

Elixir

`[h|t]=lista`

`H=head`

`T=tail`

`Lista= [1,2,3]`

`Head =1`

`Tail = [2,3]`

El simbolo “=” en elixir es Binding=fijación, que es el símbolo de pattern matching

Rebiding= vuelves a usar lo que está a la derecha del “=”.

`Lista=[1,2,3]`

`Biding= [uno|lista]=lista`

`Uno=1`

`Lista=[2,3]`

`Rebiding= [dos|lista]=lista`

`Dos= 2`

`Lista=[3]`

`Rebiding= [tres|lista]=lista`

`Tres=3`

`Lista=[]`

Suma de dos listas:

La sobre carga del símbolo de + “++”, te permite unir dos listas.

`Lista1=[1,2,3]`

`Lista2=[4,5,6]`

`Listaconvinada=lista1++lista2`

`Listaconvinada=[1,2,3,4,5,6]`

Resta de dos listas:

Con “-” restas dos listas

```
Lista1=[1,2,3]
```

```
Lista2=[2,3,4]
```

```
Listaconvinada=lista1- lista2
```

```
Listaconvinada=[1]
```

Y si tuviéramos:

```
Lista1[1,3,3]
```

```
Lista2=[2,3,4]
```

```
Listaconvinada=lista1- lista2
```

```
Listaconvinada=[1,3]
```

Poner un valor al inicio de la lista:

```
N=1
```

```
Lista=[2,3]
```

```
Lista=[n | lista]
```

```
Lista=[1,2,3]
```

Datos de elixir:

Todos los datos de elixir, todas las variables son inmutables; NO se pueden cambiar.

El operador pin=”^” protege de que la variable no se pueda modificar = lista^=[n | lista].

La función “i” es para ninspeccionar una lista = i(lista)

```
iex(59)> i(lista)
Term
  [2, 3]
Data type
  List
Reference modules
  List
Implemented protocols
  Collectable, Enumerable, IEx.Info, Inspect, List.Chars, String.Chars
iex(60)>
```

```
Enum.suma(["todo lo que este aquí separado por ',' van a ser sumados"])
```

Para el enum hay que cargar la función “i”, creo.

Es tipo de programación **declarativa**.

Variables:

Elixir es un lenguaje de programación dinámico.

- No es necesario declarar de manera explícita una variable o su tipo de dato.
- El tipo de dato de determina de acuerdo al valor contenido.
- La asignación se conoce como fijación (binding).
- Cuando se inicializa una variable con un valor, la variable se fija con ese valor.

```
iex()> dia_semana = 7 <fija (binds) el valor>
7 <resultado de la última expresión>
iex()> dia_semana <expresión que retorna el valor de la variable>
7 <valor de la variable>
iex()> dia_semana * 2
14
```

Características de las variables.

- El nombre de una variable siempre inicia con un carácter alfabético en minúscula o carácter de subrayado (_).
- Después puede llevar cualquier combinación de estos caracteres.
- La convención es usar solo letras, dígitos y subrayados.
- Pueden terminar con los caracteres “?” o “!”.

```
variable_valida
esta_variable_tambien_es_valida
esta_tambien_1
estaEsValidaPeroNoRecomendada
No_es_valida
nombre_valido?
claro_que_si!
```

Inmutabilidad.

- Los datos en Elixir son inmutables: su contenido no puede cambiarse.
- Las variables pueden ser refijadas (rebound) a un diferente valor.

```
iex()> dia_semana = 5 <se establece el valor inicial>  
5
```

```
iex()> dia_semana <verificación>  
5 <>  
iex()> dia_semana = 7 <se refija el valor inicial>  
7 <>  
iex()> dia_semana <se verifica el efecto de la refijación>  
7 <>
```

Estructura del código

Módulos y Funciones

Módulos.

- Un módulo consta de varias funciones.
- Cada función debe estar definida dentro de un módulo.
- El módulo IO permite varias operaciones de E/S (I/O), la función `puts` permite imprimir un mensaje en pantalla.

```
iex()> IO.puts("Hola Mundo")  
Hola Mundo  
:ok
```

- La sintaxis general es: *NombreModulo.nombre_funcion(args)*.
- Se utiliza el constructor “*defmodule*” para la creación de los módulos.
- Dentro del módulo con el constructor “*def*” se crean las funciones.

Funciones.

Una función siempre debe estar dentro de un módulo.

Los nombres de funciones son igual que las variables:

- El nombre de una variable siempre se inicia con un carácter alfabético en minúscula o carácter de subrayado (`_`).
- Después puede llevar cualquier combinación de estos caracteres.
- La convención es usar solo letras, dígitos y subrayados.
- Pueden terminar con los caracteres “?” o “!”..
- Por convención el “?” se utiliza cuando la función retorna “*true*” o “*false*”.

-El “!” se utiliza generalmente en funciones que podrían provocar algún error en tiempo de ejecución.

-Tanto “*defmodule*” como “*def*” NO son palabras reservadas del lenguaje, son *macros*.

Función sin argumentos (procedimiento):

```
#Función sin argumentos
defmodule HolaMundo do
  def mensaje do
    IO.puts("Hola mundo")
  end
end

iex(1)> l(Elixir.HolaMundo)
{:module, HolaMundo}
iex(2)> HolaMundo.mensaje
Hola mundo
:ok
```

Función con argumentos:

```
#Función con argumentos
#Área de un cuadrado:
defmodule Areas do
  def area_cuadrado(1) do
    1*1
  end
end

iex(3)> l(Elixir.Areas)
{:module, Areas}
iex(4)> Areas.area_cuadrado(8)
64
```

-Un módulo puede estar dentro de un archivo. Un archivo puede contener varios módulos.

Reglas de los módulos:

- Inicia con una letra mayúscula
- Se escribe con el estilo CamelCase
- Puede consistir en caracteres alfanuméricos, subrayados y puntos (.).
- Regularmente se usa para la organización jerárquica de los módulos.

```
defmodule Geometria.Cuadrado do
  def perimetro(1) do
    4*1
  end
end

defmodule Geometria.Rectangulo do
  def perimetro(l1,l2) do
    2*l1 + 2*l2
  end
end
```

También se pueden anidar de la siguiente forma:

```
#Lo anterior también se pueden anidar de la siguiente forma:
defmodule Geometria do
  defmodule Cuadrado do
    def perimetro(1) do
      4*1
    end
  end
  defmodule Rectangulo do
    def perimetro(l1,l2) do
      2*l1 + 2*l2
    end
  end
end
```

Las funciones pueden expresarse de manera condensada:

```
#Las funciones pueden expresarse de manera condensada:
defmodule Geometria do
  def perimetro_cuadrado(1), do: 4*1
  def perimetro_rectangulo(11,12), do: 2*11 + 2*12
end
```

Los paréntesis en los argumentos son opcionales:

```
iex()> Geometria.perimetro_cuadrado 4
16
iex()> Geometria.perimetro_rectangulo 4,3
14
```

Visibilidad de funciones

-Se pueden utilizar funciones privadas con el constructor defp.

-Función Pública y privada.

```
defmodule TestPublicoPrivado do
  def funcion_publica(msg) do #Pública
    IO.puts("#{msg} publico")
  end
  defp funcion_privada(msg) do #Privada
    IO.puts("#{msg} privado")
  end
end

iex(2)> TestPublicoPrivado.funcion_publica("Hola")
Hola publico
Hola privado
:ok
```

Módulo Geometría:

```
#Módulo Geometría:
defmodule Geometria do
  def perimetro1(1), do: cuadrado(1)
  def perimetro2(1), do: Geometria.cuadrado(1)
  defp cuadrado(1), do: 4*1
end
```

Para ejecutar sus privadas hay que usar un **Operador Pipeline**.

```
iex(3)> 4 |> Geometria.perimetro1
16
```

```
#Obtener el cuadrado de la suma de 2 números
#Invocando funciones
```

```
defmodule Operaciones do
  def suma(n1,n2), do: n1 + n2
  def cuadrado(n), do: n * n
end
```

```
Operaciones.cuadrado(Operaciones.suma(4,5))
```

```
{:module, Operaciones}
iex(2)> Operaciones.cuadrado(Operaciones.suma(4,5))
81
```

Capítulo 7. Estructura del código en Elixir

Estructura del código

Aridad (Arity) de funciones

-Es el nombre para el número de argumentos que una función recibe

-Una función se identifica por:

1. el módulo donde se encuentra,
2. su nombre y
3. su aridad (arity)

Polimorfismo (sobrecarga)

-Dos funciones con el mismo nombre, pero con diferente aridad son dos diferentes funciones.

```
#Haciendo que una función dependa de otra de diferente aridad, se podría realizar lo siguiente:
defmodule Calculadora do
  def suma(n) do
    suma(n, 0)
  end
  def suma(n1, n2) do
    n1 + n2
  end
end
```

```
#Argumentos por defecto
# -Se pueden especificar argumentos por defecto mediante el operador
# Este módulo genera dos funciones como en el caso anterior
defmodule Calculadora do
  def suma(n1, n2 \\ 0) do
    n1 + n2
  end
end

#Se puede utilizar cualquier combinación de argumentos por defecto:
defmodule Calculadora do
  def funcion(n1, n2 \\ 0, n3 \\ 1, n4, n5 \\ 2) do
    n1 + n2 + n3 + n4 + n5
  end
end
```