Elixir

Peeking into *Elixir's*

Processes, OTP & Supervisors

21st March 2014

benjamintanweihao@gmail.com

/benjamintanweihao

/bentanweihao

What will we learn today?

Elixir & Erlang

In less than 5 minutes

What will we learn today?

Elixir & Erlang

In less than 5 minutes

Processes 101

The Basic Concurrency Primitive

What will we learn today?

Elixir & Erlang

In less than 5 minutes

Processes 101

The Basic Concurrency Primitive

OTP

Framework and much more

What will we learn today?

Elixir & Erlang

In less than 5 minutes

Processes 101

The Basic Concurrency Primitive

OTP

Framework and much more

Supervisors

Fault Tolerance & Recovery

Ohai, Elixir!

Elixir is a functional, meta-programming aware language built on top of the Erlang VM.

Elixir & Erlang

In less than 5 minutes

Ohai, Erlang!

Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with eager evaluation, single assignment, and dynamic typing. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system.

Ohai, Erlang!

Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with eager evaluation, single assignment, and dynamic typing. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system.

Why Elixir ?

- Free lunch is over
- Hyper-threading & Multicore
- Faster software means using all cores!
- Concurrency -> Coordination
- Functional

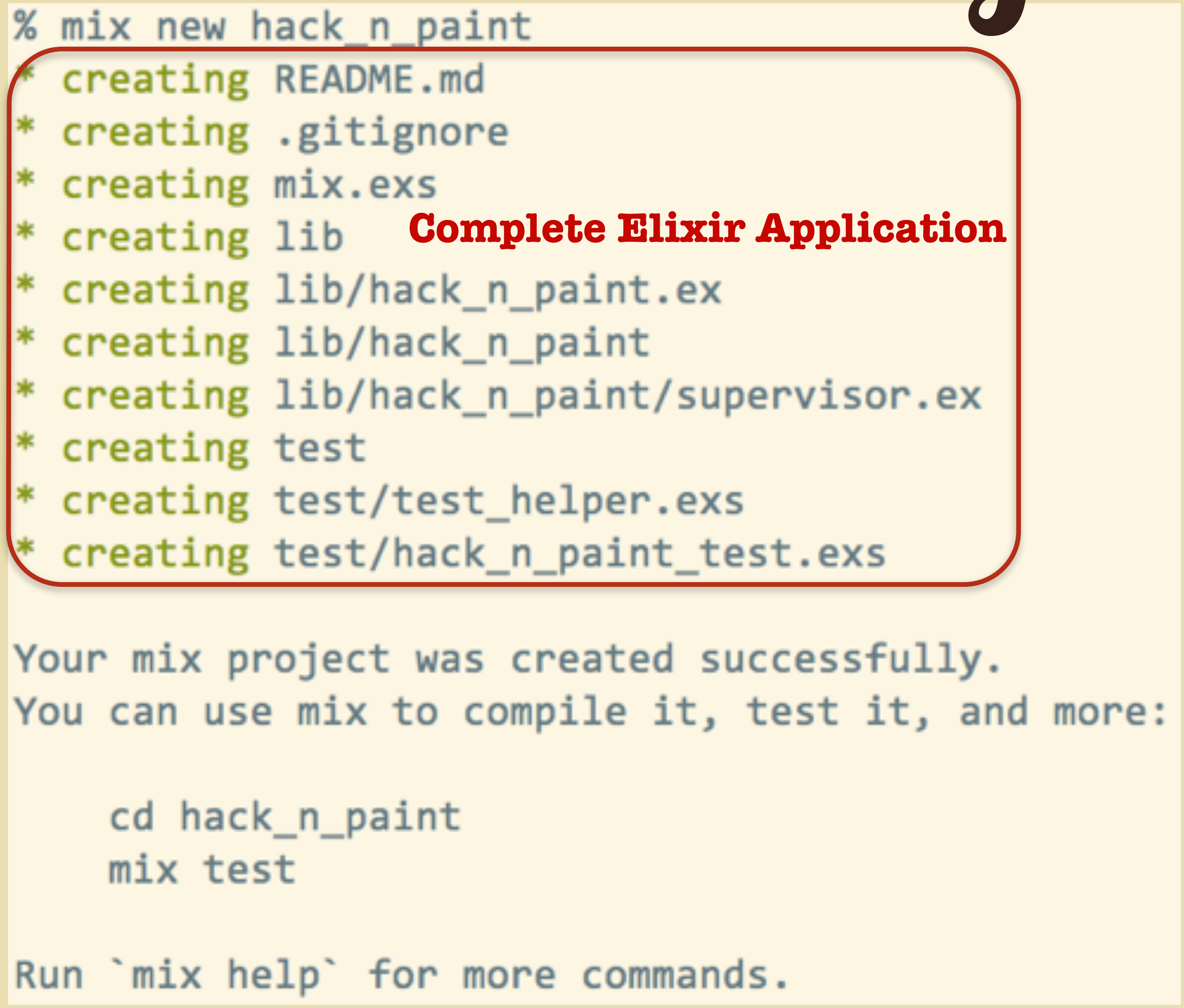
Design Goals of Elixir

1. Productivity

2. Extensibility

3. Compatibility

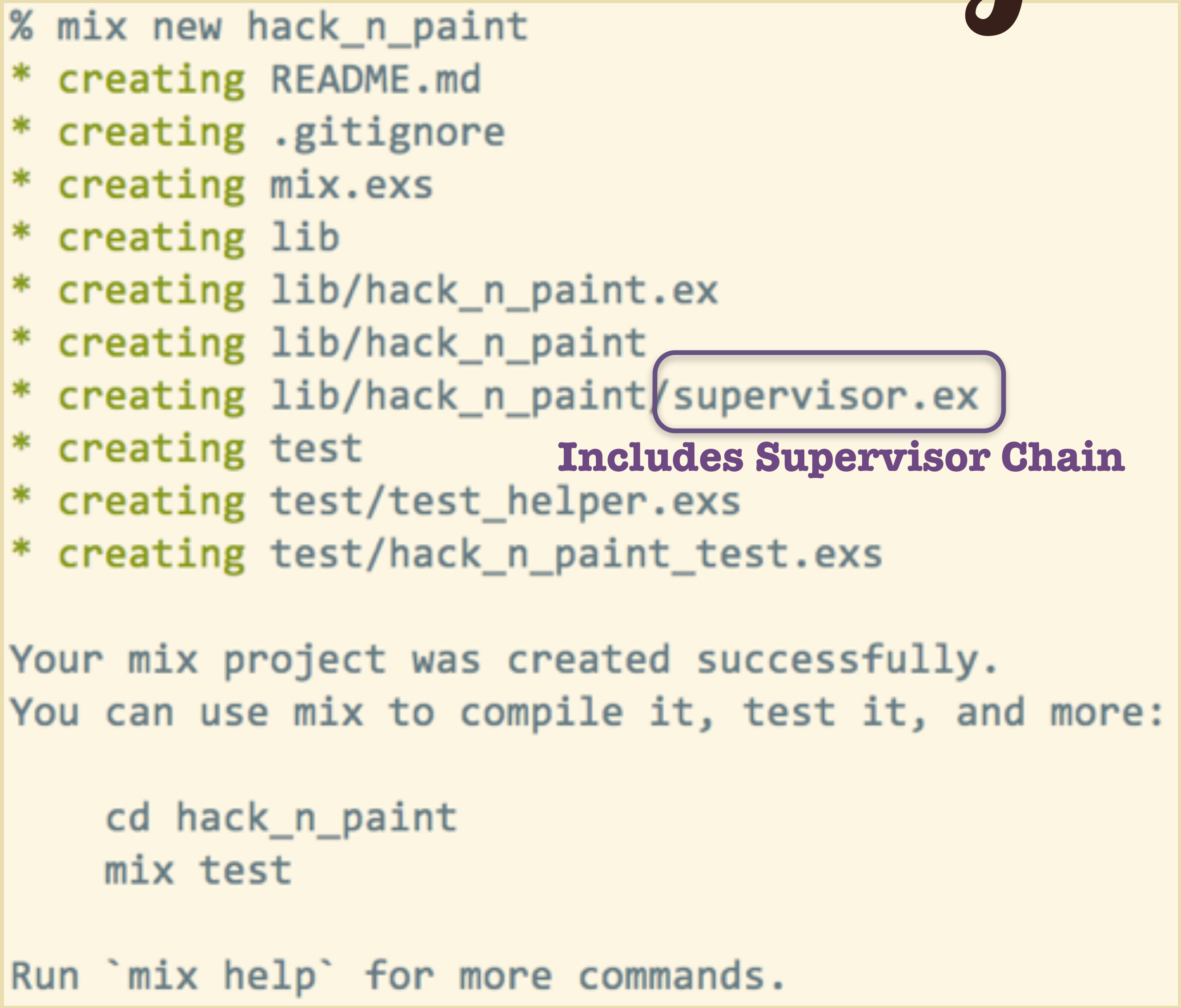
Productivity



```
% mix new hack_n_paint
* creating README.md
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/hack_n_paint.ex
* creating lib/hack_n_paint
* creating lib/hack_n_paint/supervisor.ex
* creating test
* creating test/test_helper.exs
* creating test/hack_n_paint_test.exs

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

    cd hack_n_paint
    mix test

Run `mix help` for more commands.
```

Complete Elixir Application

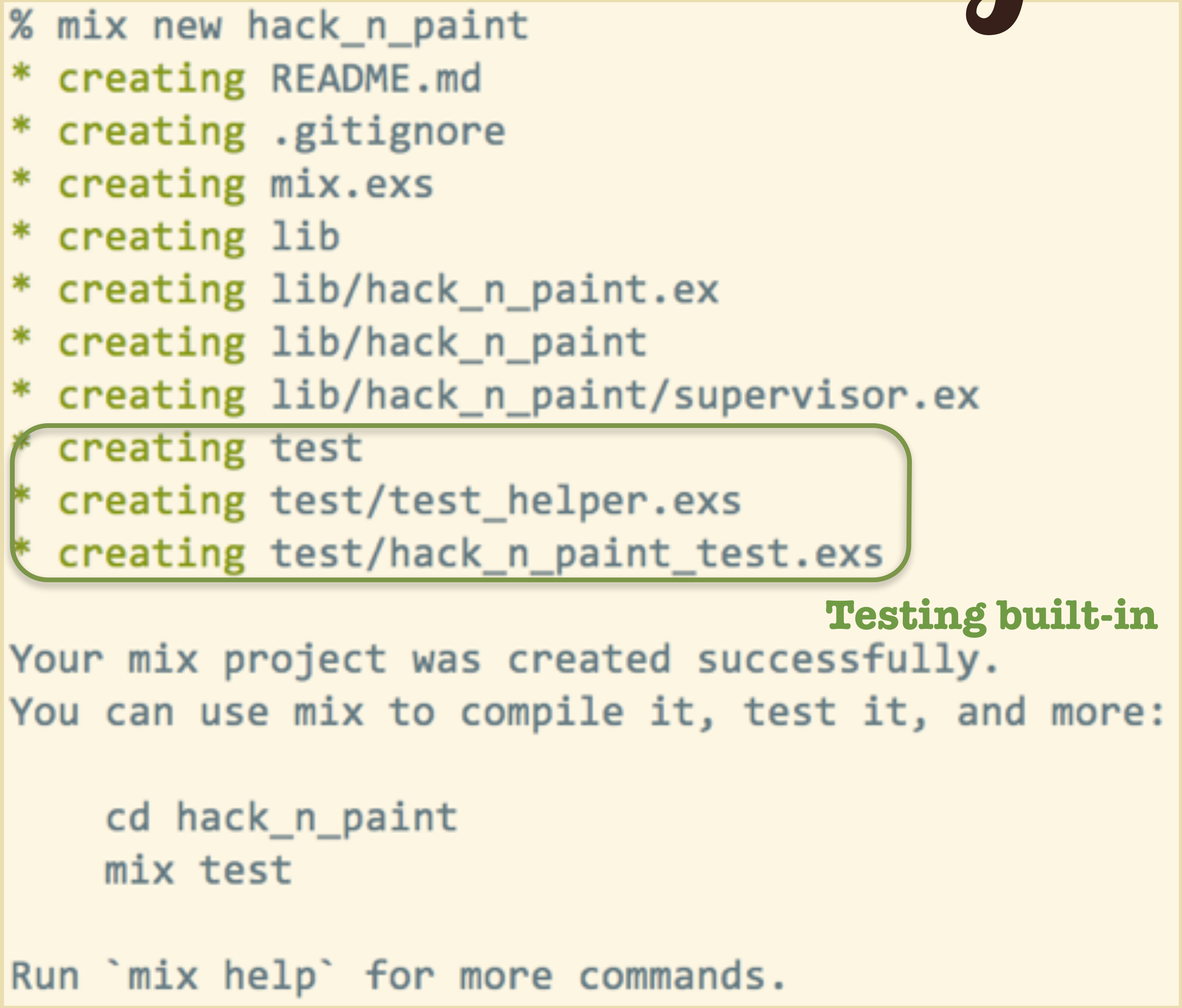
Productivity



```
% mix new hack_n_paint
* creating README.md
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/hack_n_paint.ex
* creating lib/hack_n_paint
* creating lib/hack_n_paint/supervisor.ex
* creating test
* creating test/test_helper.exs
* creating test/hack_n_paint_test.exs

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

    cd hack_n_paint
    mix test

Run `mix help` for more commands.
```

Includes Supervisor Chain

Productivity

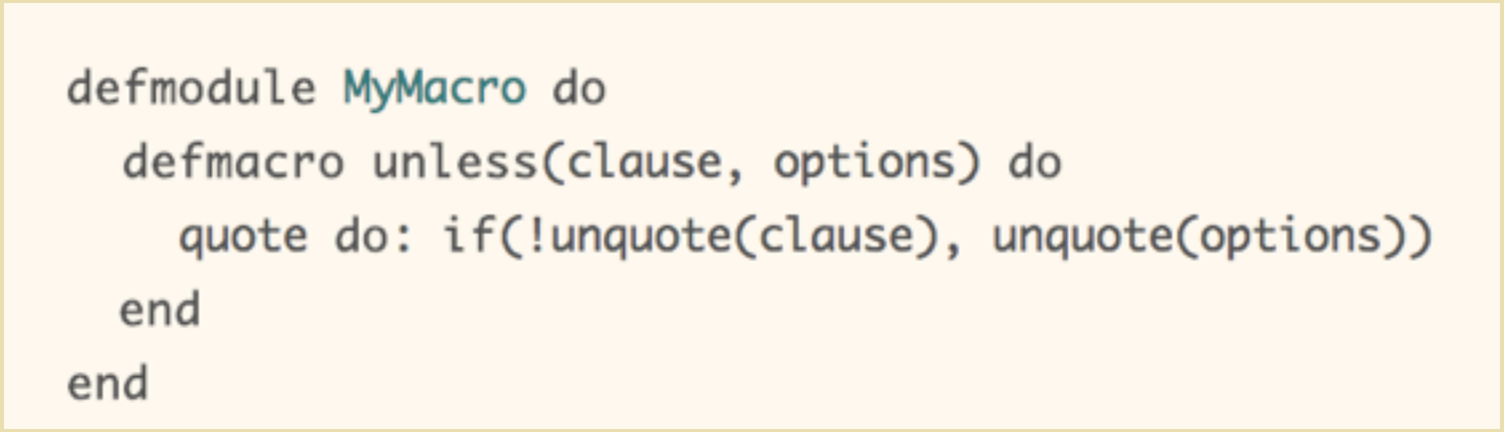


```
% mix new hack_n_paint
* creating README.md
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/hack_n_paint.ex
* creating lib/hack_n_paint
* creating lib/hack_n_paint/supervisor.ex
* creating test
* creating test/test_helper.exs
* creating test/hack_n_paint_test.exs

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

    cd hack_n_paint
    mix test

Run `mix help` for more commands.
```

Testing built-in

Extensibility

Macros & Meta-programming

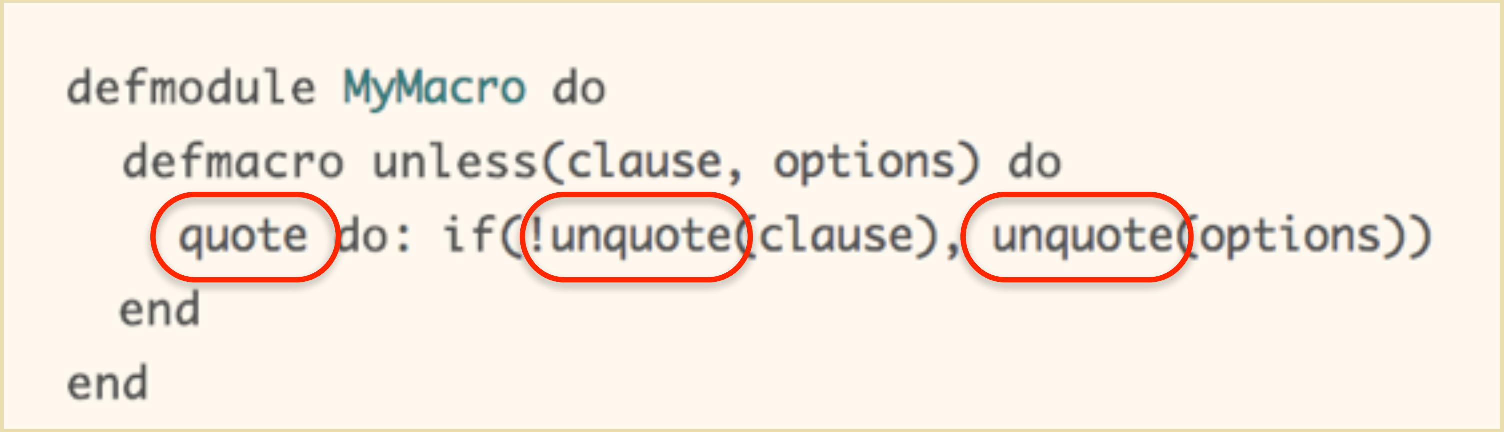


```
defmodule MyMacro do
  defmacro unless(clause, options) do
    quote do: if(!unquote(clause), unquote(options))
  end
end
```

Implementing **unless** using **if**

Extensibility

Macros & Meta-programming



```
defmodule MyMacro do
  defmacro unless(clause, options) do
    quote do: if(!unquote(clause), unquote(options))
  end
end
```

Implementing **unless** using **if**

Compatibility

Elixir can call *Erlang* code, without any conversion cost at all.

Compatibility

Elixir can use Erlang libraries!

The Actor Concurrency Model

- Actor = Process

The Actor Concurrency Model

- Actor = Process
- A process performs a specific task when it receives a message

The Actor Concurrency Model

- Actor = Process
- A process performs a specific task when it receives a message
- In turn, the process can reply to the sender

The Actor Concurrency Model

- Actor = Process
- A process performs a specific task when it receives a message
- In turn, the process can reply to the sender
- All messages go to a processes' mailbox – Q of unprocessed messages sent from other processes that are not yet consumed

The Actor Concurrency Model

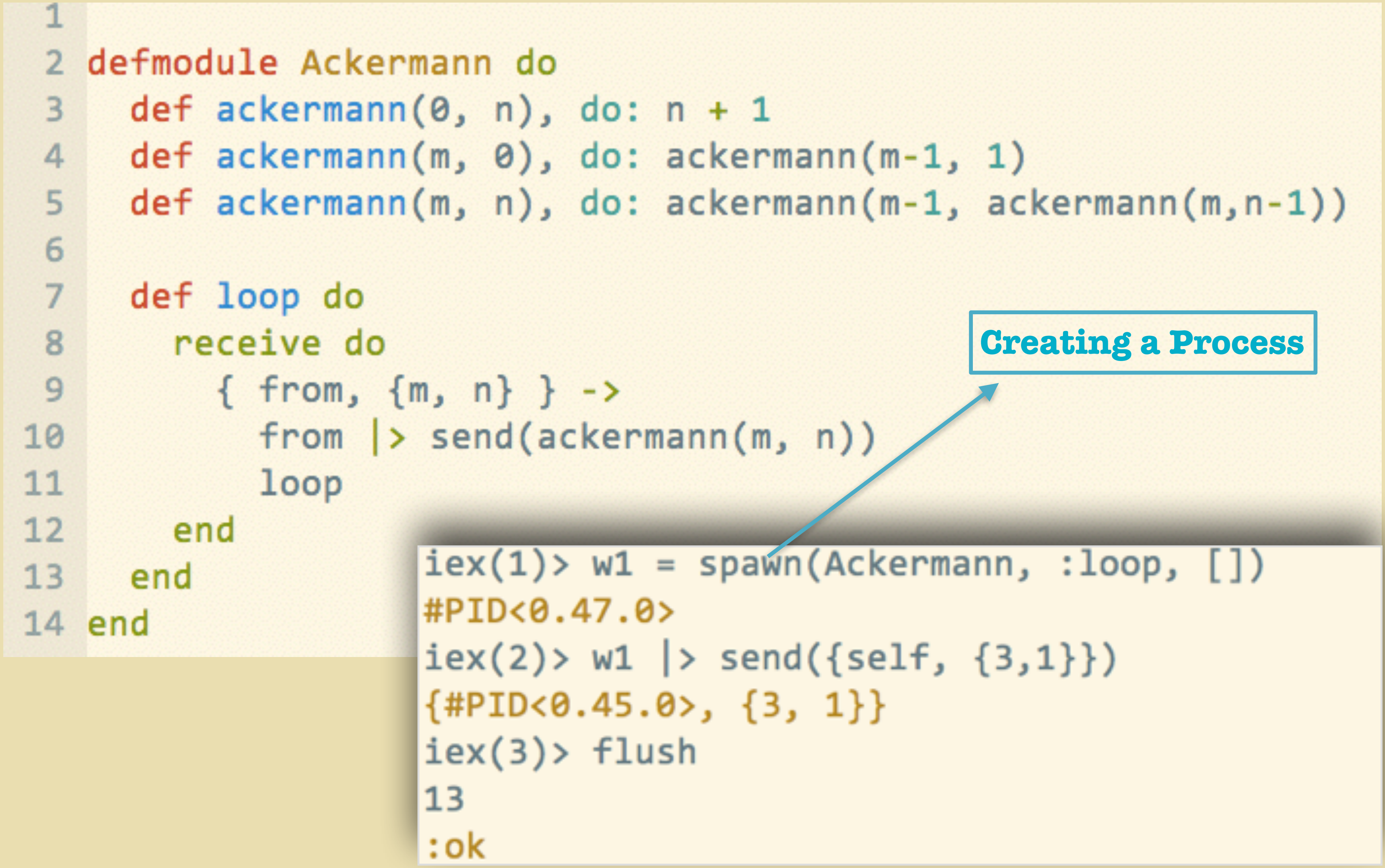
- Actor = Process
- A process performs a specific task when it receives a message
- In turn, the process can reply to the sender
- All messages go to a processes' mailbox – Q of unprocessed messages sent from other processes that are not yet consumed

Shared-nothing Async Message-passing

Processes 101

The Basic Concurrency Primitive

Creating Processes & Sending Messages in Elixir

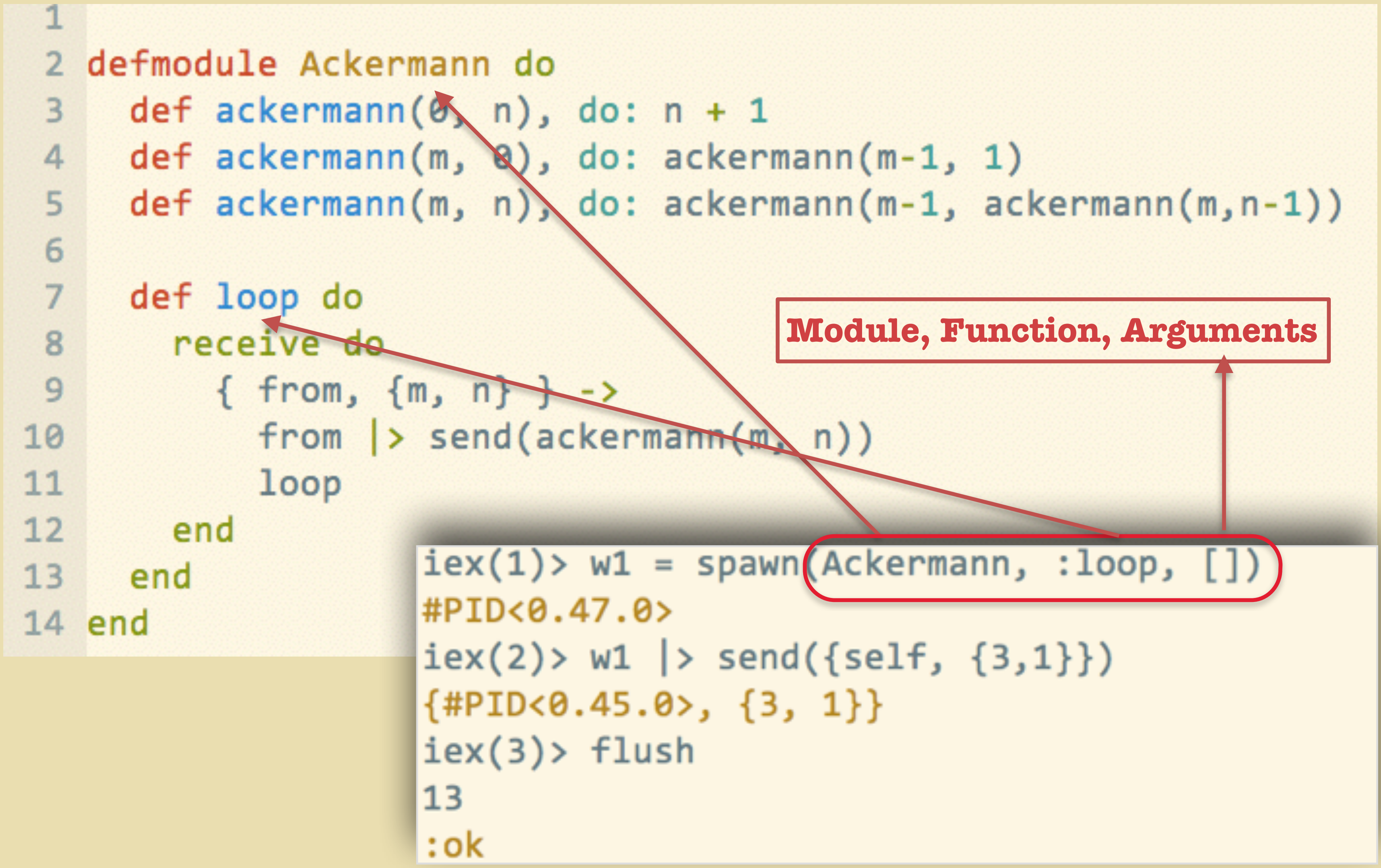


```

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```

Creating a Process

```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

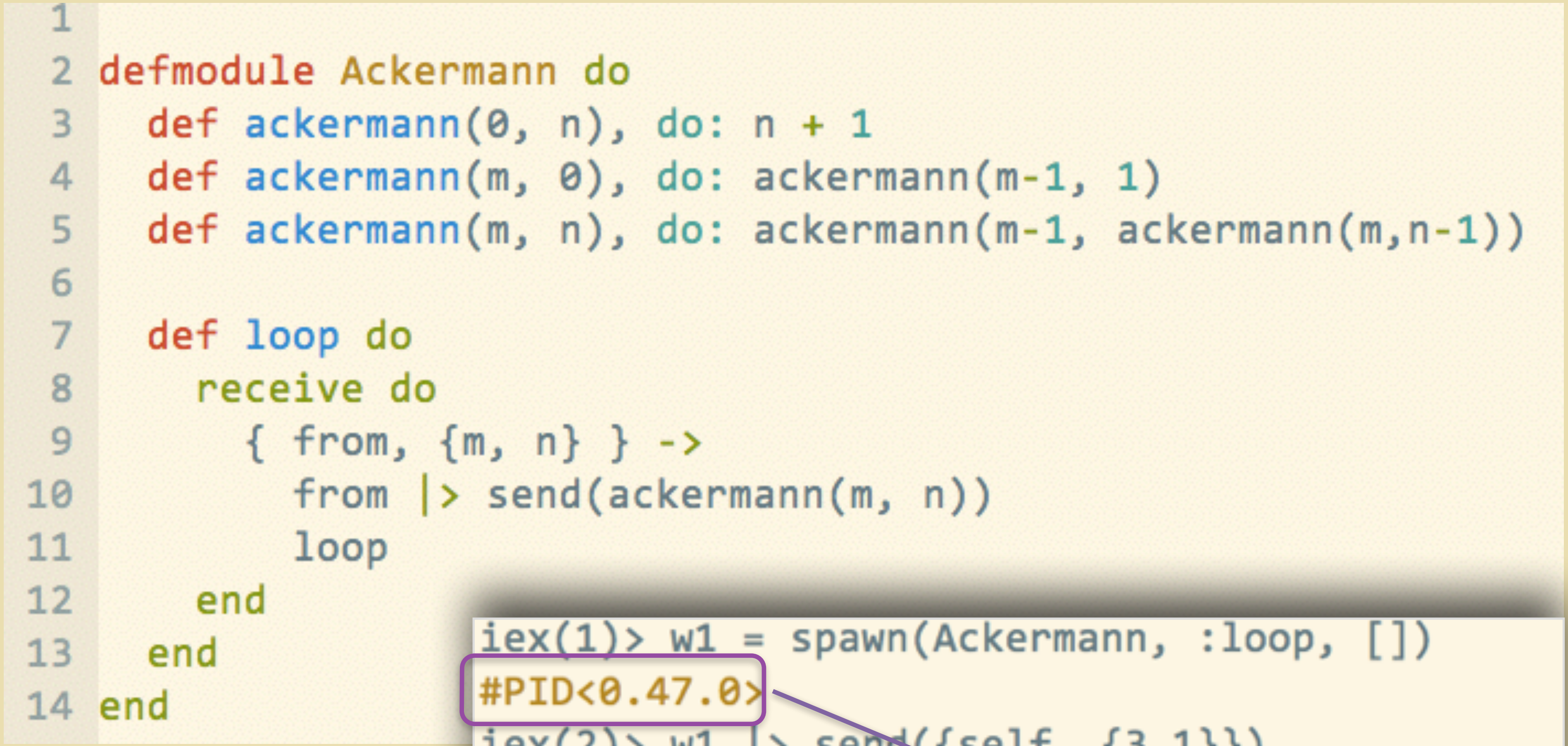
Creating Processes & Sending Messages in Elixir



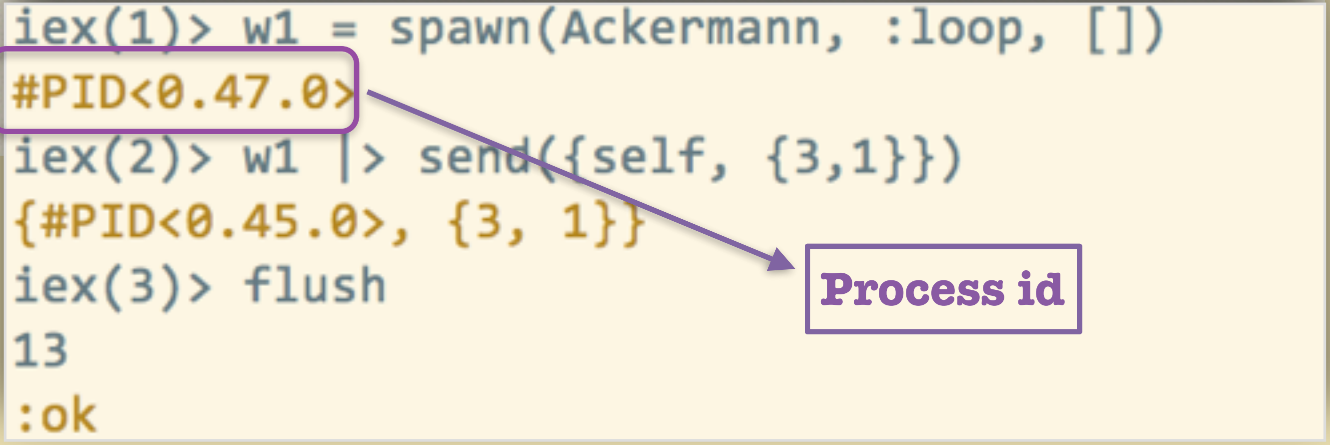
```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```

```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Creating Processes & Sending Messages in Elixir



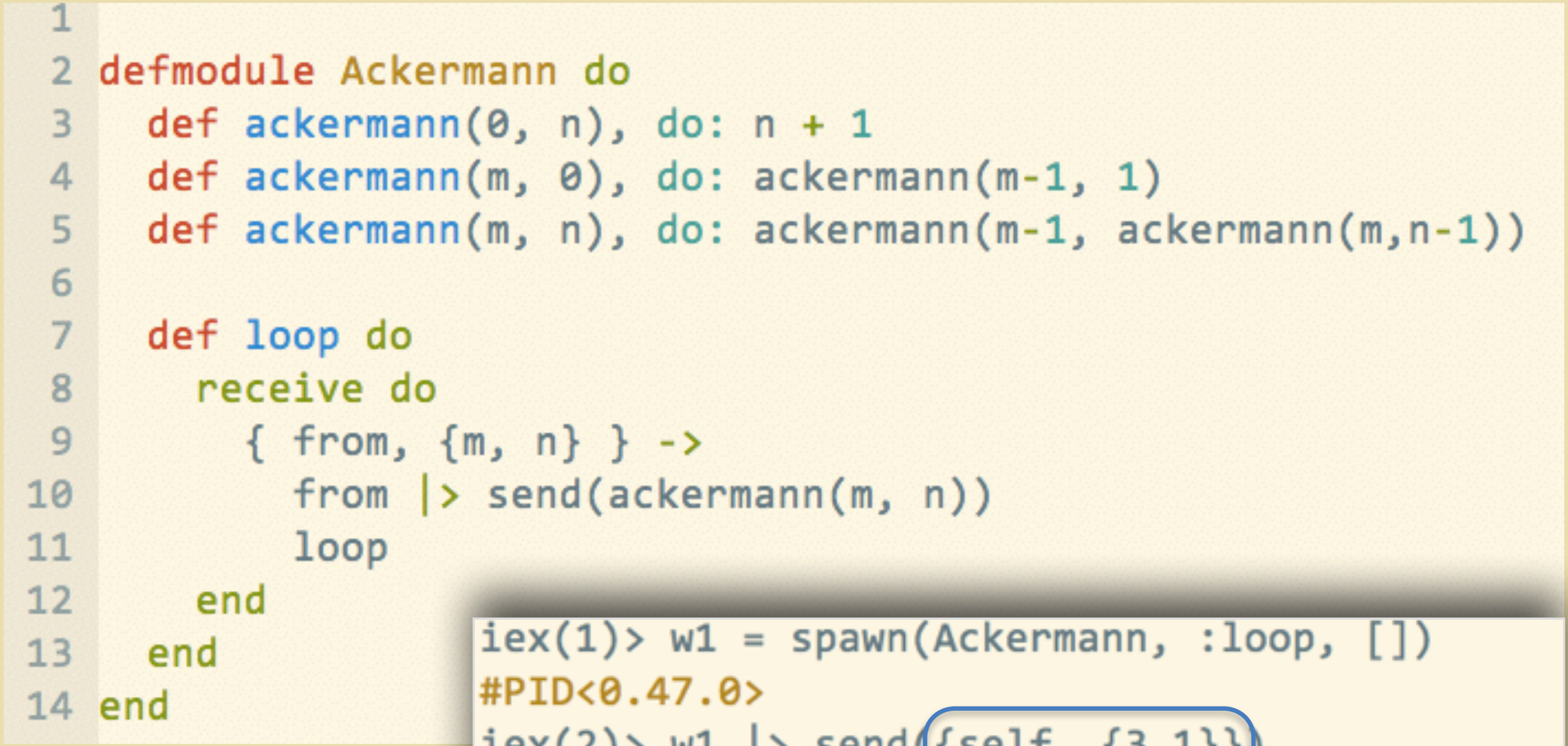
```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```



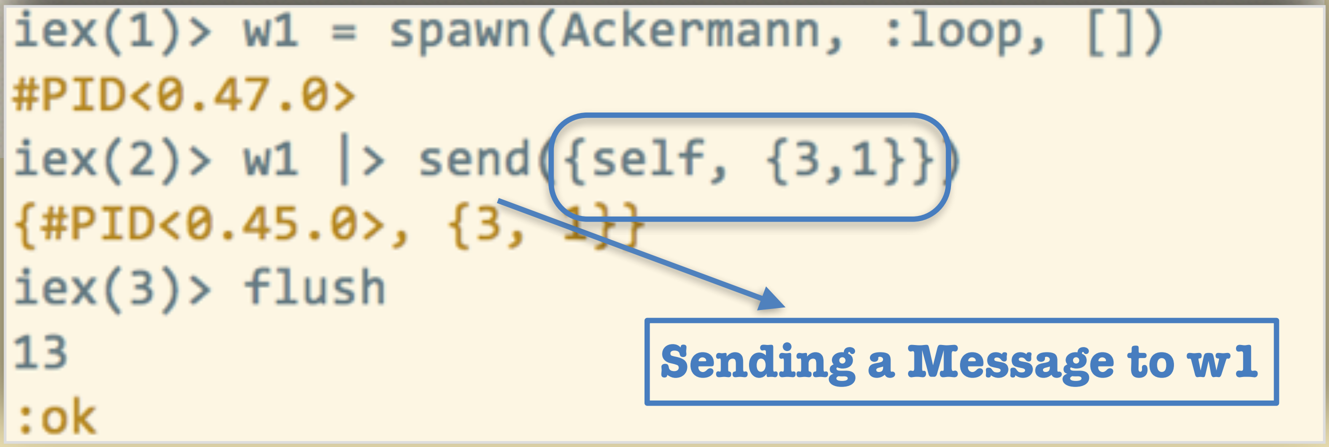
```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Process id

Creating Processes & Sending Messages in Elixir

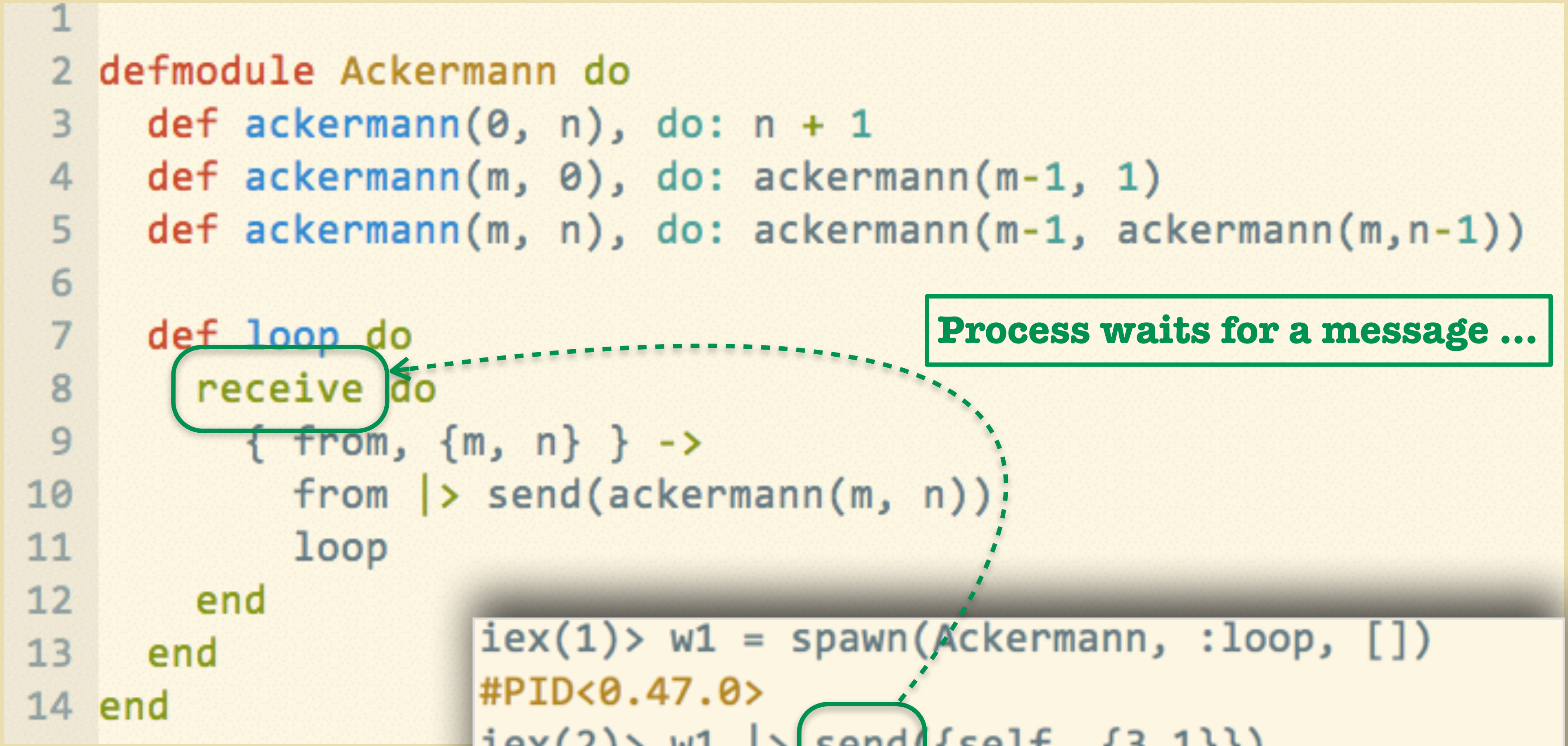


```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```



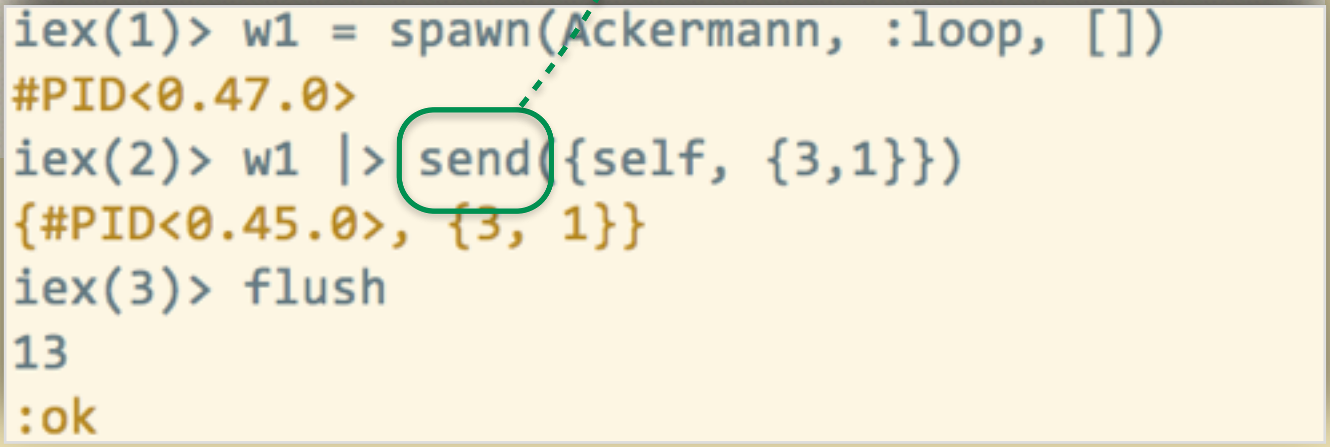
```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Creating Processes & Sending Messages in Elixir



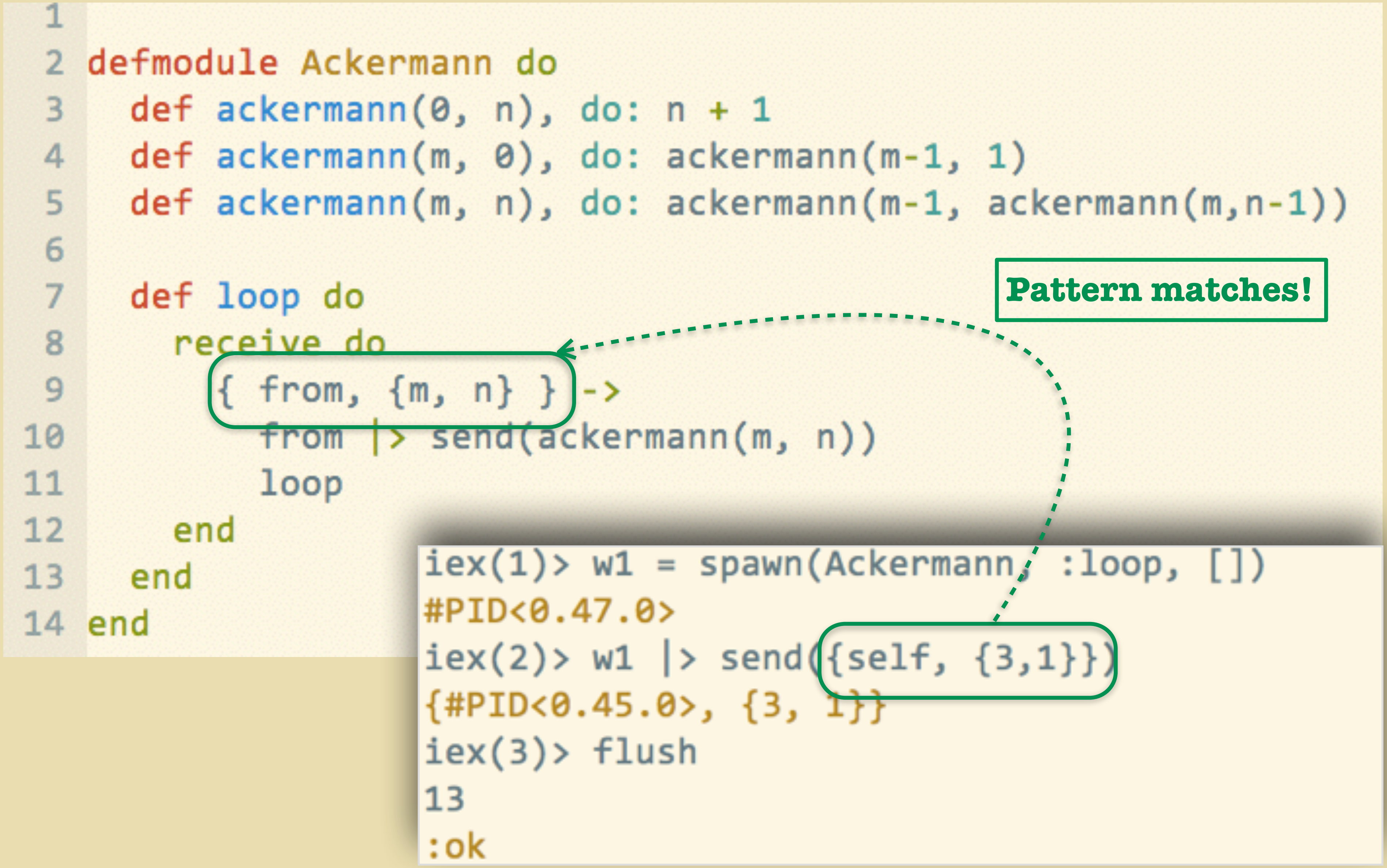
```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```

Process waits for a message ...



```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

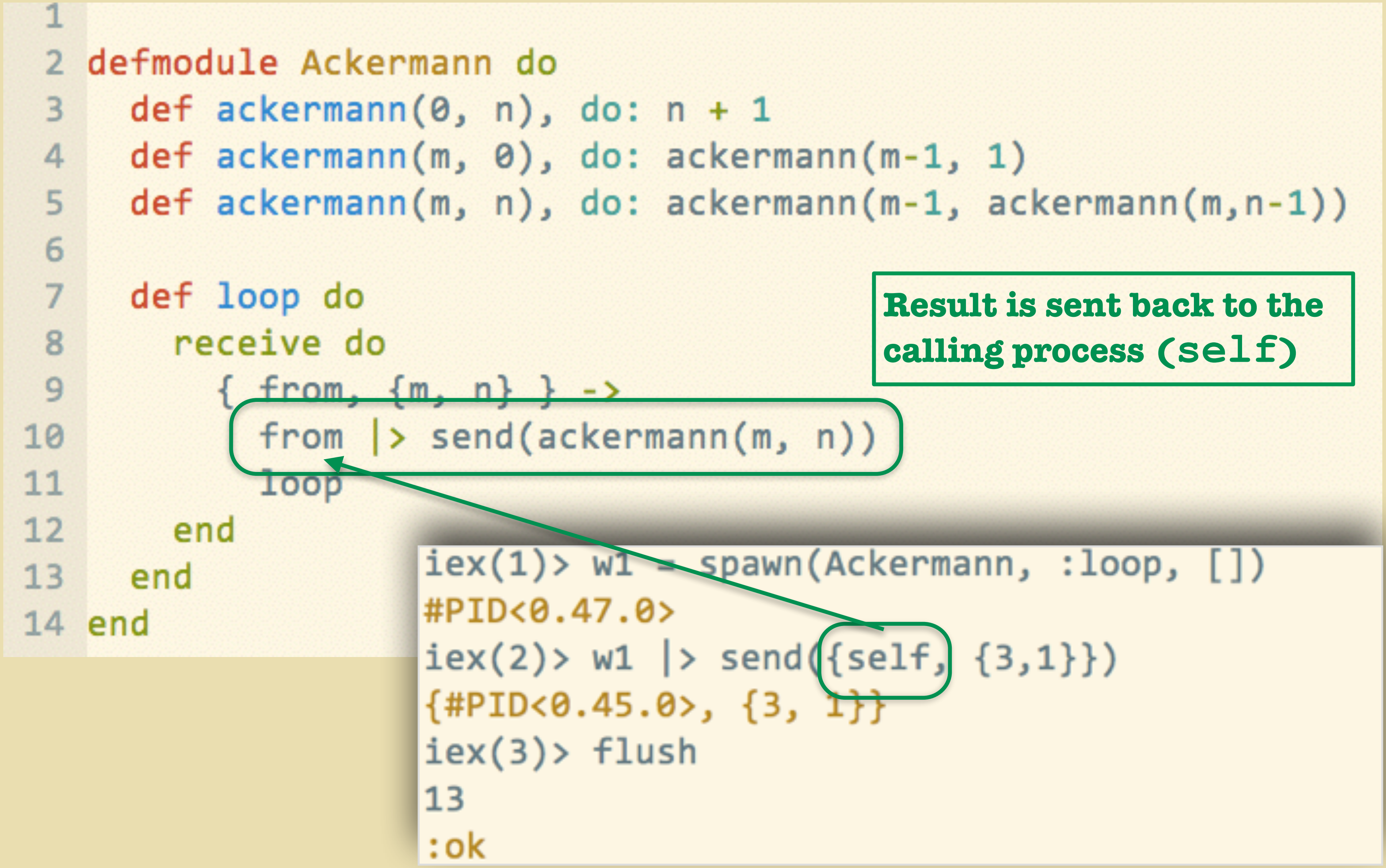
Creating Processes & Sending Messages in Elixir



```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```

```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Creating Processes & Sending Messages in Elixir



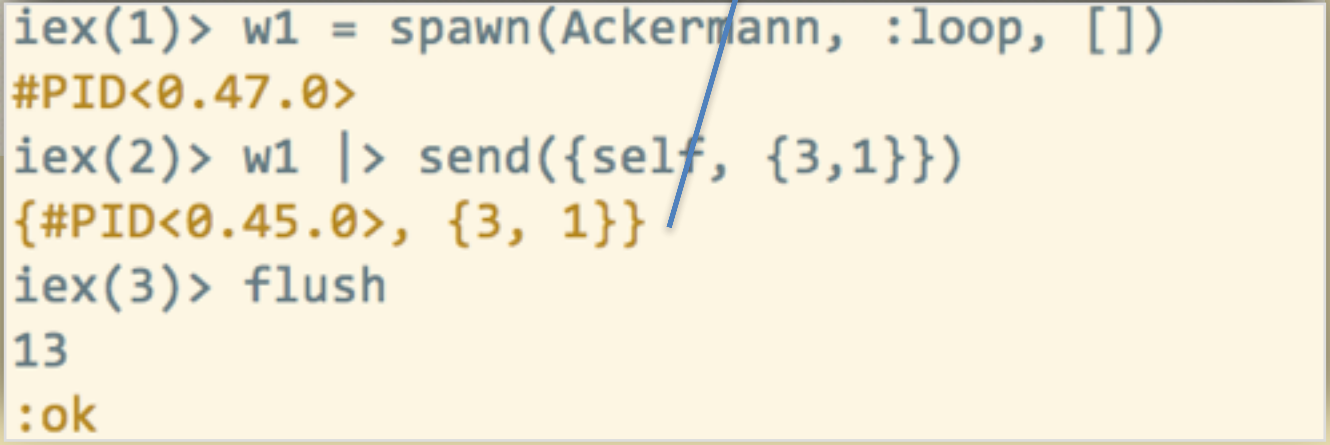
```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```

Result is sent back to the calling process (`self`)

```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Creating Processes & Sending Messages in Elixir

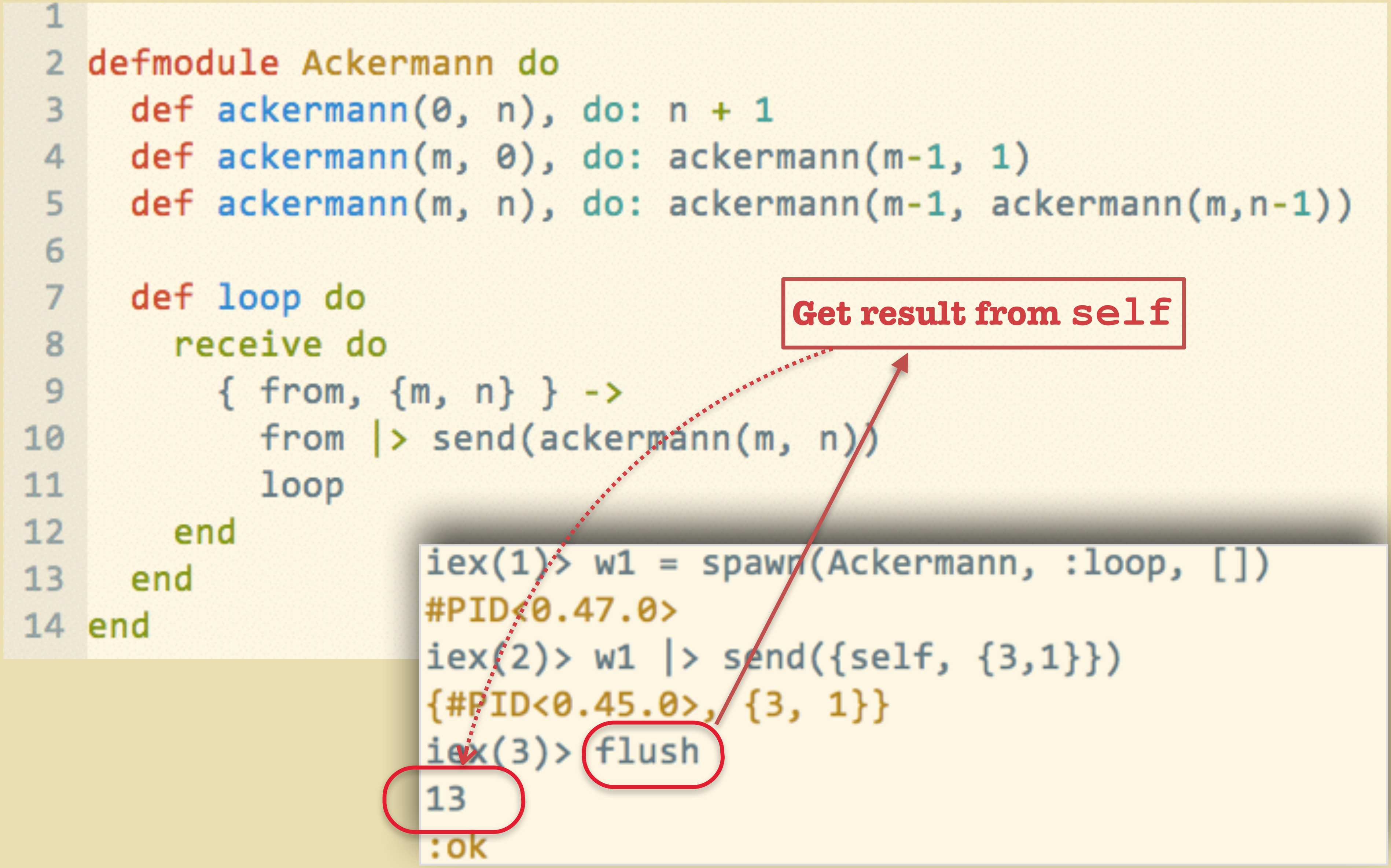
```

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```



```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Returns immediately

Creating Processes & Sending Messages in Elixir



```elixir

defmodule Ackermann do
  def ackermann(0, n), do: n + 1
  def ackermann(m, 0), do: ackermann(m-1, 1)
  def ackermann(m, n), do: ackermann(m-1, ackermann(m,n-1))

  def loop do
    receive do
      { from, {m, n} } ->
        from |> send(ackermann(m, n))
        loop
    end
  end
end
```

```
iex(1)> w1 = spawn(Ackermann, :loop, [])
#PID<0.47.0>
iex(2)> w1 |> send({self, {3,1}})
{#PID<0.45.0>, {3, 1}}
iex(3)> flush
13
:ok
```

Get result from `self`

OTP

Framework and much more

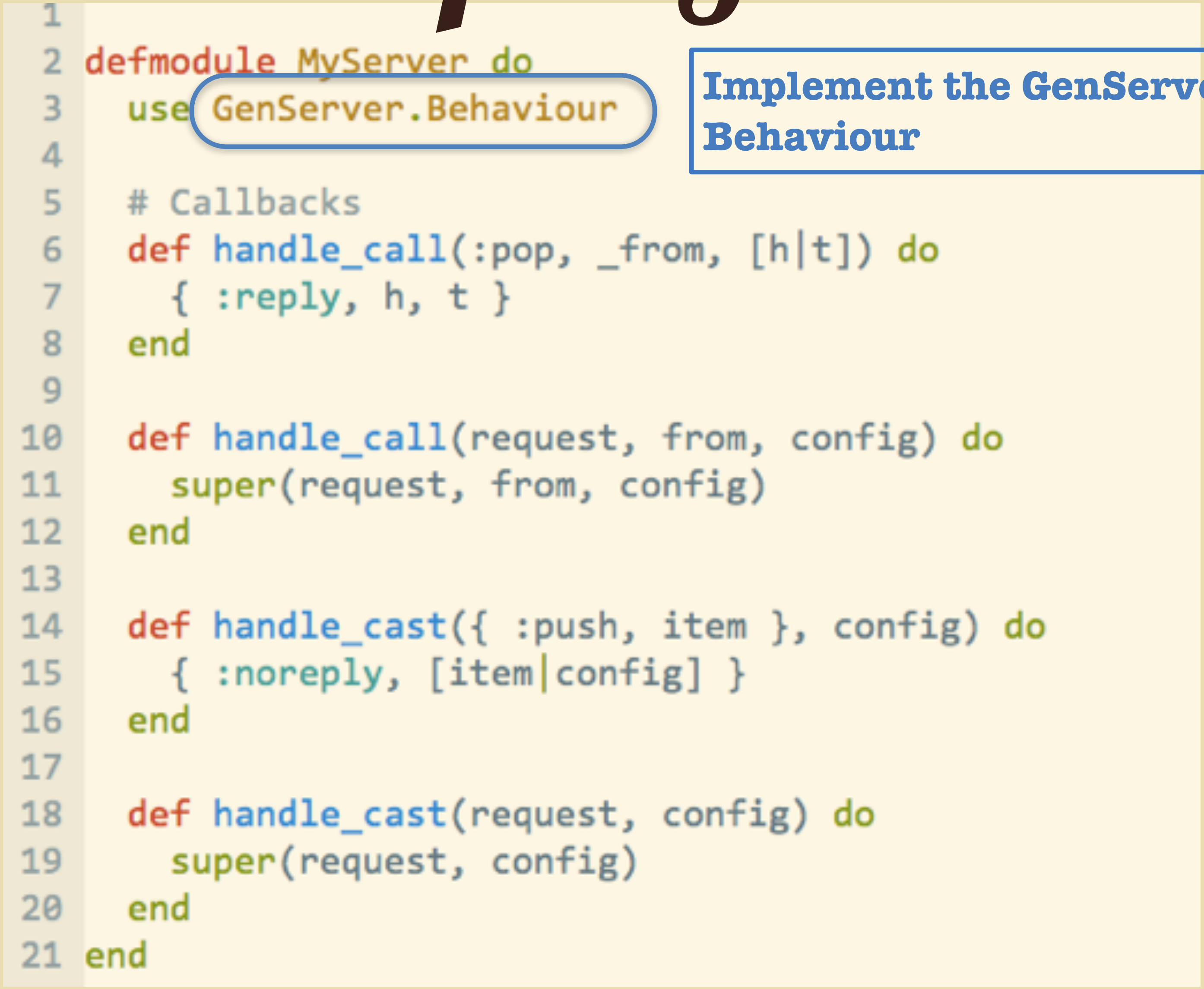
What is OTP?

- Comes with Elixir/Erlang
- Framework to build applications that are fault-tolerant, scalable, distributed
- Databases + Profilers + Debuggers

OTP Behaviours

- GenServer
- Supervisor
- Application

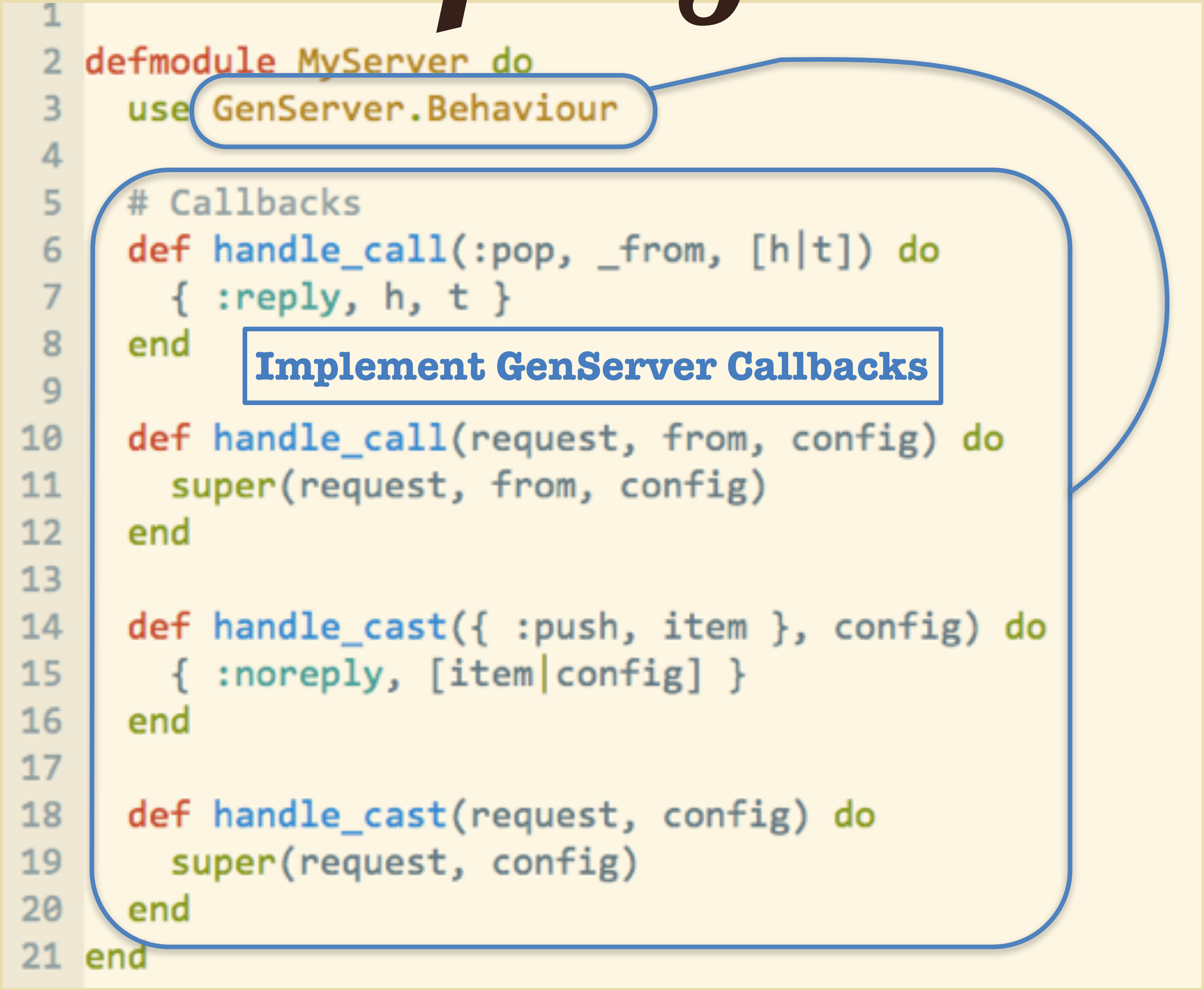
An Example GenServer



```
defmodule MyServer do
  use GenServer.Behaviour

  # Callbacks
  def handle_call(:pop, _from, [h|t]) do
    { :reply, h, t }
  end

  def handle_call(request, from, config) do
    super(request, from, config)
  end

  def handle_cast({ :push, item }, config) do
    { :noreply, [item|config] }
  end

  def handle_cast(request, config) do
    super(request, config)
  end
end
```

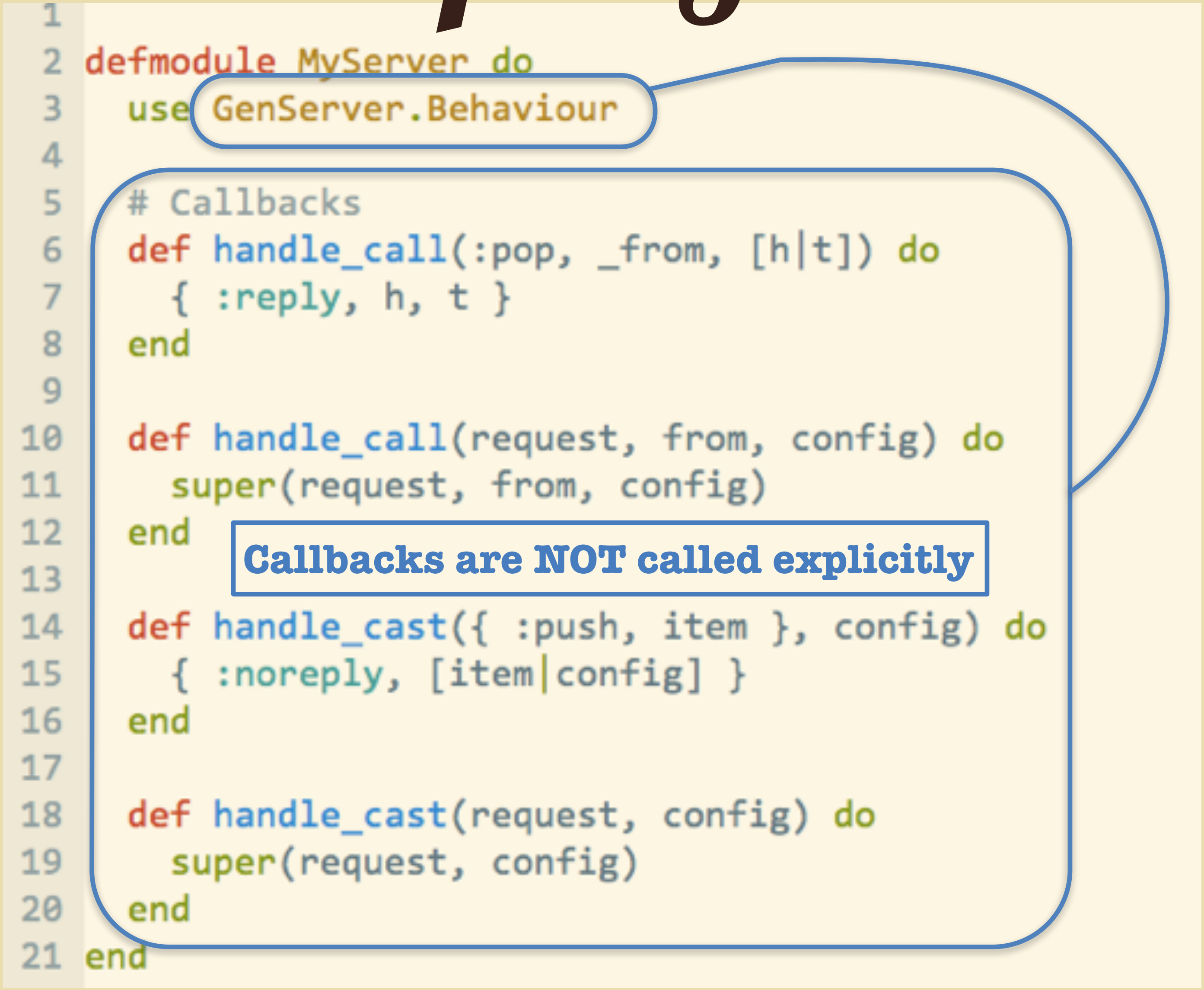
Implement the GenServer Behaviour

An Example GenServer



```elixir

defmodule MyServer do
  use GenServer.Behaviour

  # Callbacks
  def handle_call(:pop, _from, [h|t]) do
    { :reply, h, t }
  end

  def handle_call(request, from, config) do
    super(request, from, config)
  end

  def handle_cast({ :push, item }, config) do
    { :noreply, [item|config] }
  end

  def handle_cast(request, config) do
    super(request, config)
  end
end
```

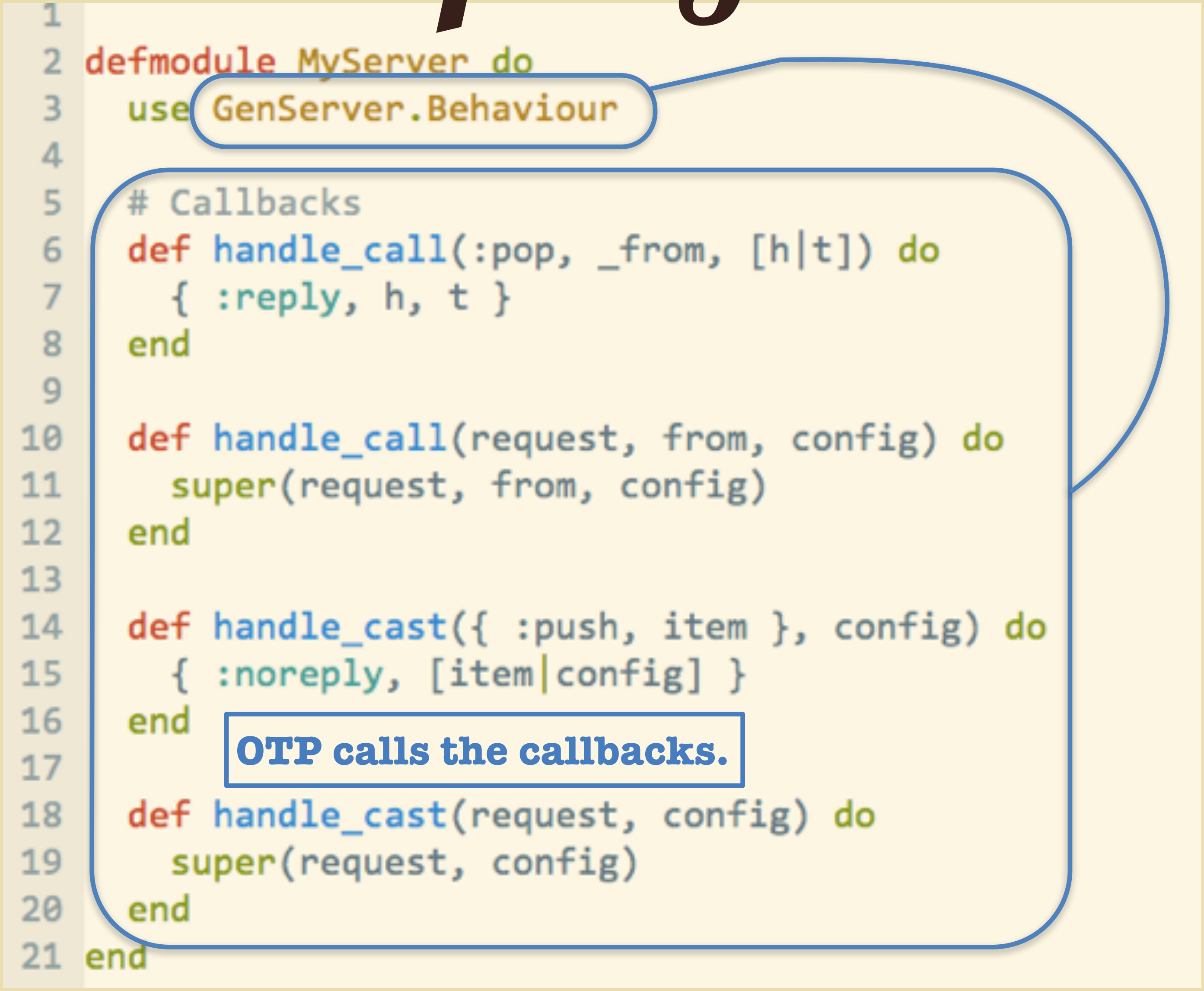
An Example GenServer



```
defmodule MyServer do
  use GenServer.Behaviour

  # Callbacks
  def handle_call(:pop, _from, [h|t]) do
    { :reply, h, t }
  end

  def handle_call(request, from, config) do
    super(request, from, config)
  end

  def handle_cast({ :push, item }, config) do
    { :noreply, [item|config] }
  end

  def handle_cast(request, config) do
    super(request, config)
  end
end
```

Callbacks are NOT called explicitly

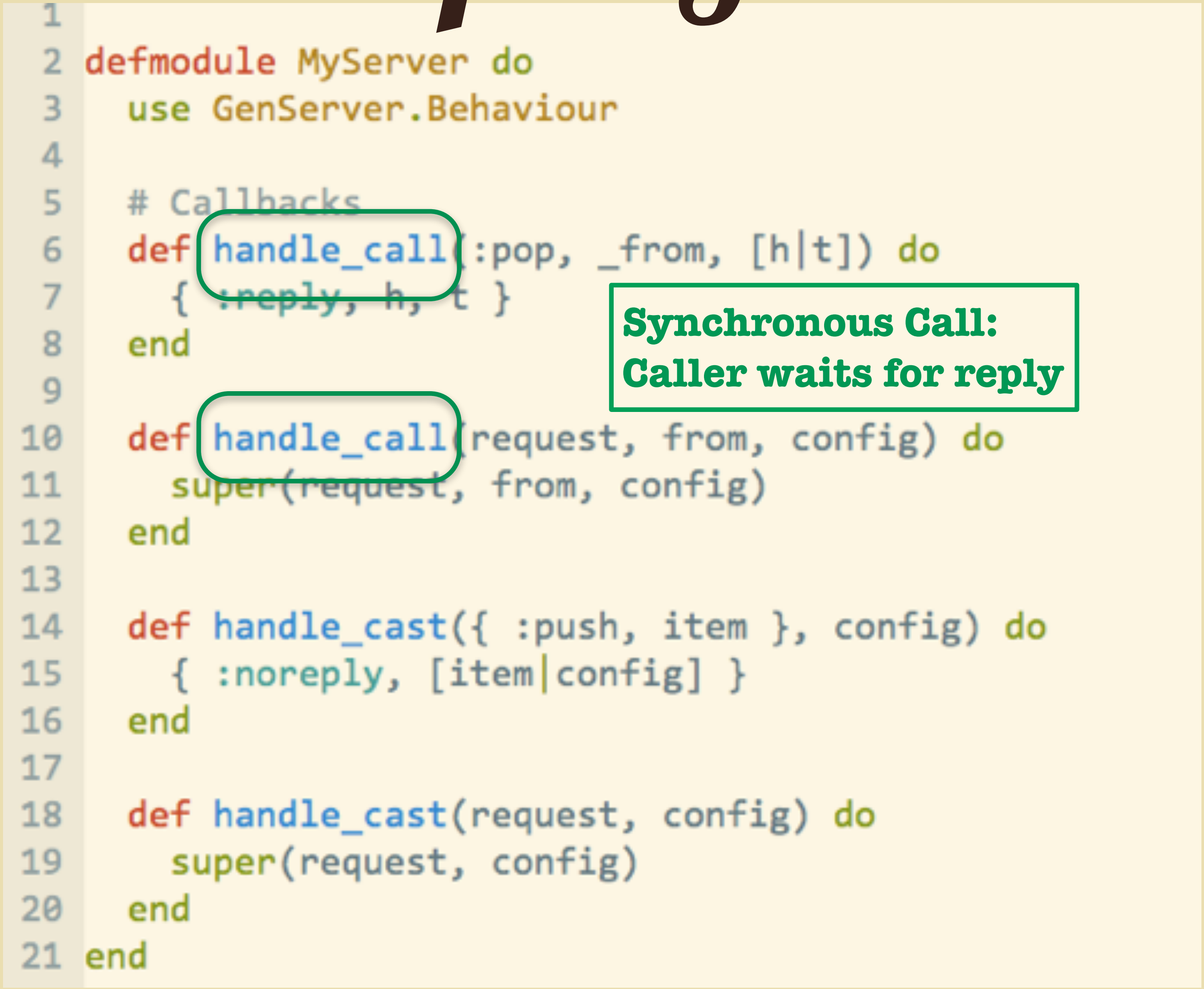
An Example GenServer



```

defmodule MyServer do
  use GenServer.Behaviour

  # Callbacks
  def handle_call(:pop, _from, [h|t]) do
    { :reply, h, t }
  end

  def handle_call(request, from, config) do
    super(request, from, config)
  end

  def handle_cast({ :push, item }, config) do
    { :noreply, [item|config] }
  end

  def handle_cast(request, config) do
    super(request, config)
  end
end
```

OTP calls the callbacks.

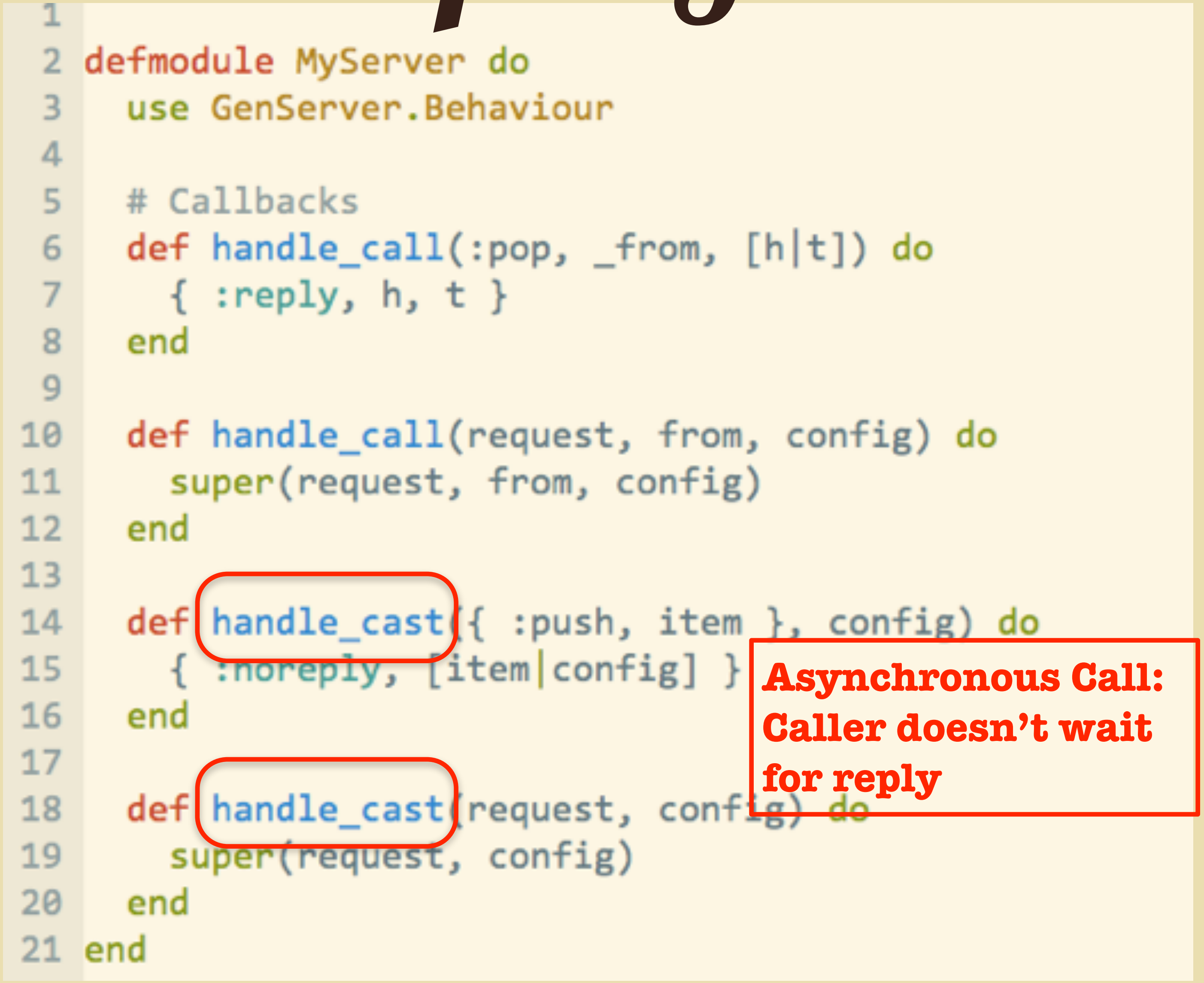
An Example GenServer



```
defmodule MyServer do
  use GenServer.Behaviour

  # Callbacks
  def handle_call(:pop, _from, [h|t]) do
    { :reply, h, t }
  end

  def handle_call(request, from, config) do
    super(request, from, config)
  end

  def handle_cast({ :push, item }, config) do
    { :noreply, [item|config] }
  end

  def handle_cast(request, config) do
    super(request, config)
  end
end
```

**Synchronous Call:
Caller waits for reply**

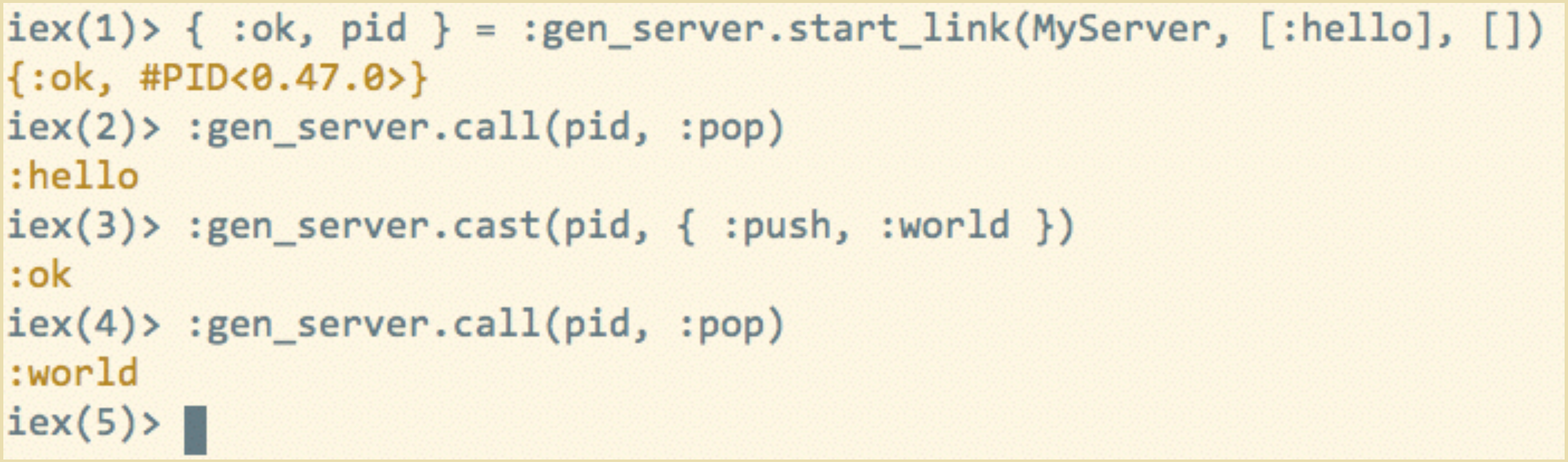
An Example GenServer



```

defmodule MyServer do
  use GenServer.Behaviour

  # Callbacks
  def handle_call(:pop, _from, [h|t]) do
    { :reply, h, t }
  end

  def handle_call(request, from, config) do
    super(request, from, config)
  end

  def handle_cast({ :push, item }, config) do
    { :noreply, [item|config] }
  end

  def handle_cast(request, config) do
    super(request, config)
  end
end
```

Asynchronous Call: Caller doesn't wait for reply

An Example GenServer Demo

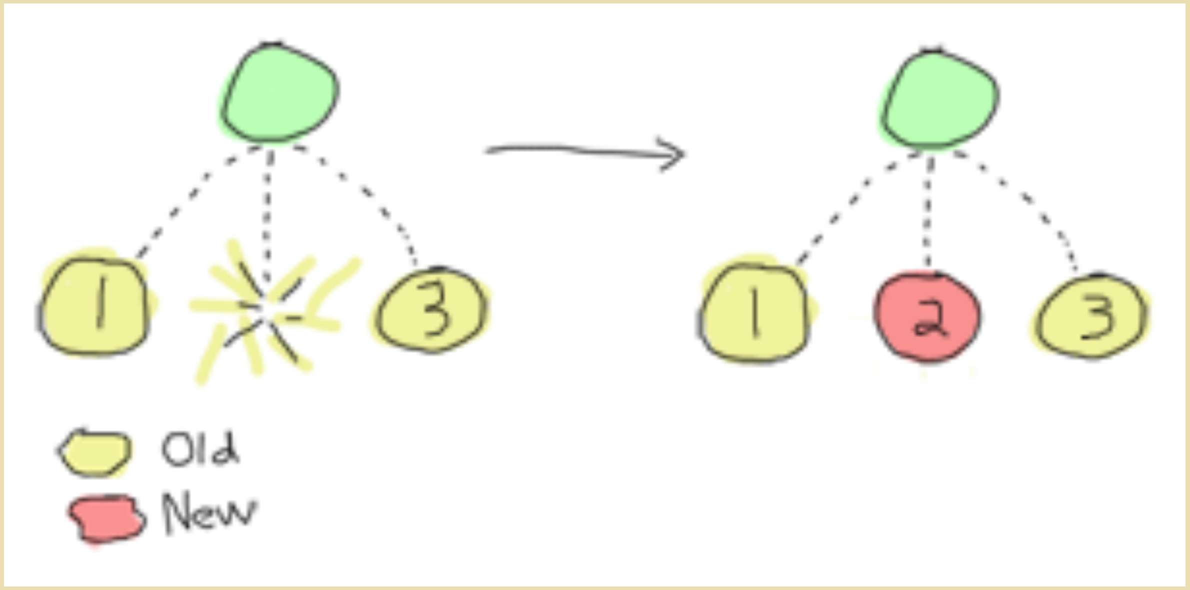


```
iex(1)> { :ok, pid } = :gen_server.start_link(MyServer, [:hello], [])
{:ok, #PID<0.47.0>}
iex(2)> :gen_server.call(pid, :pop)
:hello
iex(3)> :gen_server.cast(pid, { :push, :world })
:ok
iex(4)> :gen_server.call(pid, :pop)
:world
iex(5)>
```

Supervisors

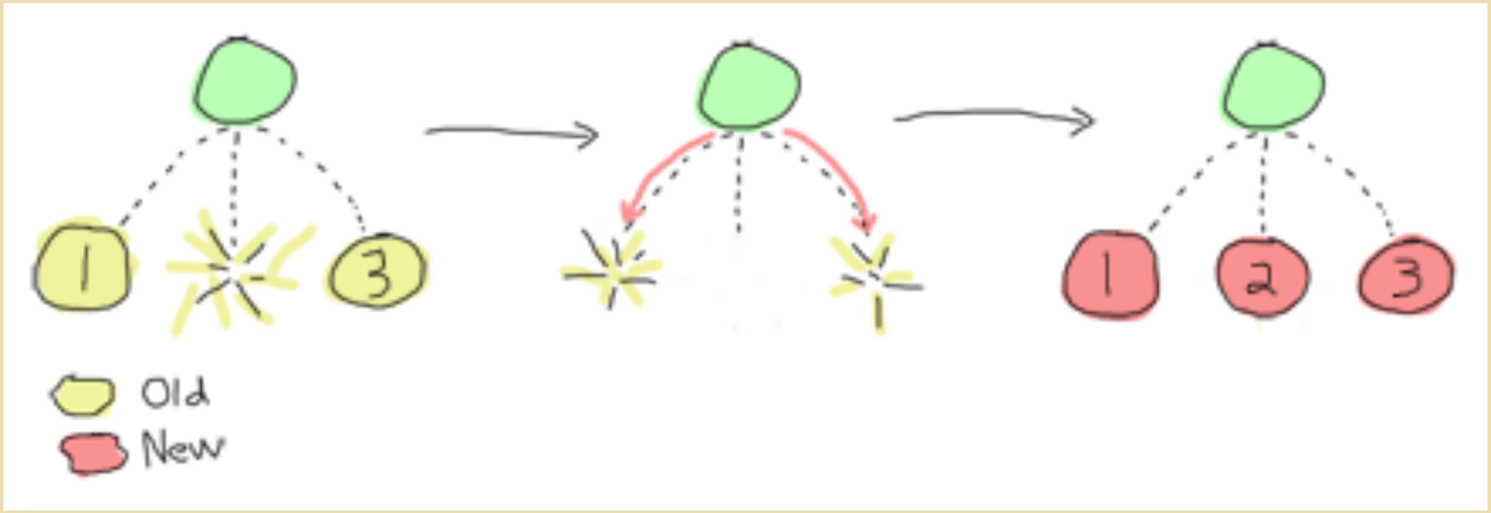
Fault Tolerance & Recovery

Supervisors for Fault Tolerance and Recovery



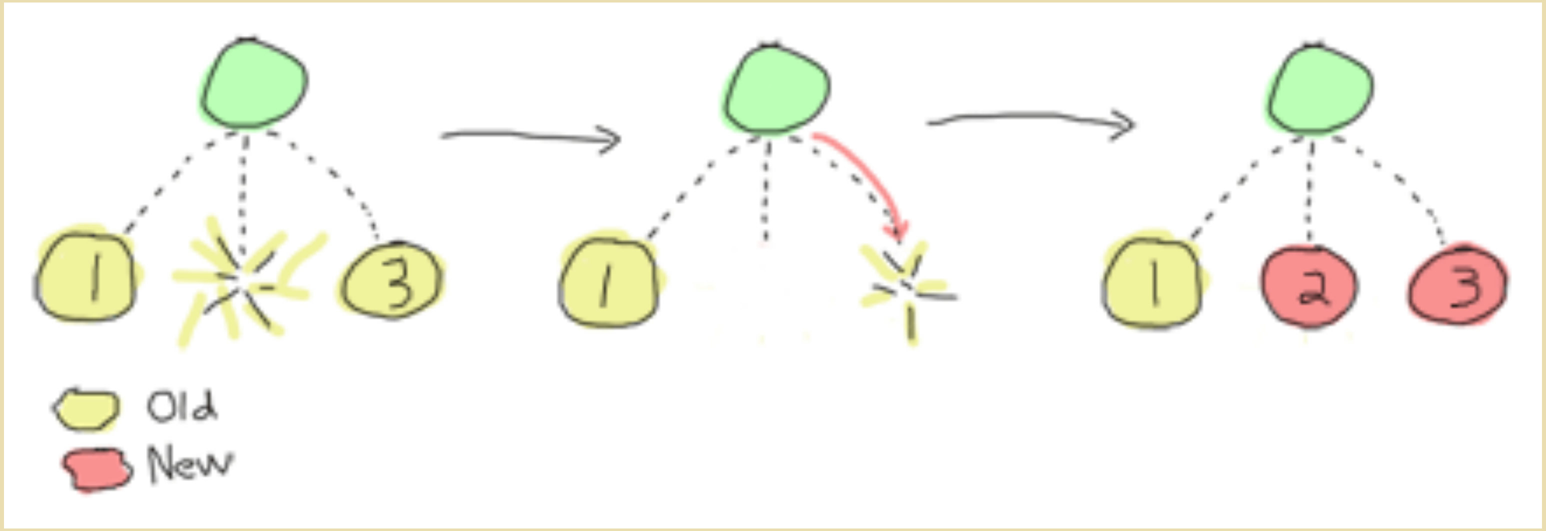
one_for_one restart strategy

Supervisors for Fault Tolerance and Recovery



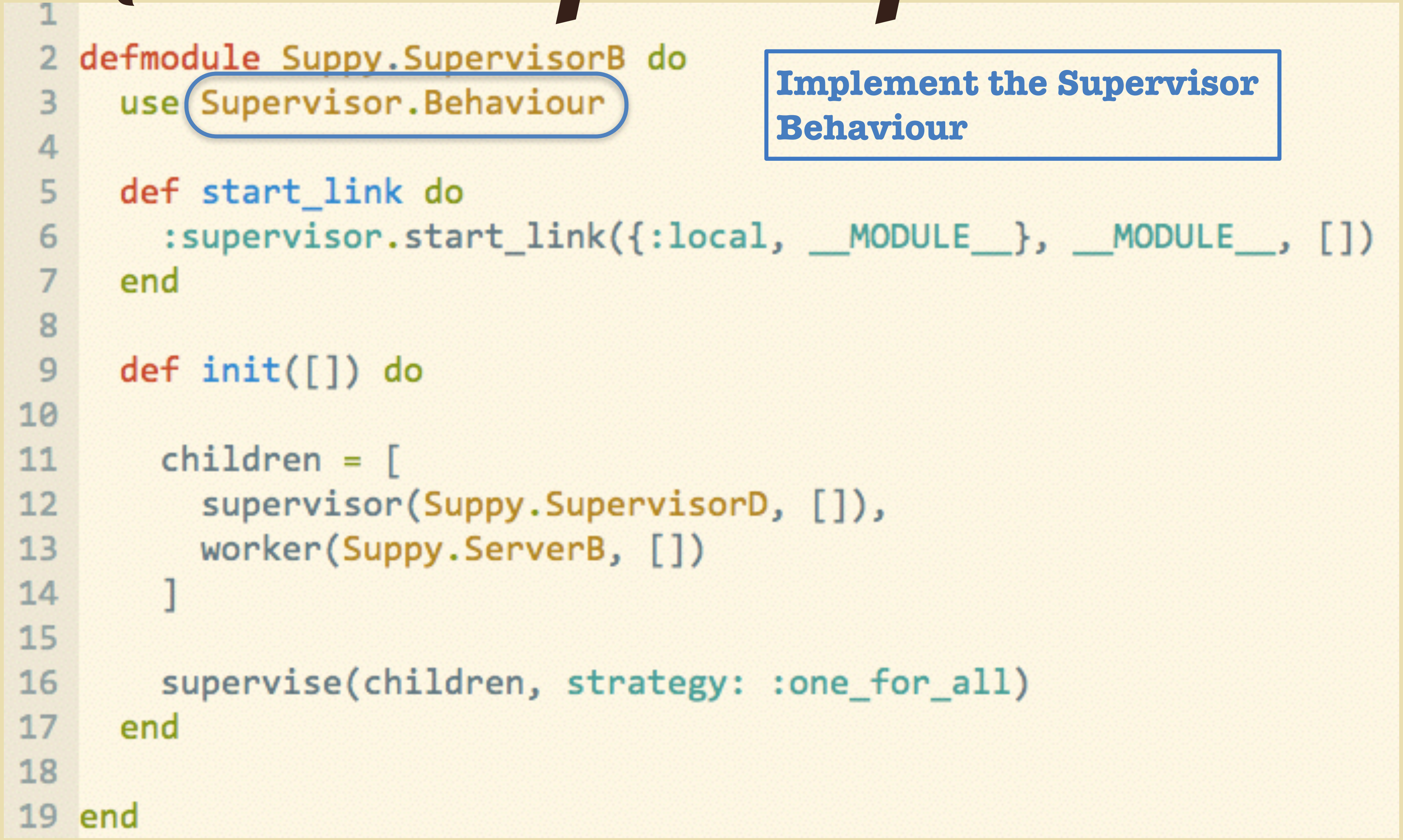
rest_for_all restart strategy

Supervisors for Fault Tolerance and Recovery



rest_for_one restart strategy

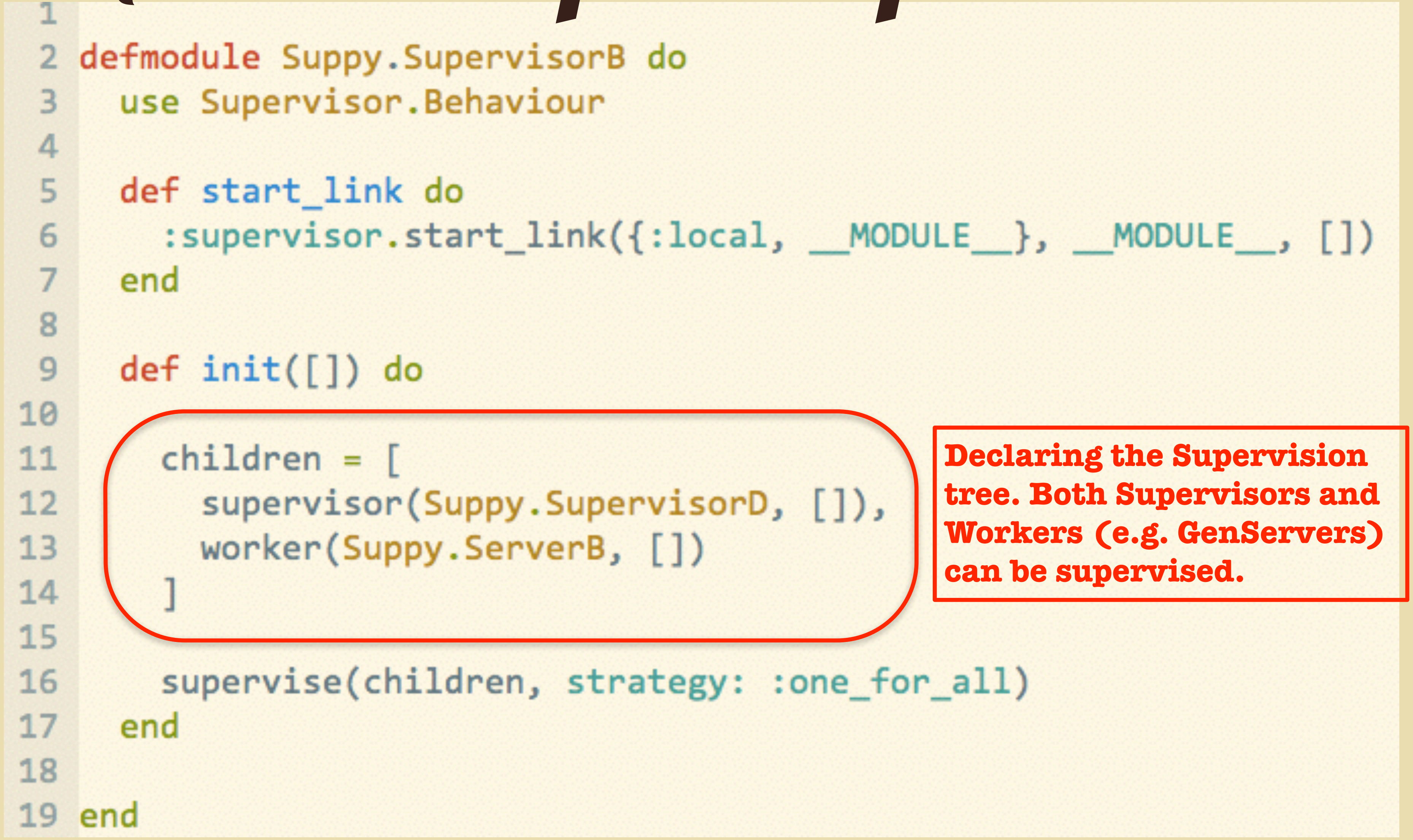
An Example Supervisor



```
defmodule Suppy.SupervisorB do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link({:local, __MODULE__}, __MODULE__, [])
  end

  def init([]) do

    children = [
      supervisor(Suppy.SupervisorD, []),
      worker(Suppy.ServerB, [])
    ]

    supervise(children, strategy: :one_for_all)
  end

end
```

Implement the Supervisor Behaviour

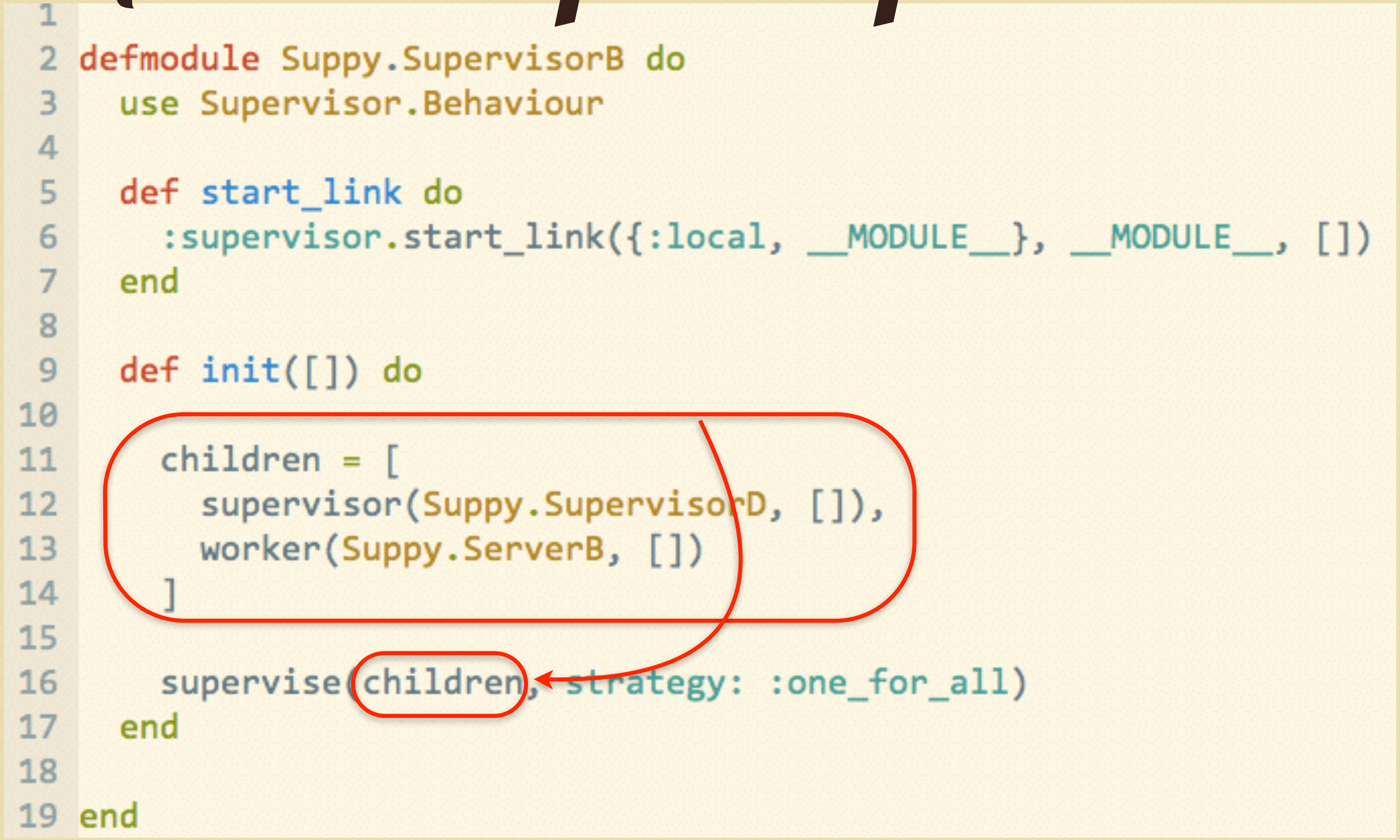
An Example Supervisor



```elixir

defmodule Suppy.SupervisorB do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link({:local, __MODULE__}, __MODULE__, [])
  end

  def init([]) do

    children = [
      supervisor(Suppy.SupervisorD, []),
      worker(Suppy.ServerB, [])
    ]

    supervise(children, strategy: :one_for_all)
  end

end
```

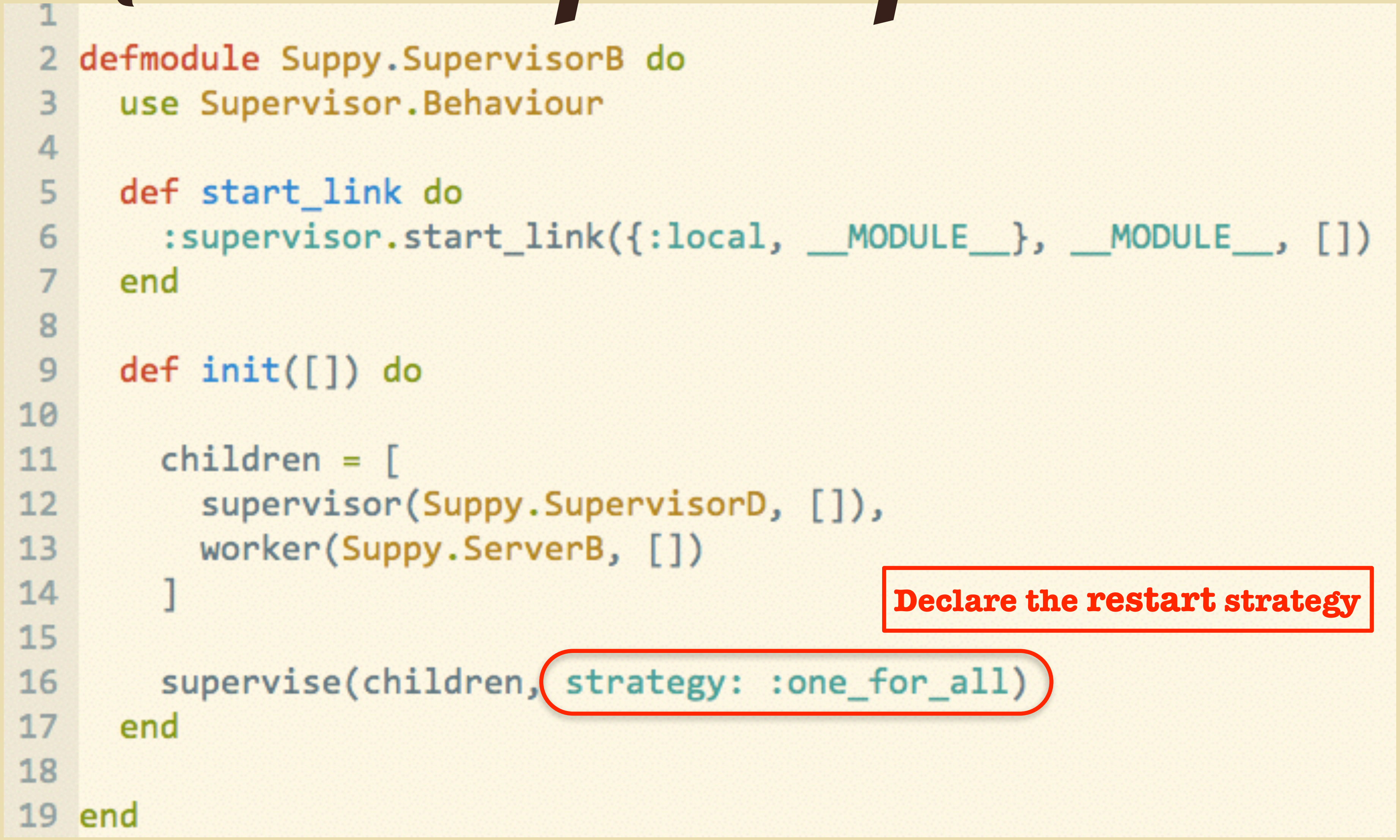
Declaring the Supervision tree. Both Supervisors and Workers (e.g. GenServers) can be supervised.

An Example Supervisor



```

defmodule Suppy.SupervisorB do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link({:local, __MODULE__}, __MODULE__, [])
  end

  def init([]) do

    children = [
      supervisor(Suppy.SupervisorD, []),
      worker(Suppy.ServerB, [])
    ]

    supervise(children, strategy: :one_for_all)
  end

end
```

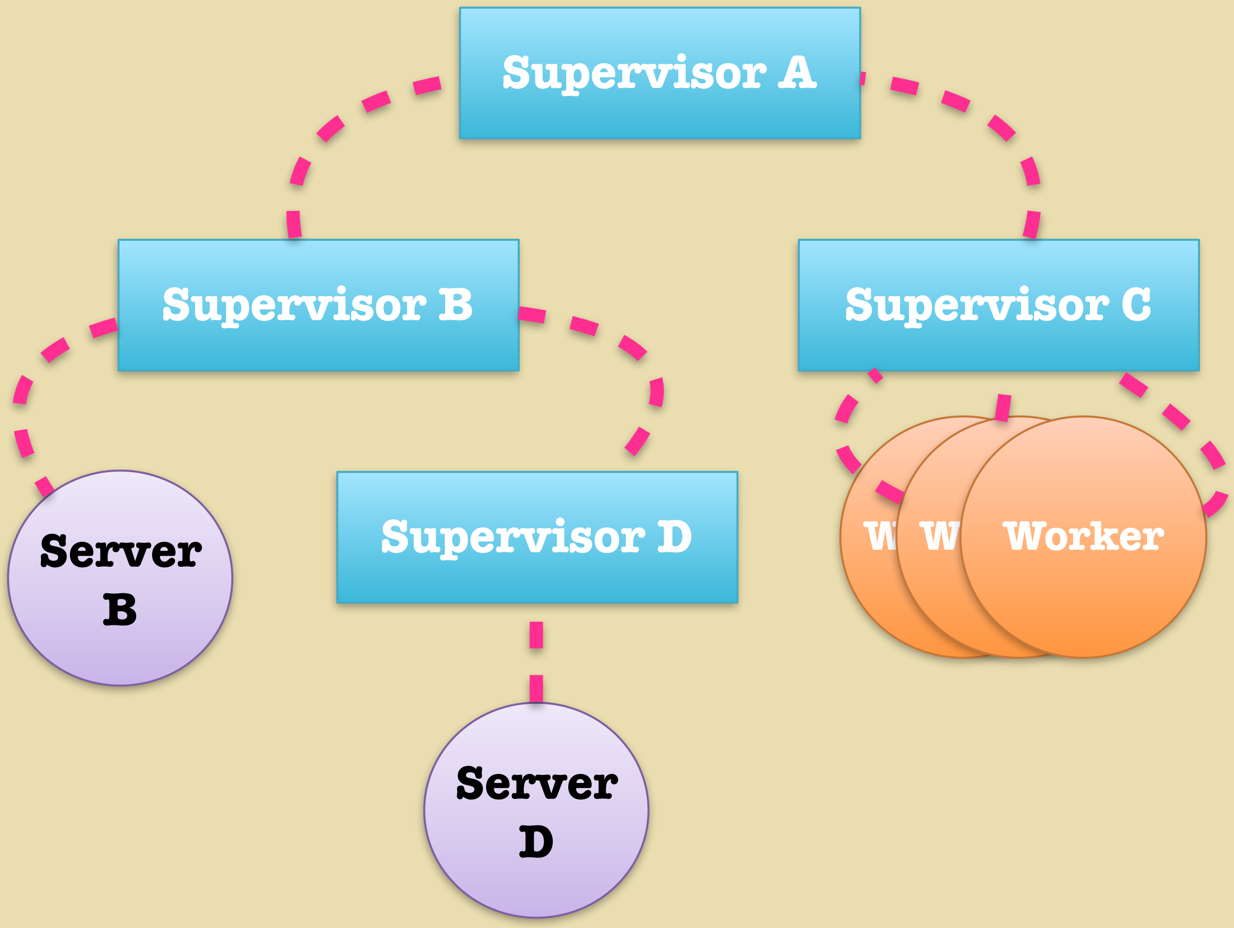
An Example Supervisor



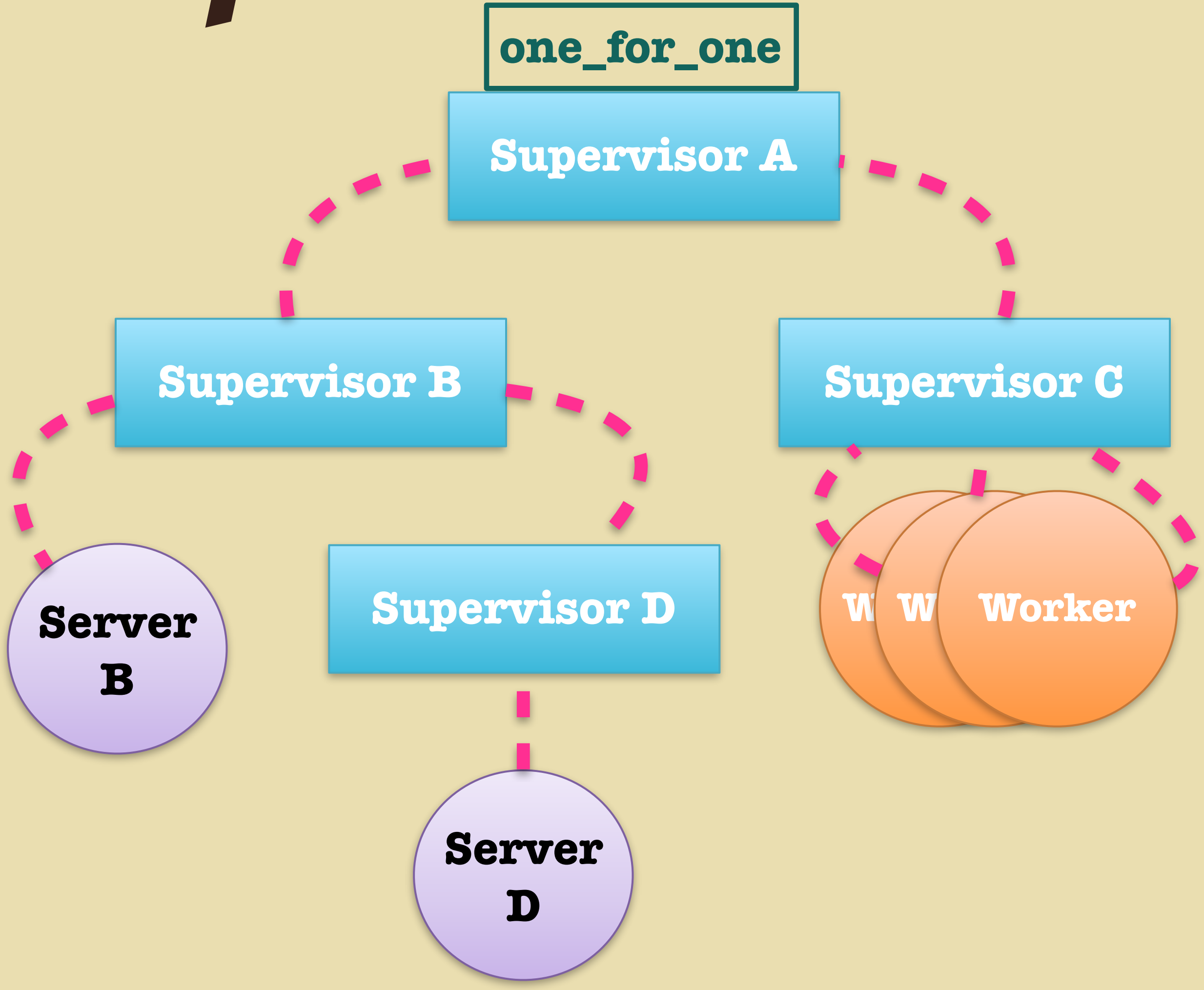
```

defmodule Suppy.SupervisorB do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link({:local, __MODULE__}, __MODULE__, [])
  end

  def init([]) do

    children = [
      supervisor(Suppy.SupervisorD, []),
      worker(Suppy.ServerB, [])
    ]

    supervise(children, strategy: :one_for_all)
  end

end
```

Declare the restart strategy

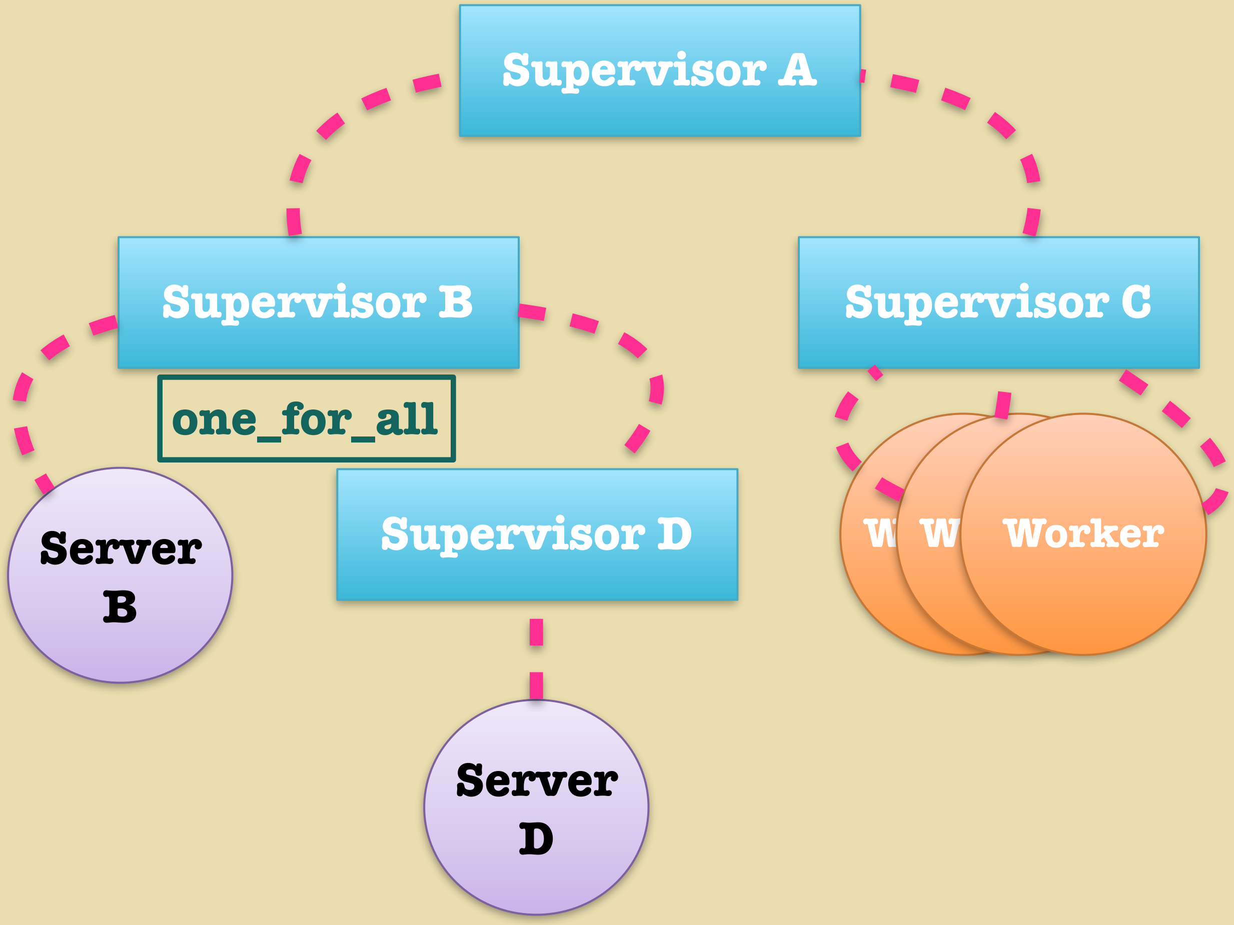
Supervisor Demo



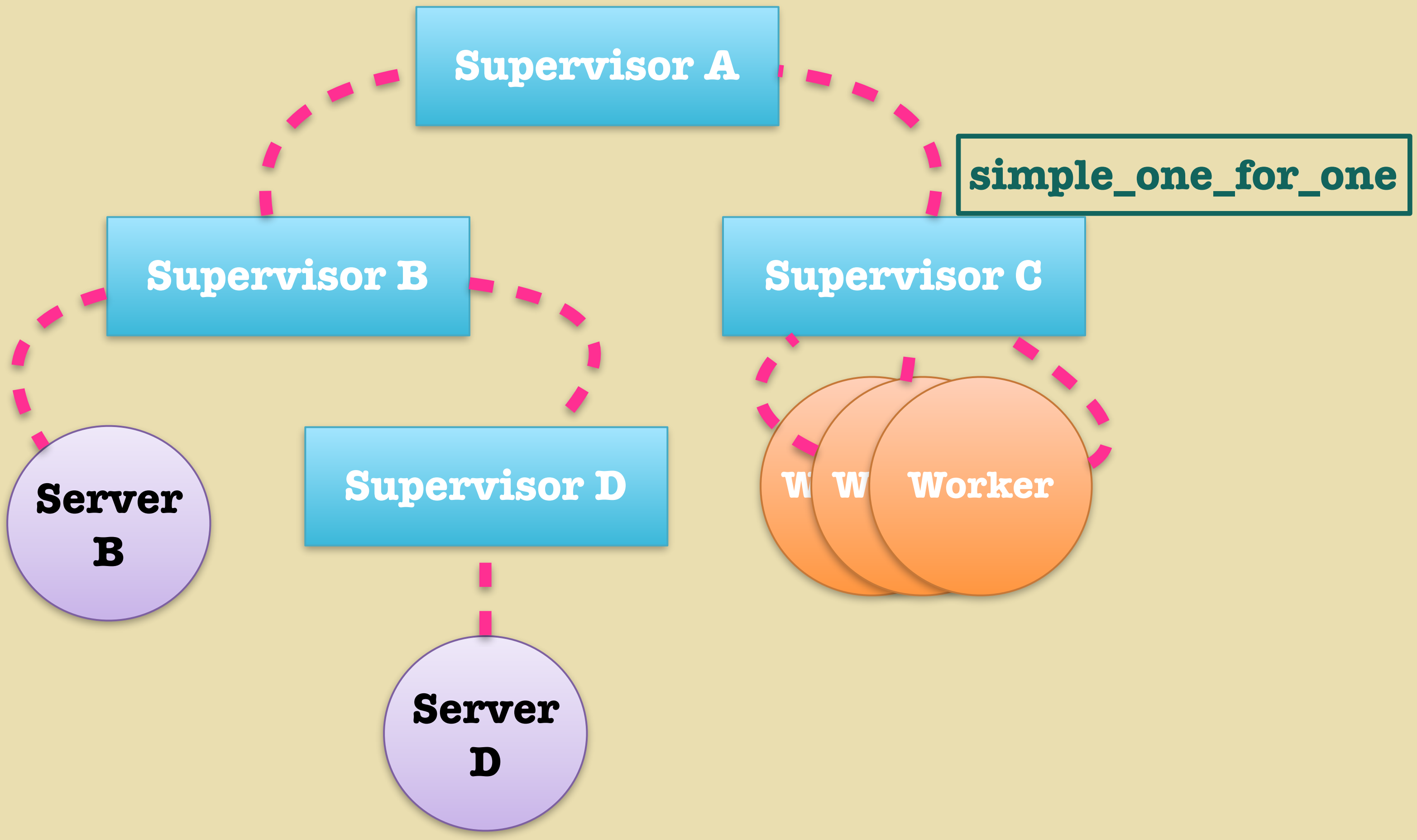
Supervisor Demo



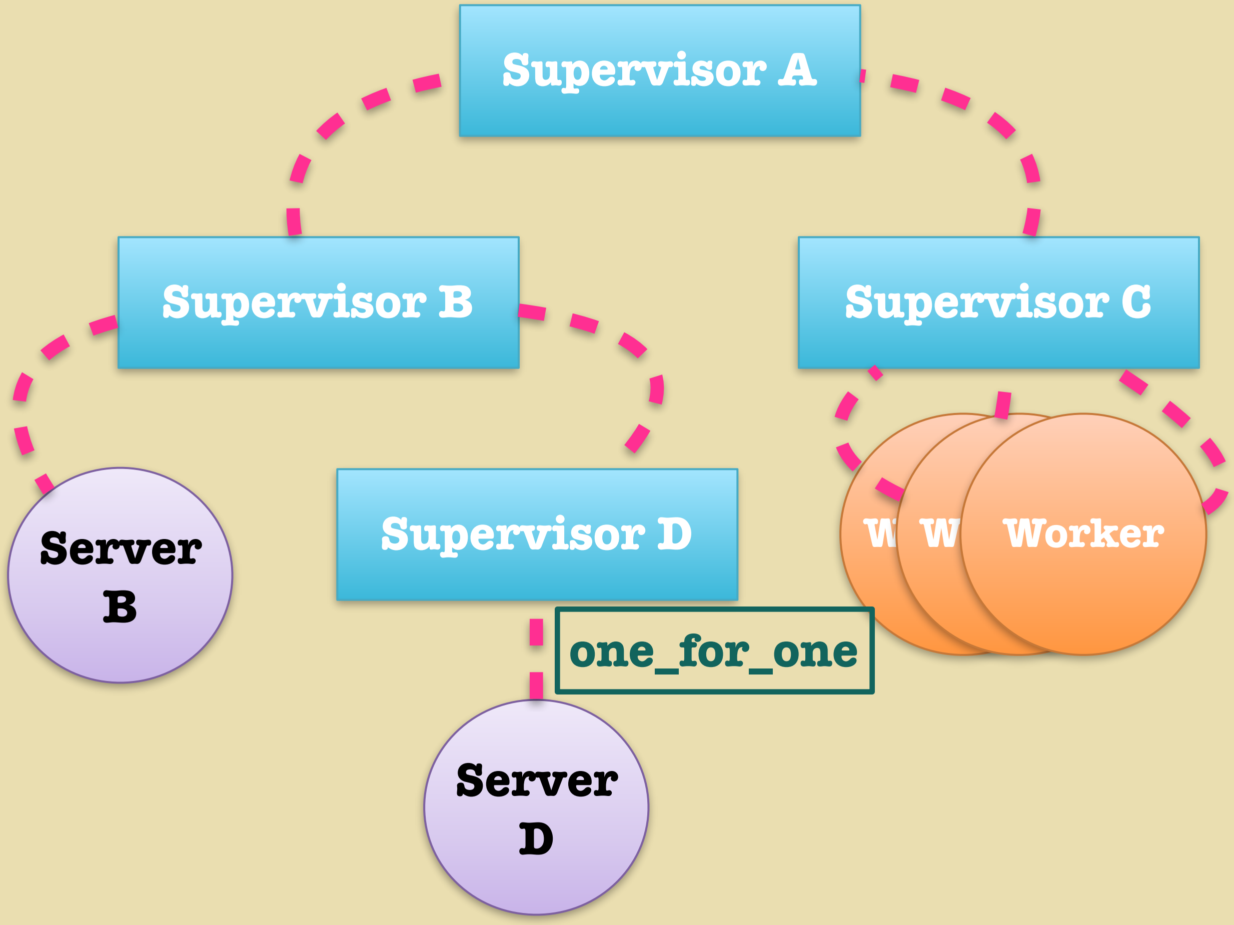
Supervisor Demo



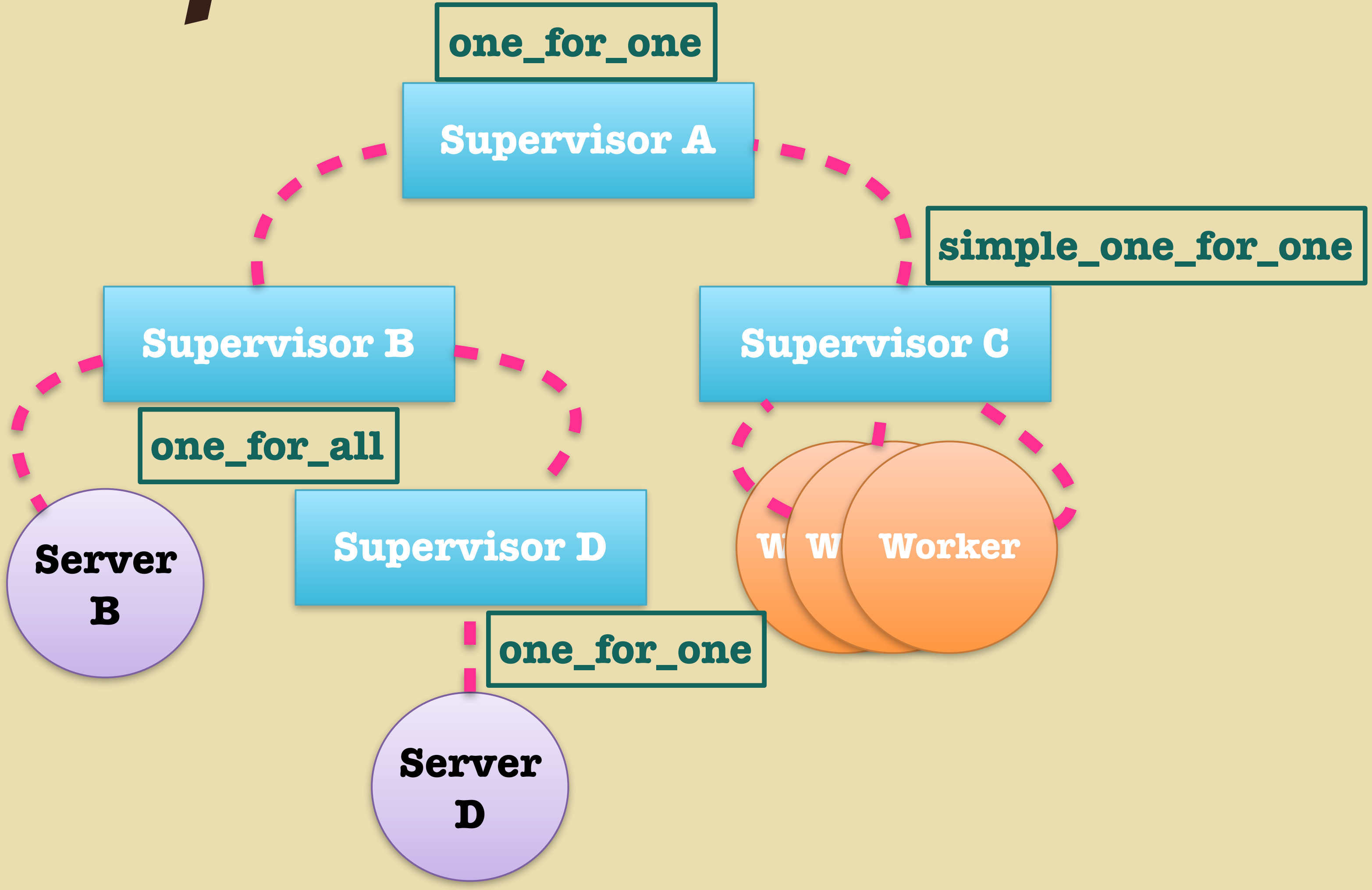
Supervisor Demo



Supervisor Demo



Supervisor Demo



Elixir & Erlang

In less than 5 minutes

Processes 101

The Basic Concurrency Primitive

OTP

Framework and much more

Supervisors

Fault Tolerance & Recovery

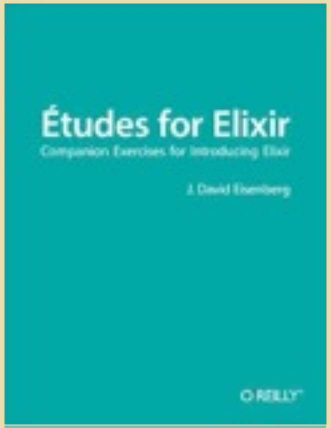
Resources

Getting Started in Functional Programming

Early Release
RAW & UNEDITED

Introducing
Elixir

O'REILLY
Simon St. Laurent & J. David Eisenberg

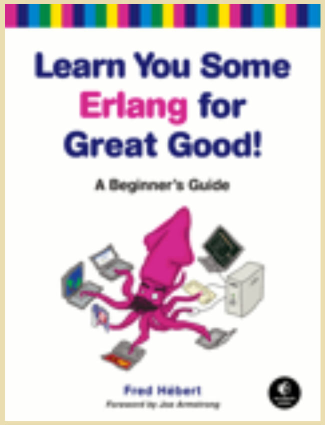

Études for Elixir
Companion Exercises for Introducing Elixir
J. David Eisenberg

O'REILLY

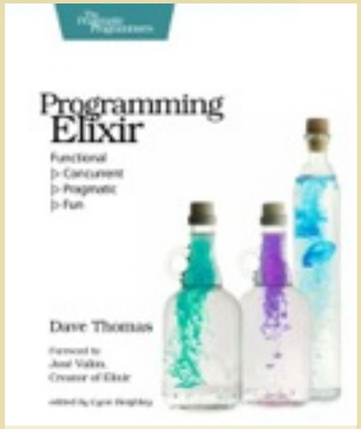
Meet elixir
by elixir creator José Valim
with Geoffrey Grosenbach

elixir sips
with Josh Adams

Learn elixir with two short videos each week.


Learn You Some Erlang for Great Good!
A Beginner's Guide

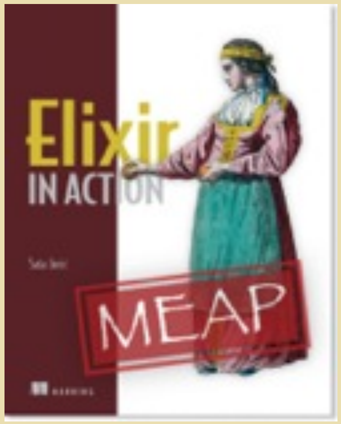
Fred Hébert
Foreword by Joe Armstrong


The Pragmatic Programmers

Programming Elixir
Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas
Foreword by
José Valim,
Creator of Elixir

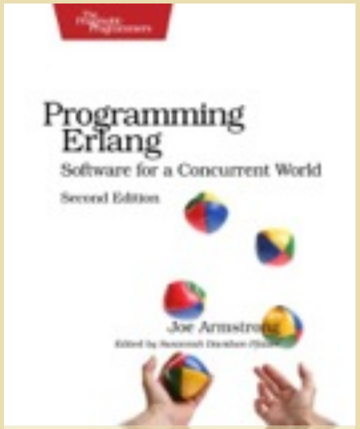
edited by Lynn Beighley


Elixir
IN ACTION

Saša Jurić

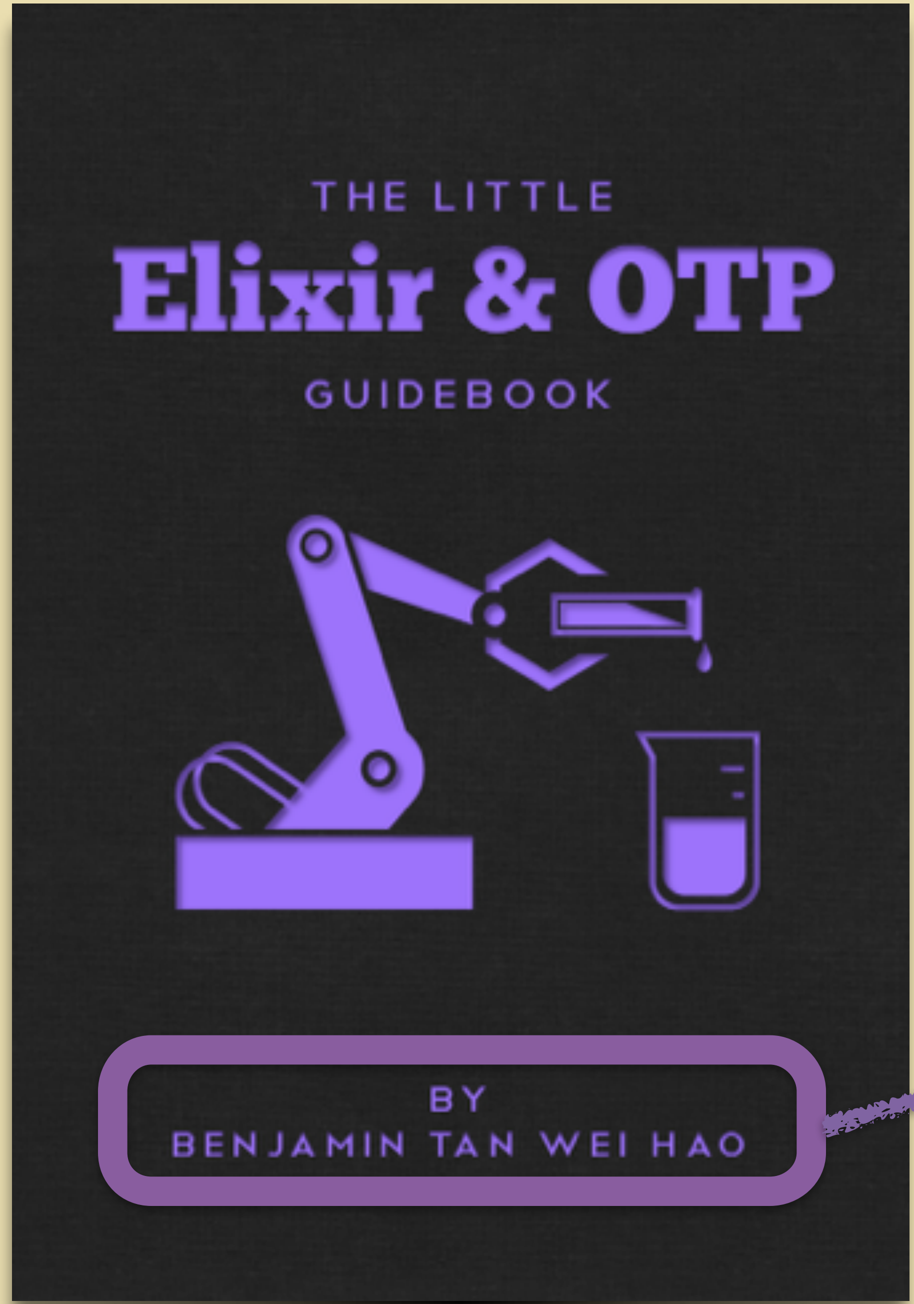
MEAP

MANNING


The Pragmatic Programmers

Programming Erlang
Software for a Concurrent World
Second Edition

Joe Armstrong



Work in Progress!

Sign up at:
http://www.exotpbook.com

Me.

Stickers!*

* Very Limited Quantity

Thanks!

benjamintanweihao@gmail.com

/benjamintanweihao

/bentanweihao