
TRY
ELIXIR

Level 1

Modules & Functions

Understanding the Functional Paradigm

TRY ELIXIR

What Is Elixir?

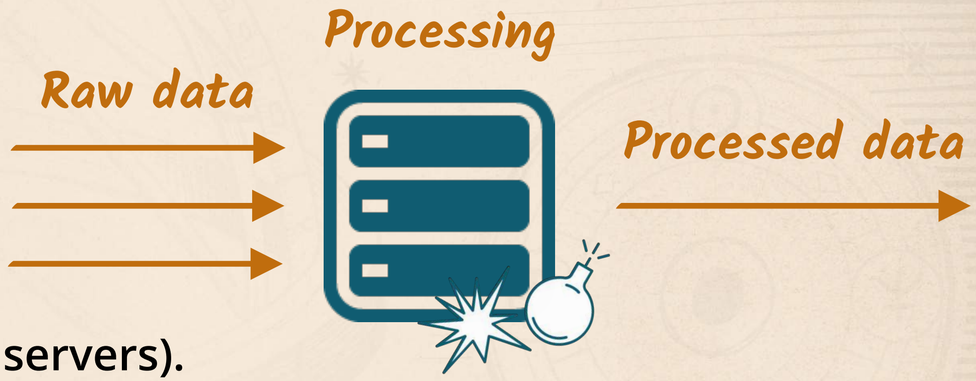
Elixir is a programming language created in 2012 and designed for building scalable and maintainable applications.

- **Functional** - Functions and modules — no objects or classes.
- **Dynamic** - Types checked at runtime.
- **Compiled** - Source code turns into bytecode.
- **Distributed and concurrent** - Tasks can run quickly and independently!
- **Runs on the Erlang Virtual Machine (BEAM)** - Stability and maturity.

What Is It Used For?

Elixir is a general-purpose language that's especially suitable for the following domain problems:

- **Data Processing**
  Consuming, computing, and transforming data.



- **Network Applications**
  Applications that communicate across the network (e.g., web servers).

- **High Availability Systems**
  Systems that must always be available despite crashes.

What We'll Learn

In this introductory course, we'll look at three of the most widely used features in Elixir:

- Functions
- The pipe operator: |>
- Pattern matching

TRY ELIXIR

The Functional Programming Style

In functional programming, computation is treated as transformation of data through **functions.**

```
f(x)
```

Invoke function *f* on data *x*

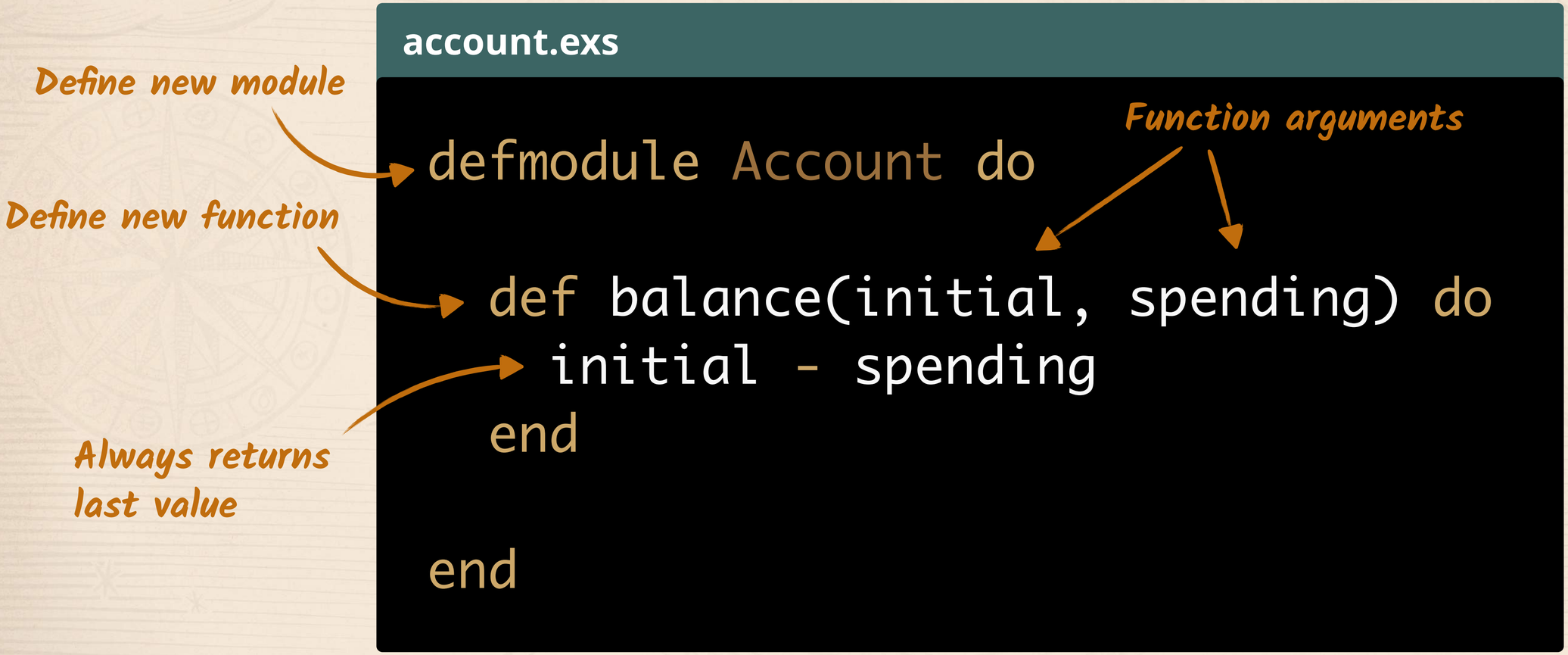
```
g(f(x), y)
```

Invoke function *g* on the return value of function *f* on data *x*, and on data *y*

TRY ELIXIR

Defining Named Functions

All named functions in Elixir must be part of an enclosing module.



Invoking Named Functions

Named functions are invoked using the **dot notation** and preceded by their module name.

account.exs

```
...

current_balance = Account.balance(1000, 200)
IO.puts "Current balance: US $#{current_balance}"
```

Variable data types are inferred by the runtime

Module acts as namespace

Print to the console

String interpolation

Running Elixir Programs

We can use the elixir **command to run programs from the command line.**

account.exs

```
...

current_balance = Account.balance(1000, 200)
IO.puts "Current balance: US $#{current_balance}"
```

```
$ elixir account.exs
```

```
Current balance: US $800
```

Visit **elixir-lang.org** for installation instructions.

Working With Pure Functions

A key part of functional programming is writing **pure functions**. In order to determine whether a function is pure, we look for these two things:

1. Return value relying entirely on arguments
2. No side effects

Always yields the same result when given the same values as arguments

All changes are represented by the return value

```
current_balance = Account.balance(1000, 200)
```

```
...
def balance(initial, spending) do
  initial - spending
end
```

Our function is pure! 👍

TRY ELIXIR

Pure Functions in FP vs. Object State in OO

The use of **pure functions** in functional programming makes programs easier to reason about.

1. Object State - *Object-oriented Languages*

(Some info is hidden from the caller)

Q: What is my account balance?

A: It depends on *how much you initially had* and *how much was spent.*

*The account **object** holds information*

```
account.currentBalance()
```

2. Pure Functions - *Functional Languages*

(Function receives all data necessary to perform operation)

Q: Starting the month with $1,000, when I spend $200, what is my account balance?

A: I can tell you! $1000 - $200 = ***$800***

```
Account.balance(1000, 200)
```

No dependencies!