



Tercer clase de erlang!



## En esta clase...

- Estructuras en Erlang:
  - Mapas (diccionarios)
  - Macros/Records
  - Archivos de cabecera (Headers)
  - Representación con Binarios
- Sockets TCP

En esta clase veremos cómo definir estructuras en Erlang.

Primero veremos Mapas o diccionarios

Luego veremos principalmente records, y la generación de archivos de cabecera para tener una separación limpia entre estructuras y código

Veremos que Erlang nos presenta/facilita el trabajo con representaciones binarias

Por último veremos cómo hacer manejo de sockets tipo TCP



Docs: <https://erlang.org/doc/man/maps.html>



Los mapas son una estructura **persistente** de datos que nos permite asociar una **clave** con un **valor**.

Los mapas son creados de forma dinámica, y liberados por el *garbage collector*.

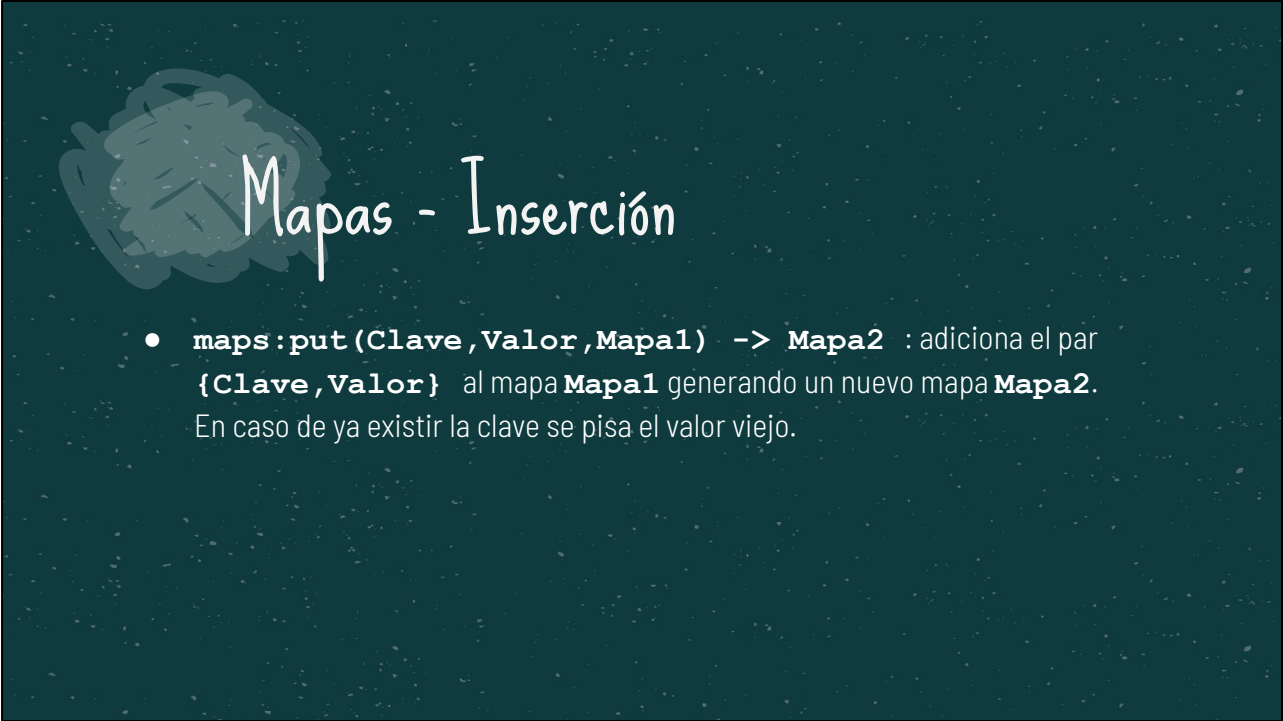
Una estructura persistente es una estructura que persiste en el tiempo, se logra no modificando la estructura original sino pensando que esta se utiliza para generar una nueva.

Es como en matemáticas, si hacemos  $2+3 \rightarrow 5$ , el 2 y el 3 siguen existiendo.



# Mapas - Creación

- `maps:new()` -> **Mapa** : crea un mapa vacío
- `maps:from_list(Lista)` -> **Mapa** : crea un mapa a partir de una lista de pares clave y valor.



# Mapas - Inserción

- `maps:put(Clave,Valor,Mapa1) -> Mapa2` : adiciona el par `{Clave,Valor}` al mapa **Mapa1** generando un nuevo mapa **Mapa2**. En caso de ya existir la clave se pisa el valor viejo.

Notar que los mapas son estructuras persistentes.



# Mapas - Búsqueda

- `maps:find(Clave, Mapa1) -> {ok, Valor} | error`
- `maps:get(Clave, Mapa1) -> Valor`
- `maps:get(Clave, Mapa1, Def) -> Valor`

Acceso a la información de un mapa.

Find nos indica si encontró un valor asociado a la clave o no.

Get(Clave, Mapa) falla en el caso de que la clave no se encuentre

Get(Clave, Mapa, Def) devuelve Def en caso de no encontrar la clave



# Mapas - Actualización

- `maps:update(Clave, Valor, Mapa1) -> Mapa2`
- `maps:update_with(Clave, Fun, Mapa1) -> Mapa2`
- `maps:update_with(Clave, Fun, Mapa1, Def) -> Mapa2`

Flia update, las primeras dos pueden fallar en el caso que no exista la clave en el Mapa1.

Mientras que update\_with/4 en el caso que la clave Clave no esté inserta el par {Clave, Fun(Def)} en Mapa1





# Mapas - Eliminación

- `maps:remove(Clave, Mapa1) -> Mapa2` : elimina, en caso de que exista, la clave **Clave** del mapa **Mapa1**

Notar que los mapas son estructuras persistentes.



# Ejercicio Cliente/Servidor de Identificadores

El objetivo del ejercicio es hacer una pequeña librería que provea el servicio de otorgar identificadores únicos para distintos nombres, y luego poder recuperar los nombres en base al identificador.

# Servidor

El servidor aceptará 4 tipos de pedidos, y responderá adecuadamente

## nuevoId

Acompañado de un **Nombre** y el **PId** del cliente. Generará un nuevo identificador para **Nombre**, se agrega al mapa, y se notifica a **PId**

## verLista

Acompañado de un **PId**, le envía el mapa en forma de lista

## buscarId

Acompañado de un **Id** y el **PId** del cliente, buscará en el mapa el nombre asociado a **Id**, y se lo enviará a **PId**

## finalizar

Acompañado de un **PId**, que finaliza el proceso servidor y responde con un **ok** a **PId**



# Librería de Acceso

- **nuevoNombre (Nombre)** : que dado un **Nombre** le pide al **Servidor** que le asigne un identificador.
- **quienEs (Id)** : que le pide a **Servidor** el nombre asociado a **Id**
- **listaDeIds ()** : que le pide a **Servidor** la lista de ids y nombres
- **iniciar ()** : que inicia un servicio de nombres
- **finalizar ()** : que finaliza el servicio de nombres



Comencemos...  
(tenemos un template para trabajar)



Macros

# Macros

Las macros sirven para identificar constantes que pueden llegar a variar en el tiempo, pero siempre serán conocidas al momento de la compilación. Hay dos tipos, **constantes** o **funciones**.

```
-define (Const, Expresión) .
```

```
-define (Func (Arg1, ..., ArgN), Expresión) .
```

Las macros son reemplazadas directamente (sin chequeo de tipos ni de sintaxis) en un momento previo de la compilación.

# Macros

Por ejemplo, podemos definir una macro para establecer el tiempo de espera en una petición como:

```
-define (TIMEOUT, 500) .  
foo() ->  
  receive  
    ...  
  after  
    ?TIMEOUT -> ...  
end.
```





# Macros

Erlang ya define algunas macros útiles:

- **?MODULE** : reemplazada por el nombre del módulo
- **?FILE** : reemplazada por el nombre del archivo
- **?LINE** : reemplazada por el número de línea
- **?MACHINE** : reemplazada por el nombre de la máquina
- **?FUNCTION\_NAME** : reemplazada por el nombre de la función donde se encuentra

Pueden encontrar un par de macros más en la documentación  
[https://erlang.org/doc/reference\\_manual/macros.html#predefined-macros](https://erlang.org/doc/reference_manual/macros.html#predefined-macros)



# Macros

Operaciones con macros:

- `-ifdef (Flag) .` : Chequea si la bandera **Flag** fue definida
- `-undef (Flag) .` : Olvida **Flag**
- `-ifndef (Flag) .` : Cheque si la bandera **Flag** no fue definida
- `-else.`
- `-endif.`

Activar banderas: `c (Nombre, [{d,Flag}]) .`

# Ejemplo: usando Macros

```
-module(macros_test).  
-export([test/0]).  
  
-define(Debug, debug).  
-ifdef(Debug).  
    -define(DBG(Str,Args), io:format(Str,Args)).  
-else.  
    -define(DBG(_Str,_Args), ok).  
-endif.  
  
test() -> ?DBG("Esto es un ~p~n",[test]).
```



Records!



# Records

Los records nos permiten definir estructuras con una cantidad fija de elementos, y asignarles un nombre a cada campo.

Similar a las macros se definen de la siguiente forma:

```
-record(Nombre, {Campo1 [= Def1] , ... , CampoN [= DefN]}).
```

Los records son reemplazados por tuplas al momento de compilación, pero nos permite identificar cada campo con su nombre.

Son similares a las estructuras de C.

```
-record(nombre, {Campo1,..., CampoN})
```

La principal ventaja de utilizar registros en lugar de tuplas es que se accede a los campos de un record por nombre, mientras que a los campos de una tupla se acceden por posición.



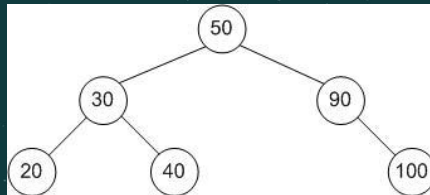
## Ejemplo: uso de Records

```
-record(persona,{nombre, apellido, tel}).  
> M = #persona{nombre = "Pepe", apellido = "Pepa", tel=155}.  
> M#persona.nombre.  
> MT = M#persona{apellido = "Perez"}.
```

# ¡Ejercicio para la casa!

Definir un record que describa árboles binarios.

Codificar el siguiente árbol:





# Archivo de Cabera (header.hrl)

Finalmente, como es común definir macros y records, estas suelen agruparse en un archivo aparte de cabecera. Los archivos de cabecera contienen código erlang, y se puede incluir directamente con:

```
-include("nombre.hrl").
```





Ejercicio: Utilizar records para mejorar la comunicación entre el servidor y cliente.

La idea es mejorar la comunicación entre el servidor y cliente anteriormente, dándole mayor estructura mediante Records!  
Yo comienzo por mi parte mostrando qué es lo que deberían hacer y después que continúen ellos.



Binarios



# Binarios

Los binarios son un tipo de datos de Erlang, que son muy utilizados para **serializar** información para enviar por la red.

Los binarios representan un segmento crudo de memoria sin tipo.

Serializar: cambiar la representación de una estructura 'profunda' como un árbol, en una cadena de bytes.



## Binarios - Bifs

- `term_to_binary/1` : traduce un término en un binario
- `binary_to_term/1` : traduce un binario en un término
- `is_binary/1` : retorna si su argumento es un binario
- `binary_to_list/1` : toma un binario y devuelve una lista con sus bytes.

Está la función de `list_to_binary` pero no recomiendo usarla.



# Binarios - Sintaxis de Bit

La sintaxis de bit permite agrupar secuencias de Binarios de forma rápida y sencilla.

> **Bin** = <<**B1**, **B2**, ... , **BN**>>.

Y podemos hacer Pattern Matching

> <<**V1**,**V2**,...,**VN**>> = **Bin**.

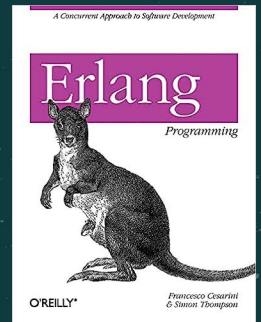


# Binarios - Cadenas de Bytes

La sintaxis de Bits permite cortar largas cadenas de bytes para ser enviadas por la red de a pedazos, utilizando la sintaxis: **Value:Size/Type**

```
send(Pid, <<Chunk:100/binary, Res/binary>>) -> Pid !  
{chunk, Chunk}, send(Pid, Res);  
send(Pid, <<Res>>) -> Pid ! {last, Res}.
```

# Sockets TCP



Lo pueden leer también del libro de Simon Thompson:  
Erlang Programming: A Concurrent Approach to Software Development.



# El retorno de los Sockets

Los sockets nos permiten establecer una comunicación para transferir cadenas de bytes con **cualquier proceso** escrito en cualquier **otro** lenguaje, incluso procesos remotos.

Es por esto que vimos cómo manipular binarios en las slides anteriores.





# El retorno de los Sockets

Erlang provee diferentes librerías amigables para interactuar con los sockets, en particular:

- **gen\_udp** : Protocolo de transmisión de paquetes
- **gen\_tcp** : Protocolo orientado a la conexión

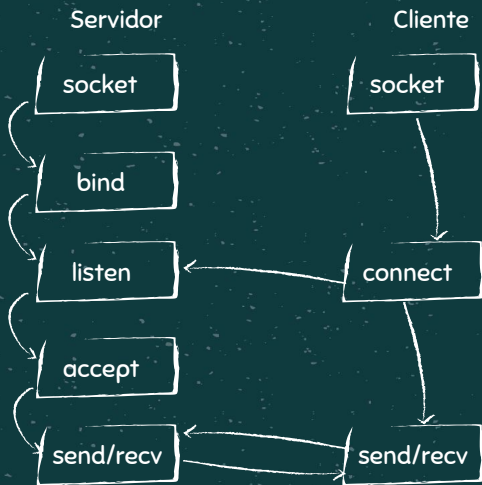
UDP: [https://erlang.org/doc/man/gen\\_udp.html](https://erlang.org/doc/man/gen_udp.html)

TCP: [https://erlang.org/doc/man/gen\\_tcp.html](https://erlang.org/doc/man/gen_tcp.html)

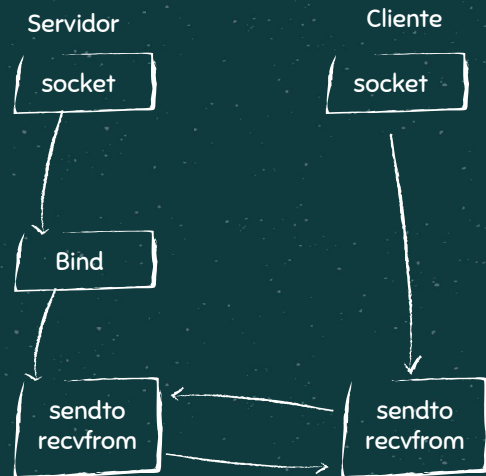
También ofrece otras librerías para el manejo unificado de sockets, pero nosotros nos concentramos en conexiones TCP!

# Modelo Cliente Servidor (Por Red)

TCP : Orientado a la Conexión



UDP: Orientado al Mensaje



Para los que necesiten recordar cómo era el tema en C



# Servidor TCP en Erlang

Asumiendo que hay un proceso esperando una conexión. Cuando aparece una petición de un cliente tenemos dos opciones en erlang:

- 1) Generar un nuevo proceso que atienda al cliente
- 2) Generar otro proceso que espere clientes



# Servidor TCP en Erlang (I)

- Proceso P1 está esperando un cliente.

- Aparece C y se comunica con P1. P1 genera un nuevo proceso P2, y le pasa la comunicación a P2.

- P1 vuelve a esperar que aparezcan clientes.

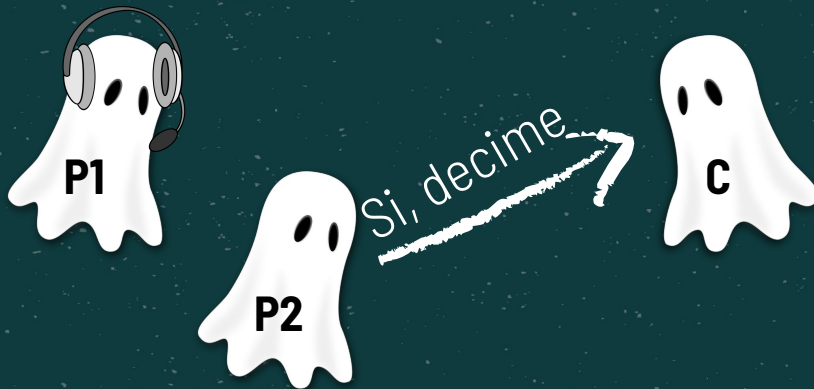
# Servidor TCP en Erlang (I)



# Servidor TCP en Erlang (I)



# Servidor TCP en Erlang (I)





## Servidor TCP en Erlang (2)

Proceso P1 está esperando un cliente.

Aparece C y se comunica con P1. P1 genera un nuevo proceso P2, quién será el nuevo proceso que reciba clientes, y P1 atenderá el pedido de C.



# Servidor TCP en Erlang (2)



## Servidor TCP en Erlang (2)



## Servidor TCP en Erlang (2)





# Recepción de Paquetes

Erlang provee un mecanismo **activo** de Sockets que los convierte a mensajes de Erlang directamente:

```
{tcp, Socket, Paquete}
```

```
{udp, Socket, DirOrigen, Puerto, Paquete}
```

Si bien es una opción tentadora, puede ser que la gran transmisión de mensajes llene los buzones de los procesos de forma innecesaria.

Es decir, asumiendo que la conexión está establecida, los paquetes pueden llegar como mensajes al buzón del proceso.



# Recepción de Paquetes

Erlang provee un mecanismo **pasivo** de Sockets en que los procesos deberán buscar los paquetes usando funciones para recibir los paquetes.

Esta será en general la opción a utilizar, ya que nos protege de cantidades masivas de mensajes en los buzones, y nos permite ir a buscarlos nosotros. Qué pasa con los paquetes? Quedan en las capas inferiores del protocolo de red, y que lo manejen ahí abajo como puedan.



# Ejercicio: Servidor Echo

Servidor Echo!