first message with a ‘receive’ block
- send messages to a process with the ‘!’ operator
- processes are addressed by their Pid, returned by spawn

Processes

```
39  start_receiver() ->
40      Pid = spawn(processes, run_receiver, []),
41      Pid ! "Joe",
42      Pid ! "Chris",
43      Pid ! 5,
44      ok.
45
46  run_receiver() ->
47      receive
48          "Chris" ->
49              io:format("Hey Chris!~n"),
50              run_receiver();
51          SomeoneElse when is_list(SomeoneElse) ->
52              io:format("Hey, ~s have you Seen Chris?~n", [SomeoneElse]),
53              run_receiver();
54          X ->
55              io:format("ack, ~w? seriously?~n", [X]),
56              error
57      end.
```

That's all folks!

- We only scratched the surface...
  - lots more features: libraries, apis, OTP
  - the interactive erl shell
  - tools: Eclipse, IntelliJ, Emacs

Questions, Comments, Concerns

- Code: http://github.com/chrisduesing/erlvjava
- Contact Info:
  - email: chris.duesing@gmail.com
  - twitter: @Simergence
  - blog: http://simergence.blogspot.com