Módulo 4:
Kotlin : Colecciones de datos.
Kotlin funcional.

<title> Objetivos del día </title>

- Recall Día Anterior.
  - Estructuras de control (when, if else, for, while, do while)
  - Clases especiales (sealed, enum, data class, generic Type)
  - Ejercicios (ejercicios finales).
- Kotlin
  - Mas tipos de datos Pairs y Triples
  - Array y Listas
  - MutableList
  - Maps and Sets
  - Sorted
- Ejercicios

Antes de continuar...

Ranges (Until)

Para excluir un elemento de un rango de iteración, se utiliza **until** este va a dejar fuera el elemento indicado.

No existe una palabra reservada para los elementos de inicio, pero como tenemos que indicarlo lo manejamos manual.

```
for (i in 0 until 10) {      // i in [0, 10), 10 is excluded
    print(i)
}
```

Tipos de datos Pair y Triple

Existe un tipo de datos en Kotlin llamado Pairs and Triples. Servirá para algunos datos que vienen en pares por ejemplo las coordenadas 2D. También las coordenadas 3D podrían ser representadas con un Triple.

```
fun main() {
    val coordinates: Pair<Int, Int> = Pair(2, 3)
    val coordinatesInferred = Pair(2, 3)
    val coordinatesWithTo = 2 to 3
    val coordinatesDoubles = Pair(2.1, 3.5) //
Inferred to be of type Pair<Double, Double>
    val coordinatesMixed = Pair(2.1, 3) //
Inferred to be of type Pair<Double, Int>
    val x1 = coordinates.first
    val y1 = coordinates.second
    val (x, y) = coordinates // x and y both
inferred to be of type Int
}
```

Tipos de datos Triple

Existe un tipo de datos en Kotlin llamado Pairs and Triples. Servirá para algunos datos que vienen en pares por ejemplo las coordenadas 2D. También las coordenadas 3D podrían ser representadas con un Triple.

```
fun main() {
    val coordinates3D = Triple(2, 3, 1)
    //val (x3, y3, z3) = coordinates3D
    val x3 = coordinates3D.first
    val y3 = coordinates3D.second
    val z3 = coordinates3D.third
}
```

Tipos de datos Any

El tipo Any se puede considerar como la madre de todos los demás tipos (excepto los tipos que aceptan valores NULL). Todos los tipos en Kotlin, ya sean Int o String, también se consideran Any. Es similar al tipo de objeto en Java, que es la raíz de todos los tipos excepto los primitivos.
Por ejemplo, es perfectamente válido para Kotlin declarar un literal Int y un literal String como Any así:

```
fun main() {
    val anyNumber: Any = 42
    val anyString: Any = "42"
}
```

Operador Elvis

Existe otra forma práctica de obtener un valor de un nullable. Se usa cuando deseamos obtener un valor de un tipo de dato nullable sin importar qué, y en el caso que sea nulo, se usará un valor predeterminado.

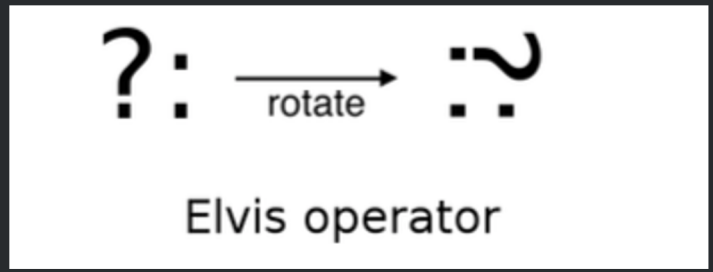
Así es como funciona: ->

El operador utilizado en la segunda línea **?:** Se conoce como el operador Elvis, ya que se parece a la estrella de rock cuando se gira 90 grados en el sentido de las agujas del reloj. El uso del operador Elvis significa que **mustHaveResult** será igual al valor dentro de nullableInt, o 0 si nullableInt contiene nulo. En este ejemplo, se infiere que mustHaveResult es de tipo Int y contiene el valor Int concreto de 10.

```
fun main() {

    var nullableInt: Int? = 10
    var mustHaveResult = nullableInt ?: 0

}
```



Operador !!

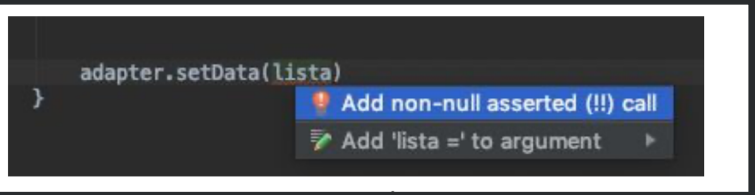
Como podemos ver, en este caso donde hay una serie de relaciones entre clases y parámetros, tendríamos que usar el operador (?) en cada nivel a medida que accedemos a la información. Ahora supongamos que en vez de usar (?) usamos otro operador.

El operador (!!) asegura que la variable nunca va a ser null, o sea le decimos al compilador, que nosotros estamos cien por ciento seguros que esta variable no será null, en caso de que lo sea, obtendremos un nullpointerException y esto provocará que la app falle mientras se ejecuta, por lo que hay que usarlo de manera muy responsable.

```
fun main() {
    var perro = Perro()
    perro.raza?.nombre
    perro.raza!!.nombre
}

class Raza {
    var nombre: String? = null
}

class Perro {
    var raza: Raza? = null
}
```



Funcional

Kotlin y la programación funcional.

Kotlin es un lenguaje que vienen con características del paradigma de programación funcional incorporados. Pero ¿qué es este paradigma?

La programación funcional se basa en el uso de funciones, que idealmente no tienen efectos secundarios. Un efecto secundario es cualquier cambio en el estado del sistema.
Un ejemplo simple de un efecto secundario es imprimir algo en la pantalla. Otro es cambiar el valor de la propiedad de un objeto. Los efectos secundarios suelen ser rampantes en la programación orientada a objetos, ya que las instancias de clase se envían mensajes entre sí y cambian su estado interno.

Otra característica clave de la programación funcional hace que las funciones sean ciudadanos de primera clase del lenguaje, La mayoría de los lenguajes de programación funcionales también se basan en un concepto llamado transparencia referencial. Este término significa efectivamente que dada la misma entrada, una función siempre devolverá la misma salida.

A diferencia de versiones anteriores a Java 8, Kolin permite utilizar ambos paradigmas en nuestros programas o aplicaciones.

Funciones de alto orden

Una higher-order function, es una función que puede tomar otra función como parámetro o tipo de retorno.

Como ya mencione, en kotlin las funciones son de primera clase, lo que significa que pueden ser guardadas en variables y estructuras de datos, pasadas como argumentos y retornadas desde otras funciones.

1. Declara una función de orden superior. Toma dos parámetros enteros, **x** e **y**. Además, toma otra operación de función como parámetro. Los parámetros de operación y el tipo de retorno también se definen en la declaración.
2. La función de orden superior devuelve el resultado de la invocación de la operación con los argumentos proporcionados.
3. Declara una función que coincide con la firma de la operación.
4. Invoca la función de orden superior pasando dos valores enteros y el argumento de la función **::** suma. **::** es la notación que hace referencia a una función por su nombre en Kotlin.
5. Invoca la función de orden superior pasando un lambda como argumento de función. Parece más claro, ¿no?

```
fun calculate(x: Int, y: Int, operation: (Int, Int) ->
Int): Int {  // 1
  return operation(x, y)
// 2
}

fun sum(x: Int, y: Int) = x + y
// 3

fun main() {
  val sumResult = calculate(4, 5, ::sum)
// 4
  val mulResult = calculate(4, 5) { a, b -> a * b
}             // 5
  println("sumResult $sumResult, mulResult
$mulResult")
}
```

Retornando un función

1. Declara una función de orden superior que devuelve una función. (Int) -> Int representa los parámetros y el tipo de retorno de la función cuadrada.
2. Declara una función que coincide con la firma.
3. Invoca la operación para obtener el resultado asignado a una variable. Aquí func se convierte en cuadrado que es devuelto por operación.
4. Invoca func. La función cuadrada se ejecuta realmente.

```
fun operation(): (Int) -> Int {                    // 1
    return ::square
}

fun square(x: Int) = x * x                         // 2

fun main() {
    val func = operation()                         // 3
    println(func(2))                               // 4
}
```

Funciones Lambda y Funciones anónimas

Las expresiones lambda y las funciones anónimas son 'literales de función', es decir, funciones que no se declaran, pero que se pasan inmediatamente como una expresión.

Las funciones lambda ("lambdas") son una forma sencilla de crear funciones ad-hoc. Las lambdas se pueden denotar de manera muy concisa en muchos casos gracias a la inferencia de tipos y la variable it implícita.

En palabras simples es una forma específica de hacer una función y definir lo que devolverá.

```
fun main() {
// All examples create a function object that performs upper-casing.
// So it's a function from String to String

  val upperCase1: (String) -> String = { str: String -> str.toUpperCase()
} // 1

  val upperCase2: (String) -> String = { str -> str.toUpperCase() }
// 2

  val upperCase3 = { str: String -> str.toUpperCase() }                  // 3

// val upperCase4 = { str -> str.toUpperCase() }                          // 4

  val upperCase5: (String) -> String = { it.toUpperCase() }              //
5

  val upperCase6: (String) -> String = String::toUpperCase               //
6

  println(upperCase1("hello"))
  println(upperCase2("hello"))
  println(upperCase3("hello"))
  println(upperCase5("hello"))
  println(upperCase6("hello"))
}
```

Collections

List, maps, sets (Collections)

La biblioteca estándar de Kotlin tiene un conjunto de herramientas para manejar y procesar colecciones o collections. Una collection es un grupo de items que comparten una relación con el problema a solucionar.

El concepto es ampliamente usado por la mayoría de los lenguajes de programación y en general se implementan en 3 tipos relevantes:

- Lista: es una colección ordenada en la que se accede a los elementos por su índice y donde el mismo elemento puede estar contenido más de una vez.
- Set: es una colección de elementos únicos donde generalmente el orden no tiene importancia.
- Mapa (o diccionario): es un grupo de pares clave-valor donde la clave (key) es única y se relaciona con solo un valor (value). Los valores pueden estar duplicados.

Read only o Mutable

La biblioteca estándar de Kotlin entrega implementaciones para las colecciones básicas como listas, sets y mapas. Además un par de interfaces representan el tipo de la colección que puede ser:

La interfaz de sólo lectura (read-only) que modela las operaciones de acceso a los elementos en una colección.

La interfaz Mutable que extiende el comportamiento de la interfaz de sólo lectura y agrega las operaciones de escritura: agregar, remover y actualizar sus elementos.

Una mutable collection no requiere de ser del tipo var : las operaciones de escritura modifican el objeto pero no su referencia.

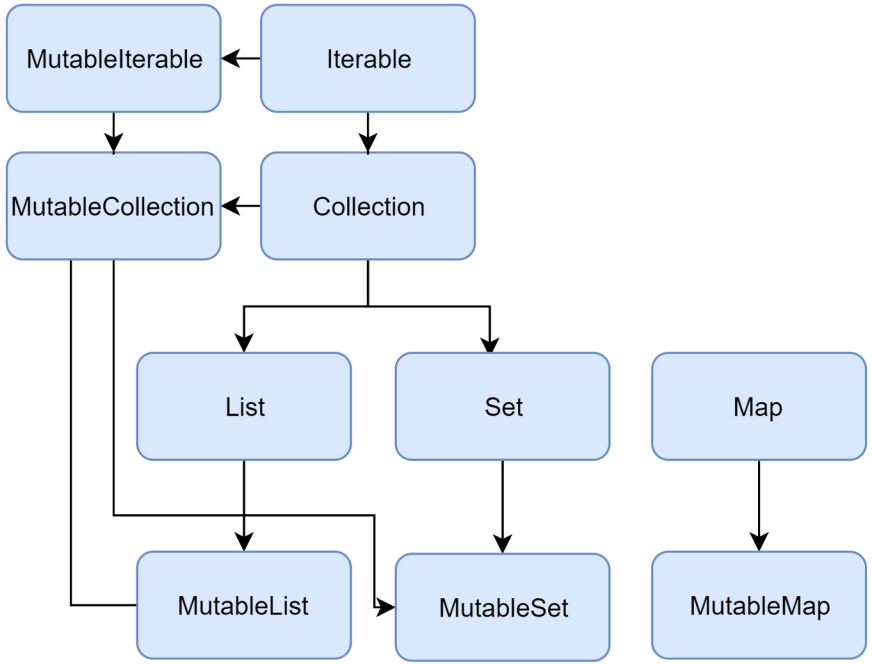
Collections

La interfaz Iterable<T> define las operaciones de iteración sobre los elementos.

La interfaz hereda de Iterable y es la raíz de la jerarquía de collection y representa el comportamiento de una colección de solo lectura.

Dado lo anterior, Collection puede usarse como parámetro de una función para distintos tipos de colecciones, por ejemplo List o Set.

Por otra parte, la interfaz MutableCollection<T> se comporta de la misma forma que Collection y agrega las operaciones de escritura como add o remove .



List

Una lista es una colección ordenada de elementos. En Kotlin, las listas pueden ser mutables (MutableList) o de solo lectura (List). Para la creación de listas, use las funciones de biblioteca estándar listOf () para listas de solo lectura y mutableListOf () para listas mutables.

1. Crea una lista MutableList.
2. Crea una lista de solo lectura de la lista.
3. Añade elementos a la lista Mutable.
4. Una función que retorna una lista inmutable.
5. Actualiza una lista mutable. Todas las otras listas de solo lectura son actualizadas, ya que hacen referencia al mismo objeto.
6. Obtiene el tamaño de la lista.
7. Itera sobre la lista e imprime el contenido.
8. Intenta escribir en una lista de solo lectura y da un error.

```
val systemUsers: MutableList<Int> = mutableListOf(1, 2, 3)
// 1
val sudoers: List<Int> = systemUsers                    // 2

fun addSudoer(newUser: Int) {                           // 3
    systemUsers.add(newUser)
}

fun getSysSudoers(): List<Int> {                        // 4
    return sudoers
}

fun main() {
    addSudoer(4)                                        // 5
    println("Tot sudoers: ${getSysSudoers().size}")     // 6
    getSysSudoers().forEach {                           // 7
        i -> println("Some useful info on user $i")
    }
    // getSysSudoers().add(5) <- Error!                 // 8
}
```

Set

Un Set es una colección desordenada que no admite duplicados. Para crear Sets, hay funciones setOf () y mutableSetOf (). Se puede obtener una vista de solo lectura de un conjunto mutable Casteando a Set.

1. Crea un set y le asigna elementos.
2. Retorna un boolean si el elemento fue actualizado o añadido.
3. Retorna un string, basado en el parámetro de input.
4. Imprime un mensaje de éxito si el elemento es añadido al set.
5. Imprime un mensaje de falla si no es posible añadir el elemento por si esta duplicado.

```
val openIssues: MutableSet<String> =
mutableSetOf("uniqueDescr1", "uniqueDescr2",
"uniqueDescr3") // 1

fun addIssue(uniqueDesc: String): Boolean {
   return openIssues.add(uniqueDesc)                    //
2
}

fun getStatusLog(isAdded: Boolean): String {
   return if (isAdded) "registered correctly." else
"marked as duplicate and rejected."          // 3
}

fun main() {
   val aNewIssue: String = "uniqueDescr4"
   val anIssueAlreadyIn: String = "uniqueDescr2"

   println("Issue $aNewIssue
${getStatusLog(addIssue(aNewIssue))}")
// 4
   println("Issue $anIssueAlreadyIn
${getStatusLog(addIssue(anIssueAlreadyIn))}")
// 5
}
```

Map

Un mapa es una colección de pares clave / valor, donde cada clave es única y se usa para recuperar el valor correspondiente. Para crear mapas, hay funciones mapOf () y mutableMapOf (). Se puede obtener una vista de solo lectura de un mapa mutable casteando a Map.

1. Crea un Mutable map.
2. Crea un Map de solo lectura.
3. Verifica si la llave existe en el mapa.
4. Lee el valor correspondiente y lo incrementa con un valor constante.
5. Itera el mapa de solo lectura e imprime el key/ value.
6. Lee los puntos antes del update.
7. Update a una cuenta existente dos veces
8. Trata de actualizar una cuenta no existente.
9. Lee los puntos de balance después del update.

```
const val POINTS_X_PASS: Int = 15
val EZPassAccounts: MutableMap<Int, Int> = mutableMapOf(1 to 100,
2 to 100, 3 to 100)   // 1
val EZPassReport: Map<Int, Int> = EZPassAccounts                    // 2

fun updatePointsCredit(accountId: Int) {
   if (EZPassAccounts.containsKey(accountId)) {                     // 3
      println("Updating $accountId...")
      EZPassAccounts[accountId] =
EZPassAccounts.getValue(accountId) + POINTS_X_PASS  // 4
   } else {
      println("Error: Trying to update a non-existing account (id:
$accountId)")
   }
}

fun accountsReport() {
   println("EZ-Pass report:")
   EZPassReport.forEach {                                           // 5
         k, v -> println("ID $k: credit $v")
   }
}

fun main() {
   accountsReport()                                                 // 6
   updatePointsCredit(1)                                            // 7
   updatePointsCredit(1)
   updatePointsCredit(5)                                            // 8
   accountsReport()                                                 // 9
}
```

filter

La función de filtro le permite filtrar colecciones. Toma un predicado de filtro como parámetro lambda. El predicado se aplica a cada elemento. Los elementos que cumplen la condición del predicado se devuelven en la colección de resultados.

1. Define una colección de números.
2. Obtiene los números positivos
3. Puede utilizar la expresión it para indicar cada uno de los elementos.

```
val numbers = listOf(1, -2, 3, -4, 5, -6)      // 1

val positives = numbers.filter { x -> x > 0 }  // 2

val negatives = numbers.filter { it < 0 }      // 3
```

map (es la función)

La función de extensión de mapa le permite aplicar una transformación a todos los elementos de una colección. Toma una función de transformador como parámetro lambda.

1. Define una colección de números.
2. Dobla el valor de todos los números.
3. Utiliza el operador it para triplicar los números.

```
val numbers = listOf(1, -2, 3, -4, 5, -6)      // 1

val doubled = numbers.map { x -> x * 2 }       // 2

val tripled = numbers.map { it * 3 }           // 3
```

any, all, none

Estas funciones verifican la existencia de elementos de colección que coinciden con un predicado dado.

any

La función **any** devuelve verdadero si la colección contiene al menos un elemento que coincide con el predicado dado.

1. Define una colección de números.
2. Verifica si hay elementos negativos.
3. Verifica si hay elementos mayores a 6.

```
val numbers = listOf(1, -2, 3, -4, 5, -6)          //
1

val anyNegative = numbers.any { it < 0 }
// 2

val anyGT6 = numbers.any { it > 6 }
// 3
```

any, all, none

all

La función **all** devuelve verdadero si todos los elementos de la colección coincide con el predicado dado.

1. Define una colección de números.
2. Verifica si todos los elementos son pares
3. Verifica si hay elementos menores a 6.

```
val numbers = listOf(1, -2, 3, -4, 5, -6)          //
1

val allEven = numbers.all { it % 2 == 0 }
// 2

val allLess6 = numbers.all { it < 6 }              //
3
```

any, all, none

none

La función **none** devuelve verdadero si no hay elementos de la colección que coincidan con el predicado dado.

1. Define una colección de números.
2. Verifica si todos los elementos son impares (todos son pares)
3. Verifica si no hay elementos mayores a 6.

```
val numbers = listOf(1, -2, 3, -4, 5, -6)          //
1

val allEven = numbers.none { it % 2 == 1 }
// 2

val allLess6 = numbers.none { it > 6 }
// 3
```

find, findLast

Las funciones find y findLast devuelven el primer o el último elemento de la colección que coincide con el predicado dado. Si no hay tales elementos, las funciones devuelven null.

1. Define una colección de palabras
2. Busca la primera palabra que contenga “some”
3. Busca la última palabra que contenga “some”
4. Busca la primera palabra que contenga “nothing”

```
val words = listOf("Lets", "find", "something",
"in", "collection", "somehow")  // 1

val first = words.find { it.startsWith("some") }
// 2
val last = words.findLast { it.startsWith("some")
}                              // 3

val nothing = words.find { it.contains("no
```

Sorted

sorted devuelve una lista de elementos de la colección ordenados según su orden de clasificación natural (ascendente).
sortedBy ordena los elementos según el orden de clasificación natural de los valores devueltos por la función de selección especificada.

1. Define una colección de números desordenados
2. Ordena los números en su orden natural.
3. Ordena los números de forma invertida a su orden natural usando -it como selector.

```
val shuffled = listOf(5, 4, 2, 1, 3)      // 1
val natural = shuffled.sorted()           // 2
val inverted = shuffled.sortedBy { -it } // 3
```

Ejercicios:

Crear un sistema en Kotlin que permita almacenar información de libros escolares. Los datos que deben almacenar son:

Nombre del libro. ISBN.

Año publicación Editorial

Cantidad de páginas Precio

Autor del libro

Tipo de libro (D: Digital / I: Impreso)

Este sistema debe tener una funcionalidad que permita devolver el precio con el signo $ en cualquier parte del proyecto. Además, debe contar con una función que imprima todos la información del libro.

Para terminar, se solicita crear la función “main” y declarar 4 variables de tipo “LibrosEscolares”, donde 3 de estas variables deben ser mutables y una variable debe ser inmutable.

Imprimir las 4 variables llamando a la función que imprime toda la información.

Instrucciones ejercicios

1. Crear el proyecto Kotlin correctamente.

2. Crear la clase LibrosEscolares con todos los atributos solicitados.

3. Crear función “precioFormateado” de forma correcta.

4. Crear función “imprimir” de forma correcta.

5. Crear función main de forma correcta.

6. Declarar 4 variables de la clase “LibrosEscolares”.

7. Llamar a la funciones creadas y tener la salida de información por consola.