Curso Elixir

Sección 12

Módulos

Como lo vimos en el capítulo 7 en Elixir un módulo es la agrupación de varias funciones nombradas y privadas. Es posible anidar módulos en Elixir, permitiéndonos ser explícitos nombrando nuestra funcionalidad.

```
    defmodule Math.Basic do
    def mul(x, y) do
        x * y
    end

    def sum(x, y), do: x + y
    end

iex> Math.Basic.mul(8, 5)
40
```

Esta anidación función como un namespace.

Atributos de un Módulo

Los atributos de un módulo son comúnmente usados como constantes en Elixir.

```
    defmodule Math.Basic do
    @pi 3.14159

    def mul(x, y) do
        x * y
    end

    def sum(x, y), do: x + y


    def circle_area(r) do
        @pi * :math.pow(r, 2)
    end
    end

    iex> Math.Basic.circle_area(2)
```

Es importante destacar que hay atributos reservados en Elixir. Los tres más comunes son:

moduledoc — Documenta el módulo actual.
doc — Documentación para funciones y macros.
behaviour — Usa OTP o comportamiento definido por el usuario.

Estructuras

Las estructuras son mapas con un conjunto definido de claves y valores. Deben ser definidas dentro de un módulo, y tomarán su nombre. Es común que una estructura sea definida únicamente dentro de un módulo.

Para definir una estructura se utiliza defstruct junto con una lista de claves y valores por defecto:

```
defmodule App.User do
  defstruct username: "", roles: []
end

iex> %App.User{}
%App.User{roles: [], username: ""}

iex> %App.User{username: "miniguez"}
%App.User{roles: [], username: "miniguez"}

iex> usuario = %App.User{username: "miniguez", roles: ["Admin", "Report"]}
%App.User{roles: ["Admin", "Report"], username: "miniguez"}
iex> usuario.username
"miniguez"
```

Para poder inspeccionar sobre la estructura, utilizaremos inspect

```
iex> inspect(usuario)
"%App.User{roles: [\"Admin\", \"Report\"], username: \"miniguez\"}"
```

Podemos ocultar campos en el inspect, usando @derive

```
defmodule App.User do
  @derive {Inspect, only: [:username]}

  defstruct username: "", roles: []
end

iex> usuario = %App.User{username: "miniguez", roles: ["Admin", "Report"]}
iex> inspect(usuario)
"#App.User<username: \"miniguez\", ...>"
```

Composición

- alias

  Nos permite darle un alias a los módulos, que son usados frecuentemente en Elixir

  ```
  defmodule Math.Basic do
  def mul(x, y) do
      x * y
  end

  def sum(x, y), do: x + y
  end

  iex> alias Math.Basic
  Math.Basic
  iex> Basic.mul(2, 4)
  8
  ```

  Si hay un conflicto entre dos alias o quieres que los alias tomen un nombre diferente, podemos utilizar la opción :as

  ```
  iex> alias Math.Basic, as: Opt
  iex> Opt.mul(2, 2)
  4
  ```

Es posible dar múltiples alias a un módulo a la vez:

```
iex> alias Math.{Basic, Opt}
```

- import

  Para importar las funciones y macros de un módulo, más que solo darle un alias, podemos utilizar import/:

```
iex> flatten([[1,2], 3, 4], [])
** (CompileError) iex: undefined function flatten/2

iex> import List

iex> flatten([[1,2], 3, 4], [])
[1, 2, 3, 4]
```

  Algunos módulos tienen los mismos nombres de funciones, al importarlos y usar las funciones podríamos tener errores, y necesariamente tendremos que especificar el nombre del módulo.

- only

  Cuando importamos un módulo por defecto se incluyen todas las funciones, en caso de que solo queramos una o algunas podemos usar :only y :except

```
iex> import List, only: [first: 1]
List
iex> first([1, 2, 3])
1
iex> last([1,2,3])
** (CompileError) iex: undefined function last/2
```

  En adición a los pares nombre/aridad, hay dos átomos especiales, :functions y :macros, las cuales importan únicamente funciones y macros

```
import List, only: :functions
import List, only: :macros
```

- require

  Aunque require/2 no es usado frecuentemente, es bastante importante. Haciendo require de un módulo asegura que está compilado y cargado. Esto es muy útil cuando necesitamos acceso a las macros de un módulo:

```
defmodule Example do
require SuperMacros

SuperMacros.do_stuff()
end
```

  Si intentamos hacer un llamado a una macro que no está cargada aún, Elixir lanzará un error.

- use

Con el macro use podemos habilitar otro módulo para modificar la definición de nuestro módulo actual. Cuando llamamos use en nuestro código, en realidad estamos invocando el callback **using**/1 definido por el módulo utilizado. El resultado de **using**/1 se convierte en parte de la definición de nuestro módulo.

```
defmodule AwesomeLibrary do
  defmacro __using__(_) do
    quote do
      def print(s), do: IO.puts("Hi, #{s}")
    end
  end
end

defmodule Example do
  use AwesomeLibrary
end

iex> Example.print("World")
```