Level 3–1

Tuples & Maps

Tuples

Creating Tuples

We use curly braces `{}` to represent tuples, an ordered collection of elements typically used as return values from functions.

A valid tuple

```
{:functional, "elixir", 2012}
```

Different data types

Tuples can hold many elements of different data types, but more often than not, we'll work with **two-element** tuples where the first element is an atom.

First element is usually an atom

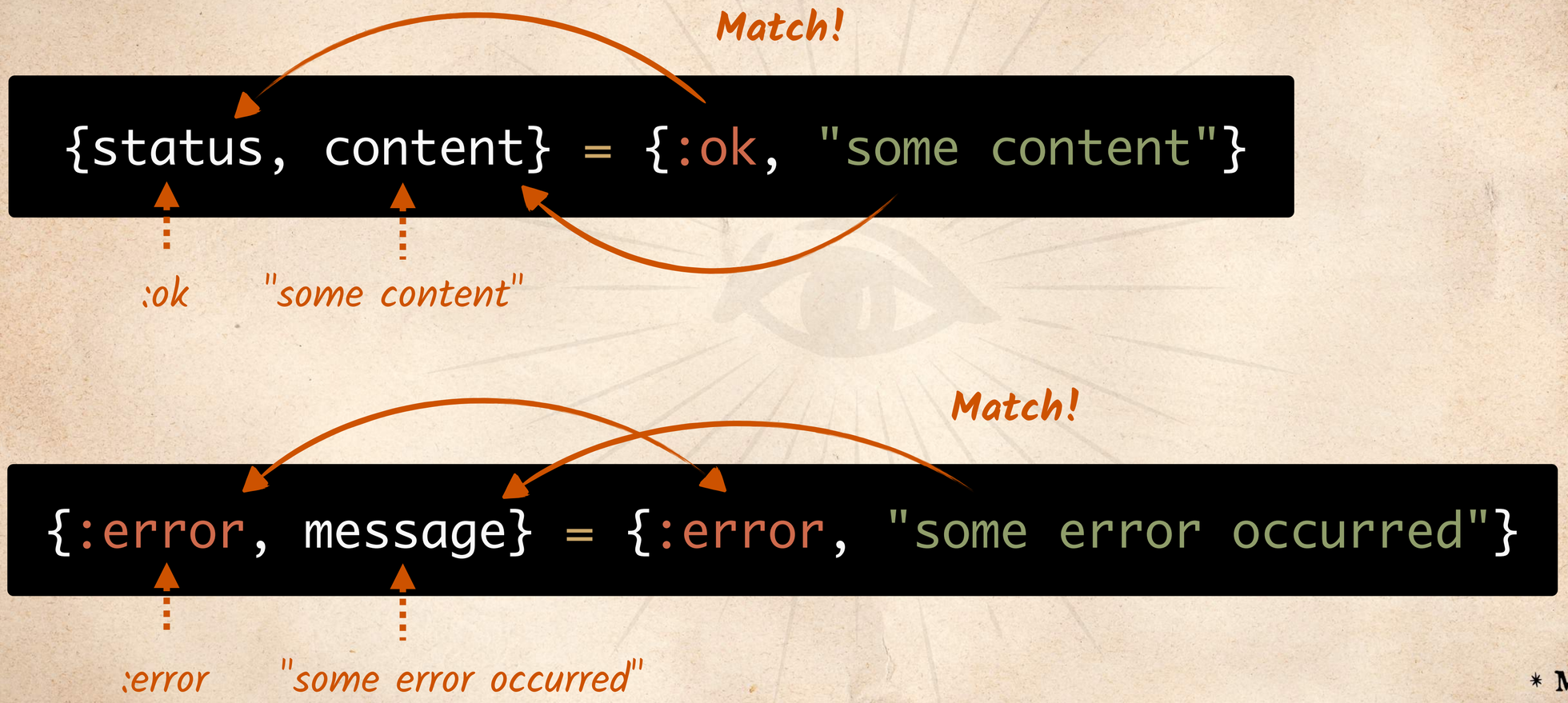
```
{:ok, "some content"}
{:error, :enoent}
```

Data type for second element will vary

atom representing an unknown file error

Tuples & Pattern Matching

We can use **pattern matching** to read elements from tuples.



* MIXING IT UP *
with
ELIXIR

Returning Tuples From Functions

The File.read **function from Elixir's standard library returns a tuple with two elements: an atom representing the status of the operation and either the content of the file or the error type.**

```
{status, content} = File.read(          )
```

Either :ok or :error

Content or error type

Path to file

```
{:ok, content} = File.read("transactions.csv")
```

```
{:ok, content} = File.read("file-that-doesnt-exist")
```

```
** (MatchError) no match of right hand side value: {:error, :enoent}
```

```
{:error, content} = File.read("file-that-doesnt-exist")
```

Pattern Matching Tuples From Functions

We can pattern match tuples in function arguments to read values passed in function calls.

```
defmodule Account do
  def parse_file({:ok, content}) do
    IO.puts "Transactions: #{content}"
  end

  def parse_file({:error, error}) do
    IO.puts "Error parsing file"
  end
end
```

This clause matches a successful File.read operation.

This clause matches an unsuccessful File.read operation.

Matching Successful Return Value

The pipe operator `|>` can be used to pass the result of reading the given file over to the newly created parse_file **function from the** Account **module.**

```
defmodule Account do
  def parse_file({:ok, content})...

  def parse_file({:error, error})...
end
```

Successful File.read matches first clause

```
File.read("transactions.csv") |> Account.parse_file()
```

Tuple { :ok, content } becomes first argument to next function

```
Content: 01/12/2016,deposit,1000.00
01/12/2016,withdrawal,10.00
01/13/2016,withdrawal,25.00,
...
```

Matching Unsuccessful Return Value

Reading a file that does not exist matches the second clause. However, in this example, a warning is raised because the error **variable is not being used from within the function.**

```
defmodule Account do
  ...
  def parse_file({:error, error}) do
    IO.puts "Error parsing file"
  end
end
```

Argument NOT used inside function body

Unsuccessful File.read matches second clause

```
File.read("does-not-exist") |> Account.parse_file()
```

Tuple { :error, error } becomes first argument to next function

```
warning: variable error is unused
  account.exs:20

Error parsing file
```

Matching Unsuccessful Return Value

The underscore character is used to explicitly **ignore unused values** and avoid compiler warnings.

```
defmodule Account do
  ...
  def parse_file({:error, _}) do
    IO.puts "Error parsing file"
  end
end
```

Explicitly ignore the value matched...

```
File.read("does-not-exist") |> Account.parse_file()
```

```
Error parsing file
```

...and no compiler warnings!