

2. Conceptos básicos de Elixir

Prog. Secuencial



UNIVERSIDAD
DEL QUINDÍO®

Res.MEN 014915 - 02 AGO 2022
RENOVACIÓN ACREDITACIÓN

Programación 3

Material basado en el de Julián Esteban Gutiérrez Posada

Programa de Ingeniería de Sistemas y Computación

Facultad de Ingeniería - 2025

UNIQUINDÍO
en conexión territorial

www.uniquindio.edu.co



Problema No. 1

Carlos, el gerente de la empresa de software Once Ltda, requiere un programa que permita mostrar un mensaje de bienvenida a los trabajadores de la empresa.

El mensaje es **Bienvenidos a la empresa Once Ltda.**



Abstracción

- ¿Qué se solicita finalmente?

Mostrar un mensaje de bienvenida

- ¿Qué información es relevante dado el problema anterior?

El mensaje a mostrar "Bienvenidos a la empresa Once Ltda".



Descomposición

- ¿Qué acciones se requieren para resolverlo?

- Mostrar el mensaje



Reconocimiento de patrones

- ¿Qué puede reutilizar de la solución de otros problemas?

De momento no hay elementos a reutilizar de ejercicios anteriores.



Codificación

- ¿Cómo escribo la solución en un lenguaje (ej: Elixir)?

Versión 1:

```
IO.puts("Bienvenidos a la empresa Once Ltda")
```



A CLARACIÓN - 1

Aunque es posible usarlo de esta forma,
por legibilidad se usará la el operador |> de encadenamiento (*pipe*).



Versión 2:

```
"Bienvenidos a la empresa Once Ltda"  
|>IO.puts()
```



A CLARACIÓN - 2

- Esta forma está bien para ciertas pruebas de un concepto, sin embargo, lo ideal, por organización y reutilización, es que todas las funcionalidades estén agrupadas en **MÓDULOS**.
- Elixir, así como otros (ej: Python) no tiene una función principal (*main*), así que, **por ahora** (luego se usarán proyectos con **mix**), y por organización, se creará un módulo particular para el ejemplo.
- Y en este caso, se debe invocar manualmente la función principal (**main**) del módulo del ejemplo



Versión 3:

```
defmodule Mensaje do
  def main do
    "Bienvenidos a la empresa Once Ltda"
    |> IO.puts()
  end
end

Mensaje.main()
```

- Desde ya se buscará tener una estructura que se pueda seguir utilizando y que esté asociada con los elementos del pensamiento computacional, principalmente con la **DESCOMPOSICIÓN** que se convertirá en el contenido de la función **main**.

Descomposición

- Mostrar el mensaje



A C L A R A C I Ó N - 3 - (2 de 2)

- Por ahora un poco de acto de fe, pero tendrá sentido un poco más adelante.
 - Se deja únicamente la función principal (*main*) para que se pueda invocar desde fuera del módulo (**def**).
 - Se creará una función privada al módulo (**defp**) con el nombre de *mostrar_mensaje* para buscar la coherencia con la descomposición.
- Tener una función *mostrar_mensaje* facilitar:
 - Reutilizar la función en otro ejercicios (*Reconocimiento de patrones*)
 - Crear una separación entre la lógica (*mostrar_mensaje*) con la forma de su implementación en el lenguaje, facilitando su futura actualización a otras formas.



Versión 4:

```
defmodule Mensaje do
  def main do
    "Bienvenidos a la empresa Once Ltda"
    |> mostrar_mensaje()
  end

  defp mostrar_mensaje(mensaje) do
    mensaje
    |> IO.puts()
  end
end

Mensaje.main()
```



A C L A R A C I Ó N - 4 - (1 de 3)

- Por organización y para facilitar el *Reconocimiento de patrones* (reutilización de código), se hará:
 - Mover a un módulo independiente la función *mostrar_mensaje*.
 - Se dejarán como funciones privadas al ejemplo, las funciones que no se tengan el potencial de retutilización.
 - Se pueden tener cuantos módulos sean necesarios por organización, por ahora, se creará un módulo llamado **Util**, para agrupar todas las funciones que será de utilidad en el desarrollo de los ejemplos.
 - Se deberán dejar únicamente como públicas las funciones que se requieren invocar (utilizar) por fuera del módulo.



A CLARACIÓN - 4 - (2 de 3)

- Crear un archivo llamando **util.ex**, observe que:
 - el nombre de los archivos de los módulos usa la notación *Snake Case*.
Por ejemplo: *utilidades_entrada_salida*.
 - el nombre de los módulos usar notación *Camel Case*.
Por ejemplo: *UtilidadesEntradaSalida*.
 - la extensión del archivo para el módulo es **.ex** , lo cual indica que será un programa para compilar (convertirlo al código binario de la máquina virtual de Erlang -**BEAM**-).
 - Es más rápido
 - Las nuevas funciones del módulo quedará disponibles para su uso.

- El nuevo módulo deberá quedar, por ahora, así:

util.ex

```
defmodule Util do
  def mostrar_mensaje(mensaje) do
    mensaje
    |> IO.puts()
  end
end
```

- Deberá compilar el módulo **cada vez que realice un ajuste en el módulo**. Para esto, desde la terminal y en el directorio donde está el módulo (luego con proyectos mix esto no será necesario), usar el comando:
`elixirc util.ex` y deberá crearse un archivo llamando *Elixir.Util.beam*.



Versión 5:

```
defmodule Mensaje do
  def main do
    "Bienvenidos a la empresa Once Ltda"
    |> Util.mostrar_mensaje()
  end
end

Mensaje.main()
```

- Ahora ya tiene una mayor coherencia con **Descomposición** (la función *main* es lo que la descomposición indicaba *mostrar_mensaje*).
- Observe que no hay necesidad de indicar explícitamente el uso del módulo.

Para comprender las ventajas de hacer esta separación entre las funciones del lenguaje (ej: `IO.puts`) y la lógica, vea el siguiente punto.

- Elixir no tiene de forma nativa **GUI** (*Graphical User Interface*), pero Python sí, así que se podría modificar `util.ex` de la siguiente manera:

`util.ex`

```
defmodule Util do
  def mostrar_mensaje(mensaje) do
    System.cmd("python3", ["mostrar_dialogo.py", mensaje])
  end
end
```



Tal vez en su equipo necesites usar:

```
defmodule Util do
  def mostrar_mensaje(mensaje) do
    System.cmd("cmd.exe",["/c", "python3", "mostrar_dialogo.py", mensaje])
  end
end
```

O incluso usar `python` en lugar de `python3`.



A CLARACIÓN - 5 - (2 de 4)

- En la misma carpeta se debe tener el siguiente *script* Python3 que muestre un mensaje en un dialogo.

```
import sys
import tkinter as tk
from tkinter.simpledialog import messagebox

root = tk.Tk()
root.withdraw()

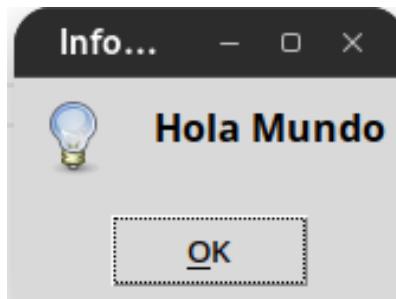
messagebox.showinfo("Información", sys.argv[1])
```



A CLARACIÓN - 5 - (3 de 4)

El programa Python3 se ejecuta de la siguiente forma:

```
python3 mostrar_dialogo.py "Hola Mundo"
```



Pero lo interesante es que **TODOS** los programas en Elixir que usen la módulo **Util** mostrarían sus mensajes de salida de forma gráfica, simplemente con ajustar la implementación de **mostrar_mensaje**.

Ejecute con `elixir mensaje.exs`. No ejecuta **Code Runner** desde **VSCode**.

También puede usar `java`

y ajustar el módulo así:

```
defmodule Util do
  def mostrar_mensaje(mensaje) do
    System.cmd("java", ["-cp", ".", "Mensaje", mensaje])
  end
end
```

También puede instalar `zenity` (principalmente en Linux)

<https://ostechnix.com/zenity-create-gui-dialog-boxes-in-bash-scripts/>

y ajustar el módulo así:

```
defmodule Util do
  def mostrar_mensaje(mensaje) do
    System.cmd("zenity", ["--info", "--text=#{mensaje}"])
  end
end
```



O si lo prefiere puede usar Java utilizando un código como:

```
import javax.swing.JOptionPane;

public class Mensaje {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, args[0]);
    }
}
```

Para verificar su correcto funcionamiento debe compilar el archivo Mensaje.java con el comando javac Mensaje.java y se puede usar:

```
java Mensaje "Hola Mundo"
```



A CLARACIÓN - 7

Es ideal que todos los programas estén documentados.

<https://elixirschool.com/es/lessons/basics/documentation>



Documentación

- `#` Documentación de línea.
- `@moduledoc` Documentación del módulo
- `@doc` Documentación de una función

Aunque es ideal el uso de la documentación, no se puede exagerar su uso.



Documentación del módulo *Util*

Por ejemplo, se podría documentar un módulo de la siguiente forma:

```
defmodule Util do
  @moduledoc """
  Módulo con funciones que se reutilizan
  - autor: Nombre del autor.
  - fecha: fecha creación
  - licencia: GNU GPL v3
  """

```

Elimine el archivo `Elixir.Util.beam` y vuelva compilar el módulo `Util.ex` usando en la terminal el comando `elixirc`.



Es de aclarar que normalmente los proyectos suelen tener, entre otros, los siguientes archivos:

- **VERSION**: Versión del proyecto
- **LICENSE**: Para indicar la licencia general de proyecto.
- **README.md**: Información adicional del proyecto en formato *Markdown*.

Este tema se retomará cuando se utilice el comando `“mix”` para crear proyectos Elixir.



Ejecute `iex` y solicite las ayudas del módulo, usando el comando `h :`

```
iex(1)> h Util
```

Util

Módulo con funciones que se reutilizan

- Autor: Nombre del Autor
- Fecha: Fecha creacion
- Licencia: GNU GPL v3 (nombre de la licencia)

Las funciones también se pueden documentar, por ejemplo:

<https://github.com/elixir-lang/elixir/blob/main/lib/elixir/lib/kernel.ex>



```
@doc """
Función para mostrar un mensaje en la pantalla.
## Parámetro
- mensaje: texto que se le presenta al usuario
## Ejemplo
  iex> Util.mostrar_mensaje("Hola Mundo")

o puede usar

  "Hola Mundo"
  |> Util.mostrar_mensaje()
"""

def mostrar_mensaje(mensaje) do
  mensaje
  |> IO.puts()
end
end
```



Elimine el archivo `Elixir.Util.beam` y vuelva compilar (`elixirc`) el módulo `Util.ex`. Luego ejecute `iex` y solicite las ayudas de la función `mostrar_mensaje` del módulo `Util`, usando el comando `h`:

```
iex(1)> h Util.mostrar_mensaje
```

```
def mostrar_mensaje(mensaje)
```

Función para mostrar un mensaje en la pantalla.

Parámetros

- `mensaje`, texto que se le presenta al usuario

Ejemplos

```
iex> Util.mostrar_mensaje("Hola Mundo")
```

...



Aclaración Final

Por **espacio en las diapositivas**, se omitirá la documentación de:

- los ejemplos y
- los módulos

Pero se recomienda que documente los programas, tal y como se evidencia en el código de Elixir.

<https://github.com/elixir-lang/elixir/blob/main/lib/elixir/lib/kernel.ex>



Julián Esteban Gutiérrez Posada
jugutier@uniquindio.edu.co
2024