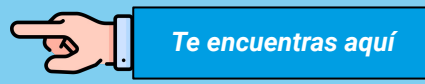
Kotlin para el desarrollo de aplicaciones

Características del Lenguaje Kotlin (Parte II)

{desafío}
latam_

Reconocer las principales características del lenguaje Kotlin para el desarrollo de aplicaciones móviles Android Nativo.

- Unidad 1: Kotlin para el desarrollo de aplicaciones.



- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.

{desafío} latam_

¿Qué aprenderás en esta sesión?

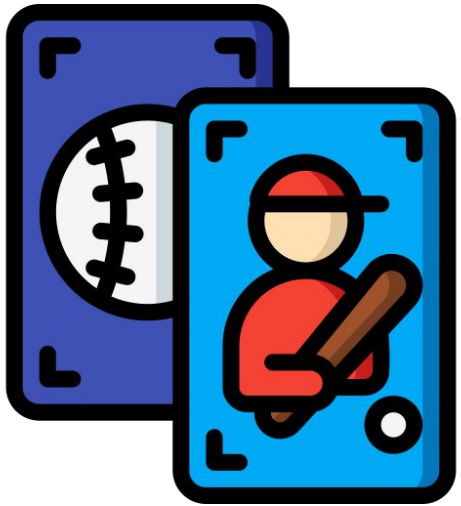
- *Utilizar la sintaxis del lenguaje Kotlin para la definición de variables, clases y funciones*

/* Estructuras de datos en Kotlin */

{desafío}
latam_

Estructuras de datos en Kotlin

Collections

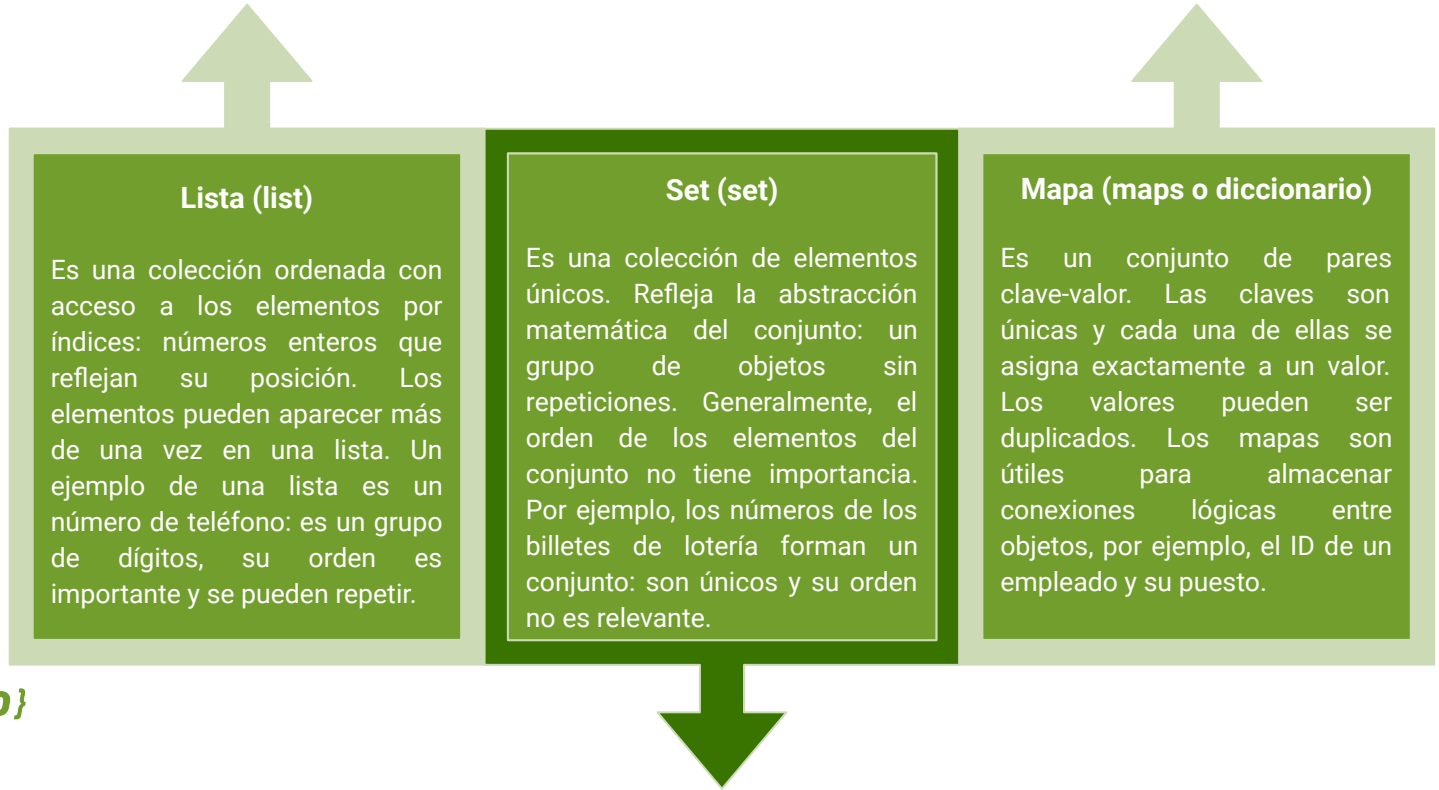


Una colección es un conjunto de objetos, normalmente, contiene objetos del mismo tipo (no obstante, este número también puede ser cero). Los objetos de una colección se denominan elementos o ítems.

Por ejemplo, todos los estudiantes de un departamento forman una colección que se puede utilizar para calcular su edad promedio.

Tipos de Colección

que son relevantes para Kotlin…



List y Set

List (`List<T>`)

Almacena elementos en un orden específico y proporciona acceso indexado a ellos. Los índices comienzan desde cero, y van a **`lastIndex`**, que es el (**`list.size - 1`**).

Ejemplo:

```
val numbers = listOf("uno", "dos", "tres",
"cuatro")
println("numero de items: ${numbers.size}")
resultado: número de ítems: 4
```

Set (`Set<T>`)

Almacena elementos únicos; su orden es generalmente indefinido. Los elementos nulos también son únicos: un conjunto puede contener solo un nulo. Dos Sets son iguales si tienen el mismo tamaño, y para cada elemento de un Set hay un elemento igual en el otro Set.

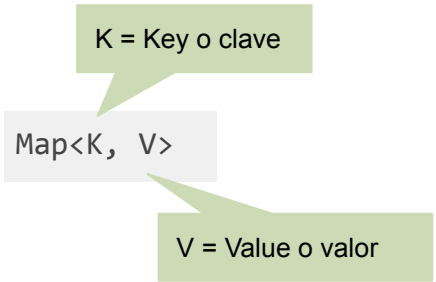
Ejemplo:

```
val numbers = setOf(1, 2, 3, 4)
println("número de ítems: ${numbers.size}")
if (numbers.contains(1)) println("1 está en el set")
```

Maps

Map (**Map<K, V>**) no es un heredero de la interfaz **Collection**, sin embargo, también es un tipo de colección de Kotlin. Un mapa almacena pares clave-valor (o entradas). Las claves son únicas, pero se pueden emparejar diferentes claves con valores iguales.

La interfaz de Map proporciona funciones específicas, como acceso a valor por clave, búsqueda de claves y valores, etc.



{desafío} latam_

Tip: No confundir con map (mapping transformation)

Algunos ejemplos de Maps

```
val map1 = mapOf(1 to 1, 2 to 2, 3 to 3)
println(map1)

resultado: {1=1, 2=2, 3=3}
```

```
val map2 = mapOf("1" to 1, "2" to 2, "3" to 3)
println(map2)

Resultado: {1=1, 2=2, 3=3}
```

¿Cuál es la diferencia entre los dos maps?
¿Por qué entregan el mismo resultado?

Parecen tener el mismo resultado en ambos maps, pero en realidad no lo tienen.

En **map1** se le asigna a cada posición dentro del map un valor entero (Int), en cambio, en el **map2** se le asignan valores String.

Mutable Maps

Colección **modificable** que contiene pares de objetos (claves y valores) y admite la recuperación eficiente del valor correspondiente a cada clave. Las claves de mapa son únicas; el mapa tiene solo un valor para cada clave.

Ejemplo:

```
val mutableMap1 = mutableMapOf(1 to "uno", 2 to "dos", 3
to "tres")
println(mutableMap1)

resultado: {1=uno, 2=dos, 3=tres}
```

Mutable Maps

- Podemos agregar un nuevo elemento al mutableMap especificando la posición (**4**) en la colección y el valor (“**cuatro**”)

```
mutableMap1[4] = "cuatro"
println(mutableMap1)
Resultado: {1=uno, 2=dos, 3=tres, 4=cuatro}
```

- Podemos modificar un elemento del mutableMap especificando la posición (**1**) en la colección y el valor (“**cuatro**”).

```
mutableMap1[1] = "cuatro"
println(mutableMap1)
Resultado: {1=cuatro, 2=dos, 3=tres}
```

Algunos ejemplos de Mutable Maps

```
val mutableMap1 = mutableMapOf(1 to "uno", 2 to "dos", 3 to "tres")
println(mutableMap1)

resultado: {1=uno, 2=dos, 3=tres}
```

- Podemos agregar un nuevo elemento al mutableMap especificando la posición (4) en la colección y el valor (“cuatro”):

```
mutableMap1[4] = "cuatro"
println(mutableMap1)
Resultado: {1=uno, 2=dos, 3=tres, 4=cuatro}
```

- Podemos modificar un elemento del mutableMap especificando la posición (**1**) en la colección y el valor (“**cuatro**”):

```
mutableMap1[1] = "cuatro"
println(mutableMap1)
Resultado: {1=cuatro, 2=dos, 3=tres}
```

Maps y Mutable Maps

Como se explica en la definición de Map (Map<K, V>), map es una colección de elementos definidos por clave (Key) y valor (Value). En los ejemplos anteriores, se asume que las claves y los valores son del mismo tipo, sin embargo, esto no es obligatorio.
En el siguiente ejemplo, se ve un mutableMap con datos de distinto tipo:

```
val mutableMap1 = mutableMapOf<Int, Any>(1 to "uno", 2 to "dos", 3 to "tres")
```

Esto significa que **`mutableMap1`** tiene claves tipo **Int** (entero) y valores tipo **Any** (cualquier tipo), por lo que ahora podríamos poder agregar otros tipos de datos a la colección:

```
mutableMap1[4] = 4 // K tipo Int, V tipo Int
mutableMap1[5] = true // K tipo Int, V tipo Boolean

println(mutableMap1)
{1=uno, 2=dos, 3=tres, 4=4, 5=true}
```

/* Map, filter, sorting */

Map

La transformación de mapeo crea una colección a partir de los resultados de una función en los elementos de otra colección.
La función de mapeo básica es **`map()`**. Aplica la función lambda dada a cada elemento subsiguiente y devuelve la lista de resultados lambda. El orden de los resultados es el mismo que el orden original de los elementos.
Para aplicar una transformación que además use el índice del elemento como argumento, use **`mapIndexed()`**

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { index, value -> "índice: $index - valor:
$value"})

resultado:
[3, 6, 9]
[índice: 0 - valor: 1, índice: 1 - valor: 2, índice: 2 - valor: 3]
```

Ejemplo de Map

En el siguiente ejemplo, y con el uso de la función **`map`**, se toma el set de datos y multiplica cada ítem por 3.

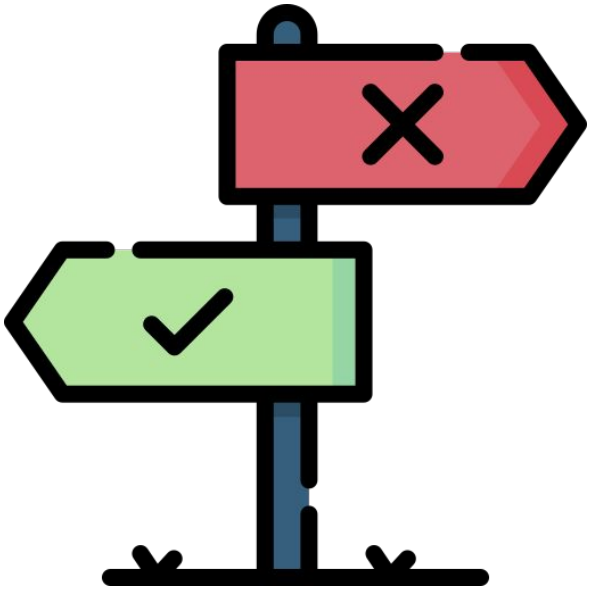
```
val numbers = setOf(1, 2, 3)

println(numbers.map { it * 3 })
resultado:
[3, 6, 9]
```

En el siguiente ejemplo se hace uso de la función **`mapIndexed`**, para imprimir por pantalla cada uno de los ítems del set **`numbers`**, junto a sus respectivas claves.

```
println(numbers.mapIndexed { index, value ->
    "índice: $index - valor: $value"
    }
)
resultado:
[índice: 0 - valor: 1, índice: 1 - valor: 2,
índice: 2 - valor: 3]
```

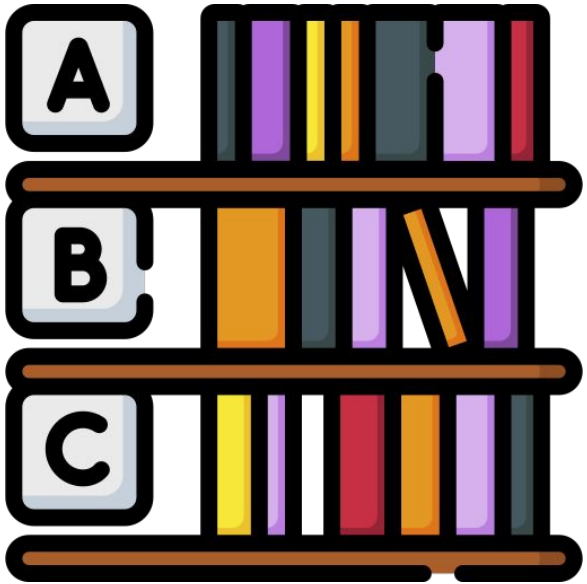
Filter



El filtrado es una de las tareas más populares en el procesamiento de colecciones. En Kotlin, las condiciones de filtrado se definen mediante predicados: funciones lambda que toman un elemento de colección y devuelven un valor booleano: verdadero significa que el elemento dado coincide con el predicado, falso significa lo contrario.

Filter

La biblioteca estándar contiene un grupo de funciones de extensión que le permiten filtrar colecciones en una sola llamada. Estas funciones dejan la colección original sin cambios, por lo que están disponibles tanto para colecciones mutables como de solo lectura.
Para operar el resultado del filtrado, debe asignarlo a una variable o encadenar las funciones después del filtrado.



Ejemplo de Filter

Ejemplo 1: Una lista (listOf) de Strings, luego esa lista se filtra con los strings que tenga más de 3 caracteres de largo y se guarda en **`longerThan3`**

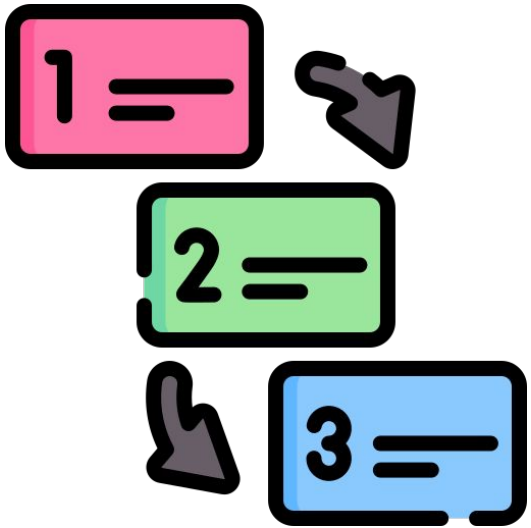
```
val numbers = listOf("uno", "dos", "tres",
"cuatro")
val longerThan3 = numbers.filter {
    it.length > 3
}
println(longerThan3)
resultado: [tres, cuatro]
```

Ejemplo 2: Un mapa (mapOf) Strings y enteros que se filtra con los valores que en su clave terminen con la letra “a” y que se valor sea mayor a 2 y luego se guarda en **`filteredMap`**

```
val numbersMap = mapOf("limon" to 1, "pera" to 2,
"manzana" to 3, "naranja" to 4)
val filteredMap = numbersMap.filter { (key, value)
->
    key.endsWith("a") && value > 2
}
println(filteredMap)

resultado: {manzana=3, naranja=4}
```

Sorting - Ordering



El orden de los elementos es un aspecto importante de ciertos tipos de colecciones. Por ejemplo, dos listas de los mismos elementos no son iguales si sus elementos están ordenados de manera diferente.

En Kotlin, el orden de los objetos se puede definir de varias formas.

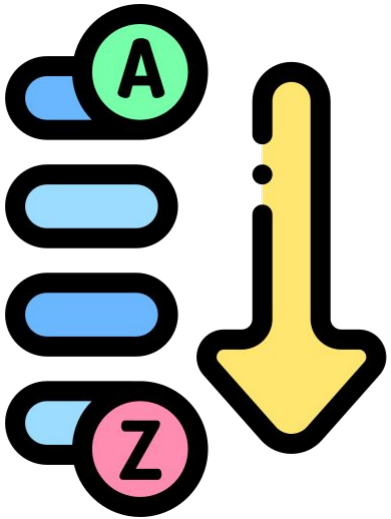
- Orden natural
- Orden Personalizado
- Orden Invertido o Reverso
- Orden Aleatorio

Ejemplo de Orden Natural

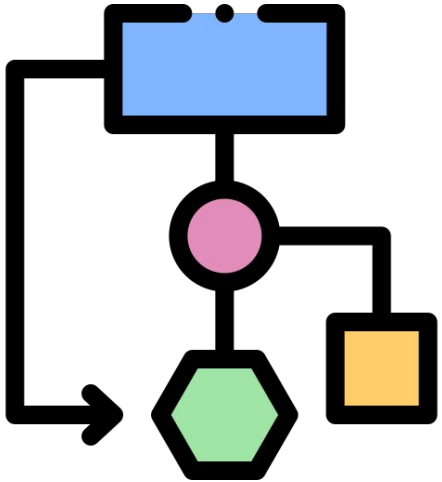
Las funciones básicas **`sorted()`** y **`sortedDescending()`** devuelven elementos de una colección ordenados en secuencia ascendente y descendente según su orden natural.

Estas funciones se aplican a colecciones de elementos comparables, por ejemplo:

```
val numbers = listOf(2,1,4,3)
println("orden ascendiente: ${numbers.sorted()}")
println("orden descendiente: ${numbers.sortedDescending()}")
orden ascendiente: [1, 2, 3, 4]
orden descendiente: [4, 3, 2, 1]
```



Ejemplo de Orden Personalizado



Para definir un orden personalizado para la clasificación de la colección, puede proporcionar su propio Comparador. Para hacer esto, llama a la función **`sortedWith()`** pasando tu Comparator.
Por ejemplo: val

```
numbers = listOf("uno", "dos", "tres", "cuatro")
println("ordenado por cantidad caracteres asc:
${numbers.sortedWith(compareBy { it.length })}")
ordenado por cantidad caracteres asc: [uno, dos, tres, cuatro]
```

Ejemplo de Orden Invertido

Devuelve una vista invertida de la misma instancia de colección, lo que significa que si la lista original cambia, entonces orden invertido también lo hará.

Por ejemplo:

```
val numbers = listOf("uno", "dos", "tres", "cuatro")
println(numbers.asReversed())
resultado: [cuatro, tres, dos, uno]
```

Ejemplo de Orden Aleatorio

Existe una función que crea una lista nueva y la entrega en orden aleatorio. Por ejemplo:

```
val numbers = listOf("uno", "dos", "tres", "cuatro")
println(numbers.shuffled())
resultado: [cuatro, dos, tres, uno]
```

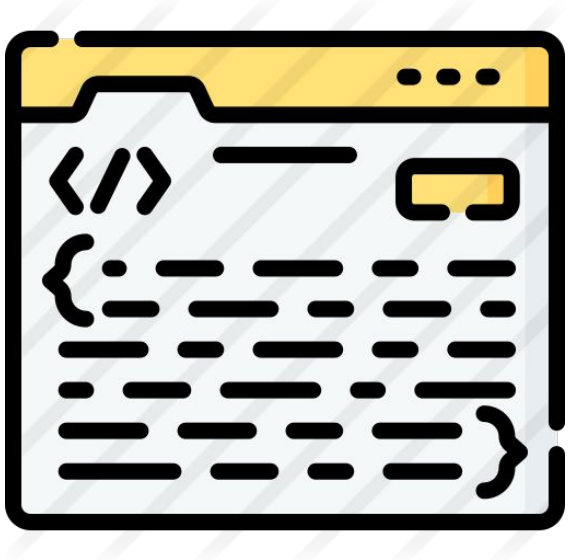
/* Kotlin standard functions y el scope */

Scope

- Scope es un concepto importante en programación.
- El Scope define dónde se puede acceder o hacer referencia a las variables.
- Si bien se puede acceder a algunas variables desde cualquier lugar dentro de un programa, es posible que otras variables solo estén disponibles en un contexto específico.

Standard Functions

- La biblioteca estándar de Kotlin contiene varias funciones cuyo único propósito es ejecutar un bloque de código dentro del contexto de un objeto.
- Cuando llama a una función de este tipo en un objeto con una expresión lambda proporcionada, forma un ámbito temporal.
- En este ámbito, puede acceder al objeto sin su nombre.
- Estas funciones se denominan funciones de alcance. Hay cinco de ellos: `let, run, with, apply y also`.



¿Cuándo utilizar let, run, with, apply o also?

	this	it	retorna
T.also{}	sin alterar	T	T
T.let{}	sin alterar	T	{body result}
T.apply{}	T	sin alterar	T
T.run{}, run{}	T	sin alterar	{body result}
with(T){}	T	sin alterar	{body result}

let y run

`let`

El objeto de contexto está disponible como argumento (`it`). El valor de retorno es el resultado lambda: `let` se puede usar para invocar una o más funciones en los resultados de las cadenas de llamadas.

`run`

El objeto de contexto está disponible como receptor (`this`). El valor de retorno es el resultado lambda; `run` es útil cuando su lambda contiene tanto la inicialización del objeto como el cálculo del valor de retorno.

Ejemplos let y run

Supongamos que tenemos la variable service y queremos cambiar el valor de port

```
val service = MultiportService("https://example.kotlinlang.org", 80)
```

con let:

```
val letResult = service.let {
    it.port = 8080
    it.query(it.prepareRequest() + " to
port ${it.port}")
}
```

con run:

```
val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

with, apply

`with`: Una función sin extensión: el objeto de contexto se pasa como argumento, pero dentro de la lambda, está disponible como receptor (**`this`**). El valor de retorno es el resultado lambda, ejemplo:

```
val numbers = mutableListOf("uno", "dos", "tres")
val firstAndLast = with(numbers) {
    "el primer ítem es: ${first()}," + " y el último
es: ${last()}"
}
println(firstAndLast)
resultado: el primer ítem es: uno, y el último es:
tres
```

`apply`: El objeto de contexto está disponible como receptor (**`this`**). El valor de retorno es el objeto mismo.

```
val user = User("Samuel").apply {
        age = 30
        city = "Bangkok"
}
println(user)
resultado: User(name=Samuel, age=30, city=Bangkok)
```

also

El objeto de contexto está disponible como argumento (it). El valor de retorno es el objeto mismo.

`also` es bueno para realizar algunas acciones que toman el objeto de contexto como argumento. Úselo también para acciones que necesitan una referencia al objeto en lugar de sus propiedades y funciones, o cuando no desea ensombrecer esta referencia desde un ámbito externo.

```
val numbers = mutableListOf("uno", "dos", "tres")
    numbers
        .also { println("lista de ítems antes de agregar uno nuevo: $it") }
        .add("cuatro")
println("lista completa: $numbers")

resultado:
lista de ítems antes de agregar uno nuevo: [uno, dos, tres]
lista completa: [uno, dos, tres, cuatro]
```

Intenta responder con tus palabras:

¿Cuándo puede ser útil utilizar map?

Intenta responder con tus palabras:

¿Cuál es la diferencia entre let y run ?

Intenta responder con tus palabras:

¿Cuándo se puede utilizar with, apply, also?

De todo lo aprendido el día de hoy ¿Qué me resultó más difícil y por qué?

{desafío} latam_

Próxima sesión...

- *Practicaremos mediante una **guía de ejercicios** todo lo aprendido en estas sesiones.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales