
MIXING IT UP
with
ELIXIR

Level 1

Citizens of the Unknown

Anonymous Functions

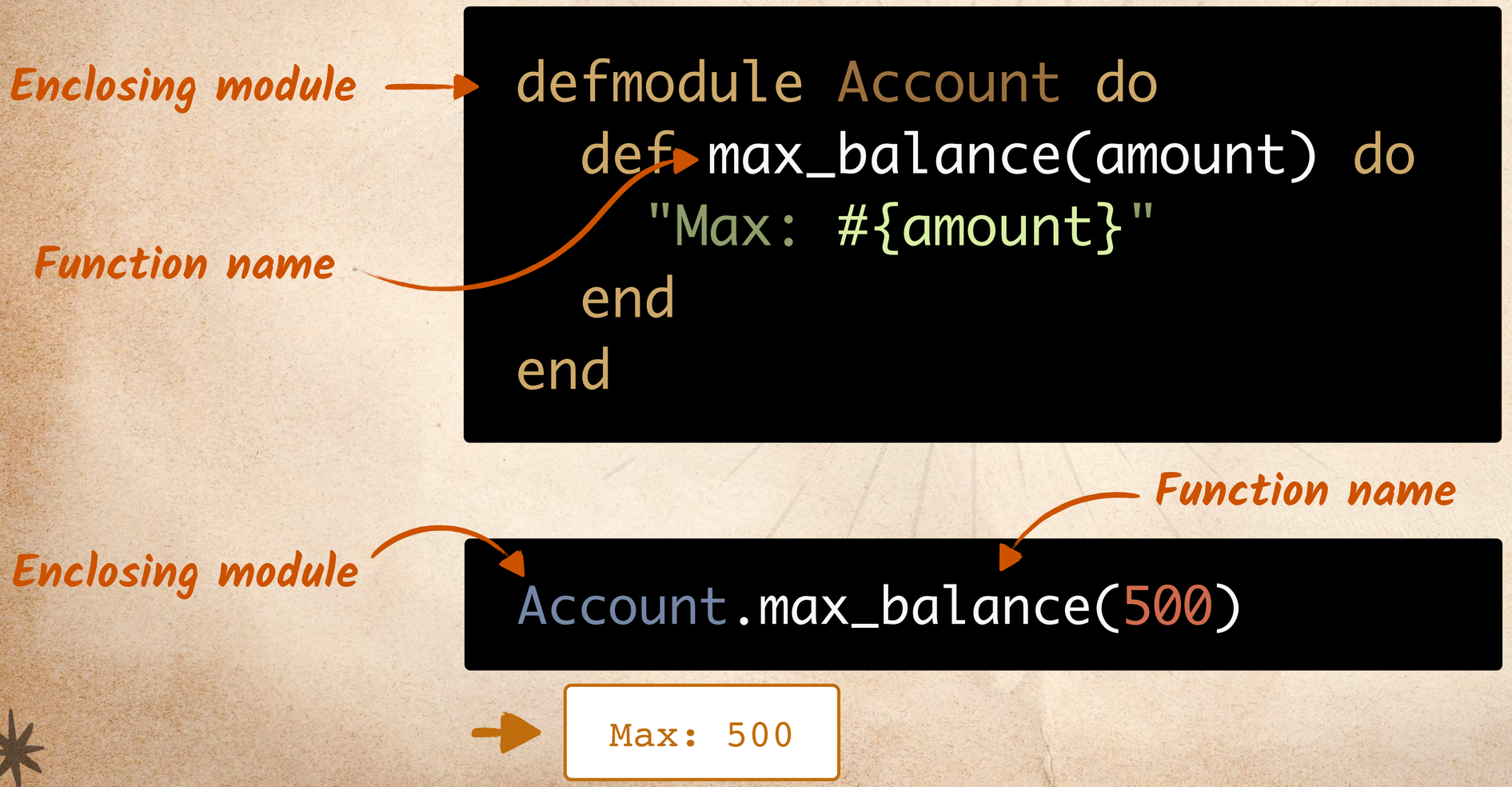
Functions Are First-class Citizens

What does this mean? It means that in Elixir, functions can:

- Be assigned to **variables**
- Be passed around as **arguments** to other functions

What We Know About Named Functions

The functions we've worked with so far have a name and belong to a module.



No Names, No Modules

Anonymous functions have **no name** and **no modules.** We create them with the `fn ->` syntax.

Single argument

```
max_balance = fn(amount) -> "Max: #{amount}" end
```

Stored in a variable

In order to invoke anonymous functions, we must use the `.( )` syntax.

```
max_balance.(500)
```

```
Max: 500
```

Must use a dot before the parenthesis

Must pass argument

```
max_balance.( )
```

```
** (BadArityError) #Function<....> with
   arity 1 called with no arguments
```

Decoupling With Anonymous Functions

Named functions can take anonymous functions as arguments. This helps promote **decoupling.**

These can be functions too!

```
Account.run_transaction(100, 20, deposit)
Account.run_transaction(100, 20, withdrawal)
```

Logic for performing the transaction...

...is decoupled from logic for each individual transaction.

How can we implement this?

Anonymous Functions as Arguments

The function signature is unchanged, but we must use `.( )` from inside the function body.

```
defmodule Account do
  def run_transaction(balance, amount, transaction) do
    if balance <= 0 do
      "Cannot perform any transaction"
    else
      transaction.(balance, amount)
    end
  end
end
```

Just like any other argument

The if statement represents logic for performing the transaction...

...and is decoupled from logic for each individual transaction.

Passing Anonymous Functions as Arguments

We can pass anonymous functions as arguments, just like with other data types.

```
deposit = fn(balance, amount) -> balance + amount end
withdrawal = fn(balance, amount) -> balance - amount end
```

```
Account.run_transaction(1000, 20, withdrawal)
Account.run_transaction(1000, 20, deposit)
```

→ `980`

→ `1020`

```
Account.run_transaction(0, 20, deposit)
```

→ `Cannot perform any transaction`

Returns immediately when the balance is 0 — remember?

Pattern Matching in Anonymous Functions

Similar to named functions, anonymous functions can also be split into **multiple clauses** using pattern matching.

The -> follows the argument list.

Clauses are broken into multiple lines.

```
account_transaction = fn
  (balance, amount, :deposit) -> balance + amount
  (balance, amount, :withdrawal) -> balance - amount
end
```

```
account_transaction.(100, 40, :deposit)
account_transaction.(100, 40, :withdrawal)
```

→ 140

→ 60

Anonymous Function Shorthand Syntax

The `&` operator is used to create helper functions in a short and concise way.

```
deposit = fn(balance, amount) -> balance + amount end
```

Turns the expression into a function

```
deposit = &(&1 + &2)
```

Numbers represent each argument.

Same thing

```
Account.run_transaction(1000, 20, deposit)
```

→ `1020`

The shorthand can be stored in a variable and passed as argument to a function, just like before!

Using the Shorthand Inline

The shorthand version of anonymous functions is often found used inline as arguments to other functions.

Can be defined inline too!

```
Account.run_transaction(1000, 20, &(&1 + &2))
```

→ `1020`

Enum.map is part of Elixir's standard library. It returns a list where each item is the result of invoking a function on each corresponding item of enumerable.

```
Enum.map([1,2,3,4], &(&1 * 2))
```

→ `[2, 4, 6, 8]`

Shorthand function that multiplies its argument by 2