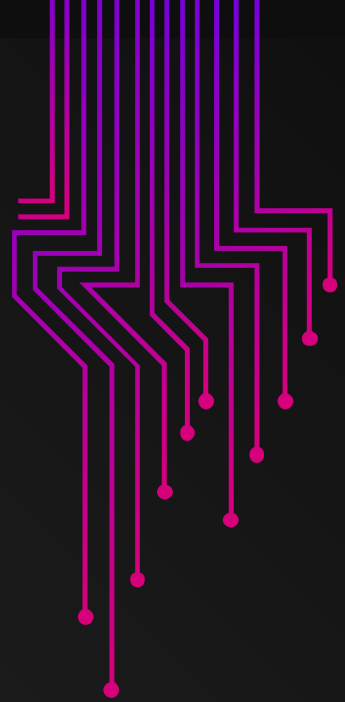


# TUTORIAL ERLANG

Programación Concurrente

Integrantes:

- Julián Alexander Manosalva Manrique
- Juan Sebastián Pachón Carvajal
- David Alexander Zambrano Bohórquez



# Contenido

**01**

**Primeros pasos**

**02**

**Tour por Erlang**

**03**

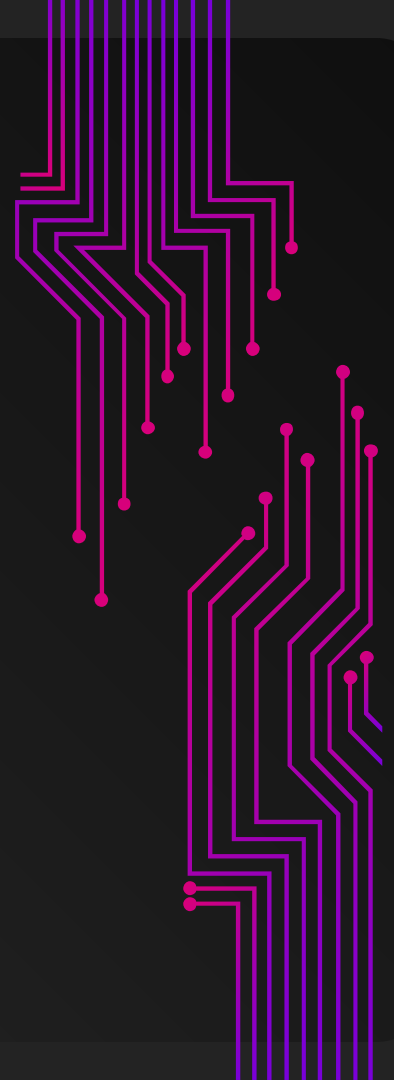
**Particularidades de  
Erlang**

**04**

**Ejemplos**

**01**

# Primeros pasos





# Instalación

## Windows

Instalador x86/x64 desde <https://www.erlang.org/downloads>

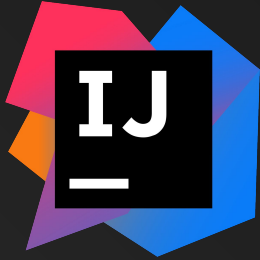
## Linux & macOS - Pre-built

```
For Homebrew on macOS: brew install erlang  
For MacPorts on macOS: port install erlang  
For Ubuntu and Debian: apt-get install erlang  
For Fedora: yum install erlang  
For ArchLinux and Manjaro: pacman -S erlang  
For FreeBSD: pkg install erlang
```

## Código fuente

```
./configure && make && make install
```

# IDES



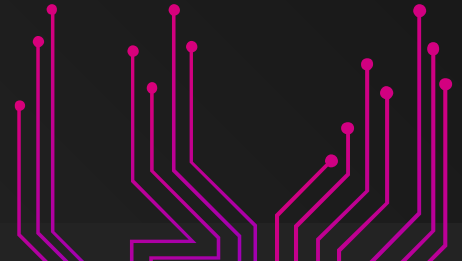
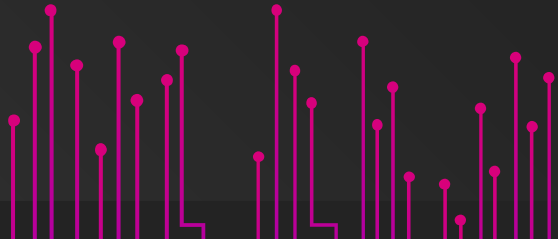
**IntelliJ IDEA**

Erlang Plugin



**VS Code**

Erlang/OTP Erlang-LS



# 02

## Tour por Erlang



# Generalidades

## Creación

Diseñado y creado por la compañía Ericsson el 1986 y cedido como código de uso libre en 1998. Inicialmente concebido para aplicaciones de telefonía.

## Características

Se distingue por manejar datos inmutables, contar con pattern matching y, sobre todo, simplificar la creación, administración y comunicación de procesos.

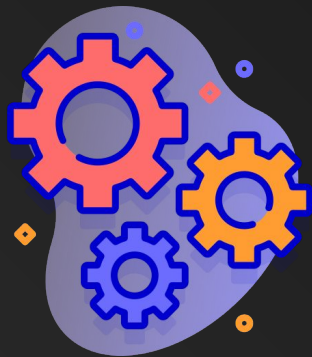
## Aplicaciones

Erlang está diseñado y concebido para: sistemas distribuidos, sistemas en tiempo real, sistemas tolerantes a fallos y aplicaciones de alta disponibilidad y de uso continuo.

## Let It Crash

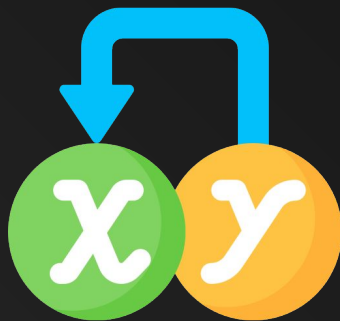
La filosofía “*let it crash*” de Erlang consiste en permitir que los errores ocurran y sean manejados adecuadamente para lograr sistemas más robustos y tolerantes a fallos.

# Introducción



## Funcional

Los programas son composiciones de funciones



## Tipos Dinámicos

Los tipos de las variables dependen del contenido



## Paso de Mensajes

Los procesos se pueden comunicar en red a través de paso de mensajes





# Introducción

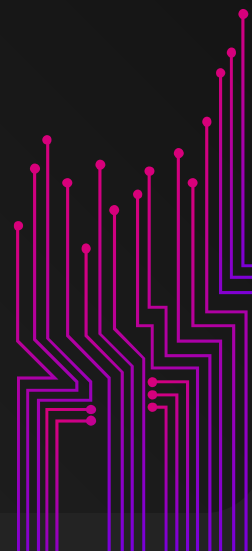
## Lenguaje Modular

Un módulo por fichero, estos deben tener el mismo nombre.  
Es necesario definir el nombre, los ficheros que importa, las funciones que exporta y su implementación.

Se pueden indicar otros metadatos como autor.

Cada módulo se compila por separado

Las funciones se invocan desde el exterior usando la siguiente sintaxis *modulo:func(args)*



# Tipos de Datos



```
42  
-10
```

## Enteros

Los enteros se representan simplemente escribiendo el número, ya sea positivo o negativo.

```
3.14  
-2.71828
```

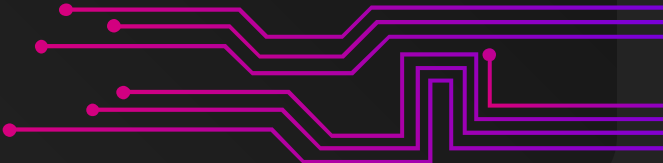
## Flotantes

Los flotantes se representan escribiendo el número con un punto decimal

```
hola  
ok  
'Átomo con espacios'
```

## Átomos

Se escriben iniciando con una minúscula o entre comillas simples si contienen caracteres especiales



# Tipos de Datos



```
"Hola, mundo!"
```

## Strings

Las cadenas de caracteres se representan entre comillas dobles

```
{1, hola, 3.14}
```

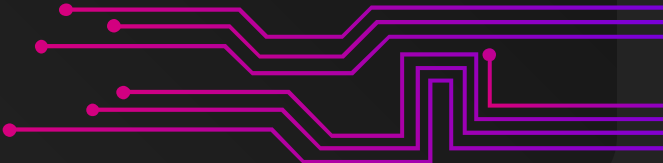
## Tuplas

Las tuplas se representan escribiendo sus elementos entre llaves


```
[1, 2, 3]  
[hola, mundo]
```

## Listas

Se representan escribiendo sus elementos entre corchetes:



# Tipos de Datos



```
#{name=>adam, age=>24, date=>{july, 29}}.
```

## Mapas

Los mapas son pares llave-valor y se representan entre llaves iniciando con #

```
-record(person, {name, age}).
```

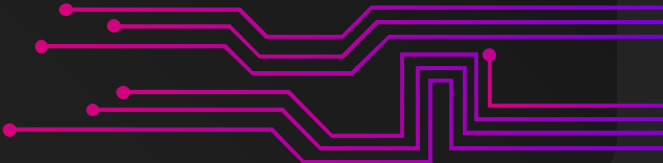
## Record

Similar a una struct en C. Se declaran con la palabra reservada record

```
true or false.  
true
```

## Booleano

Puede tener los valores de true o false



# Variables

X Emisor Msg

Empiezan con  
mayúscula

## Anónima

Se usa el símbolo `_` para  
denotar variables  
anónimas

## Asignación No-destructiva

Las variables solo se  
pueden asignar una  
única vez

# Funciones

## Parámetros

Posee un número fijo de parámetros. Si definimos otra función con el mismo nombre y distinto número de parámetros, se trata de una función distinta.

## Cláusulas

Proporcionan una forma de manejar diferentes casos o condiciones en una función y especifica qué debe hacer la función cuando se cumplen ciertos patrones o condiciones.

## Retorno

En Erlang, no se utiliza explícitamente la palabra clave "return". En su lugar, el valor de la última expresión evaluada en una función se considera el resultado de esa función.

## Ejemplo Cláusulas

```
doble(0) -> 0;  
doble(N) -> 2 * N.
```

# Funciones

## Patrones

El patrón define la estructura y los valores que se esperan en los argumentos de entrada de la función. Puede contener variables, literales o combinaciones de ellos.

## Ejemplo Patrones

```
saludar("Juan") -> "Hola, Juan!";  
saludar("María") -> "¡Hola, María!";  
saludar(_) -> "Hola, ¿cómo estás?";
```

## Guardas

Una guarda es una expresión lógica opcional que se evalúa antes de que se ejecute la cláusula. Si la guarda se evalúa como verdadera, se ejecuta la cláusula correspondiente.

## Ejemplo Guardas

```
es_positivo(N) when N > 0 -> true;  
es_positivo(_) -> false.
```

# Funciones

## Expresiones

Una expresión especifica el resultado que se devuelve cuando la cláusula es evaluada correctamente. Puede ser cualquier expresión válida en Erlang.

## Ejemplo Expresiones

```
sumar(A, B) -> A + B.
```



# Estructuras de control

## If-Else

La estructura *If-Else* nos permite tomar decisiones basadas en una condición. En esta estructura, se evalúan las condiciones en orden hasta encontrar una que sea verdadera.

La expresión asociada a esa condición se evalúa y se devuelve como resultado. Si ninguna condición es verdadera, se evalúa la expresión asociada a *true*.

## Sintaxis If-Else

```
if
  Condición1 -> Expresión1;
  Condición2 -> Expresión2;
  ...
  true -> ExpresiónN
end.
```

# Estructuras de control

## Case

La estructura *Case* nos permite evaluar múltiples patrones y ejecutar diferentes acciones en función de esos patrones.

En esta estructura, la expresión se evalúa y se compara con cada patrón en orden. Cuando se encuentra un patrón que coincide, se ejecuta la acción asociada a ese patrón. Si ningún patrón coincide, se produce un error.

## Sintaxis Case

```
case Expresión of
  Patrón1 -> Acción1;
  Patrón2 -> Acción2;
  ...
  PatrónN -> AcciónN
end.
```

# Estructuras de control

## Iteración mediante recursividad

Muchas definiciones pueden expresarse de forma recursiva mediante:

- Un caso base
- Un caso general, que se define mediante una expresión con casos más simples

Se expresan directamente como una colección de cláusulas (el caso general al final). Toda cláusula termina en ';', excepto la última, que finaliza en ''

## Ejemplo Recursividad

Factorial: El factorial de 0 es 1 (caso base), y el factorial de N es N\*factorial de N-1 (caso general).

```
fac(0) -> 1;  
fac(N) -> N*fac(N-1).
```

# Funciones Built-In

## Disponibilidad

Las funciones Built-In, también conocidas como funciones incorporadas o predefinidas, son funciones que están integradas en el propio lenguaje Erlang y están disponibles sin necesidad de importar librerías externas.

Estas funciones proporcionan una amplia variedad de características y funcionalidades útiles para el desarrollo de aplicaciones en Erlang.

## Algunas funciones

Algunas características comunes de las funciones Built-In en Erlang:

- Funciones matemáticas
- Funciones de listas
- Funciones de cadenas de caracteres
- Funciones de manejo de tiempos

Además, el lenguaje proporciona muchas más funciones integradas para diversas tareas, como operaciones de bits, manipulación de tuplas, manejo de excepciones, entre otros.

# Reducción y unificación

## Actualizaciones

En Erlang, se utiliza un enfoque llamado "cambio sobre estructuras" para manejar las actualizaciones de software.

Esto significa que, en lugar de reemplazar completamente una estructura de datos cuando se realiza una actualización, se comparten la mayor parte de los elementos entre diferentes versiones.

Esto es posible debido a la inmutabilidad de los datos en Erlang.

## Semántica write-once

En Erlang, se sigue una semántica "write-once", lo que significa que una vez que un valor se asigna a una variable, no se puede modificar.

Esta propiedad de inmutabilidad simplifica el razonamiento sobre el comportamiento de un programa, ya que se evitan los efectos secundarios inesperados causados por cambios en los valores de las variables.

# Reducción y unificación

## Pattern-matching

En Erlang, el pattern-matching es una característica poderosa que permite hacer coincidir valores con patrones específicos.

Esto permite una mayor flexibilidad en la concordancia de patrones y la toma de decisiones basada en condiciones específicas.

## Higher-order functions

En Erlang, las funciones son ciudadanos de primera clase, lo que significa que se pueden pasar como argumentos a otras funciones y también se pueden devolver como resultados.

Esto se conoce como funciones de orden superior o “higher-order functions”.

Este concepto permite una programación más modular y flexible, ya que las funciones se pueden combinar y componer de manera dinámica.

# 03

## Particularidades de Erlang



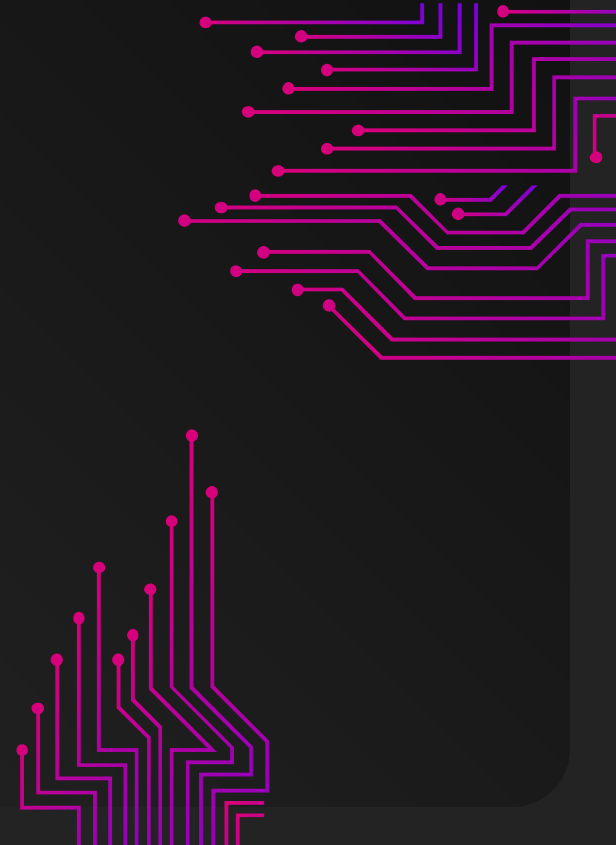
# ¿Qué lo hace especial para la concurrencia?

Es un lenguaje de programación funcional que también dispone de un entorno de ejecución.

Tiene soporte integrado para

- Concurrencia
- Distribución
- Tolerancia a fallos

Se desarrolló originalmente para ser utilizado en varios sistemas de telecomunicaciones grandes de Ericsson en 1986.





# ¿Por qué Erlang?

Debe ser fácilmente distribuible a través de una red de ordenadores.

Debe ser escalable.

Debe responder a los usuarios dentro de unos plazos estrictos.

Debe gestionar un gran número de actividades simultáneas.

Debe ser tolerante a fallos de software y hardware.

Debe poder actualizarse y reconfigurarse fácilmente sin tener que detenerse

# Concurrencia en Erlang

## Erlang



Es fácil crear hilos de ejecución paralelos



Permite que estos hilos se comuniquen entre sí



En Erlang, cada hilo de ejecución se denomina proceso.

# Procesos

## Spawn

Crea un nuevo proceso concurrente que evalúa funciones. El nuevo proceso se ejecuta en paralelo con la llamada. Teniendo en cuenta la forma en la que se definan las funciones dentro del código, la sintaxis puede ser de dos maneras diferentes.

## Sintaxis Spawn

Funciones anónimas

```
spawn(fun() -> Function_content end).
```

```
spawn(fun() -> server("Hello") end).
```

Definiendo una función

```
spawn(Module, Exported_function, List_of_arguments).
```

```
spawn(tut14, say_something, [hello,3]).
```

# Comunicación entre hilos

## “!” y receive...end

Para la comunicación entre hilos existen dos sentencias, una para enviar el mensaje y otra para recibir.

“!” envía un mensaje al proceso con identificador Pid. El envío de mensajes es asíncrono.

“Receive...end” recibe un mensaje que ha sido enviado a un proceso.

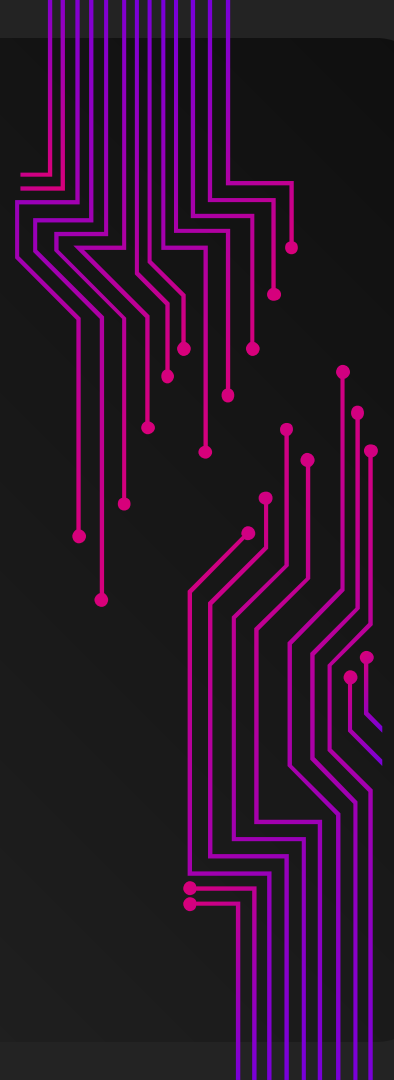
## Sintaxis

```
Pid ! Message.
```

```
receive  
  pattern1 ->  
    actions1;  
  pattern2 ->  
    action2;  
  ...  
  patternN ->  
    actionsN  
end.
```

# 04

## Ejemplos



# Referencias

## Consultas

- [Tutorial Erlang](#)
- [Getting Started with Erlang](#)
- [Erlang. Programación Distribuida y su Aplicación Bajo Internet.](#)
- [Erlang. Perdiendo el miedo a la programación concurrente.](#)

## Íconos

- [Freepik](#)

## Enlace al notebook trabajado

- 

