Elixir

CSLP 2023/24

Diogo Marto - 108298
João Dourado - 108636
Tiago Pereira - 108546

Tabela de Conteúdos

1 Introdução

2 Pontos Principais

3 Sintaxe e Semântica

💡Programação Acompanhada

1 Introdução

SOBRE ELIXIR

- *Criada por José Valim*
- *Desenvolvimento começou em 2012*
- *Lançamento oficial em Setembro de 2014*
- *Baseado na Erlang BEAM VM.*

ERLANG BEAM VM

Vantagens:

- Sistemas Distribuídos
- Garbage collector
- Portabilidade
- *Hot code loading*

Objetivos

- Produtividade
- Aplicações Robustas
- Compatível com o paradigma
- atual (high-scalability)

2. Pontos Principais

Funcional

Syntax inspirada em Ruby

Tooling

Vasta gama de diferentes ferramentas (Mix,Hex, IEx, ...)

Concorrência + Paralelismo

Programação abstraída e simplificada de conceitos complexos

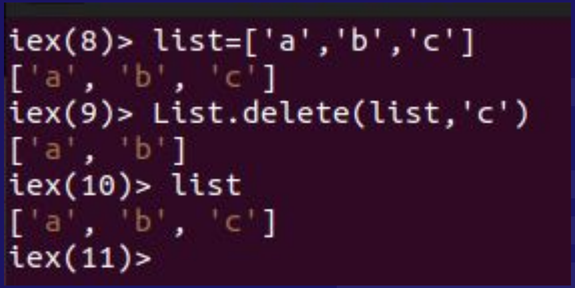
Escalável

Oferece suporte ímpar a soluções de grande escala

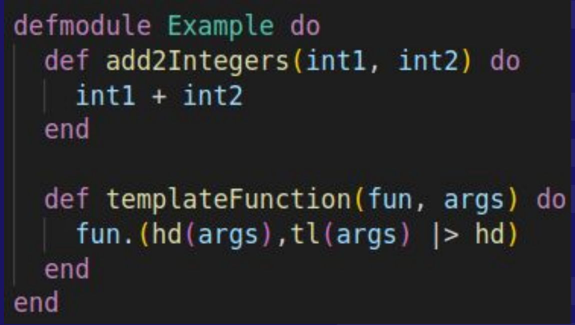
</> 2 → Funcional

- Imutabilidade de Informação

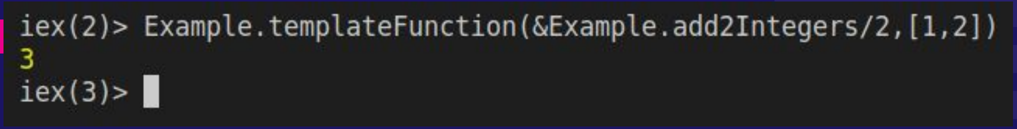
- Funções de 1ª classe



```
iex(8)> list=['a','b','c']
['a', 'b', 'c']
iex(9)> List.delete(list,'c')
['a', 'b']
iex(10)> list
['a', 'b', 'c']
iex(11)>
```



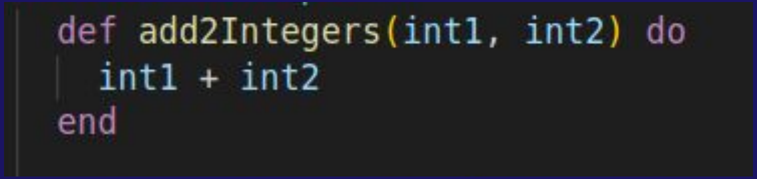
```
defmodule Example do
  def add2Integers(int1, int2) do
    int1 + int2
  end

  def templateFunction(fun, args) do
    fun.(hd(args),tl(args) |> hd)
  end
end
```



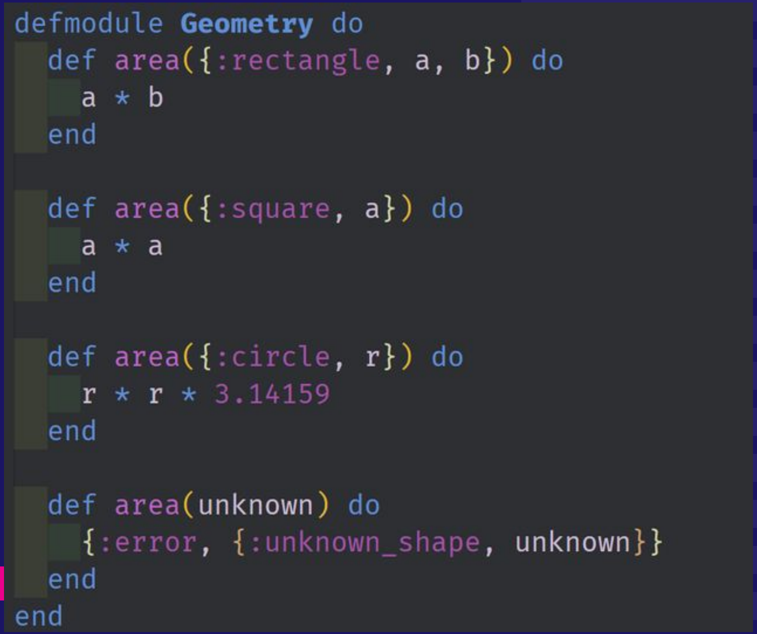
```
iex(2)> Example.templateFunction(&Example.add2Integers/2,[1,2])
3
iex(3)>
```

</> 2 → Funcional

- Funções Puras
- Correspondência de Padrões



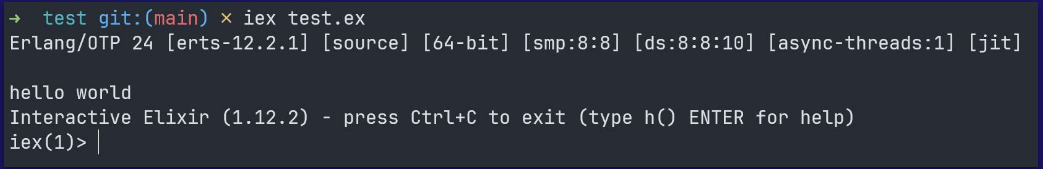
```
def add2Integers(int1, int2) do
  int1 + int2
end
```



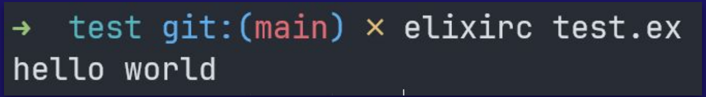
```
defmodule Geometry do
  def area({:rectangle, a, b}) do
    a * b
  end

  def area({:square, a}) do
    a * a
  end

  def area({:circle, r}) do
    r * r * 3.14159
  end

  def area(unknown) do
    {:error, {:unknown_shape, unknown}}
  end
end
```

2 → Tooling – Compiler/Interpreter

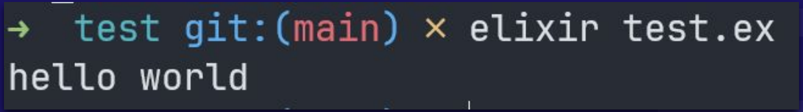
- **iex:** interactive shell
- **elixir:** interpreter
- **elixirc**: compiler - compila para BEAM VM bytecode e executa



```
➜  test git:(main) ✗ iex test.ex
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit]

hello world
Interactive Elixir (1.12.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```



```
➜  test git:(main) ✗ elixirc test.ex
hello world
```



```
➜  test git:(main) ✗ elixir test.ex
hello world
```

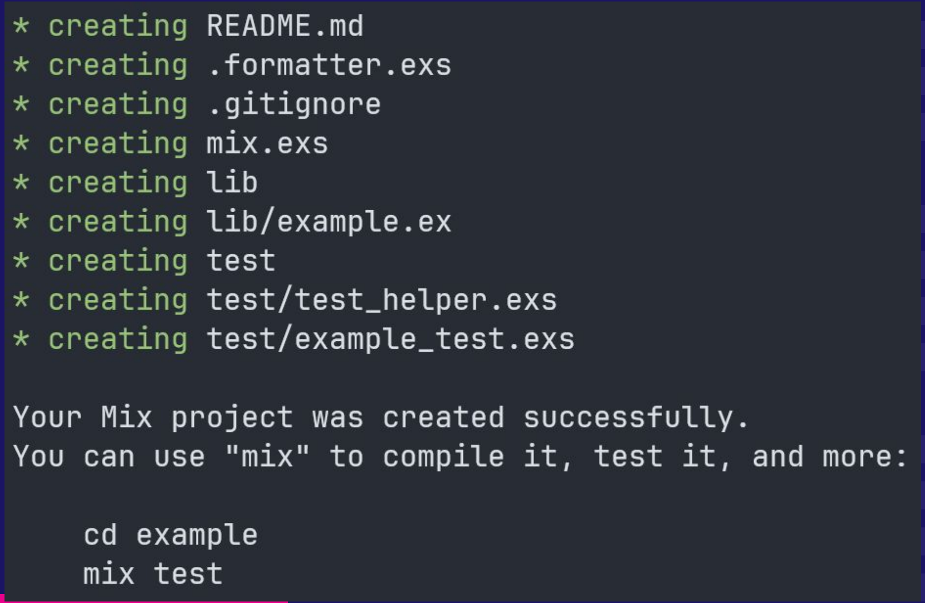
</> 2 → Tooling – Mix

- Gestor de projetos
- Build Tool
- Gestor de dependências
    - *dev*, *prod*, *test* environments
- Rico em features

</> 2 → Tooling – Mix

Criar projeto:

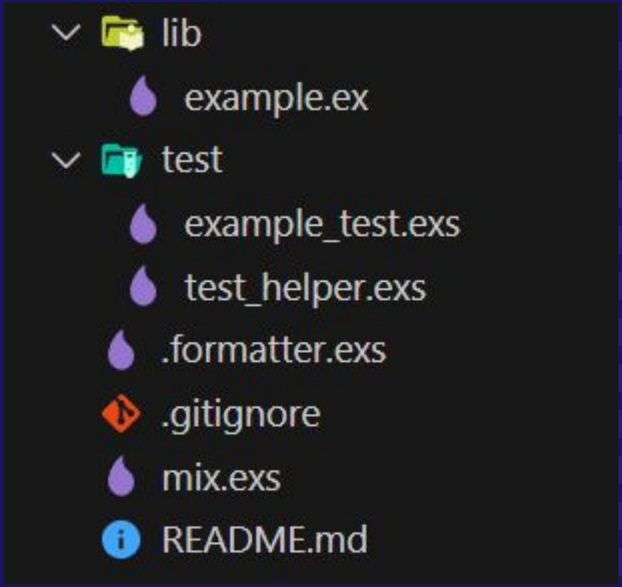
```
mix new example
cd example
```



```
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/example.ex
* creating test
* creating test/test_helper.exs
* creating test/example_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd example
    mix test
```

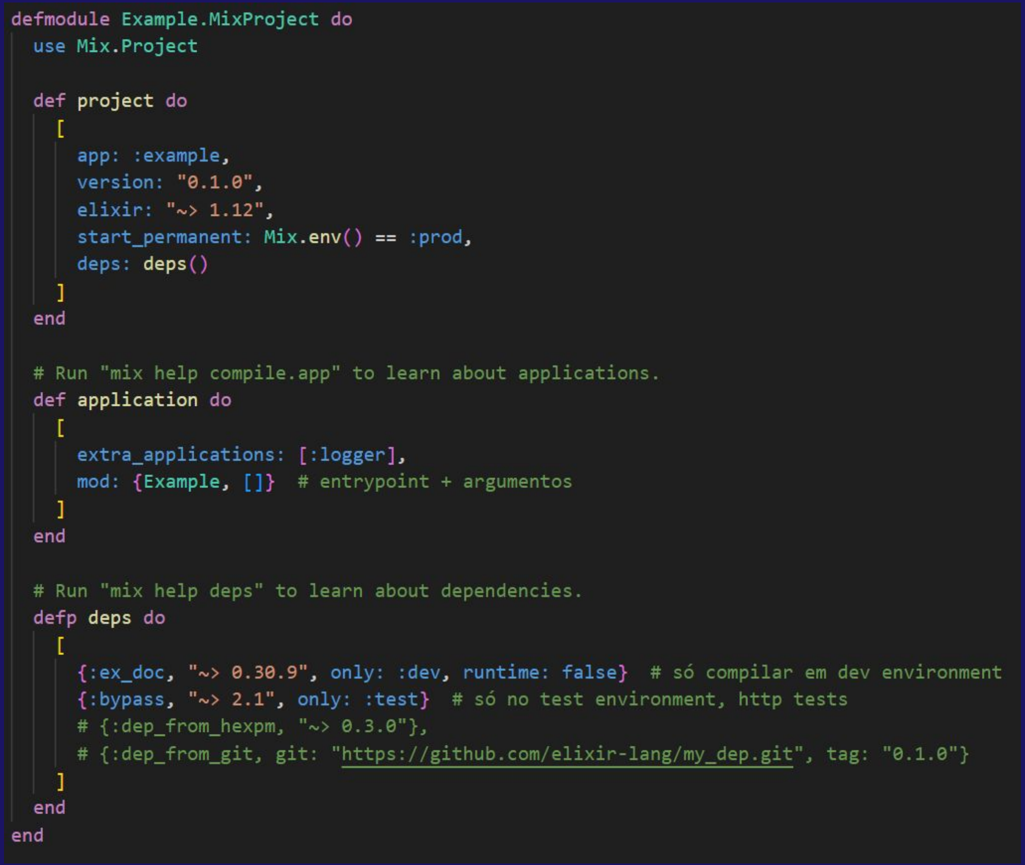
</> 2 → Mix - structure

- **Lib**: source code
- **Test**: test files
- **mix.exs**: especificação do projeto (dependencias, entrypoint, etc...)



```
lib
  example.ex
test
  example_test.exs
  test_helper.exs
.formatter.exs
.gitignore
mix.exs
README.md
```

mix.exs



```elixir
defmodule Example.MixProject do
  use Mix.Project

  def project do
    [
      app: :example,
      version: "0.1.0",
      elixir: "~> 1.12",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger],
      mod: {Example, []}  # entrypoint + argumentos
    ]
  end

  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      {:ex_doc, "~> 0.30.9", only: :dev, runtime: false}  # só compilar em dev environment
      {:bypass, "~> 2.1", only: :test}  # só no test environment, http tests
      # {:dep_from_hexpm, "~> 0.3.0"},
      # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git", tag: "0.1.0"}
    ]
  end
end
```

Mudar environment:

MIX_ENV=test mix compile

4

Sintaxe e semântica

Tópicos

- Sintaxe básica
- Pattern Matching
- Modulos
- Funções
- Colleções
- Erros
- Metaprogramação
- Concorrência
- Mix
- Concorrência OTP

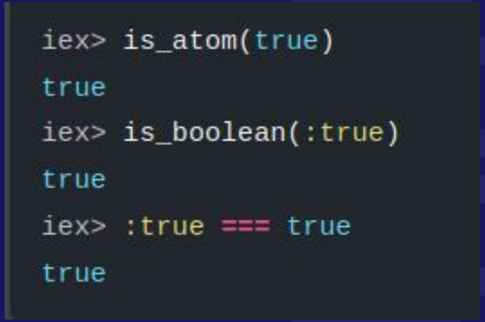
</> 3 → Basics

Data Types:

Integers, Floats, Boolean, Atom ...

Atom:

Um “atom” é uma constante cujo nome é o seu valor.

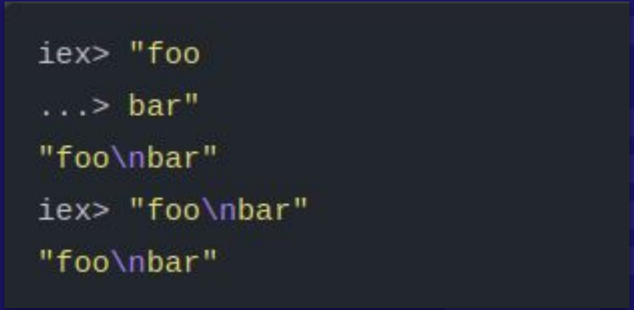


```
iex> is_atom(true)
true
iex> is_boolean(:true)
true
iex> :true === true
true
```

</> 3 → Basics

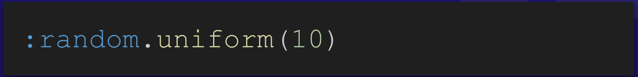
Strings:

- Envoltas em “ ” (double quotes)
- Suportam mudanças de linha



```
iex> "foo
...> bar"
"foo\nbar"
iex> "foo\nbar"
"foo\nbar"
```

Interoperabilidade com o erlang:



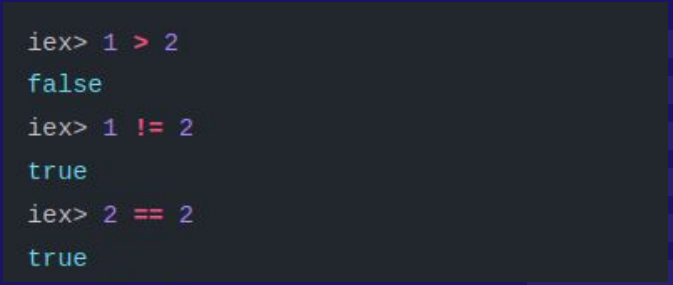
```
:random.uniform(10)
```

</> 3 → Basics

Aritmética/Lógica:

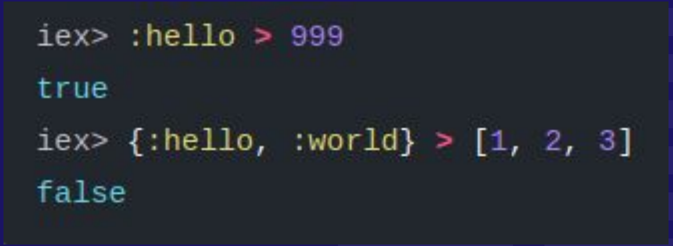
+ - * / || && ! and or not

== === < > >= <=



```
iex> 1 > 2
false
iex> 1 != 2
true
iex> 2 == 2
true
```

number < atom < reference < function < port < pid < tuple < map < list < bitstring



```
iex> :hello > 999
true
iex> {:hello, :world} > [1, 2, 3]
false
```

EXERCISE

Implementar um programa que simule o lançamento de um dado e aja de acordo.

No github: exercises/rolldie.ex

3 → Pattern Matching
→ Match Operator

```
x = 1

1 = x

2 = x
# >> (MatchError) no match of right
hand side value: 2
```

```
list = [1, 2, 3]
[1 | tail] = list
tail
# >> [2, 3]
[2 | _ ] = list
# >> ** (MatchError) no match of
right hand side value: [1, 2, 3]

{:ok, value} = {:ok, "Successful!"}
value
# >> "Successful!"
{:ok, value} = {:error}
# >> ** (MatchError) no match of
right hand side value: {:error}
```

</> 3 → Pattern Matching → Case and With

```
case {:ok, "Hello World"} do
 {:ok, result} -> result
 {:error} -> "Uh oh!"
 _ -> "Catch all"
end
# >> "Hello World"
```

```
case {1, 2, 3} do
 {1, x, 3} when x > 0 ->
   "Will match"
 _ ->
   "Won't match"
end
# >> "Will match"
```

```
user = %{first: "Sean", last:"Callan"}
case Map.fetch(user, :first) do
 {:ok, first} ->
   case Map.fetch(user, :last) do
     {:ok, last} ->
       last <> ", " <> first
     error ->
       error
   end
 error ->
   error
end

with {:ok, first} <- Map.fetch(user, :first),
    {:ok, last} <- Map.fetch(user, :last),
    do: last <> ", " <> first
# >> "Callan, Sean"
```

3 → Pattern Matching → Pin Operator

```elixir
x = 1
^x = 2
# >> ** (MatchError) no match of
right hand side value: 2
```

```elixir
x = 1
{x, ^x} = {2, 1}
# >> {2, 1}
```

```elixir
greeting = "Hello"
greet = fn
  (^greeting, name) -> "Hi #{name}"
  (greeting, name) -> "#{greeting},
#{name}"
end
```

EXERCISE

Implementar um programa que abra um ficheiro e agir consoante o resultado.

No github: exercises/openingfile.ex

3 → Modulos → Definição

```
defmodule Example do
 def greeting(name) do
   "Hello #{name}."
 end
end

Example.greeting "Sean"
# >> "Hello Sean."
```

```
defmodule Example.Greetings do
 def morning(name) do
   "Good morning #{name}."
 end

 def evening(name) do
   "Good night #{name}."
 end
end

Example.Greetings.morning "Sean"
# >> "Good morning Sean."
```

3 → Modulos → Attributes

```
defmodule Example do
 @greeting "Hello"

 def greeting(name) do
   ~s(#{@greeting} #{name}.)
 end
end
```

</3 → Modulos
→ Structs

```elixir
defmodule Example.User do
 defstruct name: "Sean", roles: []
end
```

```elixir
%Example.User{}
# >> %Example.User<name: "Sean", roles: []>
%Example.User{name: "Steve"}
# >> %Example.User<name: "Steve", roles: []>
%Example.User{name: "Steve", roles: [:manager]}
# >> %Example.User<name: "Steve", roles: [:manager]>
```

```elixir
%{name: "Sean"} = sean
# >> %Example.User<name: "Sean",
roles: [...]>
```

3 → Modulos → Composition

```
defmodule Sayings.Greetings do
 def basic(name), do: "Hi, #{name}"
end
```

```
defmodule Example3 do
 alias Sayings.Greetings, as: Hi
 def print_message(name), do:
Hi.basic(name)
end
```

```
last([1, 2, 3])
# >> ** (CompileError) iex:9:
undefined function `last/1`
import List
last([1, 2, 3])
# >> 3
```

```
import List, only: [last: 1]
```

3 → Modulos → Composition

```elixir
defmodule Example5 do
 require SuperMacros

 SuperMacros.do_stuff
end
```

```elixir
defmodule Hello do
 defmacro __using__(_opts) do
   quote do
     def hello(name), do: "Hi, #{name}"
   end
 end
end
```

```elixir
defmodule Example6 do
 use Hello
end

Example6.hello("Sean")
```

3 → Funções
→ Anonymous Functions

```
sum = fn (a, b) -> a + b end
sum.(2, 3)
# >> 5
```

```
sum = &(&1 + &2)
sum.(2, 3)
# >> 5
```

3 → Funções → Named Functions

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

Greeter.hello("Sean")
# >> "Hello, Sean"
```

```elixir
defmodule Greeter do
 def hello(), do: "Hello, anonymous person!"
 def hello(name), do: "Hello, " <> name
 def hello(name1, name2), do: "Hello, #{name1} and #{name2}"
end
```

```elixir
defmodule Greeter do
 def hello(name, language_code \\
"en") do
   phrase(language_code) <> name
 end

 defp phrase("en"), do: "Hello, "
 defp phrase("es"), do: "Hola, "
end
```

3 → Funções → Pattern Matching

```
handle_result = fn
 {:ok, result} -> IO.puts "Handling result..."
 {:ok, _} -> IO.puts "This would be never run as previous will be matched beforehand."
 {:error} -> IO.puts "An error has occurred!"
end
handle_result.({:ok, 1})
# >> Handling result...
handle_result.({:error})
# >> An error has occurred!
```

```
defmodule Length do
 def of([]), do: 0
 def of([_ | tail]), do: 1 +
of(tail)
end


Length.of []
# >> 0
Length.of [1, 2, 3]
# >> 3
```

3 → Funções → Guards

```elixir
defmodule Greeter do
 def hello(names) when is_list(names) do
   names = Enum.join(names, ", ")
   hello(names)
 end
 def hello(name) when is_binary(name) do
   phrase() <> name
 end
 defp phrase, do: "Hello, "
end

Greeter.hello ["Sean", "Steve"]
# >> "Hello, Sean, Steve"
```

EXERCISE

Implementar um função que retorna o valor do nº número de fibonacci.

No github: exercises/fibonacci.ex

3 → Coleções → List

```
list = [3.14, :pie, "Apple"]
# Prepending (fast)
["π" | list]
# >> ["π", 3.14, :pie, "Apple"]
# Appending (slow)
list ++ ["Cherry"]
# >> [3.14, :pie, "Apple",
"Cherry"]
```

```
[1, 2] ++ [3, 4, 1]
# >> [1, 2, 3, 4, 1]
["foo", :bar, 42] -- [42, "bar"]
```

```
hd [3.14, :pie, "Apple"]
# >> 3.14
tl [3.14, :pie, "Apple"]
# >> [:pie, "Apple"]
[head | tail] = [3.14, :pie,
"Apple"]
# >> [3.14, :pie, "Apple"]
head
# >> 3.14
tail
# >> [:pie, "Apple"]
```

3 → Coleções → Tuples

```
tuple = {3.14, :pie, "Apple"}

elem(tuple, 1) # fast unlike list
# >> "hello"
```

```
File.read("path/to/existing/file")
# >> {:ok, "... contents ..."}
File.read("path/to/unknown/file")
# >> {:error, :enoent}
```

3 → Coleções
→ Keyword List

```
[foo: "bar", hello: "world"]
# >> [foo: "bar", hello: "world"]
[{:foo, "bar"}, {:hello, "world"}]
# >> [foo: "bar", hello: "world"]
```

```
if(1==1,do: 1,else: 2)
# >> 1
if( 1==1 , [{ :do , 1 },{:else , 2}] )
# >> 1
```

3 → Coleções

→ Map

```elixir
map = %{:foo => "bar", "hello" => :world}
# >> %{:foo => "bar", "hello" => :world}
map[:foo]
# >> "bar"
map["hello"]
# >> :world
```

```elixir
map = %{foo: "bar", hello: "world"}
# >>  %{foo: "bar", hello: "world"}
map.hello
# >> "world"
```

```elixir
%{map | foo: "baz"}
# >> %{foo: "baz", hello: "world"}
Map.put(%{:a => 1, 2 => :b}, :c, 3)
# >> %{2 => :b, :a => 1, :c => 3}
```

</> 3 → Coleções → Enum

```
Enum.each(["one", "two", "three"],
fn(s) -> IO.puts(s) end)
# >> one
# >> two
# >> three
Enum.map([0, 1, 2, 3], fn(x) -> x -
1 end)
# >> [-1, 0, 1, 2]
Enum.filter([1, 2, 3, 4], fn(x) ->
rem(x, 2) == 0 end)
# >> [2, 4]
Enum.reduce([1, 2, 3], fn(x, acc)
-> x + acc end)
# >> 6
```

```
Enum.all?(["foo", "bar", "hello"], fn(s) ->
String.length(s) == 3 end)
# >> false
Enum.any?(["foo", "bar", "hello"], fn(s) ->
String.length(s) == 5 end)
# >> true
Enum.chunk_every([1, 2, 3, 4, 5, 6], 2)
# >> [[1, 2], [3, 4], [5, 6]]
Enum.chunk_by(["one", "two", "three", "four", "five",
"six"], fn(x) -> String.length(x) end)
# >> [["one", "two"], ["three"], ["four", "five"],
["six"]]
Enum.sort([:foo, "bar", Enum, -1, 4])
# >> [-1, 4, Enum, :foo, "bar"]
Enum.sort([%{:val => 4}, %{:val => 1}], fn(x, y) ->
x[:val] > y[:val] end)
# >> [%{val: 4}, %{val: 1}]
```

3 → Coleções → Comprehensions

```
list = [1, 2, 3, 4, 5]
for x <- list, do: x*x
# >> [1, 4, 9, 16, 25]
```

```
for {:ok, val} <- [ok: "Hello", error: "Unknown", ok: "World"],
do: val
# >> ["Hello", "World"]
```

EXERCISE

Implementar um programa multiplique todos os números de uma lista.

No github: exercises/multiplylist.ex

</> 3 → Erros
→ Try/rescue

```
raise "Oh no!"
# >> ** (RuntimeError) Oh no!
raise ArgumentError, message:
"the argument value is invalid"
# >> ** (ArgumentError) the
argument value is invalid
```

```
try do
 raise "Oh no!"
rescue
 e in RuntimeError -> IO.puts("An error occurred: " <>
e.message)
end
# >> An error occurred: Oh no!
```

```
{:ok, file} = File.open("tryrescue.ex")
try do
 # Do hazardous work
after
 File.close(file)
end
```

3 → Erros
→ New erros

```elixir
defmodule ExampleError do
 defexception message: "an example
error has occurred"
end
```

```elixir
try do
 raise ExampleError
rescue
 e in ExampleError -> e
end
# >> %ExampleError{message: "an example error has occurred"}
```

3 → Metaprogramming → Quote/Unquote

```
denominator = 2
quote do: divide(42, denominator)
# >> {:divide, [], [42, {:denominator, [], Elixir}]}
```

```
quote do: divide(42, unquote(denominator))
# >> {:divide, [], [42, 2]}
```

3 → Metaprogramming
→ Macros

```
defmodule OurMacro do
 defmacro unless(expr, do: block) do
   quote do
     if !unquote(expr), do: unquote(block)
   end
 end
end
```

```
require OurMacro
OurMacro.unless true, do: "Hi"
# >> nil
OurMacro.unless false, do: "Hi"
# >> "Hi"
```

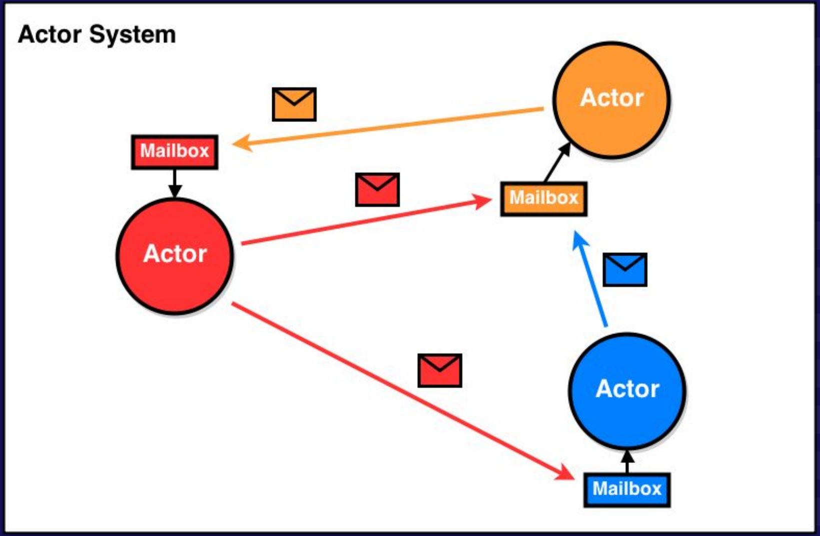
Concorrência e tolerância a falhas

</> 2 → Concorrência

Actor - entidade principal de computação concurrente

Ações do actor:
- Criar actor
- Enviar mensagem a actor
- Receber mensagens e agir

Propriedades de actors:
- Caixa de correio
- Estado interno
- Isolados entre si



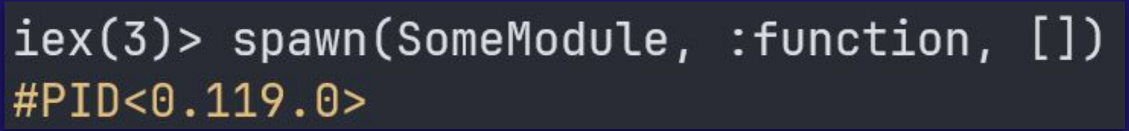
Actor Model

2 → Concorrência

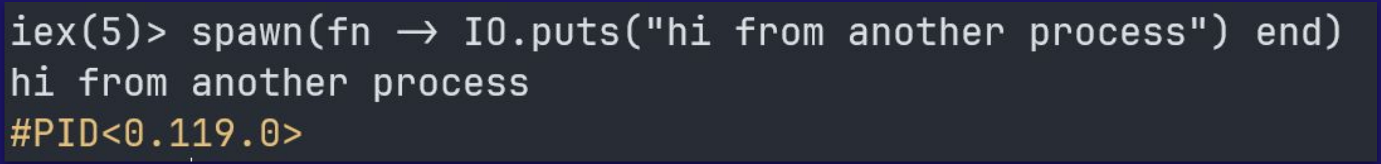
- BEAM VM implementa actors através de processes
- Lightweight processes: easy start-up, easy to take down.
- Comum haver milhões a correr num sistema de larga escala
- Não há threads, tudo são processos independentes

2 → Primitivas de concorrência

- **Spawn:** Cria processo que corre concorrentemente ao caller.

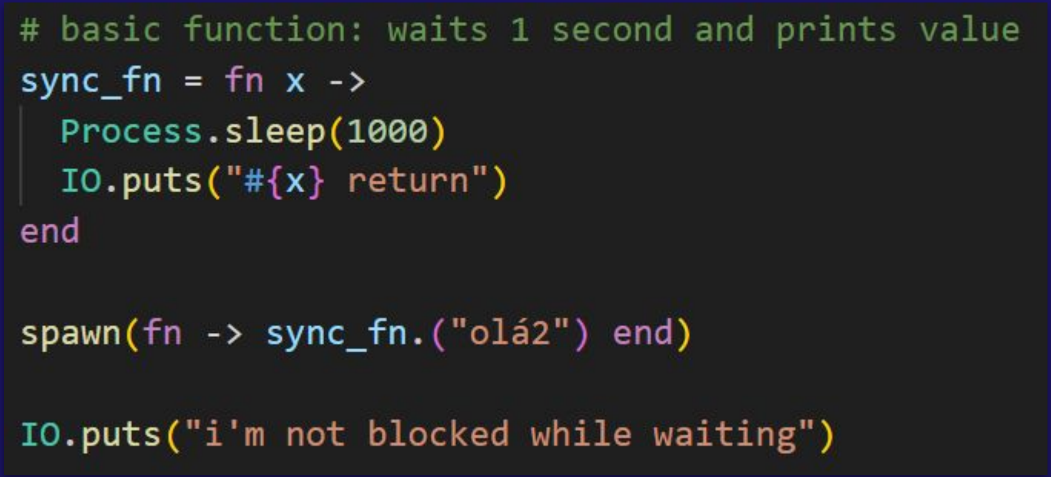


```
iex(3)> spawn(SomeModule, :function, [])
#PID<0.119.0>
```



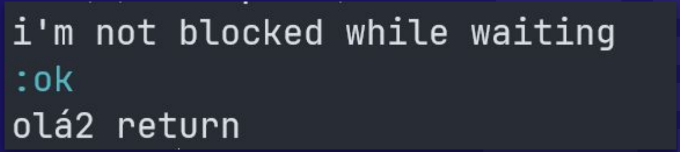
```
iex(5)> spawn(fn → IO.puts("hi from another process") end)
hi from another process
#PID<0.119.0>
```

</> 2 → Exemplo “spawn”



```elixir
# basic function: waits 1 second and prints value
sync_fn = fn x ->
  Process.sleep(1000)
  IO.puts("#{x} return")
end

spawn(fn -> sync_fn.("olá2") end)

IO.puts("i'm not blocked while waiting")
```

Output:



```
i'm not blocked while waiting
:ok
olá2 return
```

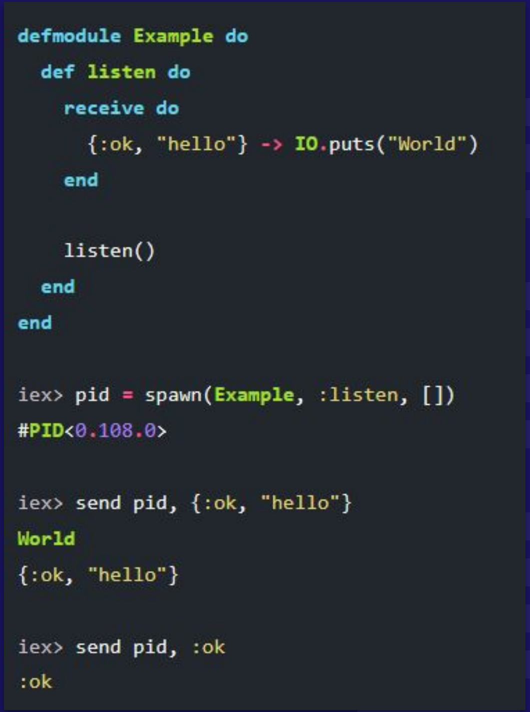
</> 2 → Message Passing

- Enviar mensagem para outro processo:
  send(destination, message)

- Receber mensagem de outros processos:

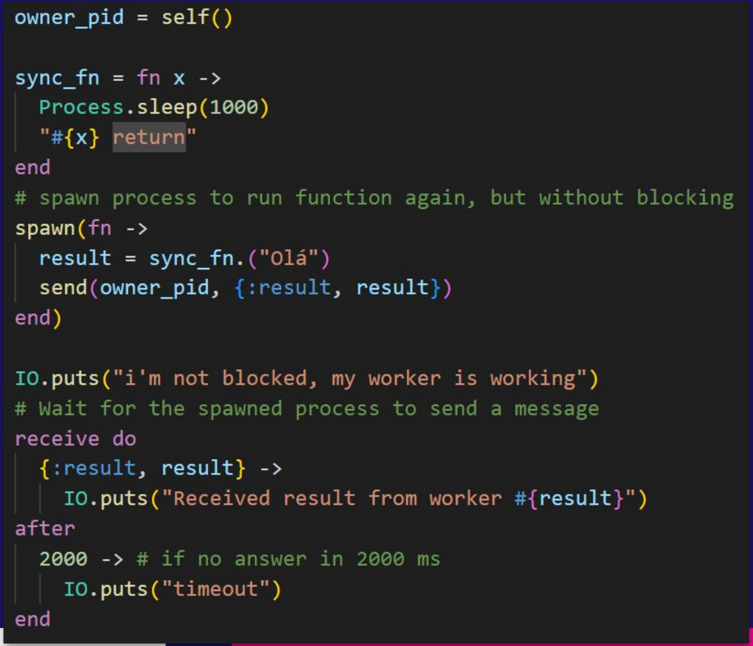
```
receive do
      pattern1 -> # stuff1
      pattern2 -> # stuff2
end
```

Blocking call



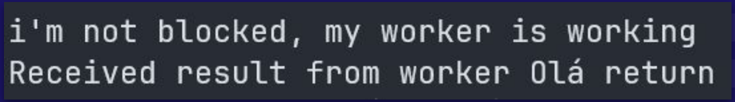
```
defmodule Example do
  def listen do
    receive do
      {:ok, "hello"} -> IO.puts("World")
    end

    listen()
  end
end

iex> pid = spawn(Example, :listen, [])
#PID<0.108.0>

iex> send pid, {:ok, "hello"}
World
{:ok, "hello"}

iex> send pid, :ok
:ok
```

</> 2 → Exemplo message passing

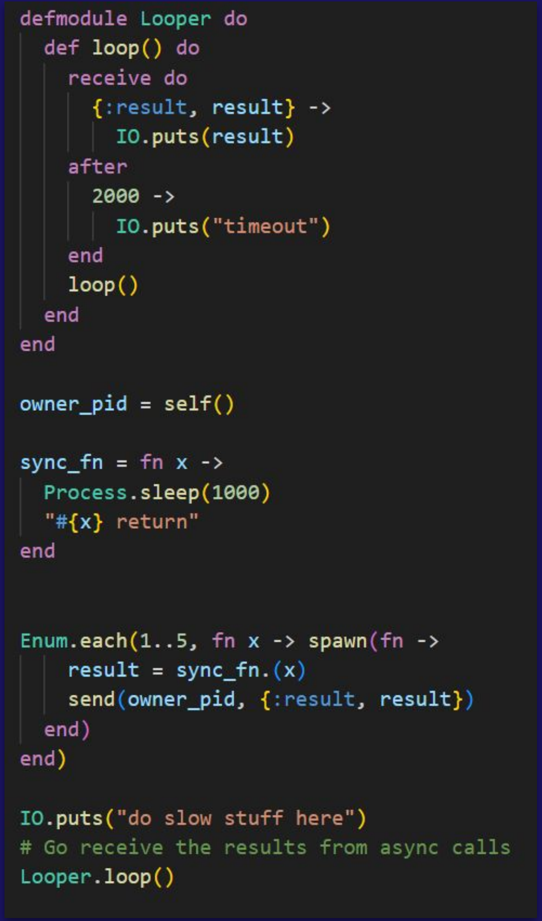


```
owner_pid = self()

sync_fn = fn x ->
  Process.sleep(1000)
  "#{x} return"
end
# spawn process to run function again, but without blocking
spawn(fn ->
  result = sync_fn.("Olá")
  send(owner_pid, {:result, result})
end)

IO.puts("i'm not blocked, my worker is working")
# Wait for the spawned process to send a message
receive do
  {:result, result} ->
    IO.puts("Received result from worker #{result}")
after
  2000 -> # if no answer in 2000 ms
    IO.puts("timeout")
end
```

Output:

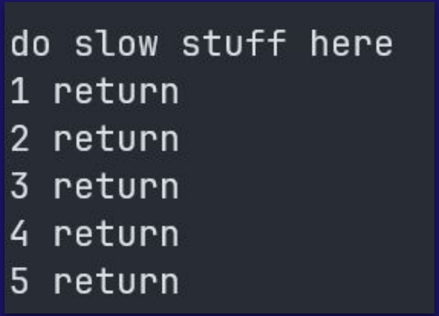


```
i'm not blocked, my worker is working
Received result from worker Olá return
```



```elixir
defmodule Looper do
  def loop() do
    receive do
      {:result, result} ->
        IO.puts(result)
    after
      2000 ->
        IO.puts("timeout")
    end
    loop()
  end
end

owner_pid = self()

sync_fn = fn x ->
  Process.sleep(1000)
  "#{x} return"
end


Enum.each(1..5, fn x -> spawn(fn ->
    result = sync_fn.(x)
    send(owner_pid, {:result, result})
  end)
end)

IO.puts("do slow stuff here")
# Go receive the results from async calls
Looper.loop()
```

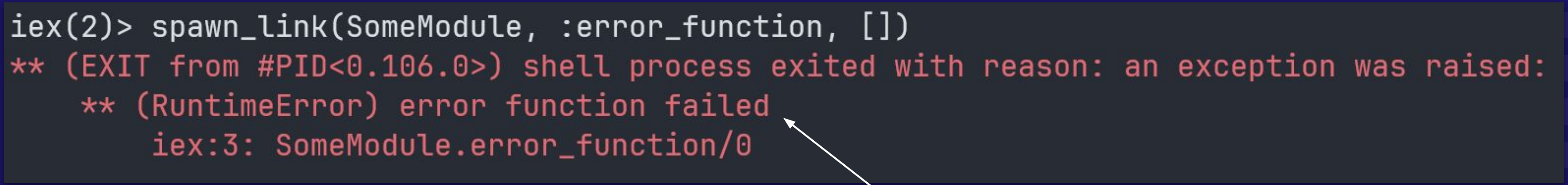
Output:



```
do slow stuff here
1 return
2 return
3 return
4 return
5 return
```

2 → Primitivas de concorrência

- **Spawn_link:** cria processo **ligado** - recebem exit notifications do outro processo a que estão ligados



```
iex(2)> spawn_link(SomeModule, :error_function, [])
** (EXIT from #PID<0.106.0>) shell process exited with reason: an exception was raised:
    ** (RuntimeError) error function failed
        iex:3: SomeModule.error_function/0
```

RuntimeError de :error_function recebido também no processo ligado

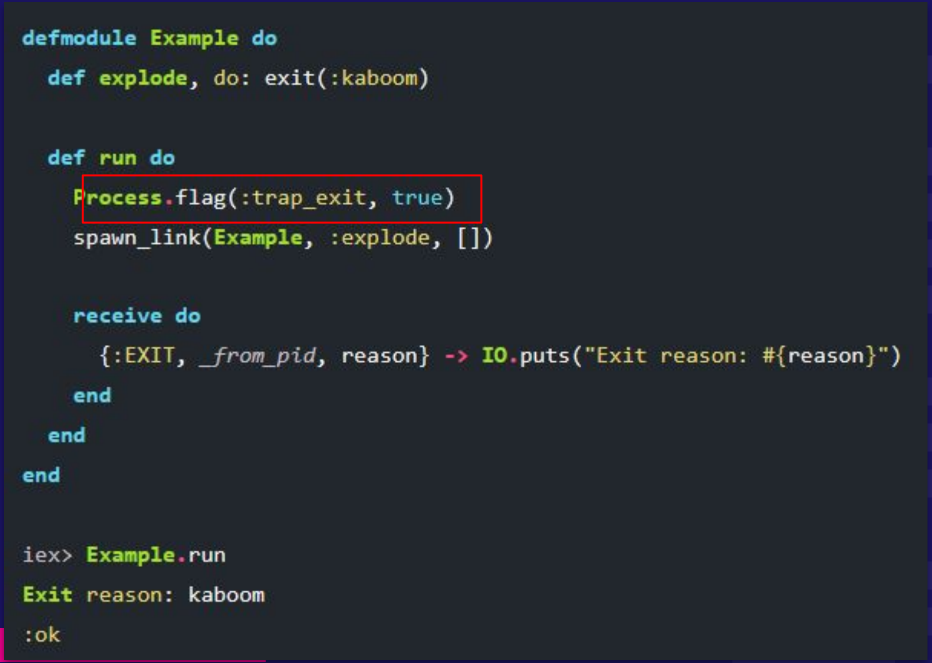
</> 2 → Tolerância a falhas

E se pudéssemos detetar os erros dos processos ligados e agir em vez de exit?

Usando:

Process.flag(:trap_exit, true)

É possível receber mensagens relativas à falha de outros processos.

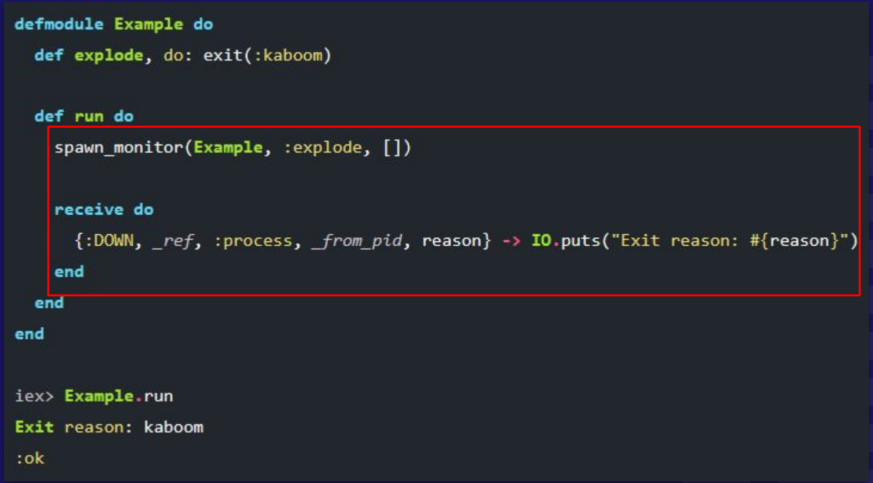


```
defmodule Example do
  def explode, do: exit(:kaboom)

  def run do
    Process.flag(:trap_exit, true)
    spawn_link(Example, :explode, [])

    receive do
      {:EXIT, _from_pid, reason} -> IO.puts("Exit reason: #{reason}")
    end
  end
end


iex> Example.run
Exit reason: kaboom
:ok
```

</> 2 → Tolerância a falhas

- Sem ligar processos e usando a flag trap_exit, é possível monitorizar outros processos.

- Monitor:
  spawn_monitor(SomeModule, :function, [])



```
defmodule Example do
  def explode, do: exit(:kaboom)

  def run do
    spawn_monitor(Example, :explode, [])

    receive do
      {:DOWN, _ref, :process, _from_pid, reason} -> IO.puts("Exit reason: #{reason}")
    end
  end
end

iex> Example.run
Exit reason: kaboom
:ok
```

Exercício
Fault Tolerance

GitHub Link

exercises/fault_tolerance/fault_tolerance.exs

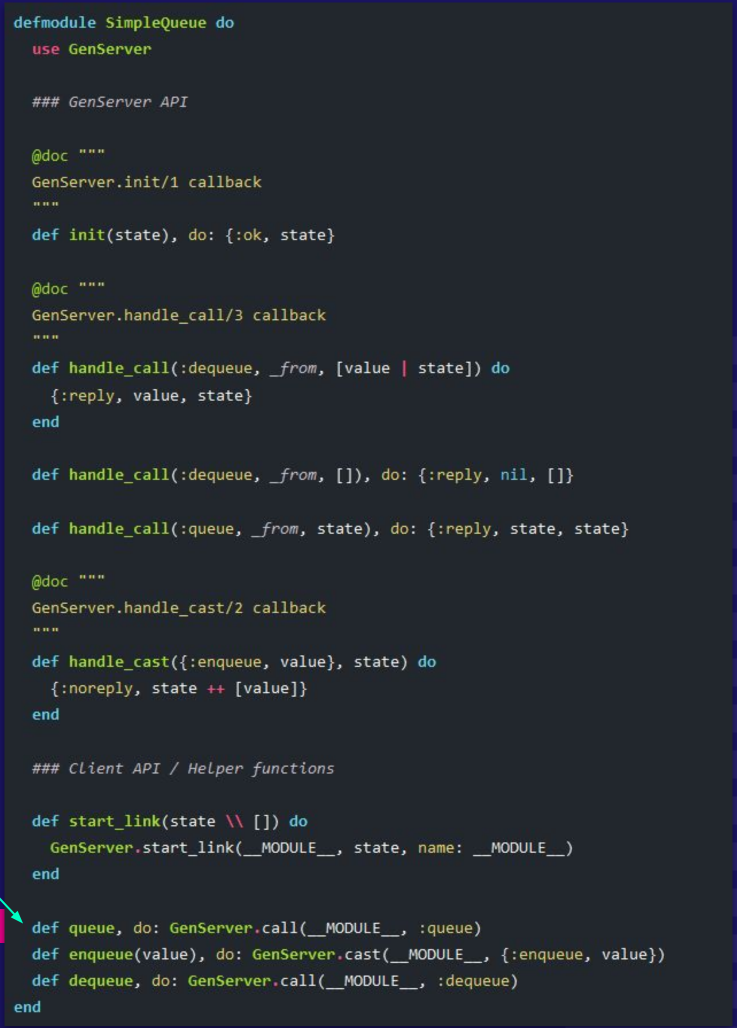
2 → OTP Concurrency

- Já mencionamos as abstrações para concorrência em Elixir, mas há outras abstrações de mais alto nível nas libraries OTP do Erlang que facilitam o uso das anteriormente mencionadas:
    - **GenServers**
    - **Supervisors**

2 → OTP GenServers

- Server genérico
- Corre num loop de message handling
- Permite async (handle_cast) e sync (handle_call) calls.

- GenServer.call / GenServer.cast para callbacks sincronos e assincronos.



```elixir
defmodule SimpleQueue do
  use GenServer

  ### GenServer API

  @doc """
  GenServer.init/1 callback
  """
  def init(state), do: {:ok, state}

  @doc """
  GenServer.handle_call/3 callback
  """
  def handle_call(:dequeue, _from, [value | state]) do
    {:reply, value, state}
  end

  def handle_call(:dequeue, _from, []), do: {:reply, nil, []}

  def handle_call(:queue, _from, state), do: {:reply, state, state}

  @doc """
  GenServer.handle_cast/2 callback
  """
  def handle_cast({:enqueue, value}, state) do
    {:noreply, state ++ [value]}
  end

  ### Client API / Helper functions

  def start_link(state \\ []) do
    GenServer.start_link(__MODULE__, state, name: __MODULE__)
  end

  def queue, do: GenServer.call(__MODULE__, :queue)
  def enqueue(value), do: GenServer.cast(__MODULE__, {:enqueue, value})
  def dequeue, do: GenServer.call(__MODULE__, :dequeue)
end
```

2 → OTP Supervisors

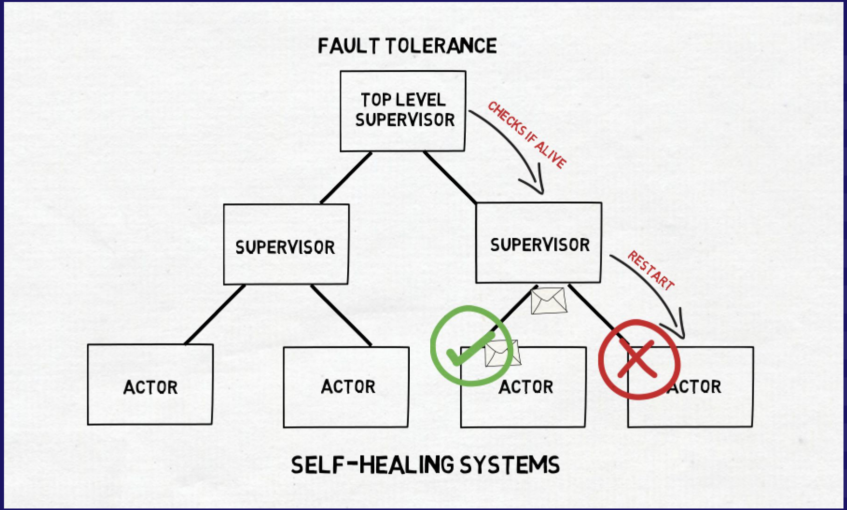
Supervisor - actor que supervisiona outros actors

Reage a falhas de actors que supervisiona e age consoante a estratégia.

Estratégias possíveis:

- One for one
- One for all
- Rest for one

Permite ~99%+ availability



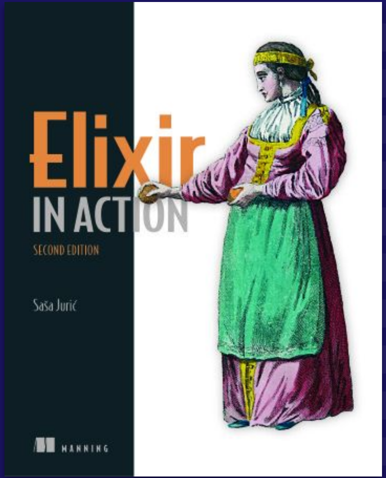
Exemplo do uso de OTP abstractions

GitHub Link

examples/OTP_abstractions/fault_tolerance

References:

- Elixir documentation: https://hexdocs.pm/
- Elixir school: https://elixirschool.com/en
- Elixir official website: https://elixir-lang.org/
- Reference book: https://www.manning.com/books/elixir-in-action-second-edition
- Exemplo fault-tolerance com OTP abstractions: https://github.com/ellyacademy/videos/tree/main/0005/elixir_supervisor/elxsuper



</>

Obrigado!

CSLP 2023/24

Diogo Marto - 108298
João Dourado - 108636
Tiago Pereira - 108546