



# ¡Introducción a Erlang!

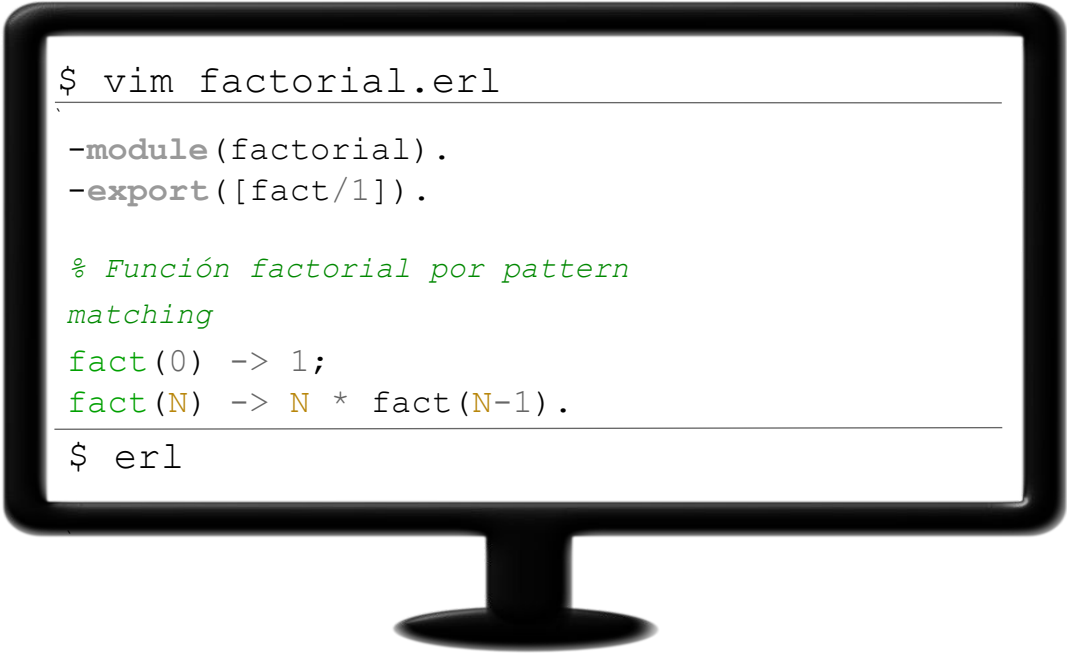
Fundamentos y Conceptos Básicos

Introducción a Erlang. Fundamentos, y conceptos básicos.

Visitar la Página de Erlang.org

*Erlang*





```
$ vim factorial.erl
-module(factorial).
-export([fact/1]).

% Función factorial por pattern
matching
fact(0) -> 1;
fact(N) -> N * fact(N-1).

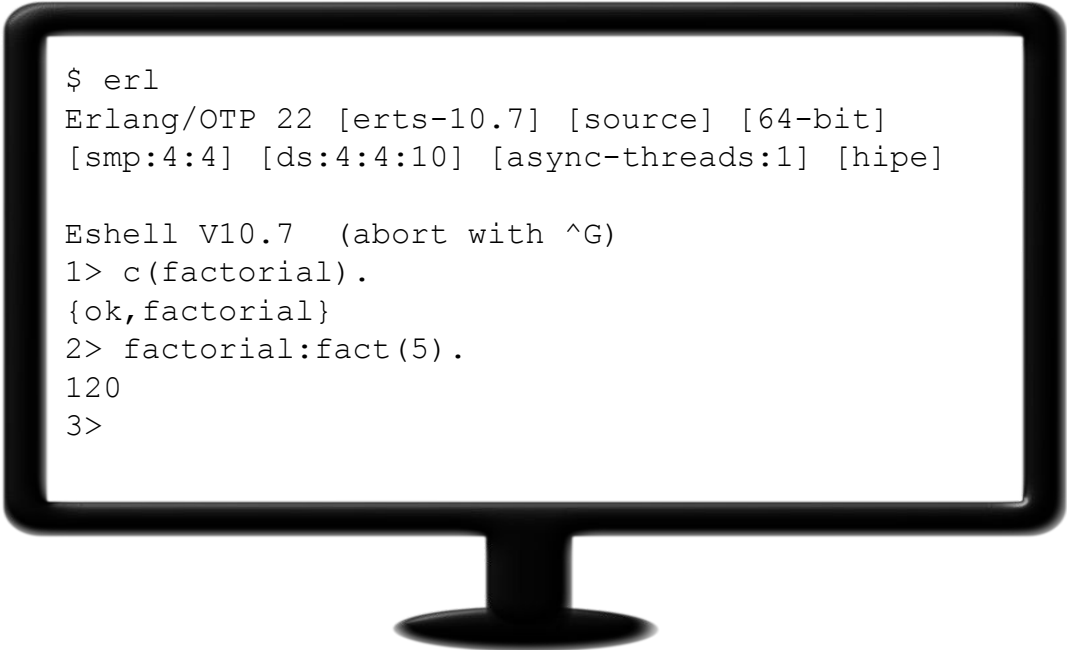
$ erl
```

Primera aproximación a código Erlang. Ejemplo clásico de función recursiva por pattern matching: Factorial!

Primero se declara el módulo `factorial`, y se declara una lista de funciones a exportar, que en nuestro caso es `fact` además avisando que toma un argumento. Notar que todas las directivas terminan con un punto (.).

El pattern matching de factorial es en el primer argumento `0` lo asocia con el fragmento de la derecha de la flecha `->`, los casos se separan por puntos y coma `;`, Y en el siguiente caso toma una variable `N` (notar que es en mayúscula), y ejecuta el código asociado a la derecha. El análisis de casos es secuencial, y se usa el primero que matchea, luego se reemplaza el valor del argumento en el cuerpo a ejecutar, y se ejecuta.

Utilizaremos la Shell de Erlang para ejecutar el código.



```
$ erl
Erlang/OTP 22 [erts-10.7] [source] [64-bit]
[smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Eshell V10.7  (abort with ^G)
1> c(factorial).
{ok,factorial}
2> factorial:fact(5).
120
3>
```

Slide que continúa de la llamada a la Erlang shell.

Cargamos el módulo que declaramos en la slide anterior ``factorial``, usando ``c(factorial).``. El archivo que creamos tiene que estar en el mismo directorio donde ejecutamos la shell.

Nos avisa que fue cargado correctamente, y ejecutamos la función ``factorial:fact(5).``

## ¿Cómo salir de la EShell?

Explicar que para salir de la EShell puede usar ``q().`` o `^G` y `q`.



## Características de Erlang

- Lenguaje **Funcional NO PURO** (No tiene transparencia referencial).
- Tiene **tipado dinámico** (como Python)
- Las funciones se definen usando **recursión** y **pattern matching** (como en Haskell)
- Es un lenguaje orientado a sistemas industriales a gran escala con requerimientos de tiempo real.
- Se basa en dos principios que ya hemos visto: la **conurrencia** y **fiabilidad**.
- Tiene además la posibilidad de ejecutar transparentemente de forma **distribuido**
- Utiliza una máquina virtual!

En un lenguaje funcional NO PURO ya que el Modelo de Actores en el que está basado no permite transparencia referencial. Por ejemplo, la transparencia referencial se rompe cuando una función recibe un mensaje, y el resultado de esta cambia según el mensaje que recibe. Es decir, el resultado de la función puede cambiar aún tomando los mismos argumentos de entrada. Observar que **los mensajes no son argumentos de la función**.

La **transparencia referencial** es un concepto clave en programación funcional que se refiere a la propiedad de una expresión o función en la que su resultado depende únicamente de sus entradas y no de su contexto o estado externo. En otras palabras, dada una expresión o función y un conjunto de valores de entrada, la transparencia referencial garantiza que siempre se obtendrá el mismo resultado, sin importar cuándo o dónde se evalúe.

Erlang es un lenguaje de programación con **tipado dinámico**, lo que significa que las variables no están asociadas con un tipo específico en tiempo de compilación. En cambio, el tipo de una variable se determina en tiempo de ejecución, en función del valor que se le asigna.

# Tipos de Datos en Erlang

## Tipos de Datos Simples

- ✗ Números: Números literales: **1**, **-123**, **23e10**, etc. Estos pueden ser enteros o flotantes.
- ✗ Átomos: por ejemplo **atomo**, **pepegrillo**, **hola**, etc. Son constantes con nombres.
- ✗ Identificadores de Procesos
- ✗ Referencias

## Tipos de Datos Compuestos

- ✗ Tuplas: **{1,a,2,3}**, **{}**, **{1,2,3,4}**, **{aa,bb,w}**. Las tuplas agrupan un número concreto de datos heterogéneos.
- ✗ Listas: Las listas son utilizadas para almacenar un número variable de elementos. Por ejemplo: **[1,2,3,a,b,c]**, **[a,1,{1,2,3},'hello']**.

Las referencias se generan con la función `make_ref/0`. No importa cuando es llamada `make_ref/0`, siempre devuelve una referencia nueva. Es útil para etiquetar mensajes y reconocer si el mensaje que recibimos es una respuesta a uno que acabamos de enviar.



# Pattern Matching

Pattern Matching es usado para asignarles valores a variables y para controlar el flujo del programa.

Erlang es de **asignación única**, es decir, una vez que se le asigna un valor a una variable no se puede cambiar.

Erlang es SSA (Single Static Assignment) y que se les puede asignar una única vez un valor a las variables.

La wildcard es `\_`, y de ser significativo pueden darle nombre `\_Nombre`



### Actividad 3: Completar el fragmento de código.

min.erl:

```
-module(min).  
-export([min/1]).  
  
min([Hd]) -> ??;  
min([Hd|Tl]) ->  
    Rest = min(Tl),  
    if  
        ??  
    end.
```

Acá introducimos otro concepto fundamental de Erlang: Ejercicio de Pattern matching!

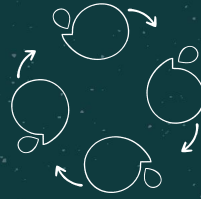
Al terminar de codear, cargar el archivo en la EShell.

Links útiles:

[https://erlang.org/doc/reference\\_manual/expressions.html#if](https://erlang.org/doc/reference_manual/expressions.html#if)

[https://erlang.org/doc/reference\\_manual/expressions.html#term-comparisons](https://erlang.org/doc/reference_manual/expressions.html#term-comparisons)

# Modelo de Actores



- Erlang se basa en el **modelo de actores para abordar la concurrencia y la comunicación entre procesos**. El modelo de actores es un paradigma de programación concurrente en el que los actores son entidades independientes que se comunican entre sí enviando mensajes. Cada actor tiene su propio estado interno y procesa los mensajes que recibe de otros actores.
- En el contexto de Erlang, los actores se implementan como procesos ligeros (*lightweight processes*) y se denominan **procesos Erlang**. Cada proceso Erlang tiene su propio espacio de memoria aislado y se comunica con otros procesos mediante el envío y recepción de mensajes.

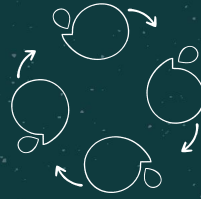
Introducción al modelo de actores. Procesos de Erlang = Actores!

Los procesos de Erlang son una implementación del modelo matemático de actores.

Link de interés:

<https://stackoverflow.com/questions/2708033/technically-why-are-processes-in-erlang-more-efficient-than-os-threads>

# Modelo de Actores



Al recibir un mensaje un actor puede:

- **Crear otros actores:** utilizando la función **spawn/3**
- **Tomar decisiones locales:** Cada proceso tiene la responsabilidad de manejar su propio estado interno y decidir cómo procesar los mensajes que recibe, sin necesidad de coordinar explícitamente con otros procesos.
- **Enviar mensajes a otros actores**
- **Responder al mensaje recibido**

En el modelo de actores en Erlang, cada proceso actúa como una entidad independiente con su propio estado y lógica de procesamiento. Un proceso puede tomar decisiones basadas en su estado actual y ejecutar acciones específicas en respuesta a los mensajes que recibe. Estas decisiones y acciones son locales en el sentido de que no dependen directamente del estado o comportamiento de otros procesos.

# Primitivas de Concurrency





# SPAWN/3

La primitiva **spawn/3** toma 3 argumentos, nombre de un Módulo, el nombre del método a ejecutar, y una lista de argumentos para dicho método. Por ejemplo, en el caso que queramos lanzar un nuevo proceso invocando al método **loop** dentro del módulo **echo** sin argumentos: **spawn(echo,loop,[ ])**.

Como resultado, **spawn**, retorna un identificador de procesos.

El **identificador de un proceso** está compuesto por **<Node.Serial.ID>**:

- **Node**, el número de nodo (0 es el nodo local, un número arbitrario para un nodo remoto)
- **Serial**, los primeros 15 bits del número de proceso (un índice en la tabla de procesos)
- **ID**, bits 16-18 del número de proceso (el mismo número de proceso que Serial)

Primitiva de creación de procesos. Esto debería hacerles acordar al ``fork()`` de C. Aunque recordemos que estamos en realidad creando un actor! (En Erlang los procesos = actores).

2

SEND/2

La primitiva de envío de mensajes se escribe utilizando un símbolo de exclamación: **PId ! Mensaje**.

Por ejemplo podemos enviarle el mensaje **{ok, 1234}** al proceso con identificador **Servidor** de la siguiente forma: **Servidor ! {ok, 1234}**.

Primitiva de envío de mensajes.

Resaltar la diferencia con send de C. La magia está en que los identificadores de procesos tienen mucha más información que un simple número de ejecución en una computadora. Esto lo veremos más adelante.

Detalles, los mensajes son evaluados antes de realizar el envío, es decir, si se quiere enviar el resultado de una función, se evalúa la función y se envía el resultado. Lo mismo puede suceder con el PID.

3

## RECEIVE

La primitiva de recepción de mensajes utiliza pattern matching, toma la siguiente forma:

```
receive
  Mensaje1 -> ...;
  Mensaje2 -> ...;
  .....;
  Mensajen -> ...
end
```

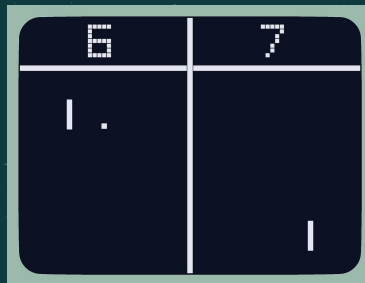
Primitiva de recepción de mensajes.

La primitiva de recepción de mensajes se queda entonces esperando a que llegue un mensaje, y cuando llegue busca el patrón que coincida con el mensaje y ejecuta el código correspondiente.

El resultado del fragmento de código será el resultado del fragmento ejecutado dependiendo del patrón del mensaje.

# Ejercicio

Escribamos un programa en Erlang que cree dos procesos, y se envíen mensajes entre ellos.





## Ejercicio

Escribamos un programa en Erlang que modele un objeto contador.

Un objeto contador es un objeto que soporta una operación que adiciona uno a su valor, y una operación de consulta del valor.

La manera de llevar un estado en Erlang es por medio de sus argumentos y llamandose recursivamente.

Por ejemplo, el estado del contador tiene que ser llevado en el argumento de una función y esta función llamarse recursivamente..

# Envío y Recepción de Mensajes



- Los envíos de mensajes **no** son al proceso sino a un **buzón de mensajes** de un proceso.
- **receive** lo que hace es mirar en el buzón del proceso actual. Esto quiere decir que **receive** es selectivo, y va a buscar el primer mensaje que se corresponda con una cláusula del Pattern Matching.

Rompemos con la idea de que los procesos se mandan mensajes entre ellos, y se introduce la noción de buzón.

Cuando se envía un mensaje, el mensaje se almacena en el buzón de un proceso.

Cuando se espera recibir un mensaje, se busca en el buzón.

Si ningún mensaje se puede asociar a algún patrón del matching, el proceso se va a dormir hasta que lo despierte la llegada de un mensaje en el buzón.

Los mensajes que no se matchean se guardan para eventual procesamiento.