ELIXIR INTRODUCTION WORKSHOP

Elixir ▷ Asturias

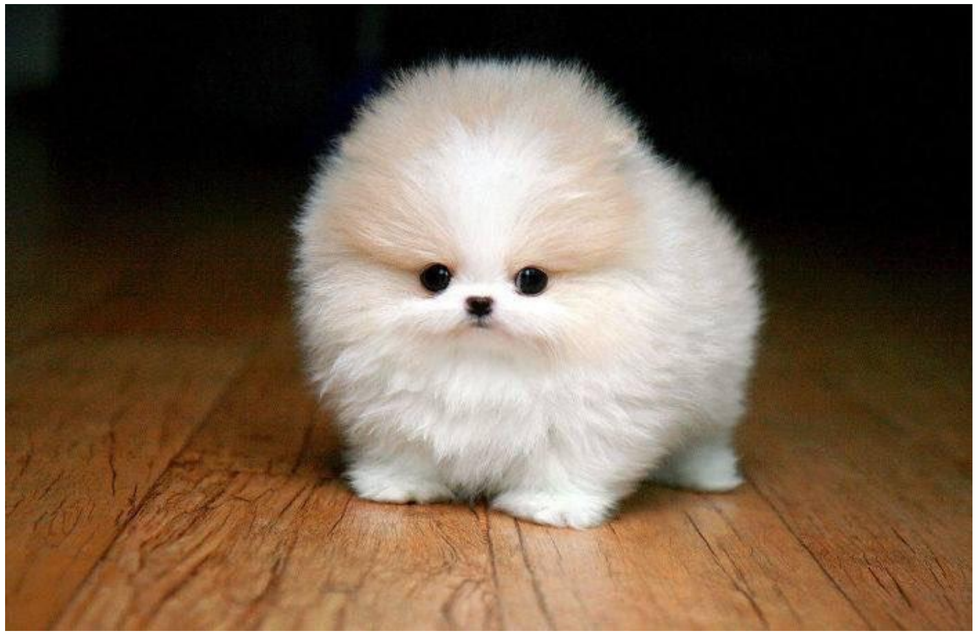
HI!

I AM IVÁN GONZÁLEZ

I **tweet** @ twitter.com/dreamingechoes

I **code** @ github.com/dreamingechoes

I **write** @ dreamingecho.es



* not an actual picture of me, but pretty close

Elixir ▷ Asturias

AGENDA

What are we going to do today?

- **Session I:** an introduction to **Elixir**, where we'll see a bit of **history**, its **main features** and **advantages** over other programming languages, as well as a series of initial concepts such as **basic types** and **operators**, **pattern matching**, **control structures**, **lists** and **maps**, etc...
- **Session II:** a bunch of **exercises** that we'll **solve together** thanks to the resources seen in the previous session.
- **Session III:** last session in which we'll develop, as a final practical exercise, a small **Twitter bot**.

SESSION I

Let’s start diving into Elixir seeing some basic concepts

HOW DID ALL THIS START?

A long time ago in a galaxy far, far away....

- Originally a proprietary language of **Ericsson**.
- Developed by **Joe Armstrong**, **Robert Virding** and **Mike Williams** in 1986.
- Designed with the aim of improving the development of telephony applications.
- The **Erlang runtime system** is known for its designs that are well suited for systems with characteristics like: **distributed**, **fault-tolerant**, **highly available**, **hot swapping**...
- The **Erlang programming language** is known for properties like: **immutable data**, **pattern matching**, **functional programming**.

THE ERLANG VIEW OF THE WORLD

As Joe Armstrong summarized in his PhD thesis

- Everything is a **process**.
- Processes are **strongly isolated**.
- Process creation and destruction is a **lightweight operation**.
- **Message passing** is the **only way** for processes to interact.
- Processes have **unique names**.
- If you know the name of a process, you can send it a message.
- Processes **share no resources**.
- Processes **do what they are supposed to do** or **fail**.

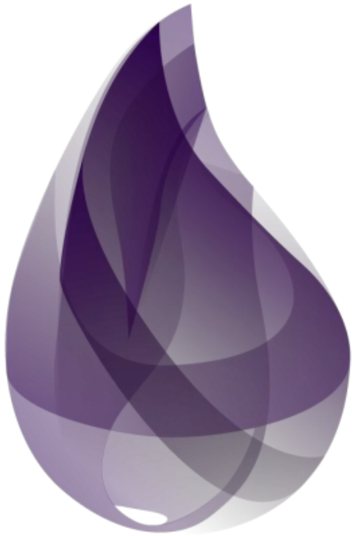
IF JAVA IS WRITE ONCE, RUN ANYWHERE, THEN ERLANG IS WRITE ONCE, RUN FOREVER.

— JOE ARMSTRONG

ELIXIR IS COMING TO TOWN

José Valim magic in action.

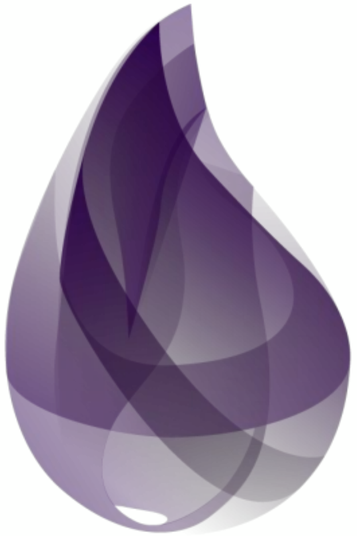
- **José Valim** is the creator of the **Elixir programming language**.
- His goals were to enable **higher extensibility** and **productivity** in the **Erlang VM** while **keeping compatibility** with **Erlang's** ecosystem.
- Builds on top of **Erlang** and shares the same abstractions for building distributed, fault-tolerant applications.
- Provides a **productive tooling** and an **extensible design**. The latter is supported by **compile-time metaprogramming** with **macros** and **polymorphism** via **protocols**.
- First appeared in **2011**, influenced by **Clojure**, **Erlang**, and **Ruby**.



SOME ELIXIR FEATURES

Why should I use Elixir?

- A language that compiles to **bytecode** for the **Erlang VM (BEAM)**.
- **Erlang** functions can be called from **Elixir** without run time impact, due to compilation to **Erlang bytecode**.
- **Meta programming** allowing direct manipulation of the **Abstract syntax tree (AST)**.
- **Shared nothing concurrent programming** via message passing.
- Emphasis on **recursion** and **higher-order functions** instead of looping.
- **Lazy** and **async** collections with **streams**.
- **Pattern matching** to promote assertive code.
- Support for documentation via **docstrings** in **Markdown**.



INTRODUCTION

INTERACTIVE MODE

https://elixir-lang.org/getting-started/introduction.html#interactive-mode

```
user@computer:$ iex
Erlang/OTP 21 [erts-10.0.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [hipe]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 40 + 2
42
iex(2)> "hello" <> " world"
"hello world"
```

RUNNING SCRIPTS

https://elixir-lang.org/getting-started/introduction.html#running-scripts

```
1 IO.puts "Hello world from Elixir Asturias!"
```

```
user@computer:$ elixir simple.exs
Hello world from Elixir Asturias!
```

BASIC TYPES

BASIC TYPES

https://elixir-lang.org/getting-started/basic-types.html

- The main **Elixir** basic types are: **integers**, **floats**, **booleans**, **atoms**, **strings**, **lists** and **tuples**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex> 1           # integer
iex> 0x1F        # integer
iex> 1.0         # float
iex> true        # boolean
iex> :atom       # atom / symbol
iex> "elixir"    # string
iex> [1, 2, 3]   # list
iex> {1, 2, 3}   # tuple
```

IDENTIFYING FUNCTIONS

https://elixir-lang.org/getting-started/basic-types.html#identifying-functions

- In **Elixir**, a function is identified by its **name** and its **arity**.
- The arity of a function is the **number of arguments** that the function takes.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> h Enum.count/1

                              def count(enumerable)

  @spec count(t()) :: non_neg_integer()

Returns the size of the enumerable.

## Examples

    iex> Enum.count([1, 2, 3])
    3

iex(2)>
```

BOOLEANS

https://elixir-lang.org/getting-started/basic-types.html#booleans

- **Elixir** supports **true** and **false** as booleans.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> true
true
iex> true == false
false
iex> is_boolean(true)
true
iex> is_boolean(1)
false
```

ATOMS

https://elixir-lang.org/getting-started/basic-types.html#atoms

- An **atom** is a constant whose name is its own value. Some other languages call these **symbols**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> :hello
:hello
iex(2)> :hello == :world
false
iex(3)> true == :true
true
iex(4)> is_atom(false)
true
iex(5)> is_boolean(:false)
true
```

STRINGS

https://elixir-lang.org/getting-started/basic-types.html#atoms

- Strings are delimited by **double quotes**, and they are encoded in **UTF-8**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> "hellö"
"hellö"
iex(2)> "hellö #{:world}"
"hellö world"
iex(3)> "hello
...> world"
"hello\nworld"
iex(4)> "hello\nworld"
"hello\nworld"
iex(5)> IO.puts "hello\nworld"
hello
world
:ok
```

ANONYMOUS FUNCTIONS

https://elixir-lang.org/getting-started/basic-types.html#anonymous-functions

- Anonymous functions can be created inline and are delimited by the keywords **fn** and **end**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex(2)> add.(1, 2)
3
iex(3)> is_function(add)
true
# check if add is a function that expects exactly 2 arguments
iex(4)> is_function(add, 2)
true
# check if add is a function that expects exactly 1 argument
iex(5)> is_function(add, 1)
false
```

LISTS

https://elixir-lang.org/getting-started/basic-types.html#linked-lists

- **Elixir** uses square brackets to specify a list of values. Values can be of any type.
- Two lists can be concatenated or subtracted using the **++/2** and **--/2** operators respectively.
- **Concatenating** to or **removing** elements from a list returns a **new list**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> [1, 2, true, 3]
[1, 2, true, 3]
iex(2)> length [1, 2, 3]
3
iex(3)> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex(4)> [1, true, 2, false, 3, true] -- [true, false]
[1, 2, 3, true]
```

TUPLES

https://elixir-lang.org/getting-started/basic-types.html#tuples

- **Elixir** uses curly brackets to specify tuples. Tuples can hold any value.
- Tuples store elements **contiguously in memory**. **Accessing a tuple** element or **getting the tuple size** is a **fast operation**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> {:ok, "hello"}
{:ok, "hello"}
iex(2)> tuple_size {:ok, "hello"}
2
iex(3)> elem(tuple, 1)
"hello"
iex(4)> put_elem(tuple, 1, "world")
{:ok, "world"}
iex(5)> tuple
{:ok, "hello"}
```

LISTS OR TUPLES?

https://elixir-lang.org/getting-started/basic-types.html#lists-or-tuples

- **Lists** are stored in memory as linked lists, meaning that each element in a list holds its value and points to the following element until the end of the list is reached. This means **accessing the length of a list is a linear operation**: we need to traverse the whole list in order to figure out its size.
- **Tuples**, on the other hand, are stored contiguously in memory. This means getting the tuple size or accessing an element by index is fast. However, **updating or adding elements to tuples is expensive**.

BASIC OPERATORS

BASIC OPERATORS

https://elixir-lang.org/getting-started/basic-operators.html

- **Arithmetic operators:** **+**, **-**, * and **/**.
- **Manipulate lists:** **++** and **--**.
- **String concatenation:** **<>**.
- **Boolean operators:** **OR**, **AND** and **NOT**. **OR** and **AND** are short-circuit operators. They only execute the right side if the left side is not enough to determine the result.
- **Comparison operators:** **==**, **!=**, **===**, **!==**, **<=**, **>=**, **<** and **>**.

PATTERN MATCHING

THE MATCH OPERATOR

https://elixir-lang.org/getting-started/pattern-matching.html#the-match-operator

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> x = 1
1
iex(2)> x
1
iex(3)> 1 = x
1
iex(4)> 2 = x
** (MatchError) no match of right hand side value: 1
iex(5)> 1 = unknown
** (CompileError) iex:5: undefined function unknown/0
```

PATTERN MATCHING

https://elixir-lang.org/getting-started/pattern-matching.html#pattern-matching-1

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex(2)> a
:hello
iex(3)> b
"world"
iex(4)> {a, b, c} = {:hello, "world"}
** (MatchError) no match of right hand side value: {:hello, "world"}
iex(5)> {a, b, c} = [:hello, "world", 42]
** (MatchError) no match of right hand side value: [:hello, "world", 42]
```

PATTERN MATCHING

https://elixir-lang.org/getting-started/pattern-matching.html#pattern-matching-1

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> {:ok, result} = {:ok, 13}
{:ok, 13}
iex(2)> result
13
iex(3)> {:ok, result} = {:error, :oops}
** (MatchError) no match of right hand side value: {:error, :oops}
iex(4)> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex(5)> head
1
iex(6)> tail
[2, 3]
```

THE PIN OPERATOR

https://elixir-lang.org/getting-started/pattern-matching.html#the-pin-operator

- The pin operator ^ allows us to **pattern match** against an existing variable's value rather than **rebinding the variable**.
- The variable _ represents a **value to be ignored in a pattern** and cannot be used in expressions.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> x = 1
1
iex(2)> ^x = 2
** (MatchError) no match of right hand side value: 2
iex(3)> {y, ^x} = {2, 1}
{2, 1}
iex(4)> y
2
iex(5)> {x, _} = {5, 6}
{5, 6}
iex(6)> x
5
```

CASE, COND AND IF

CASE

https://elixir-lang.org/getting-started/case-cond-and-if.html#case

- Allows us to compare a value against many patterns until we find a matching one.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2 in this clause"
...>   _ ->
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2 in this clause"
```

COND

https://elixir-lang.org/getting-started/case-cond-and-if.html#cond

- Allows us to check different conditions and find the first one that evaluates to **true**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

IF AND UNLESS

https://elixir-lang.org/getting-started/case-cond-and-if.html#if-and-unless

- **Elixir** also provides the macros **if/2** and **unless/2** which are useful when you need to check for **only one condition**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"
```

KEYWORD LISTS AND MAPS

KEYWORD LISTS

https://elixir-lang.org/getting-started/keywords-and-maps.html#keyword-lists

- When we have a list of tuples and the first item of the tuple (i.e. the key) is an atom, we call it a keyword list

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex(2)> list == [a: 1, b: 2]
true
iex(3)> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex(4)> [a: 0] ++ list
[a: 0, a: 1, b: 2]
iex(5)> [a: a] = [a: 1]
[a: 1]
iex(6)> a
1
```

MAPS

https://elixir-lang.org/getting-started/keywords-and-maps.html#maps

- A **key-value** store. A map is created using the **%{}** syntax.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(2)> map[:a]
1
iex(3)> map[2]
:b
iex(4)> map[:c]
nil
iex(5)> map.a
1
iex(6)> map.c
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

MODULES AND FUNCTIONS

MODULES

https://elixir-lang.org/getting-started/modules-and-functions.html

- In **Elixir** we group several functions into **modules**.
- In order to create our own modules in **Elixir**, we use the **defmodule** macro. We use the **def** macro to define functions in that module.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end
iex(2)> Math.sum(1, 2)
3
```

COMPILATION

https://elixir-lang.org/getting-started/modules-and-functions.html#compilation

- Most of the time it is convenient to write modules into files so they can be compiled and reused.
- This compilation is made using **elixirc**.
- This will generate a file named **MODULE.beam** (name of the module) containing the bytecode for the defined module. If we start **iex** again, our module definition will be available.
- When working on actual projects, the build tool called **mix** will be responsible for **compiling** and **setting up the proper paths**.

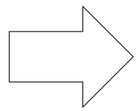
SCRIPTED MODE

https://elixir-lang.org/getting-started/modules-and-functions.html#scripted-mode

- In addition to the **Elixir** file extension **.ex**, **Elixir** also supports **.exs** files for **scripting**.
- The file will be **compiled in memory** and **executed**. **No bytecode** file will be **created**.

```elixir
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```



```
user@computer:$ elixir math.exs
```

NAMED FUNCTIONS

https://elixir-lang.org/getting-started/modules-and-functions.html#named-functions

- Inside a module, we can define functions with **def/2** and private functions with **defp/2**.
- A function defined with **def/2** can be invoked from other modules while a private function can only be invoked locally.
- Function declarations also support guards and multiple clauses.
- Giving an argument that does not match any of the clauses raises an error.

```elixir
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)    #=> 3
IO.puts Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

RECURSION

LOOPS THROUGH RECURSION

https://elixir-lang.org/getting-started/recursion.html#loops-through-recursion

- Loops in **Elixir** are written differently from imperative languages.

```elixir
defmodule Recursion do
  def print_multiple_times(msg, n) when n <= 1 do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg
    print_multiple_times(msg, n - 1)
  end
end

Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
# Hello!
```

ENUMERABLES AND STREAMS

ENUMERABLES

https://elixir-lang.org/getting-started/enumerables-and-streams.html#enumerables

- **Elixir** provides the concept of **enumerables** and the **Enum** module to work with them.
- Some enumerables we have seen are **lists** and **maps**.
- The **Enum** module provides a huge range of functions to **transform**, **sort**, **group**, **filter** and **retrieve** items from enumerables.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Enum.map(1..3, fn x -> x * 2 end)
[2, 4, 6]
iex(2)> Enum.reduce(1..3, 0, &+/2)
6
```

THE PIPE OPERATOR

https://elixir-lang.org/getting-started/enumerables-and-streams.html#the-pipe-operator

- The |> symbol is the **pipe operator**. It takes **the output from the expression on its left side** and passes it as **the first argument** to the function call on its right side.
- It's similar to the **Unix | operator**.
- Its purpose is to **highlight the data being transformed** by a series of functions.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> total_sum = 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
7500000000
```

STREAMS

https://elixir-lang.org/getting-started/enumerables-and-streams.html#streams

- As an alternative to **Enum**, **Elixir** provides the **Stream** module which supports **lazy operations**.
- Instead of generating intermediate lists, streams **build a series of computations** that are invoked only when we pass the underlying stream to the **Enum** module.
- **Streams** are useful when working with **large**, **possibly infinite**, collections.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
7500000000
```

ALIAS, REQUIRE AND IMPORT

ALIAS

https://elixir-lang.org/getting-started/alias-require-and-import.html#alias

- **Alias** allows you to set up aliases for any given module name.
- All modules defined in **Elixir** are defined inside the main **Elixir** namespace. However, for convenience, you can omit **“Elixir.”** when referencing them.
- Is **lexically scoped**, which allows you to set aliases inside **specific functions**.

```elixir
defmodule Math do
  alias Math.List, as: SomeList
  # In the remaining module definition List expands to Math.List.

  def plus(a, b) do
    alias Math.AnotherList
    # ...
  end
end
```

REQUIRE

https://elixir-lang.org/getting-started/alias-require-and-import.html#require

- **Elixir** provides **macros** as a mechanism for **meta-programming** (writing code that generates code).
- Macros are expanded at compile time.
- Public functions in modules are globally available, but in order to use macros, you need to opt-in by **requiring the module** they are defined in.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Integer.is_odd(3)
** (UndefinedFunctionError) function Integer.is_odd/1 is undefined or private. However there is a macro with
the same name and arity. Be sure to require Integer if you intend to invoke this macro
iex(2)> require Integer
Integer
iex(3)> Integer.is_odd(3)
true
```

IMPORT

https://elixir-lang.org/getting-started/alias-require-and-import.html#import

- We use **import** when we want to access **functions** or **macros** from other modules without using the **fully-qualified name**.

```elixir
defmodule Math do
  # We'll have available all the functions defined in SomeModule.
  import SomeModule

  def some_function do
    import List, only: [duplicate: 2]
    duplicate(:ok, 10)
  end
end
```

USE

https://elixir-lang.org/getting-started/alias-require-and-import.html#use

- Allows us to **inject** any code in a module, such as **importing other modules**, **defining new functions**, **setting a module state**, etc…
- For example, in order to write tests using the **ExUnit** framework, a developer should **use** the **ExUnit.Case** module.

```elixir
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    assert true
  end
end
```

STRUCTS

DEFINING STRUCTS

https://elixir-lang.org/getting-started/structs.html#defining-structs

- **Structs** are extensions built on top of **maps** that provide **compile-time checks** and **default values**.
- To define a **struct**, we use the **defstruct** construct.
- The keyword list used with **defstruct** defines what fields the struct will have along with their default values.
- **Structs** take the name of the module they're defined in.
- **Structs** provide **compile-time guarantees** that **only the fields** defined through **defstruct** will be allowed.

DEFINING STRUCTS

https://elixir-lang.org/getting-started/structs.html#defining-structs

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> defmodule User do
...>   defstruct name: "John", age: 27
...> end
iex(2)> %User{}
%User{age: 27, name: "John"}
iex(3)> %User{name: "Jane"}
%User{age: 27, name: "Jane"}
iex(4)> %User{oops: :field}
** (KeyError) key :oops not found in: %User{age: 27, name: "John"}
```

ACCESSING AND UPDATING STRUCTS

https://elixir-lang.org/getting-started/structs.html#accessing-and-updating-structs

- We can use the same techniques (and the same syntax) as **maps** to manipulate **structs**.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> john = %User{}
%User{age: 27, name: "John"}
iex(2)> john.name
"John"
iex(3)> jane = %{john | name: "Jane"}
%User{age: 27, name: "Jane"}
iex(4)> %{jane | oops: :field}
** (KeyError) key :oops not found in: %User{age: 27, name: "Jane"}
```

DEFAULT VALUES AND REQUIRED KEYS

https://elixir-lang.org/getting-started/structs.html#default-values-and-required-keys

- If we don’t specify a default key value when defining a **struct**, **nil** will be assumed.
- We can also enforce that certain keys have to be specified when creating the struct.

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe] [kernel-
poll:false]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> defmodule Car do
...>   @enforce_keys [:make]
...>   defstruct [:model, :make]
...> end
iex(2)> %Car{}
** (ArgumentError) the following keys must also be given when building struct Car: [:make]
    expanding struct: Car.__struct__/1
```

SESSION II

Practicing the concepts with some exercises

EXERCISE I

Fibonacci sequence

- Create a module with a function which implements the **Fibonacci sequence**.
  - F0 = 0
  - F1 = 1
  - Fn = Fn-1 + Fn-2

EXERCISE II

Removing case structure

- Rewrite the following module **removing** the **case** flow structure.

```elixir
defmodule Case do
  def check(argument) do
    case argument do
      %{name: name} -> IO.inspect("Your name is #{name}")
      number when is_integer(number) -> IO.inspect("You have #{number} apples")
      list when is_list(list) -> IO.inspect("The sum of the elements is #{Enum.sum(list)}")
      _ -> IO.inspect("I don't know what you say")
    end
  end
end
```

EXERCISE III

Simple calculator

- Develop a module implementing the main operations of a calculator.
  - **sum/1**
  - **sum/2**
  - **sub/2**
  - **mult/2**
  - **div/2**
- **sum/1** must accept a list of numbers and sum all its values.

EXERCISE IV

Phone agenda

- Implement a struct with the following fields:
  - **name**
  - **email**
  - **phone**
  - **address**
- Fields **name** and **email** must be required.
- All default values must be **nil**.
- Create a bunch of entries on the agenda, and implement functions to search by **name** and **email**.

SESSION III

Let’s go big! Making our first Elixir application

THANK YOU TO ALL!

AND ENJOY ELIXIR