Elixir: Scalable and Efficient Application Development

João Gonçalves

Recursion

Packt>

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

- Concept of recursion
- Designing recursive computations

An Old Function

```
def fact(0) do
  1
end

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

An Old Function

```
def fact(0) do
  1
end

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

Calls itself over and over

Recursion

- Method of dividing complex problems into smaller ones

Recursion

- Define a base case
- Compute the solution converging towards the base case

Recursion

- Base case

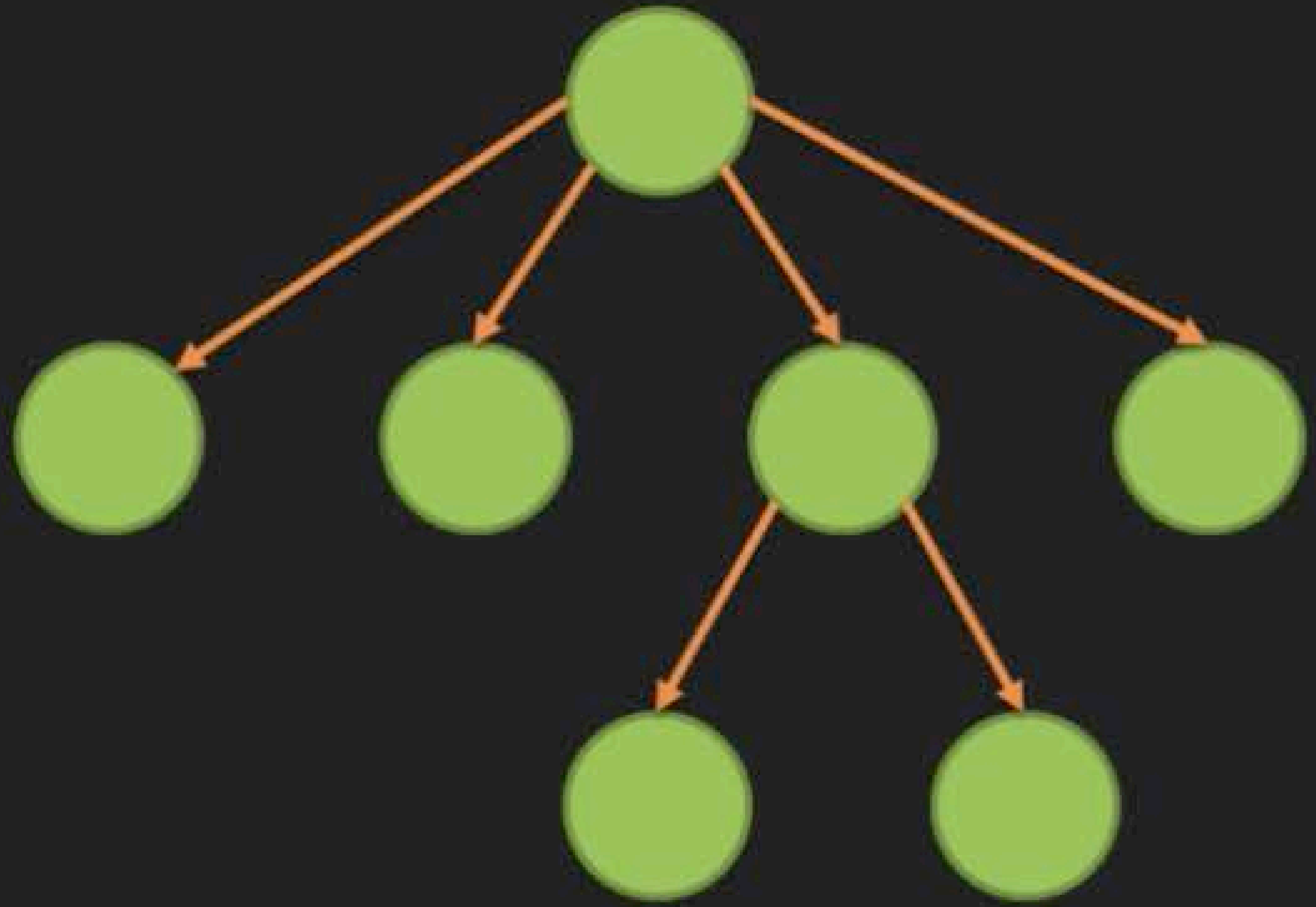
```
def fact(0) do
  1
end
```

- General function

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

A Different Problem

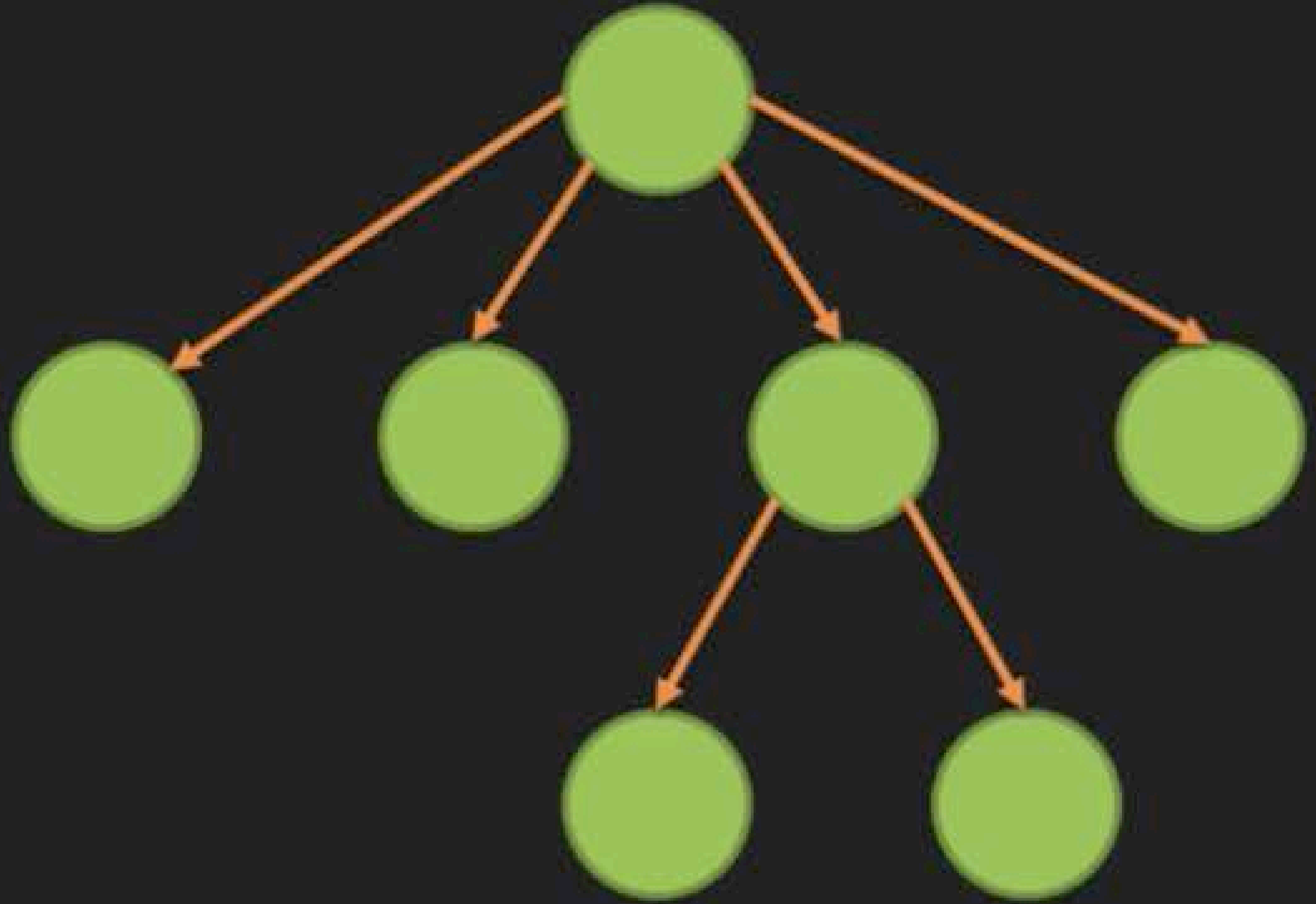
- Count the number of nodes in the graph



Packt>

A Different Problem

```
graph = %{
  children: [
    %{children: []},
    %{children: []},
    %{children: [
        %{children: []},
        %{children: []}
      ]
    },
    %{children:[]}
  ]
}
```



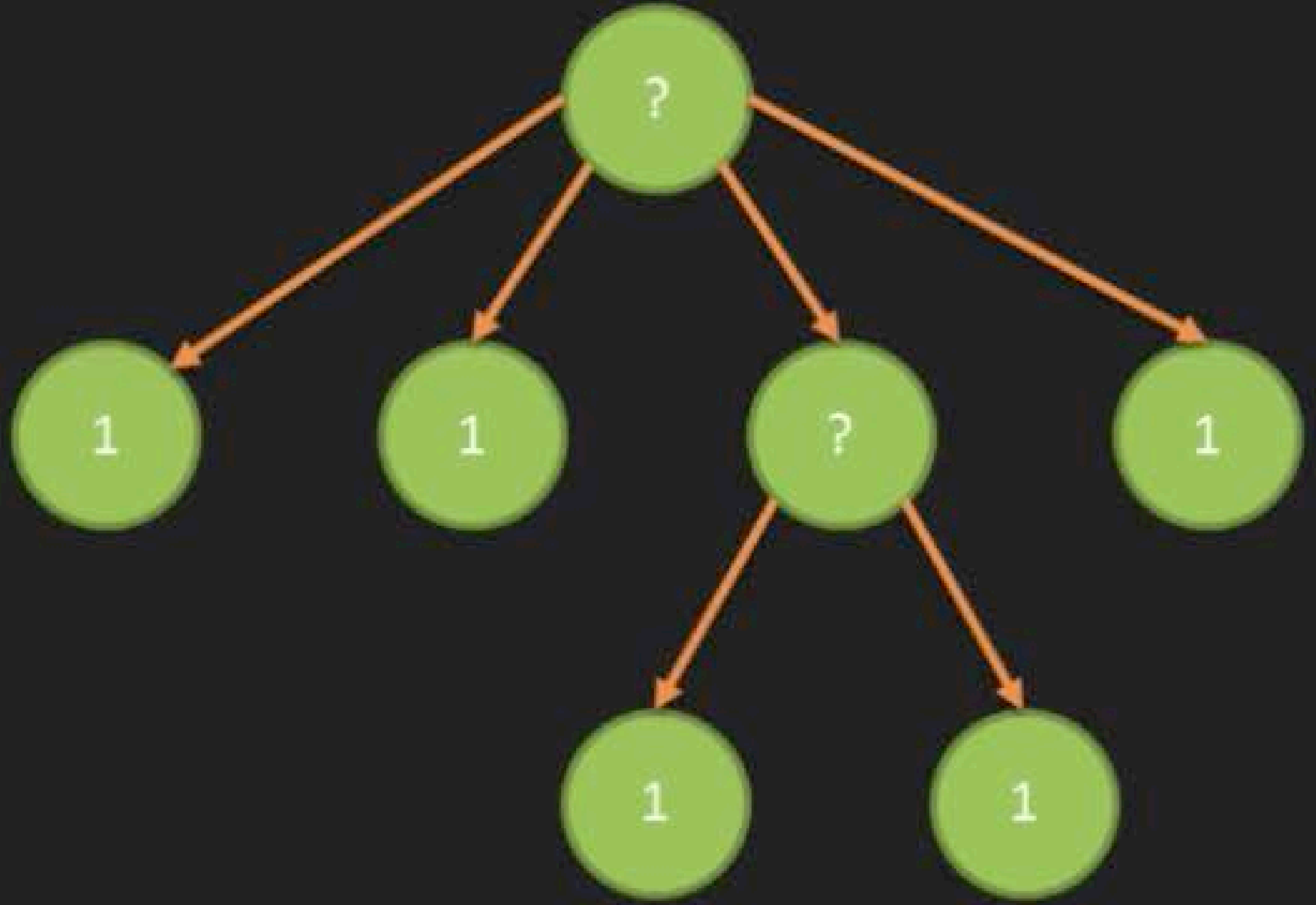
A Different Problem

- Base case

```
def count(%{children: []}) do
  1
end
```

- General function

```
def count(%{children: children}) do
  [first|rest] = children
  count(first) + count(%{children: rest})
end
```

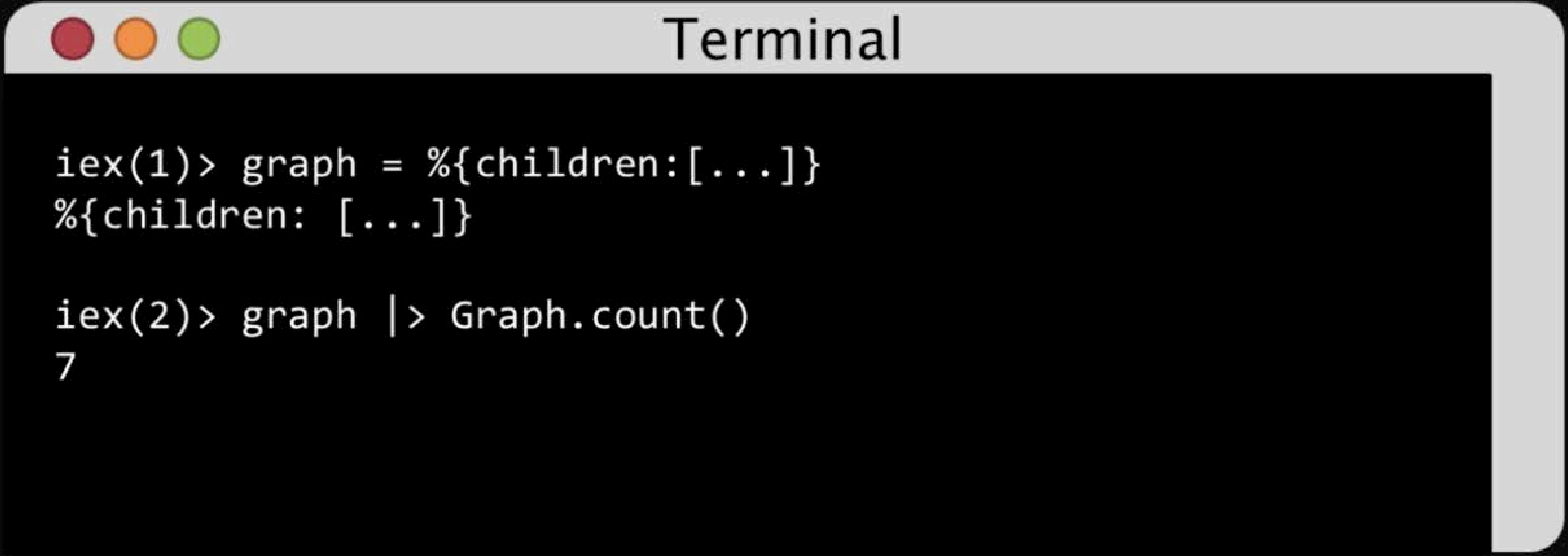


A Different Problem

```
defmodule Graph do
  def count(%{children: []}), do: 1
  def count(%{children: children}) do
    [first|rest] = children
    count(first) + count(%{children: rest})
  end
end
```

Packt>

Finally



```
iex(1)> graph = %{children:[...]}
%{children: [...]}

iex(2)> graph |> Graph.count()
7
```

João Gonçalves

Conditionals ▶

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

- Branching paths in computation
- Control flow using conditionals

Multiple Paths = Pattern Matching

- Classifying a list according to its size

```
def classify([]), do: :empty
def classify([_]), do: :single
def classify([_|_]), do: :multi
```

Age Discounts

	Child	Normal	Senior
Age range	< 12	>= 12, < 65	>= 65
Discount	60%	0%	40%

Cond

- When an expression evaluates to true, the body is executed

```
cond do
  expression -> body
  expression -> body
  ...
end
```

Age Discounts

```
def discount(age) do
  cond do
    age < 12  -> 0.6
    age >= 65 -> 0.4
    true      -> 0.0
  end
end
```

Age Discounts

```
def discount(age) do
  cond do
    age < 12  -> 0.6
    age >= 65 -> 0.4
    true      -> 0.0
  end
end
```

$\longrightarrow$ Default case (normal)

Age Discounts

```
def categorise(age) do
 ...
end

def discount(category) do
  cond do
    category == :child  -> 0.6
    category == :senior -> 0.4
    true                -> 0.0
  end
end
```

Packt>

Case

- When a pattern matches the value of the expression, the body is executed

```
case expression do
  pattern -> body
  pattern -> body
  ...
end
```

Age Discounts

```elixir
def categorise(age) do
 ...
end

def discount(category) do
  case category do
    :child  -> 0.6
    :senior -> 0.4
    _       -> 0.0
  end
end
```

Age Discounts

```
def categorise(age) do
 ...
end

def discount(category) do
  case category do
    :child  -> 0.6
    :senior -> 0.4
    _       -> 0.0
  end
end
```

→ Default case (normal)

Packt>

If/Else

- When an expression evaluates to true, the if_body is executed, otherwise the else_body is executed (if present)

```
if expression do
  if_body
end

if expression do
  if_body
else
  else_body
end
```

If/Else

- When an expression evaluates to false, the unless_body is executed

```
unless expression do
  unless_body
end
```

Age Discounts

- Changes happen, though ☹

```
def show_price(name, price) do
  discount = name
  |> Customer.find
  |> Customer.categorise
  |> Pricing.discount

  price * (1.0 - discount)
end
```

Packt>

Age Discounts

```elixir
def show_price(name, price) do
  case Customer.find(name) do
    {:ok, person} ->
      case Customer.categorise(person) do
        {:ok, category} ->
          case Pricing.discount(category) do
            {:ok, discount} ->
               {:ok, price * (1.0 - discount)}
            error -> error
          end
        error -> error
      end
    error -> error
  end
end
```

Age Discounts

!!!

```
def show_price(name, price) do
  case Customer.find(name) do
    {:ok, person} ->
      case Customer.categorise(person) do
        {:ok, category} ->
          case Pricing.discount(category) do
            {:ok, discount} ->
              {:ok, price * (1.0 - discount)}
            error -> error
          end
        error -> error
      end
    error -> error
  end
end
```

With

- If all patterns match, body is executed
- When a expression doesn't match, its value is returned

```
with
  pattern <- expression
  pattern <- expression
  ...
do
  body
end
```

Packt>

Age Discounts

```
def show_price(name, price) do
  with
    {:ok, person}   <- Customer.find(name),
    {:ok, category} <- Customer.categorise(person),
    {:ok, discount} <- Pricing.discount(category)
  do
    {:ok, price * (1.0 - discount)}
  end
end
```

Elixir: Scalable and Efficient Application Development

João Gonçalves

Exception Handling

Packt>

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

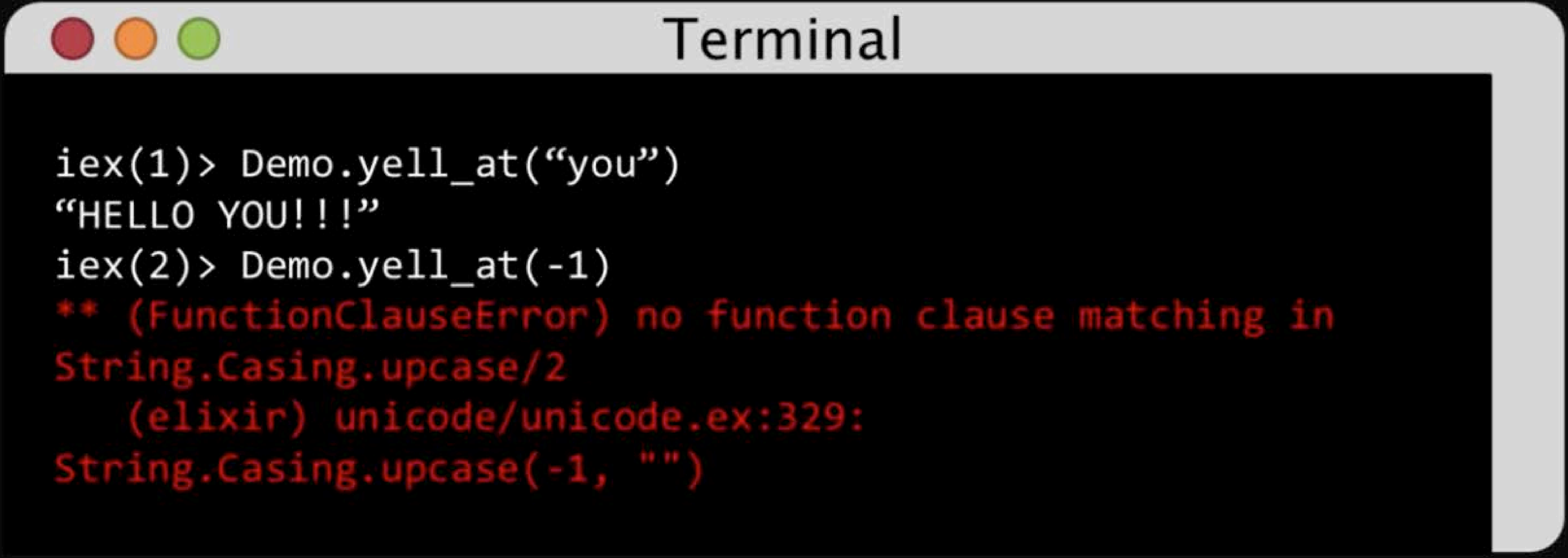
- Discussing the exceptions
- Handling exceptions

A Very Simple Function

```
def yell_at(name) do
  “HEY #{String.upcase(name)}!!!”
end
```

⍰⍰

A Very Simple Function



```
iex(1)> Demo.yell_at("you")
"HELLO YOU!!!"
iex(2)> Demo.yell_at(-1)
** (FunctionClauseError) no function clause matching in
String.Casing.upcase/2
   (elixir) unicode/unicode.ex:329:
String.Casing.upcase(-1, "")
```

Handling Any Exception

```
def yell_at(name) do
  try do
    “HEY #{String.upcase(name)}!!!”
  rescue
    e -> “HEY STRANGER!!!”
  end
end
```

Raising Exceptions

$$\begin{tikzcd} \texttt{raise “A weird error happened!”} \arrow[d] \\ \texttt{RuntimeError} \end{tikzcd}$$

Raised Exceptions

```
defmodule MyError do
  defexception message: ”A strange error”,
               number: 33
end
```

Raised Exceptions

```
defmodule MyError do
  defexception message: ”A strange error”,
               number: 33
end


raise MyError, number: 1000
```

Rescuing from Custom Exceptions

```
def catch_me() do
  try do
    raise MyError, number: 9023
  rescue
    e in MyError -> e.number
  end
end
```

Rescuing from Custom Exceptions

```
def catch_me() do
  try do
    raise MyError, number: 9023
  rescue
    e in MyError -> e.number
  after
    IO.puts(“I failed here”)
  end
end
```

Rescuing from Custom Exceptions

- after executes after the rescue, useful for tearing down resources

```
def catch_me() do
  try do
    raise MyError, number: 9023
  rescue
    e in MyError -> e.number
  after
    IO.puts(“I failed here”)
  end
end
```

Throwing and Catching

- throw/catch is far more suited for this purpose

```
def catch_me() do
  try do
    throw %{number: 9023}
  catch
    e -> e.number
  end
end
```

Quiz Time!

Quiz 5 | 2 Questions

Start Quiz Skip Quiz ›

Question 1:

Which of the following is a method of dividing complex problems into smaller ones?

- ◯ Concatenation
- ◯ Looping
- ◯ Breakdown
- ◯ Recursion

Question 2:

In recursion, for a base case, which value is returned when nodes have no children?

- ○ `Null`
- ○ `0`
- ○ `Nil`
- ○ `1`