Erlang

Perdiendo el miedo a la programación concurrente

Manuel Montenegro
Facultad de Informática

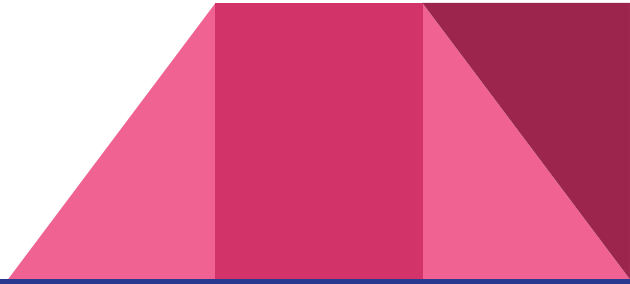
1. Primeros pasos
2. Interacción entre procesos
3. Interacción entre nodos
4. Interacción entre máquinas

El lenguaje Erlang

- Desarrollado en 1986 por Ericsson.
- Inicialmente concebido para aplicaciones de telefonía.

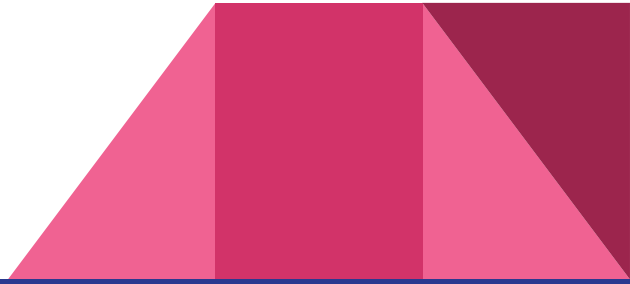
Principales características

- Integra paradigmas funcional y concurrente.
- Dinámicamente tipado.
- Enfoque pragmático.



Principios de diseño

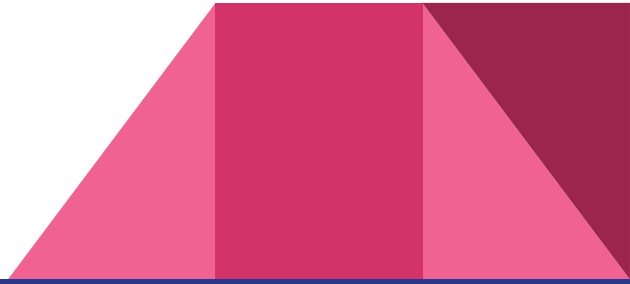
- Todo es un proceso.
- La creación y destrucción de procesos debe ser ligera y eficiente.
- Los procesos no comparten recursos.
- La única manera de interacción entre procesos es el paso de mensajes.
- Filosofía *Let it crash*.



1. Primeros pasos

Iniciar una *shell*

- Ejecuta el fichero `single_node.bat`.
- Se iniciará una *shell* de Erlang en la que podemos introducir comandos.



Rareza nº 1

Hay que escribir un punto (.) en la shell para finalizar una expresión.

Operaciones aritméticas

```
1> 1 + 5.
6
2> 2 + (1 / 2).
1.5
3> 3 * 9 - 4.
23
```

Cuidado con el punto!

Cuidado con el punto!

Cuidado con el punto!

Átomos

- Son constantes simbólicas, similares a los enumerados de C.
- Comienzan por letra minúscula y no tienen espacios.

```
1> hola.
hola
2> ok.
ok
3> qué_tal_estás.
qué_tal_estás
4> true.
true
5> 5 =< 4.
false
```

Los booleanos son átomos

Variables

- Comienzan por letra mayúscula.

```
1> X = 4.
4
2> 2 * X + 6.
14
3> Edad = 35.
35
4> Jubilacion = 2018 + (67 - Edad).
2050
5> Z = 20, W = 15, Z * W.
300
6> X = 25.
** exception error: no match of right hand side value 25
```

Secuencia de expresiones

???

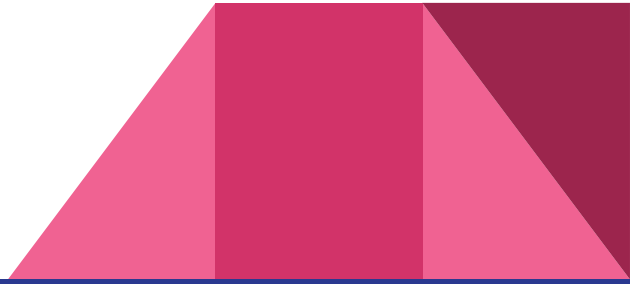
Rareza nº 2

Todas las variables son inmutables.

Inmutabilidad

- Una vez que una variable toma un valor, no se le puede asignar otro distinto.
- Esto es muy característico de los lenguajes funcionales.

```
1> X = 4.
4
2> X = 5.
** exception error: no match of right hand side value 5
```



¿Y en la *shell*?

- ¿Tenemos que estar inventándonos variables continuamente para hacer pruebas?
- No, porque tenemos el comando `f()` para “olvidar” variables.

```
1> X = 4, Y = 10, X + Y.
14
2> f(X).
ok
3> X + 1.
* 1: variable ‘X’ is unbound.
4> f().
ok
```

Borrar valor de X

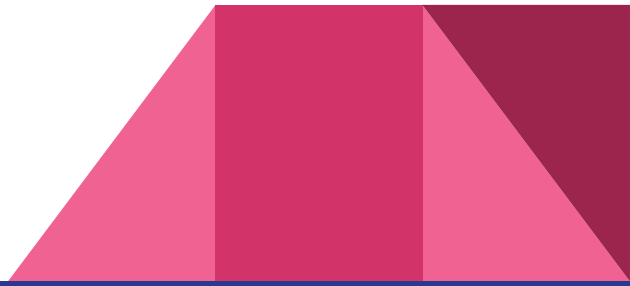
Borrar valor de todas las variables definidas hasta el momento

Nuestra primera función

Fichero `hola_mundo.erl`

```
saludar() ->
  io:format("Hola, mundo!~n").
```

~n = Fin de línea



Nuestra primera función

- Toda función debe estar definida dentro de un **módulo**.
- El módulo indica mediante `-export` qué funciones son visibles desde fuera del módulo.

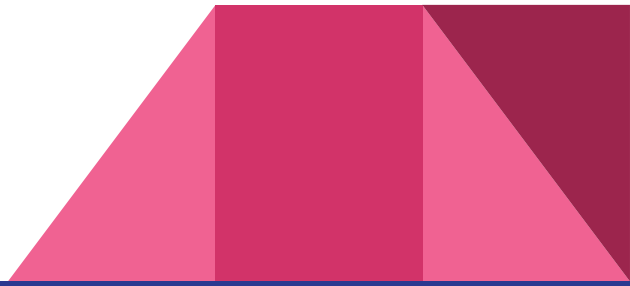
```
-module(hola_mundo).

-export([saludar/0]).

saludar() ->
  io:format("Hola, mundo!~n").
```

saludar/0 = Función saludar con cero parámetros

Compilar el módulo y llamar a sus funciones

- Desde la *shell*:

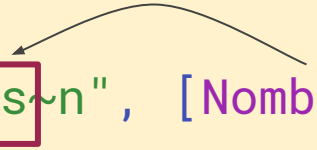
```
1> c(hola_mundo).
{ok, hola_mundo}
2> hola_mundo:saludar().
Hola, mundo!
ok
```



Creamos una nueva función

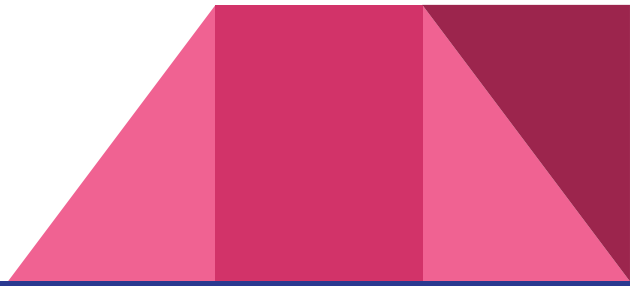
- Esta vez tiene un parámetro.

```
-module(hola_mundo).


-export([saludar/0, saludar/1]).
...
saludar(Nombre) ->
  io:format("Hola, ~s~n", [Nombre]).
```



Volvemos a la *shell*

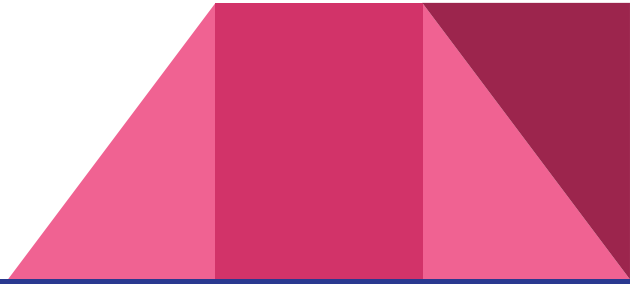
```
1> c(hola_mundo).
{ok, hola_mundo}
2> hola_mundo:saludar("Gloria").
Hola, Gloria!
ok
```



Una modificación

- Saludar tantas veces como el número indicado por el primer parámetro.

```
1> hola_mundo:saludar(4, “Gloria”).
Hola, Gloria! (4)
Hola, Gloria! (3)
Hola, Gloria! (2)
Hola, Gloria! (1)
ok
```



Rareza nº 3

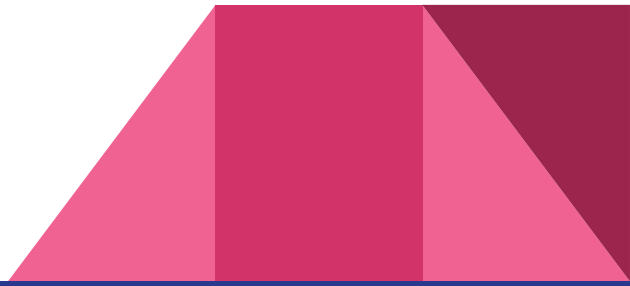
En Erlang no hay bucles

...pero sí hay recursión :-)

```
saludar(0, Nombre) -> ok;

saludar(Contador, Nombre) when Contador > 0 ->
  io:format("Hola, ~s (~w)~n", [Nombre, Contador]),
  saludar(Contador - 1, Nombre).
```

Caso base

Caso recursivo

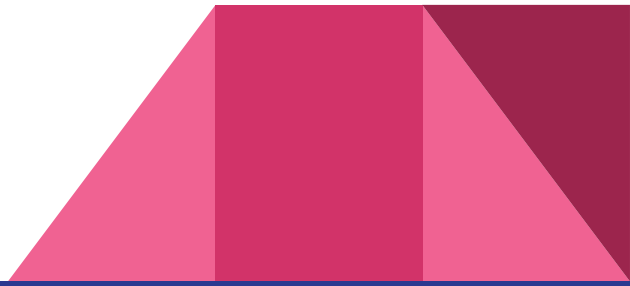


...pero sí hay recursión :-)

```
saludar(0, _) -> ok;

saludar(Contador, Nombre) when Contador > 0 ->
  io:format("Hola, ~s (~w)~n", [Nombre, Contador]),
  saludar(Contador - 1, Nombre).
```

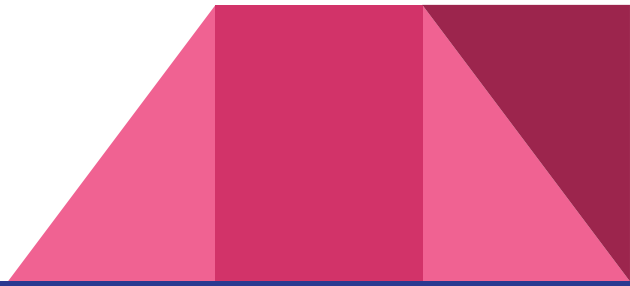
Caso base

Caso recursivo



¿Y si quiero un contador ascendente?

```
1> hola_mundo:saludar_asc(4, “Gloria”).
Hola, Gloria! (1)
Hola, Gloria! (2)
Hola, Gloria! (3)
Hola, Gloria! (4)
ok
```



¿Y si quiero un contador ascendente?

- Utilizamos un parámetro adicional.

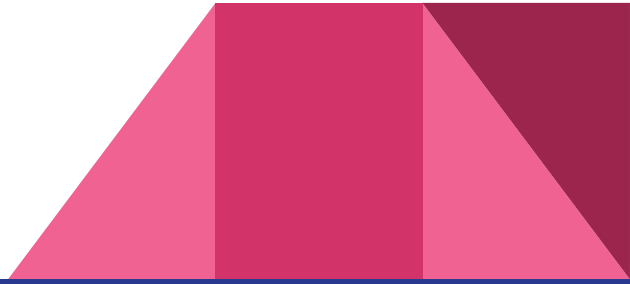
```
saludar_asc(Contador, Max, _) when Contador > Max -> ok;
saludar_asc(Contador, Max, Nombre) ->
  io:format("Hola, ~s (~w)~n", [Nombre, Contador]),
  saludar_asc(Contador + 1, Max, Nombre).

saludar_asc(Max, Nombre) -> saludar_asc(1, Max, Nombre).
```

Llamada inicial

Tuplas

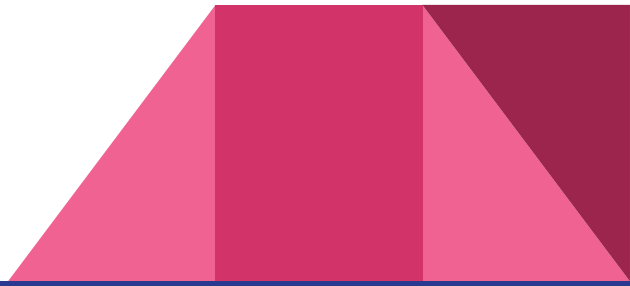
- Utilizadas para agrupar varios resultados.
- Sus componentes se delimitan entre { y }

```
decrement(0)   -> {error, already_zero};
decrement(Num) -> {ok, Num - 1}.
```



Tuplas: ejemplo de uso

```
1> tuple_examples:decrement(5).
{ok, 4}
2> tuple_examples:decrement(0).
{error, already_zero}
```



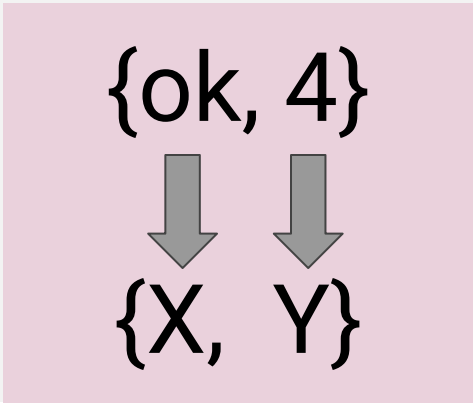
Rareza nº 4

El ajuste de patrones

Acceder a los componentes de una tupla

- Mediante **ajuste de patrones**.

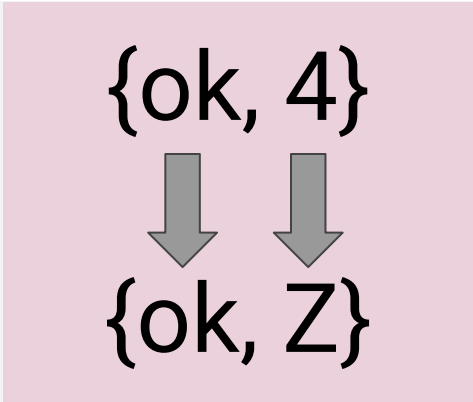
```
1> {X, Y} = tuple_examples:decrement(5).
{ok, 4}
2> X.
ok
3> Y.
4
```



Acceder a las componentes de una tupla

- Si conozco que el resultado va a ser de la forma `{ok, …}`

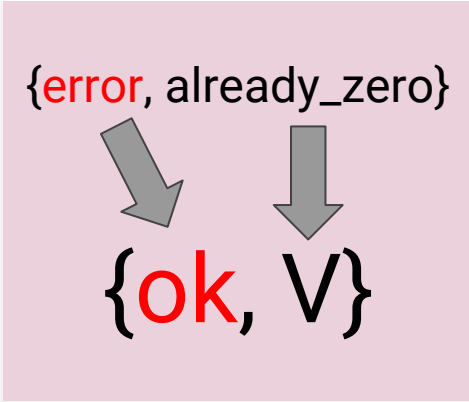
```
1> {ok, Z} = tuple_examples:decrement(5).
{ok, 4}
2> Z.
4
3> {ok, 2} = tuple_examples:decrement(3).
{ok, 2}
4> {ok, V} = tuple_examples:decrement(0).
** exception error: no match of right hand side value
{error,already_zero}
```



Acceder a las componentes de una tupla

- Si conozco que el resultado va a ser de la forma `{ok, …}`

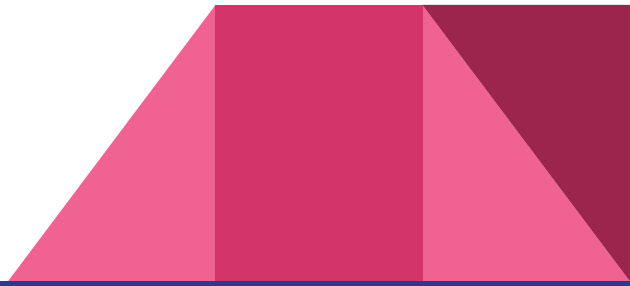
```
1> {ok, Z} = tuple_examples:decrement(5).
{ok, 4}
2> Z.
4
3> {ok, 2} = tuple_examples:decrement(3).
{ok, 2}
4> {ok, V} = tuple_examples:decrement(0).
** exception error: no match of right hand side value
{error,already_zero}
```



Expresiones `case`

- Distinción de casos + ajuste de patrones

```
decrement_print(N) ->
  case decrement(N) of
    {ok, NDec} ->
      io:format("Resultado: ~w~n", [NDec]);
    {error, _} ->
      io:format("Error al decrementar.~n")
  end.
```



Listas

```
1> Xs = [1, 3, ok, 10, 4, “Hola”].
[1,3,ok,10,4,"Hola"]
2> length(Xs).
6
3> length([]).
0
4> Xs ++ [10, 30].
[1,3,ok,10,4,"Hola",10,30]
5> lists:reverse(Xs).
["Hola",4,10,ok,3,1]
6> lists:member(3, Xs).
true
```

[ ] = Lista vacía

++ = Concatenación de listas

Listas intensionales

```
1> Xs = [1, 2, 3, 4, 5].
[1,2,3,4,5]
2> [ 2*X + 1 || X <- Xs ].
[3,5,7,9,11]
3> [ 3*X || X <- Xs, X rem 2 == 0 ].
[6,12]
4> length([ X || X <- Xs, X rem 2 == 0]).
2
5> [ {X, Y} || X <- [1,2,3], Y <- [a,b] ].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

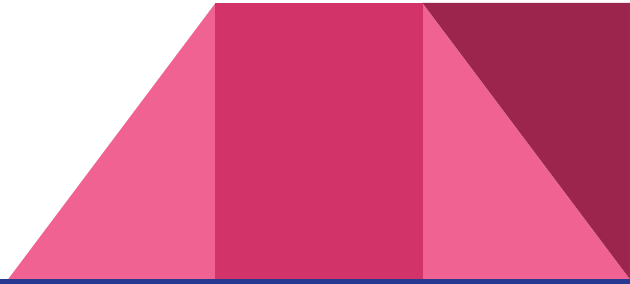
"Para todo X en Xs, devolver 2 * X + 1"

"Para todo X de Xs, tal que X es par, devolver 3 * X"

Número de elementos pares

Caracteres

```
1> $A.
65
2> $v.
118
3> $0.
48
4> $□.
128526
5> $ .
32
```



Rareza nº 5

Los caracteres se representan mediante números enteros.

Rareza nº 6

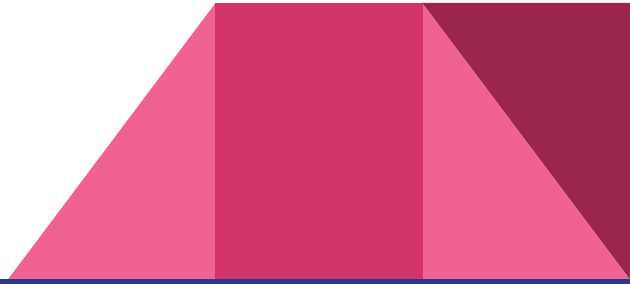
Las cadenas de texto son listas de caracteres.

Cadenas: listas de caracteres

```
1> Cadena = “Qué hay de nuevo?”.
"Qué hay de nuevo?"
2> [$H, $o, $l, $a].
“Hola”
3> [72, 111, 108, 97].
“Hola”
4> lists:reverse(Cadena).
"?oveun ed yah éuQ"
5> length(Cadena)
17
6> lists:member($a, Cadena).
true
7> [ C || C <- Cadena, not lists:member(C, “aeiou”)].
"Qé hy d nv?"
```

Ejemplo

- Queremos hacer una función que reciba una cadena de texto y sustituya todas las letras en el rango **A..Z** por caracteres de subrayado (_).



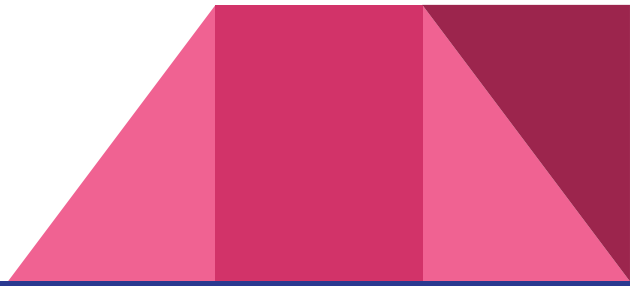
Ejemplo

- Función para transformar un carácter individual:

```
hide_char(X) when X >= $A, X =< $Z -> $_;
hide_char(X) -> X.
```

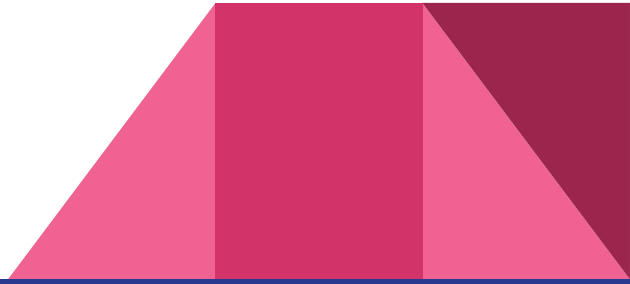
- Función para transformar una cadena entera:

```
hide_string(String) -> [ hide_char(X) || X <- String ].
```



Ejecución de `hide_string/1`

```
1> string_utils:hide_string("HOLA, ME LLAMO MANU").
"____, __ _____ ____"
```



Una mejora

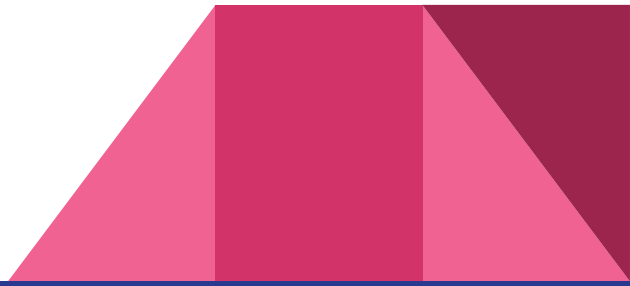
- Función `hide_string(Cadena, Chars)`

  Realiza la sustitución de las letras en **A..Z** por caracteres de subrayado, **excepto aquellas que se encuentren la lista** `Chars`.

Una mejora

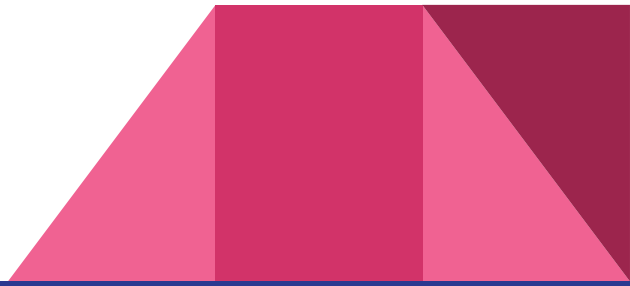
```
hide_char(X, Chars) when X >= $A, X =< $Z ->
  case lists:member(X, Chars) of
    true -> X;
    false -> $_
  end;
hide_char(X, _)  -> X.
```

```
hide_string(String, Chars) ->
   [ hide_char(X, Chars) || X <- String ].
```



Ejecución de `hide_string/2`

```
1> string_utils:hide_string("HOLA, ME LLAMO MANU", "ALM").
"__LA, M_ LLAM_ MA__"
```



2. Interacción entre procesos

El “Hola mundo” de los procesos (...literal)

- ¿Recuerdas la función `saludar/2`?
- Vamos a añadir un retardo de 5 segundos entre mensajes:

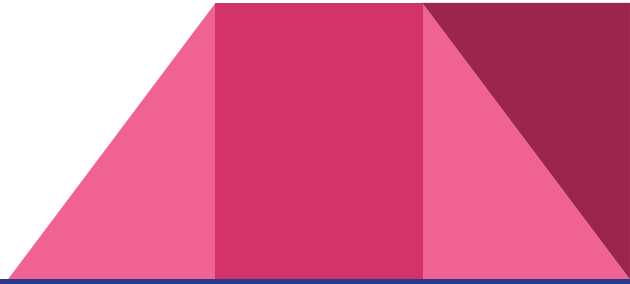
```
saludar(0, _) -> ok;
saludar(Contador, Nombre) when Contador > 0 ->
  io:format("Hola, ~s (~w)~n", [Nombre, Contador]),
  timer:sleep(5000),
  saludar(Contador - 1, Nombre).
```

Retardo

La función spawn

```
spawn(Modulo, Funcion, [Arg1, …, Argn])
```

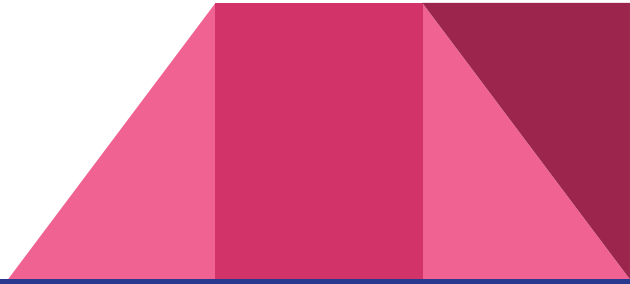
- Ejecuta la `Funcion` (perteneciente a `Modulo`) pasándole los parámetros `Arg1, …, Argn`.
- La ejecución de la función se realiza de modo concurrente a la *shell*, en un proceso nuevo.



La función spawn

```
1> spawn(concurrente, saludar, [4, “Gloria”]).
Hola, Gloria (4)
<0.146.0>
Hola, Gloria (3)
Hola, Gloria (2)
Hola, Gloria (1)
```

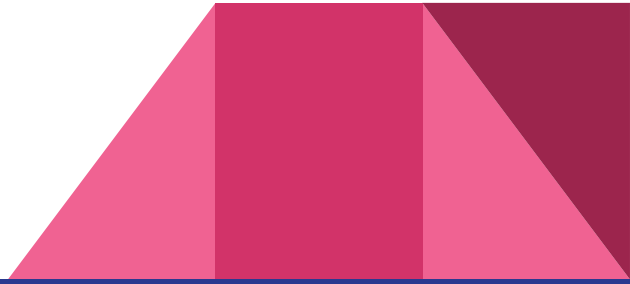
¿Qué es eso?



PID (*Process IDentifier*)

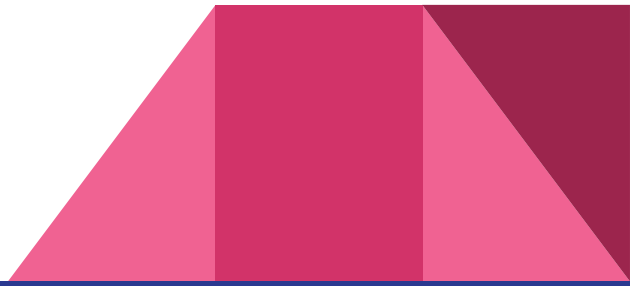
- El valor `<0.146.0>` es el identificador del proceso recién creado.
- La función `spawn` crea un proceso y devuelve su PID, el cual podemos guardar en una variable.

```
1> Pid = spawn(concurrente, saludar, [4, “Gloria”]).
…
2> Pid.
<0.146.0>
```



¿Para qué sirve?

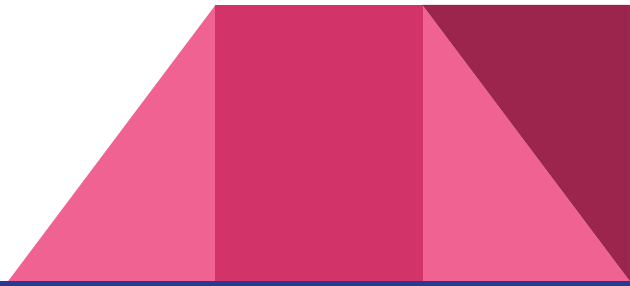
- El PID nos permite hacer referencia al proceso recién creado para:
  - Poder enviar mensajes al proceso.
  - Forzar la terminación del proceso.
  - Asignar un nombre al proceso.
  - etc.



Recibir mensajes: la cláusula `receive`

- La cláusula `receive` detiene la ejecución de un proceso hasta que llegue un mensaje a dicho proceso.

```
recibir_mensaje() ->
  receive
    Mensaje ->
      io:format("He recibido un mensaje: ~w~n", [Mensaje])
  end.
```



Enviar mensajes

- Se utiliza el operador `!`

```
Pid ! Mensaje
```

- Envía el `Mensaje` al proceso con el `Pid` dado.

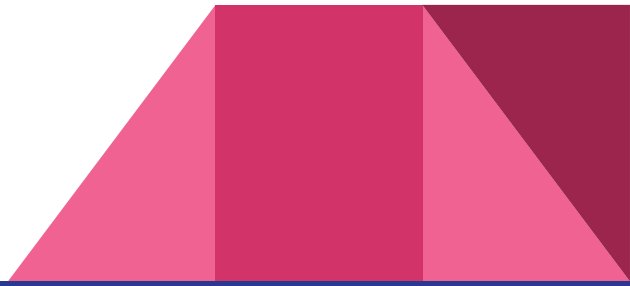
```
1> Pid = spawn(concurrente, recibir_mensaje, []).
…
2> Pid ! “Hola”.
He recibido un mensaje: "Hola"
“Hola”.
2> Pid ! 34.
...
```

¡Ojo! El proceso está muerto

Finalización de un proceso

- Cuando finaliza la función `recibir_mensaje`, el proceso creado por `spawn` muere.

```
recibir_mensaje() ->
  receive
    Mensaje ->
      io:format("He recibido un mensaje: ~w~n", [Mensaje])
  end.
```



Bucle de mensajes

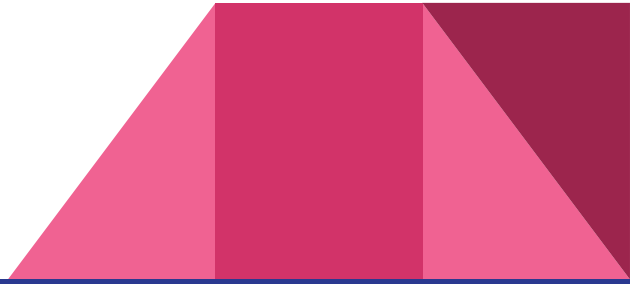
- Si queremos que siga recibiendo mensajes, debemos hacer una llamada recursiva:

```
recibir_varios_mensajes() ->
  receive
    Mensaje ->
      io:format("He recibido un mensaje: ~w~n", [Mensaje]),
      recibir_varios_mensajes()
  end.
```

Llamada recursiva

Bucle de mensajes

```
1> Pid = spawn(concurrente, recibir_varios_mensajes, []).
…
2> Pid ! “Hola”.
He recibido un mensaje: "Hola"
“Hola”
3> Pid ! {ok, [1,2,3]}.
He recibido un mensaje: {ok,[1,2,3]}
{ok,[1,2,3]}
```



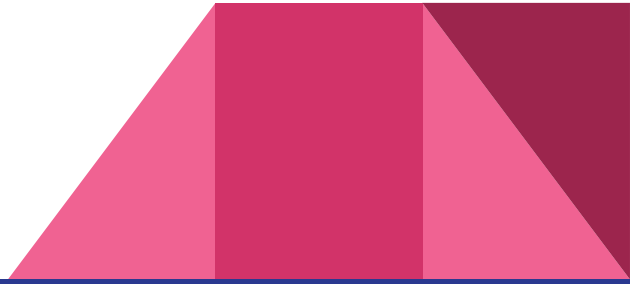
Ajuste de patrones sobre el mensaje

- Podemos tener varias ramas en un `receive`, cada una de ellas ajustándose a un patrón.

```
aritmetica() ->
  receive
    {suma, X, Y} ->
      io:format("~w + ~w = ~w~n", [X, Y, X + Y]),
      aritmetica();
    {resta, X, Y} ->
      io:format("~w - ~w = ~w~n", [X, Y, X - Y]),
      aritmetica();
    ...
  end.
```

Bucle de mensajes

```
1> PA = spawn(concurrente, aritmetica, []).
…
2> PA ! {suma, 10, 3}.
10 + 3 = 13
{suma,10,3}
3> PA ! {resta, 10, 3}.
10 - 3 = 7
{resta,10,3}
```



Guardar información entre mensajes (*estado*)

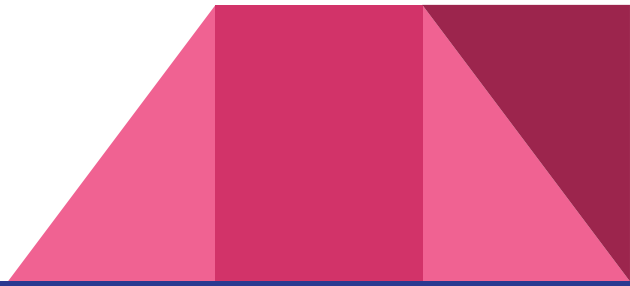
- Podemos utilizar los parámetros del bucle de mensajes.

```
contador(N) ->
  receive
    imprimir ->
      io:format("Valor actual del contador: ~w~n", [N]),
      contador(N);
    incrementar ->
      contador(N + 1)
  end.
```

Guardar información entre mensajes (*estado*)

```
1> PC = spawn(concurrente, contador, [0]).
…
2> PC ! imprimir.
Valor actual del contador: 0
3> PC ! incrementar.
4> PC ! incrementar.
5> PC ! incrementar.
6> PC ! imprimir.
Valor actual del contador: 3
```

Valor inicial = 0

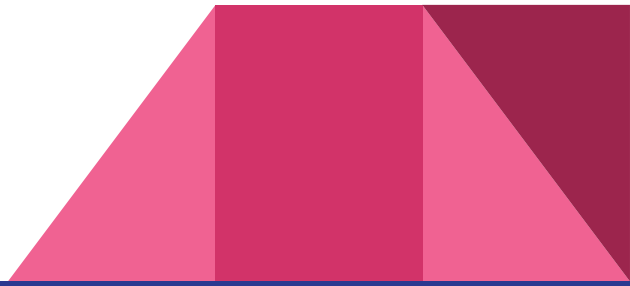


Registro de procesos

- Es posible asignar un nombre a un proceso, y utilizar este nombre para enviar mensajes, en lugar de PID.

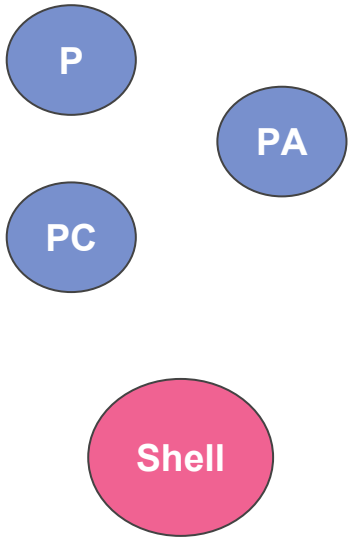
```
1> register(mi_contador, PC).
true
2> mi_contador ! imprimir.
Valor actual del contador: 3
```

Asignamos el nombre `mi_contador` al proceso cuyo PID es PC



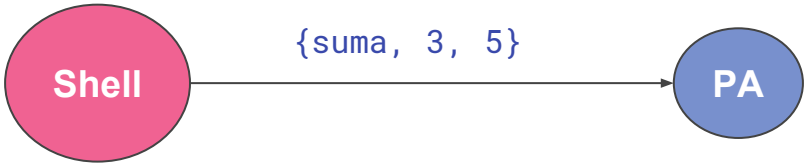
Comunicación entre procesos

- Estamos ejecutando varios procesos:
  - Proceso que imprime los mensajes que recibe.
  - Proceso que realiza operaciones aritméticas.
  - Proceso contador.
- ¿Hay alguno más?
  - Sí. Muchos más que no vemos.
  - Entre ellos, **la shell**.



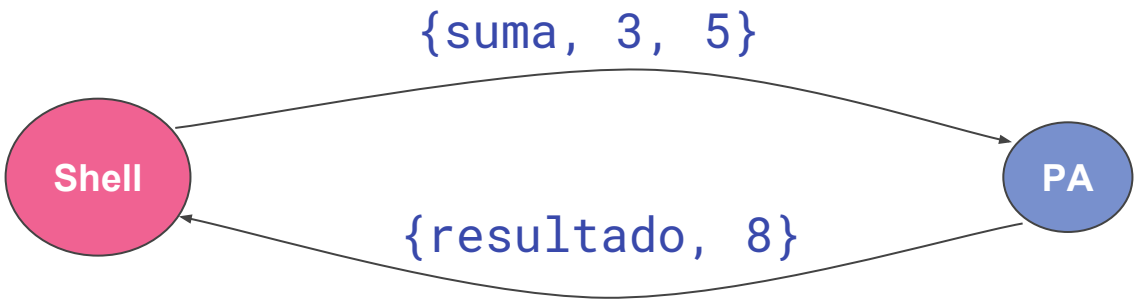
Comunicación entre procesos

- Cuando desde la *shell* tecleamos `PA ! {suma, 3, 5}`, el proceso *shell* envía un mensaje al proceso que realiza operaciones aritméticas.



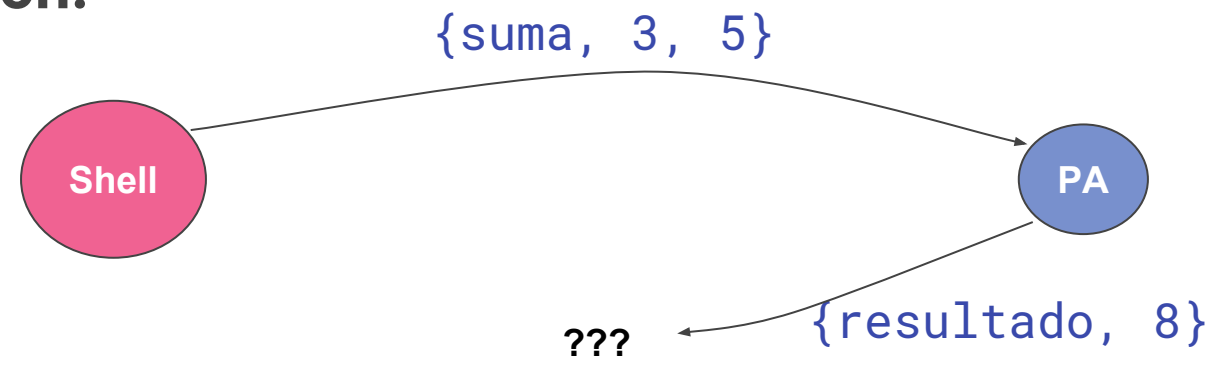
- Hasta ahora, la comunicación es unidireccional.

¿Podemos hacerla bidireccional?



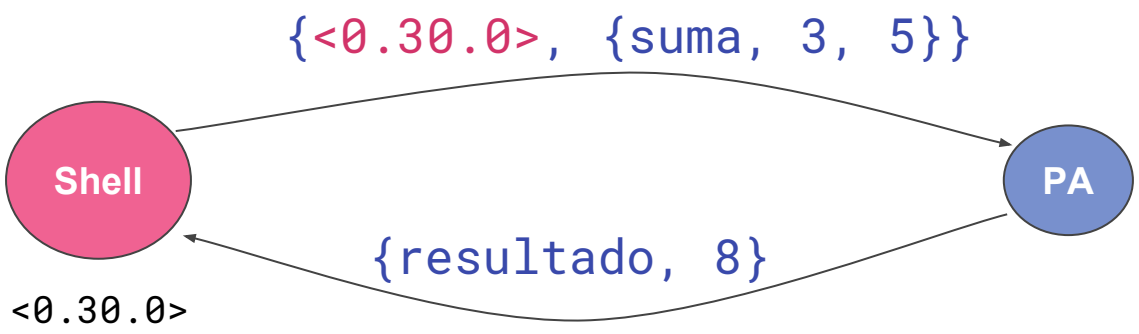
¿Podemos hacerla bidireccional?

- Sí, pero el servidor PA ha de enviar la respuesta mediante el uso del operador `!`
- Para ello, **necesita saber el PID del proceso que envió la petición.**



¿Podemos hacerla bidireccional?

- El remitente ha de incluir **su PID** en su petición.

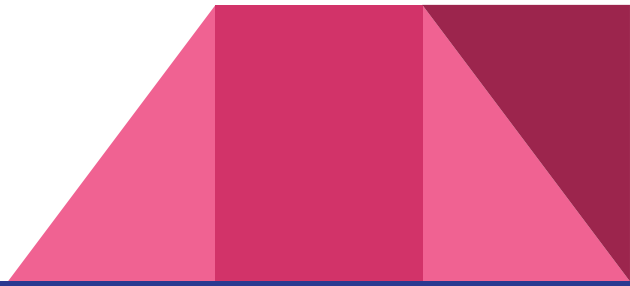


Comunicación bidireccional

```
aritmetica_respuesta() ->
  receive
    {From, {suma, X, Y}} ->
      From ! {resultado, X + Y},
      aritmetica_respuesta();
    ...
  end.
```

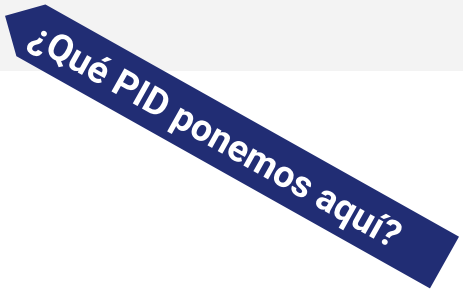
Remitente

Enviar respuesta al remitente



Comunicación bidireccional

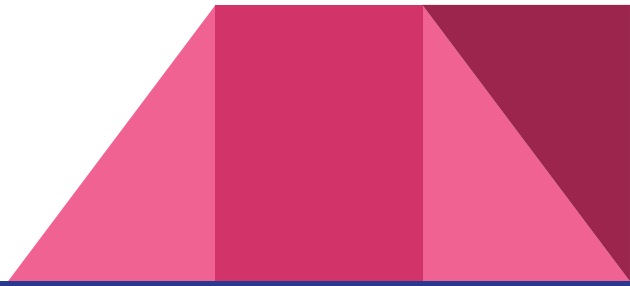
```
1> PA = spawn(concurrente, aritmetica_respuesta, []).
…
2> PA ! { ??????, { suma, 3, 5 }}.
```



Comunicación bidireccional

```
1> PA = spawn(concurrente, aritmetica_respuesta, []).
…
2> PA ! { self(), { suma, 3, 5 }}.
```

La función `self()` devuelve el PID del proceso que la invoca.

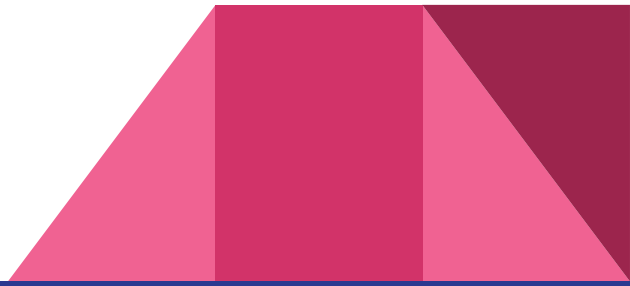


Comunicación bidireccional

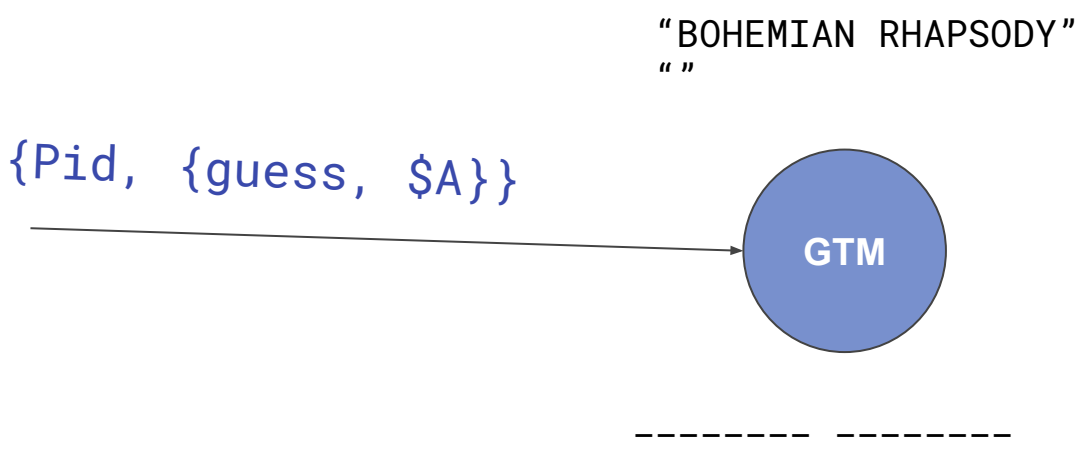
- La shell ha recibido una respuesta del proceso PA.
- ¿Cómo puedo ver los mensajes recibidos por la *shell*?

```
1> flush().
Shell got {resultado,8}
ok
```

Imprimir mensajes recibidos por la *shell*



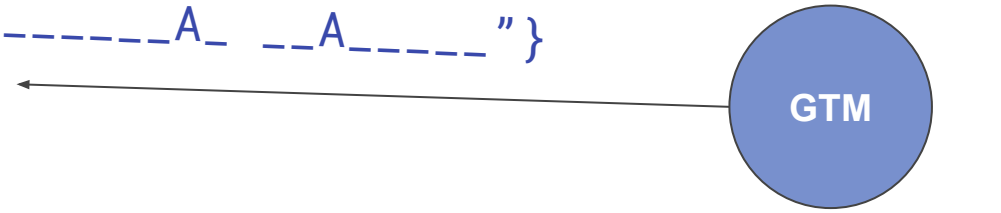
Guess the movie



Guess the movie

“BOHEMIAN RHAPSODY”
“A”

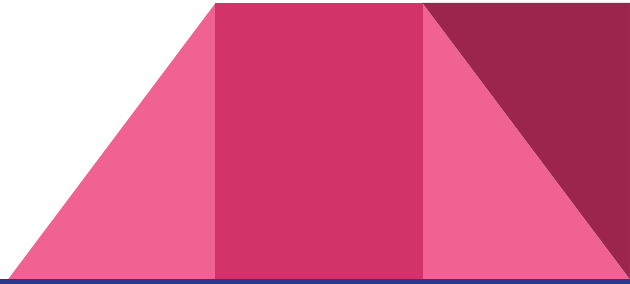
{hit, 2, “_____A_ __A_____”}



_____A_ __A_____

Guess the movie

```
1> GTM = gtm_server:start("Bohemian Rhapsody").
-------- --------
…
2> GTM ! {self(), {guess, $O}}.
_O______ _____O__
3> flush().
Shell got {hit, 2, “_O______ _____O__”}
4> GTM ! {self(), {guess, $T}}.
_O______ _____O__
5> flush().
Shell got {miss, “_O______ _____O__”}
```

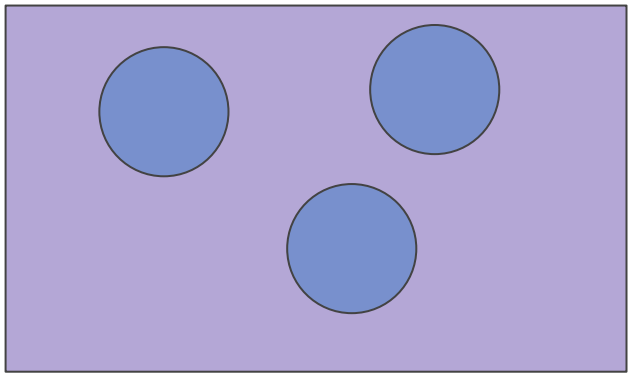


3. Interacción entre nodos

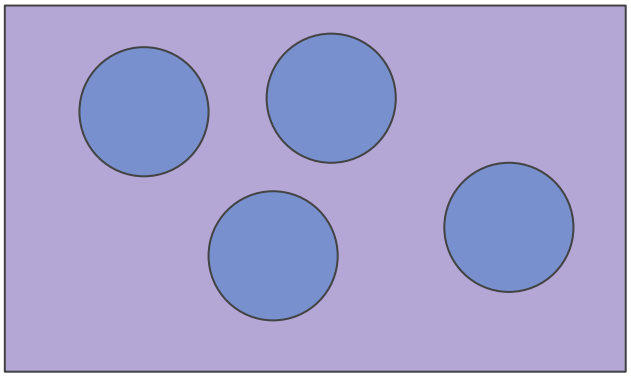
Nodos

- Un nodo es un sistema de Erlang en ejecución.
- Cada nodo puede estar identificado con un **nombre**.
- En un nodo puede haber varios procesos ejecutándose.

nodo1@PT00520



nodo2@PT00520



Envío de mensajes entre nodos

- Para enviar un mensaje a un proceso situado en otro nodo, también se utiliza el operador `!`

```
{ NombreProceso, NombreNodo } ! Mensaje
```

El proceso ha de estar registrado con un nombre

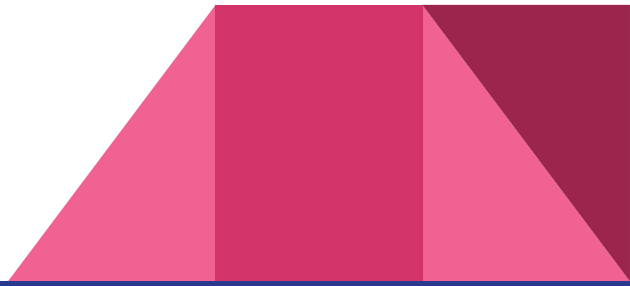
Envío de mensajes entre nodos

nodo1@PTO0520

```
1> P = gtm_server:start("Bohemian
Rhapsody").
-------- --------
…
2> register(gtm, P).
true
------A_ __A_____
```

nodo2@PTO0520

```
1> {gtm, nodo1@PTO0520} ! {self(), {guess, $A}}.
2> flush().
Shell got {hit,2,"------A_ __A_____"}
```



4. Interacción entre máquinas

Instrucciones

- Ejecuta el fichero `play_gtm.bat`
- Se abrirá un nodo con el nombre `nodoclient@PTO05XX`
- En la shell que aparezca, teclea:

```
gtm_server_enhaced:start("Tu nombre", "Nombre Pelicula")
```

- Se creará un proceso registrado con el nombre `gtm`.

¡Que empiece el juego!

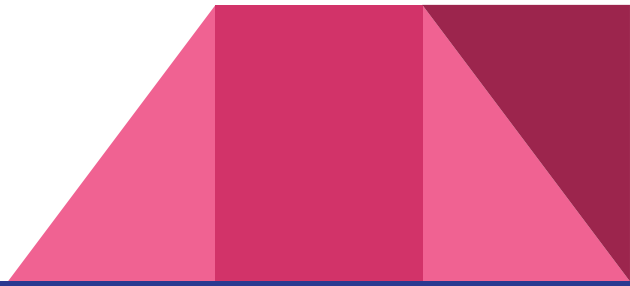
- Para obtener el título oculto de tu compañero, envíale un mensaje:

```
{gtm, nodoclient@PTO05XX} ! {self(), get_title}
```

- Para intentar descubrir letras:

```
{gtm, nodoclient@PTO05XX} ! {self(), {guess, $X}}
```

- Tras hacer una petición, no olvides obtener la respuesta recibida. Para ello utiliza `flush()`.



Bibliografía

Joe Armstrong
Programming Erlang. Software for a Concurrent World (2nd edition)
The Pragmatic Bookshelf, 2013

Francesco Cesarini, Simon Thompson
Erlang Programming. A Concurrent Approach to Software Development
O'Reilly, 2009

Fred Hébert
Learn you some Erlang for Great Good!
https://learnyousomeerlang.com/