



Michał Ślaski

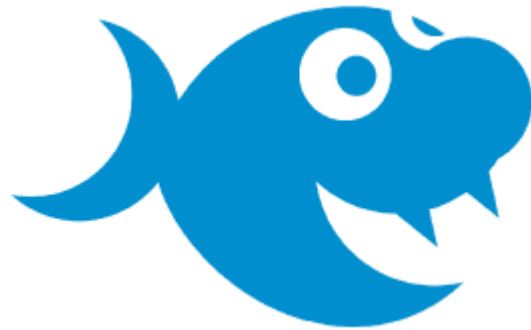
Technical Lead @ Erlang Solutions Ltd.

Wykład w KI

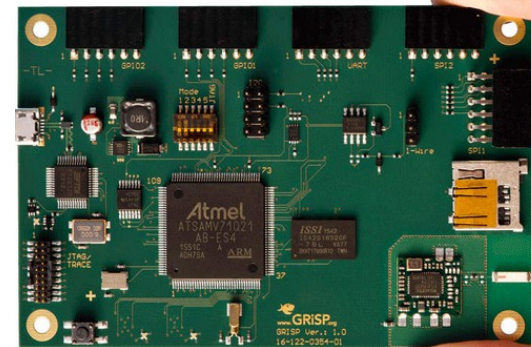
17 maja 2019, godzina 11:15
Sala 2.41

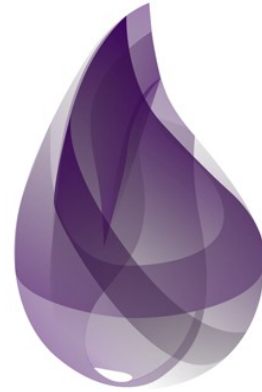


<https://twitter.com/soerlang/status/1126469409558671360>



Erlang
SOLUTIONS





elixir



Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

José Valim is the creator of the Elixir programming language, an R&D project of Plataformatec. His goals were to enable higher extensibility and productivity in the Erlang VM while **keeping compatibility with Erlang's ecosystem**





José Valim

Founder and Director of R&D
@ Plataformatec

Wykład w KI

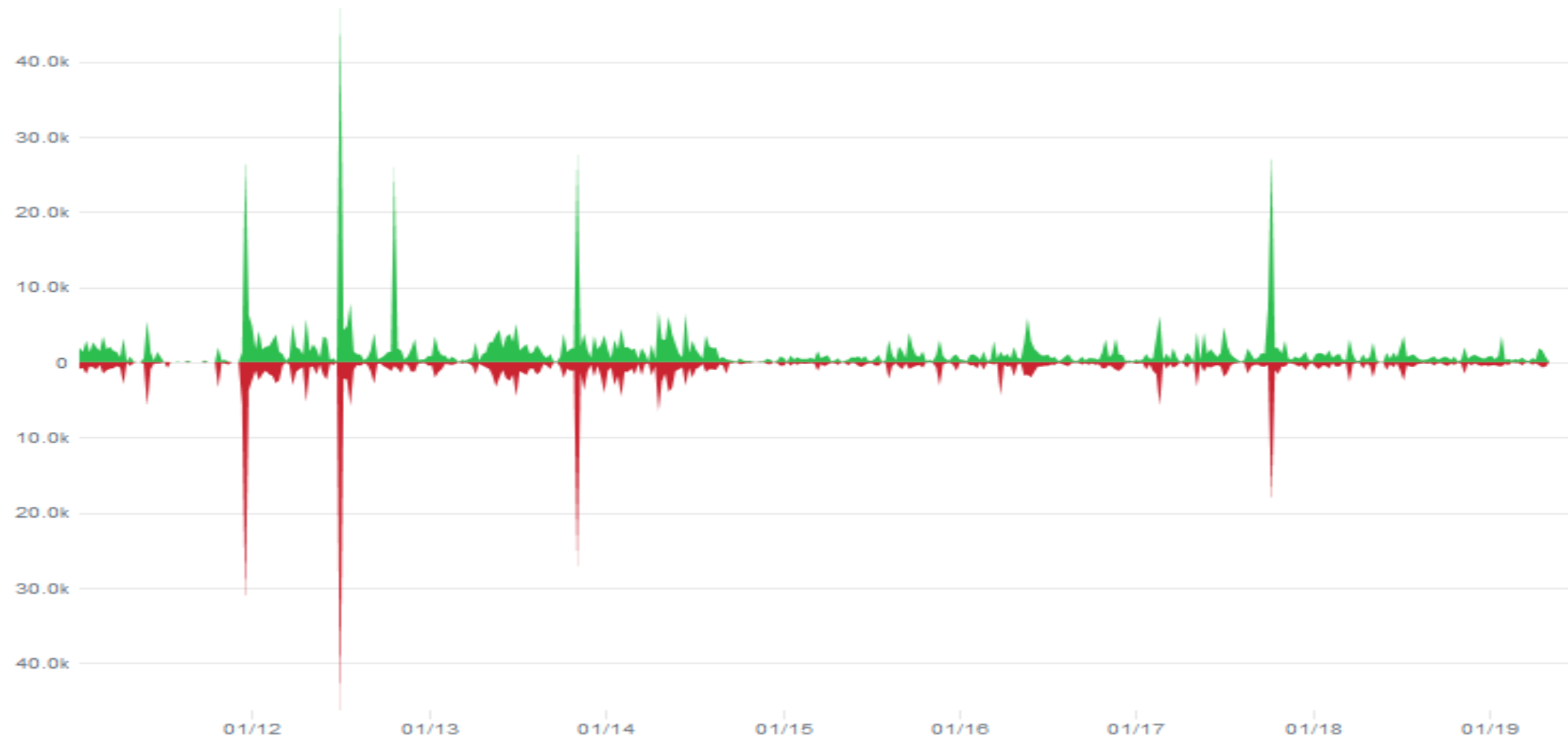
31 maja 2019, godzina 11:15
Sala 2.41



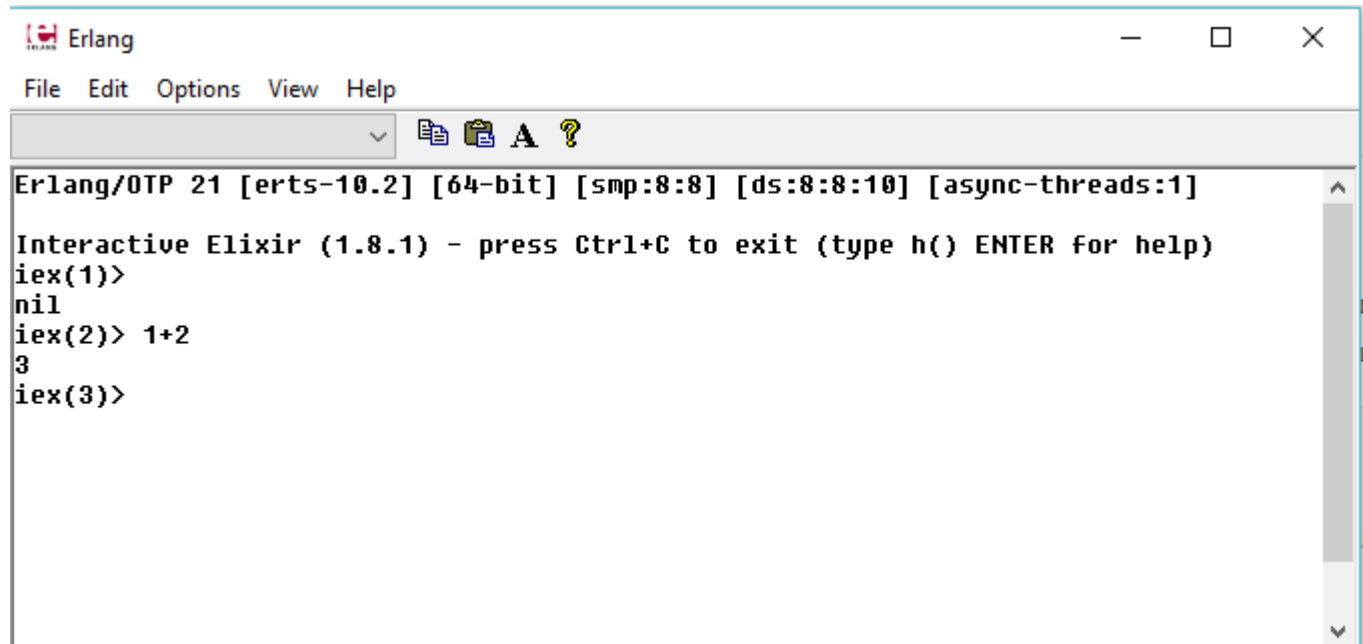
- Narodziny języka Elixir: 2011
- 2019, ver. 1.8

[Watch](#) 719
 [Star](#) 15,205
 [Fork](#) 2,189

Additions and Deletions per week



- Instalacja: <http://elixir-lang.org/install.html>
 - Kilka sposobów
- iex - interaktywny interpreter



```
Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
nil
iex(2)> 1+2
3
iex(3)>
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Podstawowe elementy składni

Podstawowe ciekawostki

- Nie ma kropek, przecinków; średników.
- Są końce linii
- Jest *nil*, czyli funkcja może zwrócić nic, czyli... ?

```
iex(14)>
nil
iex(15)>
nil
iex(16)> 2
2
iex(17)> 2 + 2
4
iex(18)> 2 +
...(18)> 3 *
...(18)> 5
17
iex(19)> (2 * 3
...(19)> )
6
```

- Są liczby
- Integer
- Float
- Dwójkowe, ósemkowe, szesnastkowe

```
iex(27)> 3 + 4.4
7.4
iex(28)> 1.0e3
1.0e3
iex(29)> 1.0e3 + 2
1002.0
iex(30)> 0b1100101001
809
iex(31)> 0o1234567
342391
iex(32)> 0xdead + 0xbeef
105884
iex(33)>
```

- Są, ale poprzedzamy je :dwukropkiem
- Chyba, że nazwiemy je Zduzejliter

```
iex(56)> :atom
:atom
iex(57)> atom
** (CompileError) iex:57: undefined function atom/0

iex(57)> :Atom
:Atom
iex(58)> is_atom(:Atom)
true
iex(59)> Atom
Atom
iex(60)> is_atom(Atom)
true
```

Boolean

- *true*, *false* - słowa kluczowe,
- równoważne atomom :true i :false

```
iex(59)> true == true
true
iex(60)> true == :true
true
iex(61)> true === :true
true
iex(62)> Atom == :Atom
false
iex(63)> :atom == :btom
false
iex(64)> true == 3
false
```

- Wydajność list znaków nie jest wystarczająca
- *String* - nowy typ, nadbudowany nad *binary*
- Lista znaków też jest: 'alamakota'

```
iex(72)> "ciąg znaków"
"ciąg znaków"
iex(73)> "ciąg znaków" <> " i jeszcze jeden"
"ciąg znaków i jeszcze jeden"
iex(74)> "ciąg
...(74)> znaków "
"ciąg \nznaków "
iex(75)> 'lista znaków'
[108, 105, 115, 116, 97, 32, 122, 110, 97, 107, 243, 119]
iex(76)> 'lista znakow' ++ ' jest'
'lista znakow jest'
iex(77)>
```

Zmienne

- Zmienne nazywamy literką małą...
- Zmienne są zmienne - mutable variables...

```
iex(94)> z = 2
2
iex(95)> y = 3
3
iex(96)> z + y
5
iex(97)> zmienna = Niezmienna
Niezmienna
iex(98)> zmienna = Awlasniezezmienna
Awlasniezezmienna
iex(99)> zmienna
Awlasniezezmienna
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the page.

Kolekcje

- Tu niewiele się zmienia

```
iex(122)> {zmienna, 2, :atomek}  
{Awlasniezezmienna, 2, :atomek}
```


- Tu niewiele się zmienia

```
iex(125)> list = [1, 2, :alamakota]
[1, 2, :alamakota]
iex(126)> list ++ [3, 4.0]
[1, 2, :alamakota, 3, 4.0]
iex(127)> [head | tail] = list
[1, 2, :alamakota]
iex(128)> head
1
iex(129)> tail
[2, :alamakota]
iex(130)> [1 | [2,3]]
[1, 2, 3]
iex(131)> [:kotamaala | list]
[:kotamaala, 1, 2, :alamakota]
```

Proplists, Keyword lists

- Cukier syntaktyczny
- Klucze to atomy

```
iex(137)> [ala: "makota", akot: "maale"]  
[ala: "makota", akot: "maale"]  
iex(138)> [{:ala, "makota"}, {:akot, "maale"}]  
[ala: "makota", akot: "maale"]
```

- Tak samo, tylko inaczej

```
iex(142)> mapa = %{:key => "value", 4 => 5}
%{4 => 5, :key => "value"}
iex(143)> mapa[4]
5
iex(144)> mapa = %{mapa | :key => "value3"}
%{4 => 5, :key => "value3"}
iex(145)> mapa = %{:key => "value", key2: "value2"}
%{key: "value", key2: "value2"}
iex(146)> mapa.key
"value"
iex(147)> mapa = %{mapa | key2: "value3"}
%{key: "value", key2: "value3"}
iex(148)> mapa = %{mapa | key3: "value4"}
** (KeyError) key :key3 not found in: %{key: "value", ...
iex(149)> Map.put(mapa, :key3, "value4")
%{key: "value", key2: "value3", key3: "value4"}
iex(150)> Map.delete(mapa, :key)
%{key2: "value3"}
```

Pattern Matching

- Działa...

```
iex(112)> x = 1
1
iex(113)> 1 = x
1
iex(114)> 2 = x
** (MatchError) no match of right hand side value: 1

iex(114)> list = [1,2,3]
[1, 2, 3]
iex(115)> [a, b, 3] = list
[1, 2, 3]
iex(119)> [a, b, 3] = [1,5,3]
[1, 5, 3]
iex(120)> [b, b, 3] = [1,5,3]
** (MatchError) no match of right hand side value: [1, 5, 3]
iex(120)> [b, ^b, 3] = [1,5,3]
[1, 5, 3]
iex(121)> b
1
```

List Comprehensions, więcej niż działa

```
iex(102)> for x <- [1,2,3,4], do: x+1  
[2, 3, 4, 5]  
  
iex(106)> for x <- 1..10, rem(x,2) == 0, do: x  
[2, 4, 6, 8, 10]  
  
iex(109)> for {k, v} <- [k1: 1, k2: 2, k3: 3], do: {k, v*v}  
[k1: 1, k2: 4, k3: 9]  
  
iex(112)> for {_k, v} <- %{:k1=>1, :two=>"two", 3=>:three}, do: v  
[:three, 1, "two"]  
  
iex(115)> for x <- [2,3], y <- [10, 20], do: {x,y,x*y}  
[{2, 10, 20}, {2, 20, 40}, {3, 10, 30}, {3, 20, 60}]  
  
iex(120)> for k <- [:a, :b, :c], v <- [10,20], do: {k,v}  
[a: 10, a: 20, b: 10, b: 20, c: 10, c: 20]  
  
iex(123)> for k <- [:a, :b, :c], v <- [10,20], into: %{}, do: {k,v}  
%{a: 20, b: 20, c: 20}
```

Warunki

if

```
iex(154)> ala = :makota
:makota
iex(155)> if ala == :mapsa do
...(155)>   "Ala ma psa"
...(155)> else
...(155)>   ala = :juzniemakota
...(155)>   "Ala miała kota"
...(155)> "
...(155)> end
"Ala miała kota\n
iex(163)> if (ala == :juzniemakota) do "nie ma kota..." end
"nie ma kota..."
iex(164)> if (ala == :makota) do "nie ma kota..." end
nil
```

- Odpowiednik *if* z Erlanga

```
iex(168)> cond do
...(168)>  ala == :makota -> "Ma kota"
...(168)>  ala == :mapsa -> "Ma psa"
...(168)>  true -> "nie ma psa, nie ma sensu..."
...(168)> end
"nie ma psa, nie ma sensu..."
```


case

```
iex(149)> a
1

iex(150)> case a do
... (150)>   {12} -> 12
... (150)>   :aa -> :aa
... (150)>   z when is_atom(z) -> z
... (150)>   _ -> :cos
... (150)> end
:cos
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Moduły i funkcje

Definiowanie funkcji

- Da się w plikach, da się też w iex

```
defmodule Hi do
  def hello(name) do
    "Hello, " <> name
  end
end

iex(170)> Hi.hello("Jan")
"Hello, Jan"
iex(171)> defmodule Abc do
... (171)>   def a(a) do
... (171)>     a * a
... (171)>   end
... (171)> end
{:module, Abc,
 <<70, 79, 82, 49, 0, 0, 4, 164, 66, 69, 65, 77, 69, 120, 68, 99,
    131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 10
    95, 118, 49, 108, 0, 0, 0, 4, 104, 2, ...>>, {:a, 1}}
```

iex(173)> Abc.a(3)
9

Definiowanie funkcji

- W jednym pliku może być kilka modułów,
- Nazwy nie muszą być zgodne

```
defmodule ModuleA do
  def a() do
    :sayA
  end
  def b() do :sayBB end
end
defmodule ModuleA.Sub do
  def b() do
    :sayB
  end
end
```

```
iex(18)> l(Filename)
{:module, Filename}
iex(19)> ModuleA.a()
:sayA
iex(20)> ModuleA.Sub.b()
:sayB
```

Funkcje eksportowane

- Wszystkie *def* są eksportowane
- *defp* nie są eksportowane (private)

```
defmodule ModuleA do
  def a() do
    :sayA
  end
  defp b() do :sayBB end
end
```

```
iex(18)> l(ModuleA)
{:module, ModuleA}
iex(19)> ModuleA.a()
:sayA
iex(20)> ModuleA.b()
** (UndefinedFunctionError) function ModuleA.b/0
   is undefined or private
ModuleA.b()
```

Ewaluacja funkcji

```
defmodule Recursion do
  def factorial(0) do 1 end
  def factorial(n) do factorial(n-1)*n end
  def makota() do :makota end
end
```

```
iex(13)> Recursion.factorial(6)
720
iex(15)> Recursion.factorial 7
5040
iex(16)> Recursion.makota()
:makota
iex(17)> Recursion.makota
:makota
```

Ewaluacja funkcji

```
defmodule Recursion do
  def niemakota do :niemakota end
  def ilekotow k do "kotow #{k} jest" end
  def ilekotowapsow k, l do "kotow #{k} jest, psow: #{l}" end
end
```

```
iex(18)> Recursion.niemakota
:niemakota
iex(19)> Recursion.niemakota()
:niemakota
```

```
iex(22)> Recursion.ilekotow 5
"kotow 5 jest"
iex(25)> Recursion.ilekotowapsow 3, 4
"kotow 3 jest, psow: 4"
```

Domyślne wartości parametrów

```
defmodule ABC do
  def fuf(a, b, c), do: a + b + c
  def fum(a, b \\ 2, c \\ 3), do: a + b + c
  def fuj(a \\ 1, b, c \\ 3), do: a + b + c
end
```

```
iex(19)> ABC.fuf(1,2,3)
```

```
6
```

```
iex(20)> ABC.fum(1,2,3)
```

```
6
```

```
iex(21)> ABC.fuj(1,2,3)
```

```
6
```

```
iex(22)> ABC.fum(1,2)
```

```
6
```

```
iex(23)> ABC.fum(1)
```

```
6
```

```
iex(24)> ABC.fuj(1,2)
```

```
6
```

```
iex(25)> ABC.fuj(1)
```

```
5
```


Funkcje anonimowe

- Są. Nie ma rekurencyjnych....

```
iex(33)> sum = fn (a, b) -> a + b end
#Function<12.52032458/2 in :erl_eval.expr/5>
iex(34)> sum(3, 4)
** (CompileError) iex:34: undefined function sum/2

iex(34)> sum.(3, 4)
7
iex(35)> sum2 = fn a, b -> a + b end
#Function<12.52032458/2 in :erl_eval.expr/5>
iex(36)> sum2.(3, 4)
7
iex(21)> up = &String.upcase/1
iex(22)> up.("alamakota")
"ALAMAKOTA"
iex(37)> sum3 = &(&1+&2)
&:erlang.+/2
iex(38)> sum3.(3, 4)
7
iex(39)> & &1 * 2 * &2
```



```
iex(67)> String.split(String.upcase("ala ma kota"))
["ALA", "MA", "KOTA"]

iex(72)> "ala ma kota" |> String.upcase |> String.split()
["ALA", "MA", "KOTA"]

to_string(Enum.map(String.to_charlist("ala ma kota"), &(&1+1)))
"bmb!nb!lpub"

iex(52)> "ala ma kota" |>
  String.to_charlist() |>
  Enum.map(&(&1+1)) |>
  to_string

"bmb!nb!lpub"
```

Atrybuty modułów

```
defmodule Makota do
  @makota "ma koteczka!"

  def ma kto do
    "#{kto} #{@makota}"
  end
end

iex(90)> Makota.ma "Alicja"
" Alicja ma koteczka!"
```

- Mapy posypane cukrem
- Tylko jedna struktura w module

```
defmodule Person do
  defstruct name: "", surname: "", sex: :unknown
end

iex(94)> %Person{}
%Person{name: "", sex: :unknown, surname: ""}
iex(96)> alamakota = %Person{name: "Ala", surname: "Makota"}
%Person{name: "Ala", sex: :unknown, surname: "Makota"}
iex(97)> alamakota.name
"Ala"
iex(98)> alamakota = %{alamakota | sex: :female}
%Person{name: "Ala", sex: :female, surname: "Makota"}
```

Integracja z Erlangiem

Beam nas połączył...

- Kod Elixira jest kompilowany do kodu maszyny Erlanga
→ Integracja to nie problem
- Elixir załaduje kod z plików .beam w dostępnych katalogach

```
iex(7)> :server.start_link(4)
Server init
{:ok, #PID<0.87.0>}
iex(8)> :server.getValue()
Server returns value
4
iex(9)> :server.incValue
Server increments value
:ok
iex(10)> :server.getValue
Server returns value
5
```

```
start_link(InitialValue) ->
    gen_server:start_link(
        {local, var_server},
        server,
        InitialValue, []).

init(InitialValue) ->
    io:format("Server init~n"),
    {ok, InitialValue}.

%% user interface
getValue() ->
    gen_server:call(var_server, {getValue}).
incValue() ->
    gen_server:cast(var_server, {incValue}).
stop() ->
    gen_server:cast(var_server, stop).

%% callbacks
handle_call({getValue}, _From, Value) ->
    io:format("Server returns value~n"),
    {reply, Value, Value}.
handle_cast({incValue}, Value) ->
    io:format("Server increments value~n"),
    {noreply, Value+1};
handle_cast(stop, Value) ->
    {stop, normal, Value}.
```

Procesy, OTP

Procesy

```
defmodule Proc do
  def printAndSpawn(0) do nil end
  def printAndSpawn(n) do
    spawn(Proc, :printAndSpawn, [n-1])
    Process.sleep(:rand.uniform(1000))
    IO.puts "terminating #{n}"
  end
end
```

```
iex(17)> Proc.printAndSpawn(6)
terminating 5
terminating 4
terminating 1
terminating 2
terminating 6
:ok
terminating 3
iex(18)>
```


- Nie ma !,
- *funk!(a)* to legalna nazwa funkcji

```
defmodule Receiver do
  def printMessage do
    receive do
      message -> IO.puts "got: #{message}"
    end
    printMessage
  end
end

iex(5)> pid = spawn(Receiver, :printMessage, [])
#PID<0.89.0>
iex(6)> send pid, "Ala Makota"
got: Ala Makota
"Ala Makota"
iex(7)>
```

OTP – jest...

```
defmodule VarServer do
  use GenServer

  def start_link(initValue \\ 0) do
    GenServer.start_link(__MODULE__, initValue,
                        name: __MODULE__)
  end

  def init(initValue) do
    IO.puts "Server init"
    {:ok, initValue}
  end

  ## user interface
  def get_value, do: GenServer.call(__MODULE__, {:get_value})
  def inc_value do
    GenServer.cast(__MODULE__, {:inc_value})
  end
  def stop do
    GenServer.cast(__MODULE__, {:stop})
  end

  ## callbacks
  def handle_call({:get_value}, _from, value) do
    IO.puts "Server returns value"
    {:reply, value, value}
  end
  def handle_cast({:inc_value}, value) do
    IO.puts "Server increments value"
    {:noreply, value + 1}
  end
  def handle_cast({:stop}, value) do
    {:stop, :normal, value}
  end
  def terminate(reason, value) do
    IO.puts "exit with value #{value}"
    reason
  end
end
```

```
-module(server).
-behaviour(gen_server).

-export([start_link/1, init/1, handle_call/3,
        handle_cast/2, terminate/2]).
-export([stop/0, get_value/0, inc_value/0]).

start_link(InitialValue) ->
  gen_server:start_link(
    {local, var_server},
    server,
    InitialValue, []).

init(InitialValue) ->
  io:format("Server init~n"),
  {ok, InitialValue}.

%% user interface
get_value() ->
  gen_server:call(var_server, {get_value}).
inc_value() ->
  gen_server:cast(var_server, {inc_value}).
stop() ->
  gen_server:cast(var_server, stop).

%% callbacks
handle_call({get_value}, _from, Value) ->
  io:format("Server returns value~n"),
  {reply, Value, Value}.
handle_cast({inc_value}, Value) ->
  io:format("Server increments value~n"),
  {noreply, Value+1};
handle_cast(stop, Value) ->
  {stop, normal, Value}.
terminate(Reason, Value) ->
  io:format("exit with value ~p~n", [Value]),
  Reason.
```

Makro wstrzykujące kod do modułu

```
defmodule VarServer do
  use GenServer

  def start_link(initValue \\ 0) do
    GenServer.start_link(__MODULE__, initValue,
                        name: __MODULE__)
  end

  def init(initValue) do
    IO.puts "Server init"
    {:ok, initValue}
  end

  ## user interface
  def get_value, do: GenServer.call(__MODULE__, [:get_value])
```

Domyślna wartość parametru

Cukier, równoważny do
[name: __MODULE__]
czyli
[{:name, __MODULE__}]

Skrótowny zapis funkcji
z jedną instrukcją

Protokoły, Enum, Stream

- Próba wprowadzenia interfejsów do języka
- Implementacja tych funkcji dla różnych modułów

```
defprotocol Empty do
  def empty?(t)
end
defimpl Empty, for: List do
  def empty?(l), do: length(l) == 0
end
defimpl Empty, for: Map do
  def empty?(m), do: map_size(m) == 0
end
iex(66)> Empty.empty?([])
true
iex(67)> Empty.empty?({})
** (Protocol.UndefinedError) protocol Empty not implemented
    iex:61: Empty.impl_for!/1
    iex:63: Empty.empty?/1
iex(67)> Empty.empty?(%{})
true
```

- Próba wprowadzenia interfejsów do języka
- Implementacja tych funkcji dla różnych modułów

```
defmodule User do
  defstruct [:name, :surname, :age]
end
defimpl Empty, for: User do
  def empty?(%User{name: nil, surname: nil}), do: true
  def empty?(_), do: false
End

iex(1)> Empty.empty?(%User{age: 11})
true
iex(2)> Empty.empty?(%User{name: "ala", surname: "ola"})
false
```

- *Enumerable* used by Enum and Stream modules to take values out of collections
- *Collectable* used by Enum and Stream modules to insert enumerables into collections
- *Inspect* used to transform a data structure into a textual form that is easily readable
- *List.Chars* used for converting a data structure to a charlist
- *String.Chars* used for converting a data structure to a string

- Czyli abstrakcja nad funkcjami z modułów lists, maps, ...
- all?
- any?
- chunk
- chunk_by
- map_every
- each
- map
- min
- max
- reduce
- sort

```
iex(76)> Enum.any?(["foo", "bar", "hello"],  
... (76)> fn(s) -> String.length(s) == 5 end)  
true
```

```
iex(77)> Enum.map([0, 1, 2, 3],  
... (77)> fn(x) -> x - 1 end)  
[-1, 0, 1, 2]
```

```
iex(81)> Enum.reduce(  
... (81)> %{1=>"A", 2=>"l", 3=>"a"},  
... (81)> {"", 0},  
... (81)> fn ({k,v}, {s,n}) -> {s<>v, n+k} end)  
{"Ala", 6}
```


- Stream, czyli leniwy Enum

```
iex(77)> Enum.map([0, 1, 2, 3],  
... (77)> fn(x) -> x - 1 end)  
[-1, 0, 1, 2]
```

```
iex(87)> Stream.map([0, 1, 2, 3],  
... (87)> fn(x) -> x - 1 end)  
#Stream<[enum: [0, 1, 2, 3], funs: [#Function<47.687886....
```

```
iex(98)> 1..5 |> Stream.filter(& &1 > 2) |>  
... (98)> Stream.map(fn x -> x*2 end) |>  
... (98)> Enum.sum  
24
```

Więcej... dużo więcej...

<https://elixirschool.com/en/lessons/basics/basics/>

<http://elixir-lang.org/getting-started/introduction.html>

<https://learnxinyminutes.com/docs/elixir/>

- Prezentacje mikro-projektów (3 punkty)
- Brak kartkówki i zadania
- Mikro-projekty dwuosobowe - Lightning talk
 - 12 minut
 - Prezentacja + pokaz na żywo
 - Podłączanie projektora...
 - Oceniamy wkład pracy i jakość prezentacji
- Zapisy na tematy
 - Środa, 15 maja, godz 15:00
 - <https://tinyurl.com/y2abnhku>
 - Temat max 1 raz w grupie, max 3 razy na roku
 - Własne tematy...

