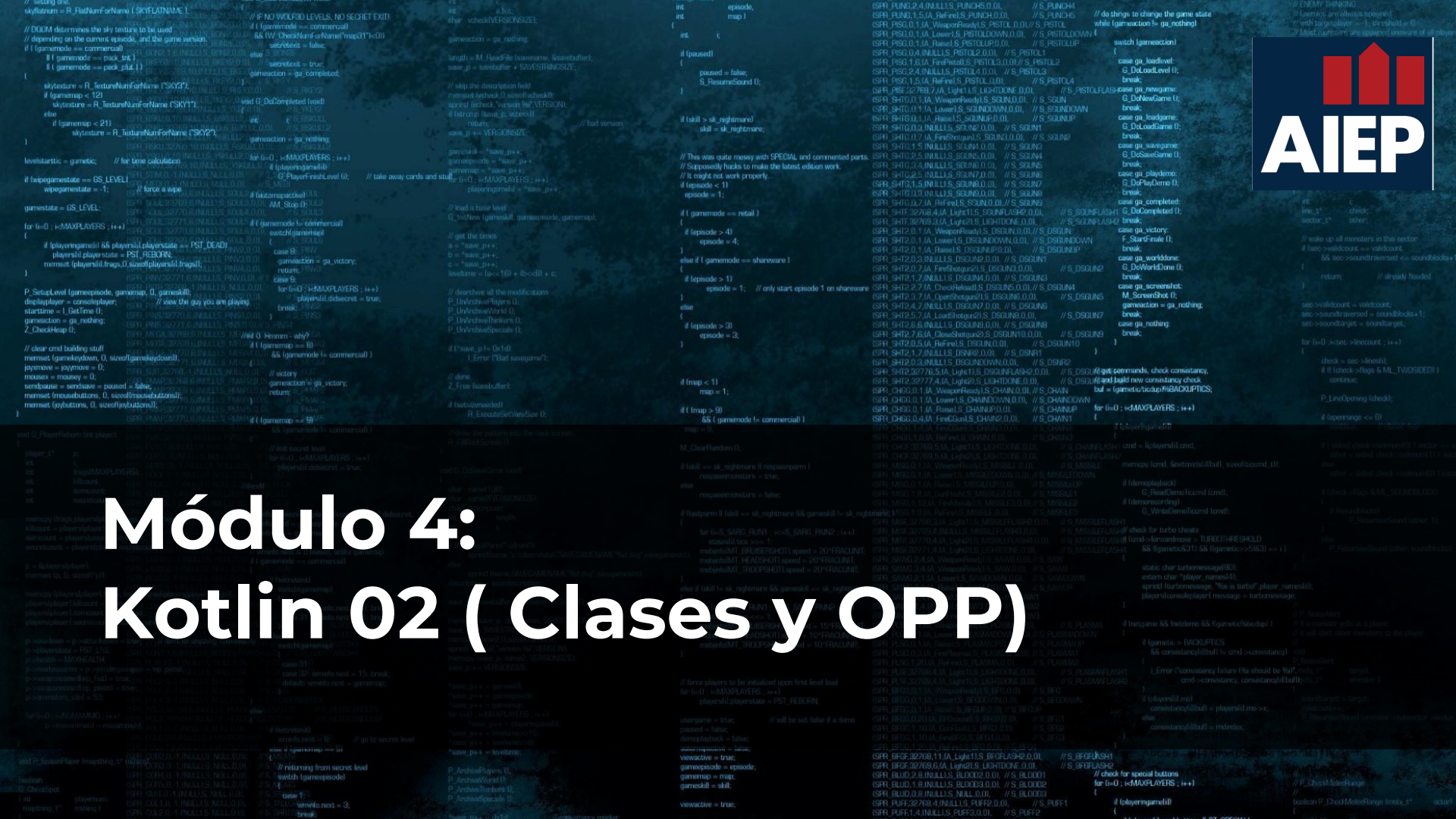

Módulo 4:
Kotlin 02 (Clases y OPP)

<title> Objetivos del día </title>

- Recall Día Anterior.
- Kotlin
  - Clases
  - Estructuras de control
  - Ciclos de iteración.
- Ejercicios Kotlin.

Clases en Kotlin

Clases

Al igual que en Java una clase es una plantilla que nos permite crear nuestros propios objetos.

Permitiéndonos definir las propiedades y características de ellas.

En kotlin las clases se definen con la palabra reservada **class**. Luego en () se definen los parámetros de su constructor primario).
Luego con llaves {} se define su cuerpo. Ambos son opcionales.

```
class Persona(text: String){
}

class OtraPersona
```

Ejemplos de Clases

1. Declara una clase llamada Customer sin ninguna propiedad o constructor. En este tipo de clases, igual Kotlin deja un constructor por defecto vacío.
2. Declaración de una clase con dos propiedades: un inmutable id y un mutable email ambas dentro de su constructor.
3. Crea una instancia de la clase constructor a través del constructor por defecto. Nota no existe la palabra reservada new en Kotlin.
4. Crea una instancia de la clase Contact usando un constructor con dos argumentos.
5. Accede e imprime la propiedad id.
6. Actualiza el valor de la propiedad email.

```
class Customer                                  // 1

class Contact(val id: Int, var email: String)
// 2

fun main() {

    val customer = Customer()                   // 3

    val contact = Contact(1,
"mary@gmail.com")  // 4

    println(contact.id)                         // 5
    contact.email = "jane@gmail.com"
// 6
}
```

Herencia de clases (open class)

1. En Kotlin las clases son final por defecto, esto quiere decir que no se puede utilizar directamente herencia. Para poder utilizar este comportamiento, debemos utilizar el modificador **open**.
2. Las funciones también son final por defecto, por lo tanto también debemos usar el modificador open para permitir que puedan ser sobreescritas.
3. Una clase que hereda de su superclase, para hacer este comportamiento usamos el símbolo **:** luego el nombre de la superclase y su constructor.
4. Para sobrescribir una función también se ocupa la palabra reservada **override.**

```
open class Dog {                // 1
    open fun sayHello() {       // 2
        println("wow wow!")
    }
}

class Yorkshire : Dog() {       // 3
    override fun sayHello() {   // 4
        println("wif wif!")
    }
}

fun main() {
    val dog: Dog = Yorkshire()
    dog.sayHello()
}
```

Herencia de clases, constructor con parámetros.

1. Si necesitamos la superclase necesita parámetros en su constructor, podemos pasarlo cuando creamos heredamos de ella.

```
open class Tiger(val origin: String) {
    fun sayHello() {
        println("A tiger from $origin says: grrhhh!")
    }
}

class SiberianTiger : Tiger("Siberia")
// 1

fun main() {
    val tiger: Tiger = SiberianTiger()
    tiger.sayHello()
}
```

Pasando argumentos a la superclase (Herencia)

1. En este ejemplo al declarar la clase Asiatic y heredar de Lion, no se indica val o var estos valores serán pasados desde la superclass Lion
2. Ejemplo de instanciación donde le pasamos un parámetro y el otro está en el constructor de la clase.

```
open class Lion(val name: String, val origin: String) {
    fun sayHello() {
        println("$name, the lion from $origin says: graoh!")
    }
}

class Asiatic(name: String) : Lion(name = name, origin = "India") // 1

fun main() {
    val lion: Lion = Asiatic("Rufo")
// 2
    lion.sayHello()
}
```

Data class (POJOS)

1. Un POJO hace referencia a una clase que contiene atributos y getters y setters.
2. En kotlin se definen con la palabra reservada **data** antes de class.

```
data class Persona(val nombre: String, val rut: String, val edad: Int)

fun main(){
   val persona = Persona("Juan", "1-9", 39)
   println("Nombre Persona = ${persona.nombre}")
   println("Rut Persona = ${persona.rut}")
   println("Edad Persona = ${persona.edad}")
}
```

Data class (Ejemplos)

1. Define una data class.
2. Por defecto está generado el método toString lo que permite imprimir la clase sin problemas.
3. El autogenerado equals considera dos instancias equivalentes si todas sus propiedades son iguales.
4. Dos instancias data class tienen equal hashChode()
5. El método copy permite crear una nueva instancia.
6. Con copy permite pasar datos en el orden del constructor.
7. usando copy con parámetros nombrados, cambiar el valor no importando el orden.

```
data class User(val name: String, val id: Int)           // 1

fun main() {
    val user = User("Alex", 1)
    println(user)                                        // 2

    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    println("user == secondUser: ${user == secondUser}")   //
3
    println("user == thirdUser: ${user == thirdUser}")

    println(user.hashCode())                             // 4
    println(thirdUser.hashCode())

    // copy() function
    println(user.copy())                                 // 5
    println(user.copy("Max"))                            // 6
    println(user.copy(id = 2))                           // 7

}
```

Puntos claves Kotlin

- Tipos de datos:
  - Enteros, coma flotante
  - String y char
  - Unit
- Funciones:
  - Función main()
  - Funciones con parámetros
  - Funcion con parametros nombrados
  - Funciones Unit
  - Inferencia en Funciones.
  - open funcions
- Funciones de extensión.
  - Añadir funciones de extensión.
- Clases y Herencia
  - Constructor por defecto
  - Open class

Pendiente:

- Herencia con parámetros
- Data class
- Constructores secundarios o múltiples.

Null Safety

Para eliminar los errores NullPointerExeption, las variables en Kotlin no permiten ser asignadas como nulas. Para poder asignarlo debemos utilizar el operador **?** al final del tipo de la variable.

1. Declara una variable
2. Intenta asignar el valor null (Error de compilación)
3. Asigna null a una variable con el operador ?
4. Asigna null a la variable.
5. Cuando se intenta inferir el tipo, el sistema siempre infiere otro tipo excepto null.
6. Cuando se le asigna null a una variable que fue inferida, da un error.
7. Se asigna una función que puede retornar null
8. No podemos asignar un valor null como parámetro si la función lo requiere.

```
fun main() {

var neverNull: String = "This can't be null"            // 1

neverNull = null                                        // 2

var nullable: String? = "You can keep a null here"      // 3

nullable = null                                         // 4

var inferredNonNull = "The compiler assumes non-null"   //
5

inferredNonNull = null                                  // 6

fun strLength(notNull: String): Int {                   // 7
   return notNull.length
}

strLength(neverNull)                                    // 8
strLength(nullable)                                     // 9

}
```

Trabajando con Null

A veces, los programas necesitan trabajar con valores nulos, como cuando interactúan con código Java externo o cuando representan un estado verdaderamente ausente. Kotlin proporciona seguimiento nulo para lidiar con elegancia con tales situaciones.

1. Una función que recibe un atributo que puede ser null y retorna su descripción.
2. Si el valor entregado no es null y no está vacío, retorna el largo del string.
3. En caso contrario, retorna un string indicando que es null o vacio.

```
fun describeString(maybeString: String?):
String {              // 1
  if (maybeString != null && maybeString.length
> 0) {        // 2
      return "String of length
${maybeString.length}"
  } else {
      return "Empty or null string"
// 3
  }
}
```

Generics y clases de este tipo.

Los genéricos son un mecanismo que se ha convertido en estándar en los lenguajes modernos. Las clases y funciones genéricas aumentan la reutilización del código al encapsular la lógica común que es independiente de un tipo genérico particular. Ejemplo : como la lógica dentro de **List <T>** es independiente de lo que es **T**.

1. Define una clase genérica MutableStack <E> donde E se denomina parámetro de tipo genérico. En use-site, se asigna a un tipo específico como Int al declarar un MutableStack <Int>.
2. Dentro de la clase genérica, **E** se puede utilizar como parámetro como cualquier otro tipo.
3. También se puede utilizar E como tipo de retorno.

```
class MutableStack<E>(vararg items: E) {
// 1

  private val elements = items.toMutableList()

  fun push(element: E) =
elements.add(element)        // 2

  fun peek(): E = elements.last()                  //
3

  fun pop(): E =
elements.removeAt(elements.size - 1)

  fun isEmpty() = elements.isEmpty()

  fun size() = elements.size

  override fun toString() =
"MutableStack(${elements.joinToString()})"
}
```

Funciones genéricas.

También se pueden crear funciones genéricas, si su lógica es independiente de su tipo específico.

En el ejemplo se utiliza una función para crear mutable stack de la clase anterior.

El compilador puede inferir el tipo genérico de los parámetros de forma que no tengamos que escribirlo o declararlo.

```
fun <E> mutableStackOf(vararg elements: E) =
MutableStack(*elements)

fun main() {
    val stack = mutableStackOf(0.62, 3.14, 2.7)
    println(stack)
}
```

Control Flow

Flujos de control

When (declaración)

Para reemplazar el ya conocido y utilizado switch, en kotlin tenemos **when** el cual nos proporciona mayor flexibilidad de construcción. Puede ser usado como una declaración o expresión.

1. Es una declaración **when**
2. verifica que “obj” sea equivalente a “One”.
3. Verifica que “obj” sea equivalente a “Hello”
4. Verifica el tipo de dato.
5. Verifica el tipo de dato de forma inversa.
6. Declaración por defecto. (puede ser omitida).

Todas las posibles alternativas son verificadas en forma secuencial hasta que se cumpla una, solo se ejecutara esa alternativa.

```
fun cases(obj: Any) {
    when (obj) {                                    // 1
        1 -> println("One")                         // 2
        "Hello" -> println("Greeting")              // 3
        is Long -> println("Long")                  // 4
        !is String -> println("Not a string")       // 5
        else -> println("Unknown")                  // 6
    }
}

class MyClass

fun main() {
    cases("Hello")
    cases(1)
    cases(0L)
    cases(MyClass())
    cases("hello")
}
```

When (Expression)

1. Esta es una expresión de **when**
2. Asigna el value “one” si “obj” es equivalente a 1.
3. Asigna el value 1 si “obj” es equivalente a “hello”.
4. Asigna el value a false is “obj” es una instancia de Long.
5. Asigna el value a 42 si ninguna de las condiciones anteriores fueron satisfactorias.

A diferencia del when de declaración, la opción default es requerida cuando se utiliza como expresión, excepto cuando se cubran todas las alternativas.

```
fun whenAssign(obj: Any): Any {
    val result = when (obj) {                    // 1
        1 -> "one"                               // 2
        "Hello" -> 1                             // 3
        is Long -> false                         // 4
        else -> 42                               // 5
    }
    return result
}

class MyClass


fun main() {
    println(whenAssign("Hello"))
    println(whenAssign(3.4))
    println(whenAssign(1))
    println(whenAssign(MyClass()))
}
```

Loops (For)

Kotlin soporta casi todos los loops conocidos: for , while, do-While

for

Funciona al igual que en todos los otros lenguajes.

1. Itera por cada uno de los elementos de la lista.

```
fun main() {
   val cakes = listOf("carrot", "cheese",
"chocolate")

   for (cake in cakes) {                          // 1
      println("Yummy, it's a $cake cake!")
   }

}
```

Loops (While, do While)

Los ciclos iteradores while y do while tambien funcionan al igual que en otros lenguajes.

1. Ejecuta el bloque while cuando la condición es verdadera.
2. Ejecuta el bloque primero y luego chequea la condición.

```
fun eatACake() = println("Eat a Cake")
fun bakeACake() = println("Bake a Cake")

fun main(args: Array<String>) {
    var cakesEaten = 0
    var cakesBaked = 0

    while (cakesEaten < 5) {                    // 1
        eatACake()
        cakesEaten ++
    }

    do {                                        // 2
        bakeACake()
        cakesBaked++
    } while (cakesBaked < cakesEaten)

}
```

Ranges (Rangos)

Existe un conjunto de Herramientas para definir rangos en Kotlin.

1. Itera en un rango que parte desde el 0 hasta el 3 inclusive.
2. Itera en un rango que tiene un incremento en elementos consecutivos.
3. Itera sobre un rango en orden inverso.

```
fun main(args: Array<String>) {
    for(i in 0..3) {              // 1
        print(i)
    }
    print(" ")

    for(i in 2..8 step 2) {       // 2
        print(i)
    }
    print(" ")

    for (i in 3 downTo 0) {       // 3
        print(i)
    }
    print(" ")
}
```

Ranges (Rangos)

También se pueden utilizar char para los Rangos.

1. Itera sobre un rango de char en orden alfabético.
2. char ranges también soporta step y downTo.

También podemos usar Rangos en declaraciones **if.**

1. Verifica si el valor está en el rango.
2. !in es lo contrario a in.

```
...
for (c in 'a'..'d') {        // 1
    print(c)
}
print(" ")

for (c in 'z' downTo 's' step 2) { // 2
    print(c)
}
print(" ")
....

val x = 2
if (x in 1..5) {             // 1
    print("x is in range from 1 to 5")
}
println()

if (x !in 6..10) {           // 2
    print("x is not in range from 6 to 10")
}
```

Equality Checks

Kotlin usa == para comparación estructural y === para comparación referencial.

1. Retorna true porque llama a authors.equals(writers) y no le importa el orden de los elementos.
2. Retorna false porque los listados son distintas referencias.

```
val authors = setOf("Shakespeare",
"Hemingway", "Twain")
val writers = setOf("Twain", "Shakespeare",
"Hemingway")

println(authors == writers)   // 1
println(authors === writers)  // 2
```

Expresiones condicionales

No existen operadores ternarios (if ternarios)
ej: condition ? then : else en Kotlin. Acá debemos utilizar if , un ejemplo sería:

1. if expresión solo retorna un valor. (99).

```
99
```

```
fun max(a: Int, b: Int) = if (a > b) a else b        //
1

println(max(99, -42))
```

Clases especiales

Constructores de las clases

Como hemos revisado el constructor primario en una clase Kotlin va entre paréntesis al comienzo de esta clase.

Como no podemos poner código en estos constructores, podemos inicializar las variables al interior de un bloque init {} este se ejecutara en el orden de aparición en la clase.

Los constructores secundarios deben llevar la palabra reservada constructor.

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }


}
```

Enum Classes

Las clases Enum al igual que en Java se utilizan para representar un número finito de distintos valores, como direcciones, estados ,modos etc.

1. Define una clase Enum con tres instancias u objetos enum. El número de instancias siempre es finita son todas distintas.

1. Accede a una instancia Enum por el nombre de la clase.

1. En el ejemplo, se utiliza la estructura de control when para inferir el tipo de expresión buscada.

```
enum class State {
    IDLE, RUNNING, FINISHED
// 1
}

fun main() {
    val state = State.RUNNING                    //
2
    val message = when (state) {                 //
3
        State.IDLE -> "It's idle"
        State.RUNNING -> "It's running"
        State.FINISHED -> "It's finished"
    }
    println(message)
}
```

Enum Classes

Las clases Enum también pueden contener propiedades y funciones como otras clases, Pero debe ir separadas las **;**

1. Define una clase enum con una propiedad y función.
2. Cada instancia debe pasar un argumento definido en el constructor.
3. Los members o miembros (Función en este caso) deben separarse de los atributos por **;**
4. El default de toString devolverá el nombre de la instancia, en el ejemplo “RED”.
5. LLamada a una función desde una instancia de un enum.
6. Llamada a una función por el nombre de la clase enum.

```
enum class Color(val rgb: Int) {                    //
1
    RED(0xFF0000),                                  // 2
    GREEN(0x00FF00),
    BLUE(0x0000FF),
    YELLOW(0xFFFF00);

    fun containsRed() = (this.rgb and 0xFF0000
!= 0)  // 3
}

fun main() {
    val red = Color.RED
    println(red)                                    // 4
    println(red.rgb)
    println(red.containsRed())                      // 5
    println(Color.BLUE.containsRed())               // 6
}
```

Sealed Class

Las clases Sealed nos permiten restringir el uso de la herencia. Una vez declarada la clase sealed, No podrá utilizarse como subclase desde fuera de este archivo donde fue declarado como sealed.

Las clases selladas se utilizan para representar jerarquías de clases restringidas, cuando un valor puede tener uno de los tipos de un conjunto limitado, pero no puede tener ningún otro tipo. Son, en cierto sentido, una extensión de las clases de **enum**: el conjunto de valores para un tipo de enum también está restringido, pero cada constante de enumeración existe solo como una única instancia, mientras que una subclase de una clase sellada puede tener múltiples instancias que pueden contener estado.
Para declarar una clase sellada, coloca el modificador sellado antes del nombre de la clase. Una clase sellada puede tener subclases, pero todas deben declararse en el mismo archivo que la propia clase sellada.

```
sealed class Mammal(val name: String)   // 1

class Cat(val catName: String) : Mammal(catName)  // 2
class Human(val humanName: String, val job: String) :
Mammal(humanName)

fun greetMammal(mammal: Mammal): String {
    when (mammal) {                                    // 3
        is Human -> return "Hello ${mammal.name}; You're
working as a ${mammal.job}"     // 4
        is Cat -> return "Hello ${mammal.name}"        // 5
    }                                                  // 6
}

fun main() {
    println(greetMammal(Cat("Snowy")))
    println(greetMammal(Human("jon", "singer")))
}
```

Ejercicios

Ejercicio 01:

1. Crear archivo “Main.kt” con la función main.
2. Imprimir “Mi nombre es: {Nombre del alumno}".
3. Declarar 3 variables numéricas con los valores 10, 20 y 30.
4. Imprimir la suma de las 3 variables.
5. Declarar 1 variable String y otra variable Char
6. Asignarle el valor “Ark Reigen” a la variable String y “A” a la variable Char .
7. Imprimir la cantidad de caracteres que tiene la variable String y modificar la variable char a otro valor distinto.
8. Declarar una variable de tipo Float de forma explícita.
9. Asignar un valor a la variable Float.
10. Imprimir los valores máximos que pueden almacenar las variables Byte y Short.

- Imprimir los valores mínimos que pueden almacenar las variables Int y Long.
- Declarar una variable Boolean con true o false e imprimir su valor.
- Crear una función llamada “imprimiendoParametros” que recibe 2 parámetros String y imprime el total de caracteres de los dos parámetros.
- Crear una función llamada “obtieneIVA” que devuelve el valor del IVA (19%).
- Desde la función main llamar a estas dos funciones de forma correcta.