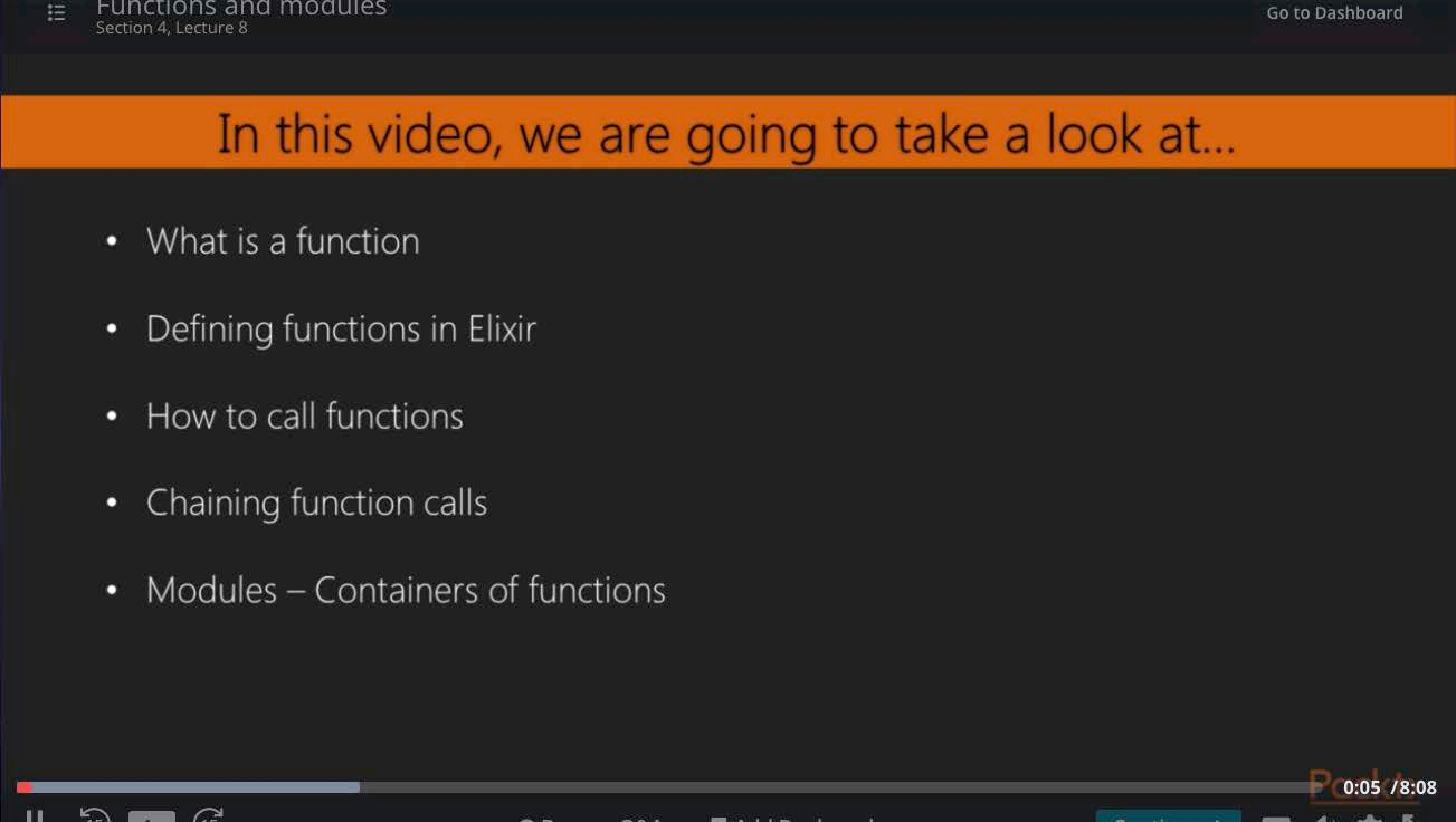
Elixir: Scalable and Efficient Application Development

João Gonçalves

Functions and Modules

Packt>



In this video, we are going to take a look at...

- What is a function
- Defining functions in Elixir
- How to call functions
- Chaining function calls
- Modules – Containers of functions

What is a Function?

$$f(x) \rightarrow y$$

What is a Function?

$$\underset{\text{“Name”}}{f}(\underset{\text{Arguments}}{x}) \to \underset{\text{Return value}}{y}$$

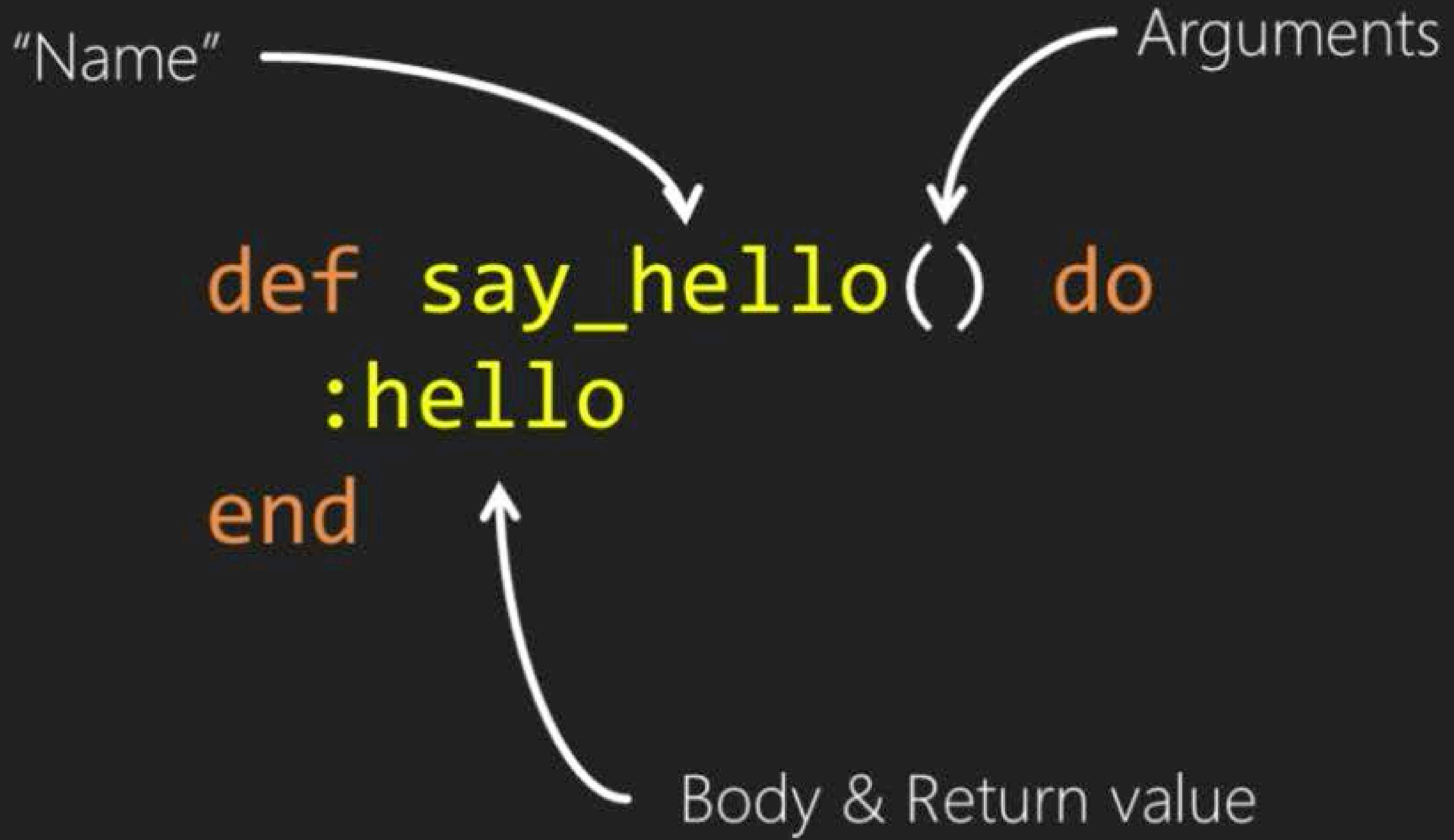
Why Functions?

- Reuse computations
- Combine to express more powerful computations

A Function in Elixir

```
def say_hello() do
  :hello
end
```

A Function in Elixir



```
def say_hello() do
  :hello
end
```

Calling a Function

$$\begin{tikzcd}\texttt{say\_hello()} \arrow[d] \\ \texttt{:hello}\end{tikzcd}$$

Function Notation

- Elixir allows the definition of functions with the same name but with different arity

`say_hello/0` ⟵ Arity

Default Arguments

```
def say_hello(name) do
  “Hello #{name}”
end
```

Packt>

Default Arguments

```
def say_hello(name\\ “you”) do
  “Hello #{name}”
end
```

Chaining Function Calls

```
def person do
  %{first_name: “Joe”, last_name: “Smith”}
end

def full_name(person) do
  “#{person.first_name} #{person.last_name}”
end

def say_hello(name, from) do
  “#{from} says: Hello #{name}!”
end
```

```
say_hello(full_name(person), “Jeff”)
```

Chaining Function Calls

```
def person do
  %{first_name: “Joe”, last_name: “Smith”}
end

def full_name(person) do
  “#{person.first_name} #{person.last_name}”
end

def say_hello(name, from) do
  “#{from} says: Hello #{name}!”
end
```

```
person |> full_name |> say_hello(“Jeff”)
```

Chaining Function Calls

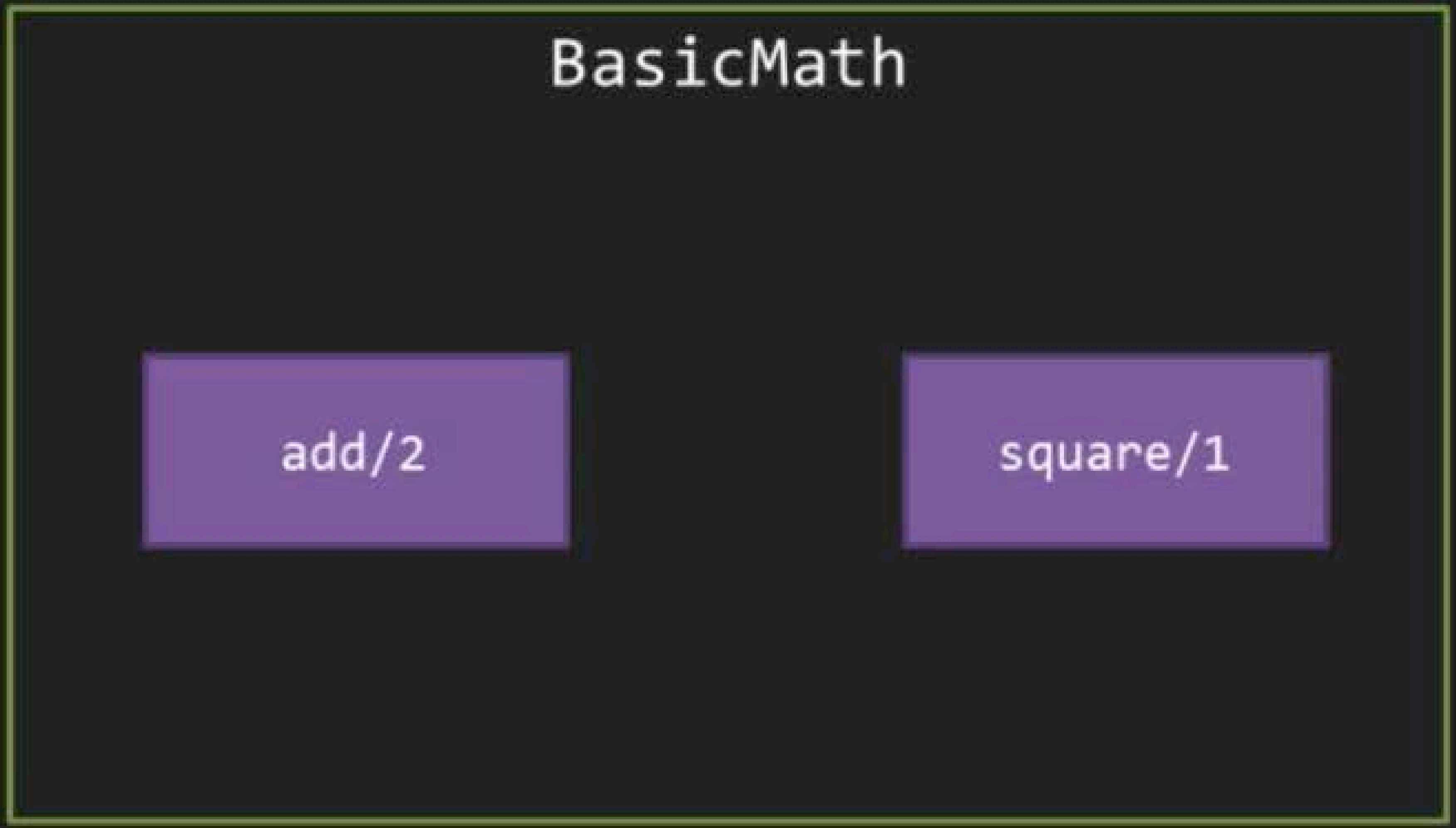
- Injects the value on the left as the first argument of the function on the right

"Pipe" operator

```
person |> full_name |> say_hello("Jeff")
```

Modules

- A group of closely related functions



Modules in Elixir

```elixir
defmodule BasicMath do
  def add(x,y), do: x + y
  def square(x) do
    x * x
  end
end
```

Calling a Function from a Module

```
BasicMath.add(2,5)
```

$$\downarrow$$

```
7
```

Composing Modules

```
defmodule ComplexMath do

  def cube(x) do
    BasicMath.square(x) * x
  end
end
```

Packt>

Composing Modules

```
defmodule ComplexMath do
  alias BasicMath, as: Math
  def cube(x) do
    Math.square(x) * x
  end
end
```

Composing Modules

```
defmodule ComplexMath do
  import BasicMath
  def cube(x) do
    square(x) * x
  end
end
```

Composing Modules

```
defmodule ComplexMath do
  import BasicMath, only: [square: 1]
  def cube(x) do
    square(x) * x
  end
end
```

Composing Modules

`alias`

Reference a module by a different name

`import`

Include the functions of a module

Private Functions

```
defmodule Example do
  def hello, do: say_hello
  def say_hello do
    :hello
  end
end
```

Private Functions

```
defmodule Example do
  def hello, do: say_hello
  defp say_hello do
    :hello
  end
end
```

Only visible by functions in the module →

Packt>

Constants

```
defmodule Example do
  @hello = :hello
  def hello do
    @hello
  end
end
```

Constants

Constant

```
defmodule Example do
  @hello = :hello
  def hello do
    @hello
  end
end
```

Value is bound at compile-time

Packt>

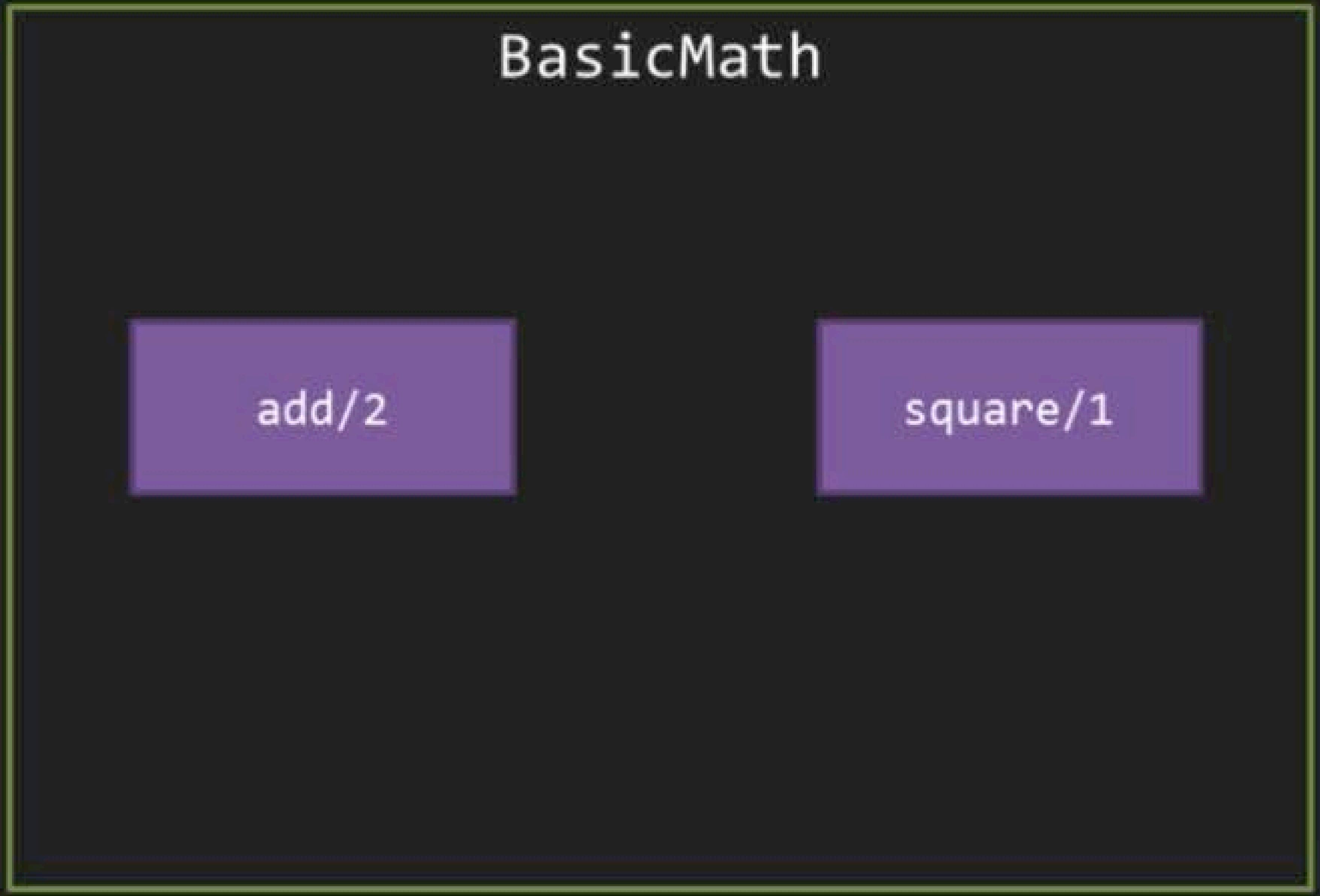
João Gonçalves

Pattern Matching in Functions

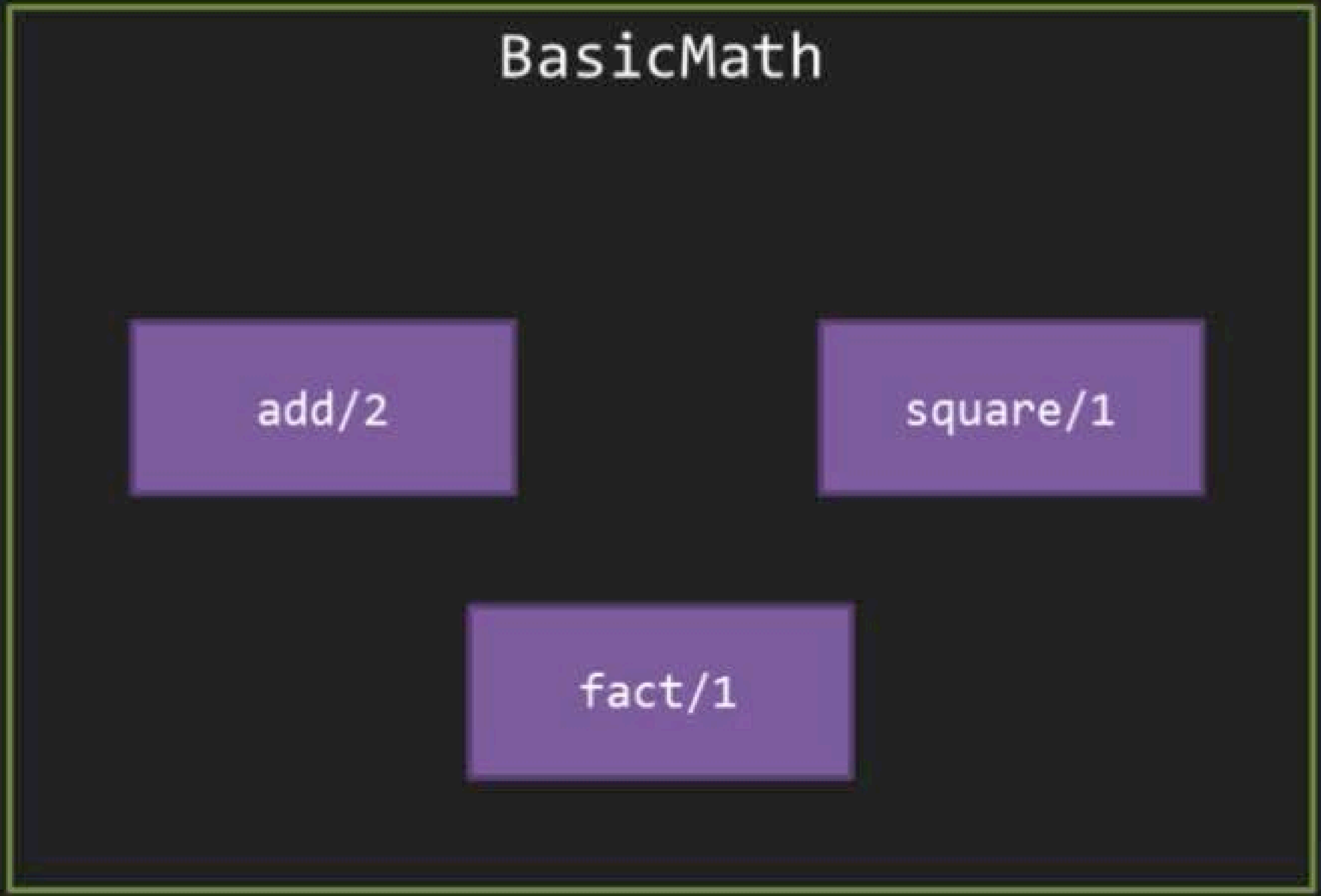
In this video, we are going to take a look at…

- How to leverage pattern matching in function calls
- Using function guards

Extending BasicMath



Extending BasicMath



Packt>

The Factorial Function

$$n! = \prod_{k=1}^{n} k$$

The Factorial Function

$$n! = f(n) = \begin{cases} n \times f(n-1) \;\; if\; n > 0 \\ 1\; if\; n \leq 0 \end{cases}$$

The Factorial Function

```
def fact(n) do
  if (n > 0) do
    n * fact(n - 1)
  else
    1
  end
end
```

And this is perfectly fine ☺
But we can do better

Packt›

The Factorial Function

```elixir
def fact(0) do
  1
end
```

The Factorial Function

```
def fact(0) do
  1
end

def fact(n) do
  n * fact(n - 1)
end
```

The Factorial Function

```
def fact(n) do
  n * fact(n - 1)
end

def fact(0) do
  1
end
```

Pattern Matching

```
def process({:ok, result}) do
  result
end

def process({:error, _}) do
  :failure
end

def process(_) do
  :unknown
end
```

Something to Note

```
def fact(0) do
  1
end

def fact(n) do
  n * fact(n - 1)
end
```

Guard Clauses

```
def fact(n) do
  n * fact(n - 1)
end
```

Guard Clauses

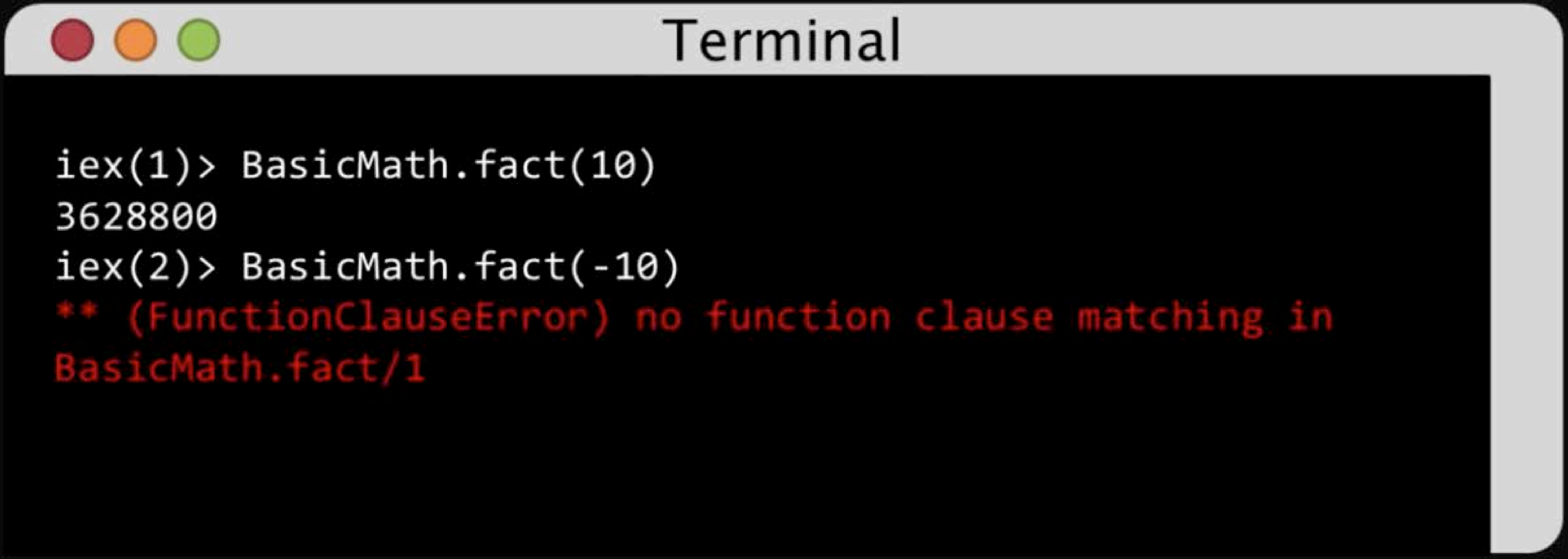
```
def fact(n) when is_integer(n) do
  n * fact(n - 1)
end
```

The Final Factorial Function

```
def fact(0) do
  1
end

def fact(n) when is_integer(n) and n > 0 do
  n * fact(n - 1)
end
```

The Final Factorial Function



```
iex(1)> BasicMath.fact(10)
3628800
iex(2)> BasicMath.fact(-10)
** (FunctionClauseError) no function clause matching in
BasicMath.fact/1
```

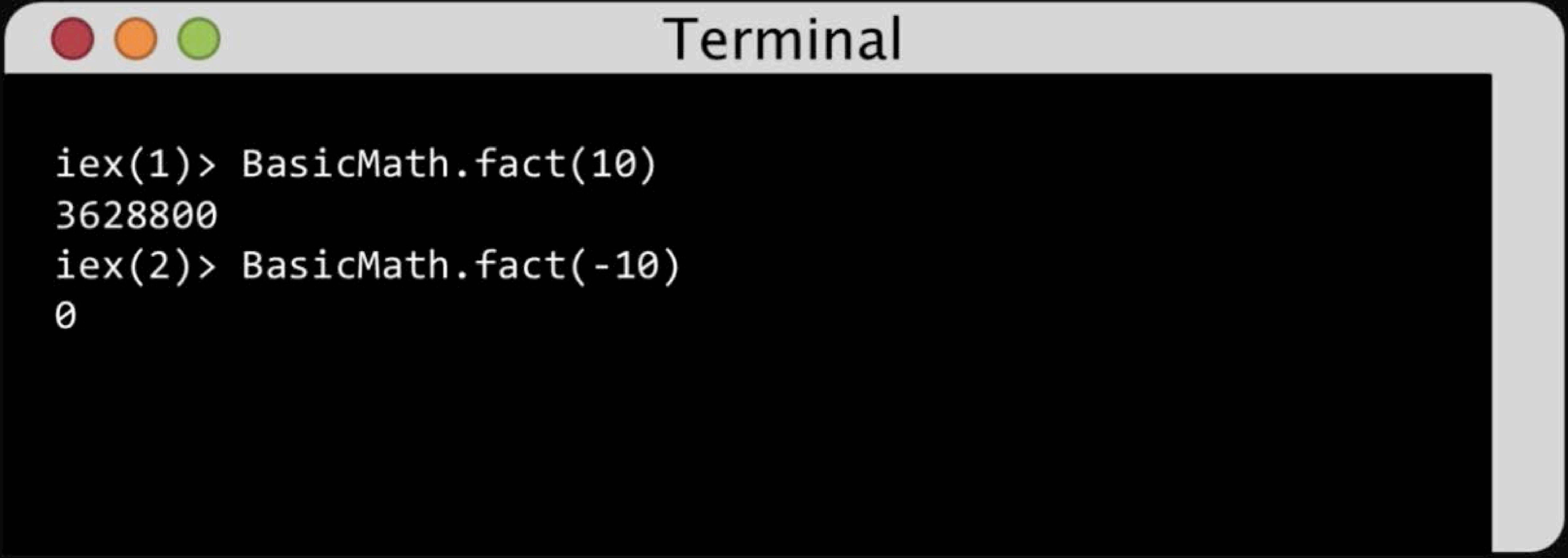
Back to the Drawing Board

```
def fact(0) do
  1
end

def fact(n) when is_integer(n) and n > 0 do
  n * fact(n - 1)
end

def fact(_) do
  0
end
```

Finally



```
iex(1)> BasicMath.fact(10)
3628800
iex(2)> BasicMath.fact(-10)
0
```

João Gonçalves

Anonymous Functions

In this video, we are going to take a look at…

- Anonymous functions
- Concept of high order functions
- Using anonymous functions in the Enum module

Validation

- Build a function that validates that a person's age is above 18

```
%{name: "Jack", age: 30}
```

Validation

```
def validate(person) do
  person.age > 18
end
```

Validation

```
def validate(person) do
  person.age > 18
end

def validate_2(person) do
  person.age > 20 and person.age < 60
end
```

Anonymous Functions

```
fn (person) ->
  person.age > 18
end
```

Anonymous Functions

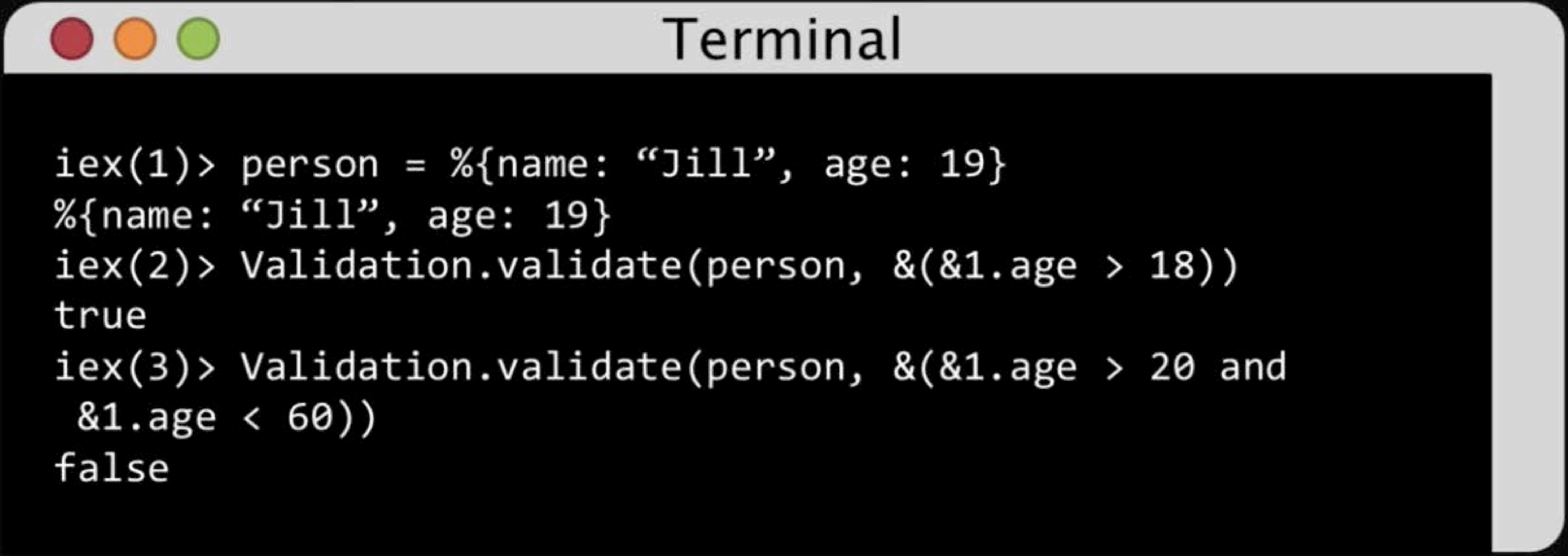
```
fn (person) ->
  person.age > 18
end

&(&1.age > 18)
```

Validation

```
def validate(person, validation) do
  validation.(person)
end
```

Validation



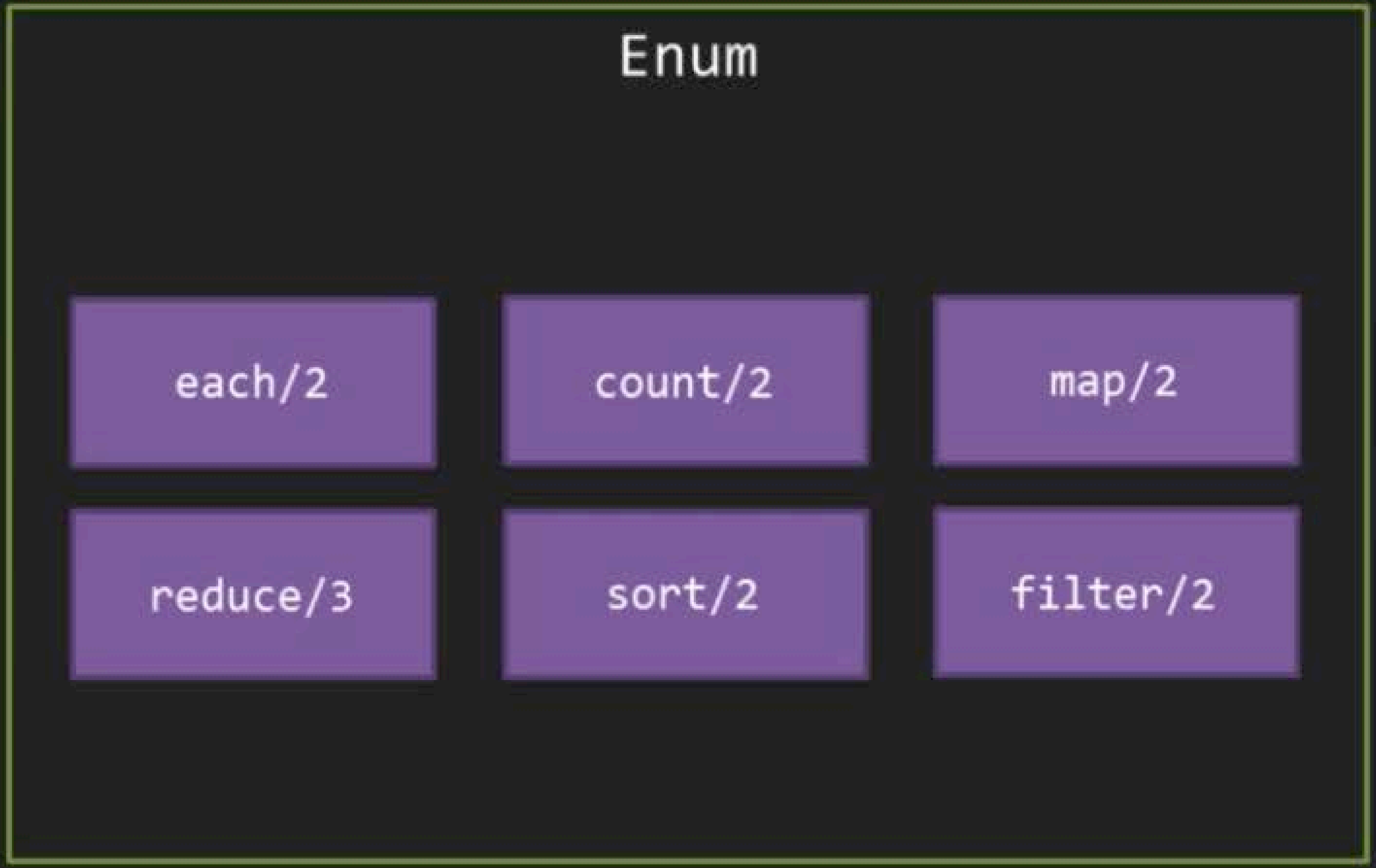
```
iex(1)> person = %{name: “Jill”, age: 19}
%{name: “Jill”, age: 19}
iex(2)> Validation.validate(person, &(&1.age > 18))
true
iex(3)> Validation.validate(person, &(&1.age > 20 and
 &1.age < 60))
false
```

Packt>

High Order Functions

- Receive functions as arguments
- Return functions

The Enum Module



The Enum Module

- Take out all the odd numbers, multiply each number by itself and count the ones above 20

- `[1,2,3,4,5,6,7,8,9,19]`

The Enum Module

```
[1,2,3,4,5,6,7,8,9,19]
```

```
def do_it(list) do
  list
  |> Enum.reject(&(is_odd(&1)))
  |> Enum.map(&BasicMath.square/1)
  |> Enum.count(&(&1 > 20))
end
```

Quiz Time!

Quiz 4 | 2 Questions

Start Quiz

Skip Quiz ›

Question 1:

Which of the following options are known as the containers of functions?

- ○ Properties
- ○ Tuples
- ○ Modules
- ○ Data types

Question 2:

Which of the following is a conditional statement at the top of a function that bails out as soon as it can?

- ◯ Loops
- ◯ Factorial functions
- ◯ Guard clauses
- ◯ Conditional statements