1. Introducción a las estructuras de datos en Kotlin

a. ¿Qué son las estructuras de datos y para qué se utilizan?

**R/=** También llamados tipos de colección, estos permiten almacenar varios valores, generalmente del mismo tipo de datos, de manera organizada. Estos pueden llegar a ser una lista ordenada, una agrupación de valores únicos o una asignación de valores de un tipo de datos a valores de otro tipo.

b. Ventajas de utilizar estructuras de datos en Kotlin

**R/=**

* Permite organizar datos del mismo tipo.
* Utilizando estructuras de datos es mucho más organizado.

c. Diferencias entre las estructuras de datos en Kotlin y Java

**R/=**

* Velocidad de codificación: Ambos tienen casi la misma velocidad de codificación, pero Java requiere escribir más código.
* Estabilidad: Java es un lenguaje establecido desde hace muchos años, teniendo versiones de soporte a largo plazo, mientras que Kotlin al ser más joven no tiene versiones con soporte a largo plazo.
* Funciones de extensión: Mientras en Java no está disponible, los programadores deben crear una nueva clase para extender la funcionalidad de la clase existente. En Kotlin los desarrolladores añaden métodos a las clases sin realizar cambios en el código fuente.
* Corrutinas: En Kotlin se puede utilizar corrutinas que se basan en conceptos establecidos de otros lenguajes, mientras que en Java se crea un hilo de fondo para el trabajo intensivo, para administrar varios subprocesos incrementa la complejidad y la posibilidad de los errores en el código.
* Seguridad nula: *NullPointerExceptions* es uno de los problemas más molestos para los desarrolladores de Java, mientras que Kotlin tiene una seguridad nula incorporada en la cual se debe marcar explícitamente la variable como anulable.
* Tipos de comodines: Mientras Java tiene disponibles diferentes tipos de comodines, Kotlin no cuenta con estos, sin embargo tiene dos alternativas: varianza del sitio de declaración y las proyecciones de tipo.

2. Arreglos en Kotlin

a. ¿Qué es un arreglo?

**R/=** Un arreglo es una estructura con valores de datos, que están almacenados de forma contigua en memoria, todos los elementos son referenciados por un mismo nombre y tienen el mismo tipo de dato. Los elementos estarán indexados tomando como base el 0 y el tamaño declarado del arreglo será fijo.

b. Creación de arreglos en Kotlin

**R/=** Para declarar un array en el código, usa la función *arrayOf()*, esta toma los elementos del array como parámetros y muestra un array del tipo que coincide con los parámetros que se pasaron. Ejemplo:

val rockPlanets = arrayOf<String>("Mercury", "Venus", "Earth", "Mars")

c. Accediendo a los elementos de un arreglo

**R/=** Para acceder a los elementos de un arreglo primero se llama el nombre del array, luego se abren y cierran los corchetes y por último se indica el índice al cual se quiere acceder. Ejemplo:

println(rockPlanets[2])

d. Modificando los elementos de un arreglo

**R/=** Para modificar los elementos de un array se llama el nombre del array, seguido de un corchete de apertura y otro de cierre que contienen el índice, luego sigue el operador de asignación ( = ) y un valor nuevo. Ejemplo:

rockPlanets[3] = "Second Earth"

e. Recorriendo un arreglo

**R/=** Para recorrer un arreglo se utiliza el ciclo For normal de Kotlin. Ejemplo:

val rockPlanets = arrayOf<String>("Mercury", "Venus", "Earth", "Mars")

for(i in rockPlanets) {

println(i)

}

f. Funciones útiles para trabajar con arreglos en Kotlin

R/= La función *joinToString()* permite crear un String legible para imprimir. Aunque también puedes usar *contentToString()*.

3. Listas en Kotlin

a. ¿Qué es una lista?

**R/=** Una [lista](https://kotlinlang.org/docs/collections-overview.html#list) es una colección genérica de elementos que se caracteriza por almacenarlos de forma ordenada, donde pueden existir duplicados (incluso un ítem null) y se indexan los elementos con base 0.

b. Creación de listas en Kotlin

**R/=** La función *listOf()* toma los elementos como parámetros, pero muestra un elemento List en lugar de un array. Ejemplo: En main(), llama a *listOf()* para crear un elemento List de frutas llamado fruits.

fun main() {

val fruits = listOf("Banana", "Coco", "Pear", "Strawberry", "Apple", "Grape", "Pineapple", "Kiwi")

}

c. Accediendo a los elementos de una lista

**R/=** Es muy parecido a la sintaxis de los arrays se pueden acceder de la misma manera o también usando el método *get()*. Ejemplos:

println(fruits[2])

println(fruits.get(2))

Además de obtener un elemento a través de su índice, también se puede buscar el índice de un elemento específico con el método *indexOf()*. Si el elemento no aparece en la lista, se muestra -1.

d. Modificando los elementos de una lista

**R/=** Para modificar los elementos de una lista se debe llamar específicamente a la función *mutableListOf()* en lugar del *listOf().* De esta función salen dos versiones:

* La primera función *add()* tiene un solo parámetro del tipo de elemento de la lista y lo agrega al final de la lista. Ejemplo:

val fruits = mutableListOf("Banana", "Coco", "Pear", "Strawberry", "Apple", "Grape", "Pineapple", "Kiwi")

fruits.add("Watermelon")

* La otra versión de *add()* tiene dos parámetros. El primer parámetro corresponde a un índice en el que se debe insertar el nuevo elemento. El segundo parámetro es el elemento que se agrega a la lista. Ejemplo:

val fruits = mutableListOf("Banana", "Coco", "Pear", "Strawberry", "Apple", "Grape", "Pineapple", "Kiwi")

fruits.add(3, "Blackberry")

Para modificar los elementos en un índice específico es prácticamente igual al array, se llama el nombre de la lista, seguido de un corchete de apertura y otro de cierre que contienen el índice, luego sigue el operador de asignación ( = ) y un valor nuevo. Ejemplo:

fruits[3] = "Mango"

Para modificar los elementos quitándoles de la lista, se puede pasarlo al método *remove()* o utilizar su índice mediante *removeAt()*. Ejemplos:

fruits.remove("Mango")

fruits.removeAt(3)

e. Recorriendo una lista

**R/=** Para recorrer una lista se utiliza el ciclo For. Ejemplo:

val fruits = listOf("Banana", "Coco", "Pear", "Strawberry", "Apple", "Grape", "Pineapple", "Kiwi")

for (i in fruits) {

println(i)

}

f. Funciones útiles para trabajar con listas en Kotlin

**R/=**

* La función *contains()* acepta un solo valor entre paréntesis y devuelve un booleano verdadero o falso dependiendo de si ese valor existe o no dentro de una lista, no necesita cambiar el contenido de la lista, por lo que puede usarse en una lista inmutable, o si lo desea también en una lista mutable.
* La función *random()* también solo requiere acceso de solo lectura y devuelve un elemento aleatorio de la lista.

4. Conjuntos en Kotlin

a. ¿Qué es un conjunto?

**R/=** Un conjunto es una colección que no tiene un orden específico y no permite valores duplicados, estos son más aplicables en programas que trabajan con conjuntos de datos que deban contener valores no duplicados, como nombres de usuario o contraseñas.

Para poder crear colecciones de estas el secreto es el código hash, este es un Int que produce el método *hashCode()* de cualquier clase de Kotlin.

b. Creación de conjuntos en Kotlin

**R/=** Los conjuntos de Kotlin se dividen en dos categorías: mutables e inmutables, y los conjuntos no son una excepción.

Un conjunto inmutable se declara usando la palabra clave *setOf* e indica un conjunto cuyos valores no pueden cambiar a lo largo de un programa. Ejemplo:

var animals = setOf(cat, dog, lion)

Si hay valores repetidos los elimina imprimiendo sólo un valor de los repetidos.

Con los conjuntos mutables, se puede agregar, eliminar o reemplazar tantos elementos como se quieran, permitiendo trabajar tanto con conjuntos que se completan con valores como con conjuntos vacíos, que son conjuntos que se inicializan sin ningún valor pero que se pueden completar en un momento posterior.

Para declarar los conjuntos mutables se usa el término *mutableSetOf* seguido del tipo de dato que esperamos que contenga el conjunto y un par de paréntesis vacíos. Ejemplo:

var animals = mutableSetOf<String>()

c. Accediendo a los elementos de un conjunto

**R/=** Se utiliza esta función *elementAt()* para acceder y recuperar elementos, esta acepta un valor entero y devuelve el elemento en esa posición. Esta función es útil para colecciones que no poseen acceso por medio de un índice. Ejemplo:

println(animals.elementAt(2))

Si un elemento no se encuentra en una determinada posición el compilador podría lanzar una excepción, para prevenir esto Kotlin cuenta con una seguridad nula, en este caso la función quedaría así *elementAtOrNull()*, siendo esta más segura.

d. Modificando los elementos de un conjunto

R/= Para modificar cualquier elemento de un conjunto se pueden utilizar *addAll()* y *clear()* que son son funciones muy útiles, rápidas y eficientes para agregar o eliminar grandes cantidades de datos de una colección a la vez. Ejemplo:

var listAnimals = listOf(cat, dog, lion, cat, dog, spider)

var animals = mutableSetOf<String>()

*animals.addAll(listAnimals)*

println(animals)

*animals.clear()*

e. Recorriendo un conjunto

R/= Se puede recorrer un conjunto con la ayuda del *forEach()* o también usar un ciclo *For*.

f. Funciones útiles para trabajar con conjuntos en Kotlin

R/= Algunas de las muchas funciones útiles son:

* La función *first()* retorna el primer valor almacenado en el conjunto.
* La función *last()* retorna el último valor almacenado en el conjunto.
* La función *sum()* retorna la suma de todos los elementos presentes en el conjunto.
* La función *average()* retorna el promedio de los elementos almacenados en el conjunto.

5. Mapas en Kotlin

a. ¿Qué es un mapa?

**R/=** Un map es una colección que consta de claves y valores, esto quiere decir que a cada clave le corresponde un solo valor y será única como si se tratase de un identificador.

Las claves de un mapa son únicas. Sin embargo, los valores de un mapa no lo son. Dos claves diferentes podrían mapear al mismo valor.

b. Creación de mapas en Kotlin

**R/=** Los mapas se pueden declarar con las funciones *mapOf()* o *mutableMapOf()*, los mapas requieren dos tipos genéricos separados por comas: uno para las claves y otro para los valores. Para propagar un mapa con valores iniciales, cada par clave-valor consta de la clave, seguida del operador *to* y del valor, cada par está separado por una coma. Ejemplo:

val countries = mutableMapOf(

"Italy" to 0,

"Colombia" to 0,

"Mexico" to 1,

"France" to 2,

"EEUU" to 79,

"Egypt" to 82,

"Greece" to 27,

"Canada" to 14

)

c. Accediendo a los elementos de un mapa

**R/=** Se puede acceder a valores con el método *get()*. Ya sea que se use sintaxis de subíndice o se llame a *get()*, es posible que la clave que se pase no esté en el mapa y si no hay un par clave-valor, se mostrará un valor nulo. Ejemplo:

println(countries.get("Canada"))

d. Modificando los elementos de un mapa

**R/=** Se puede usar esta sintaxis de subíndice para configurar pares clave-valor adicionales. Ejemplo:

countries["Brazil"] = 8

Al igual que las listas y los conjuntos, Map brinda una propiedad size. Ejemplo:

println(countries.size)

El método *remove()* quita el par clave-valor con la clave especificada. También muestra el valor que se quitó o null, si la clave especificada no está en el mapa. Ejemplo:

countries.remove("Brazil")

El método *put()*, también puede modificar el valor de una clave que ya existe.

e. Recorriendo un mapa

**R/=** Debido a la naturaleza de los mapas en Kotlin, es posible [desestructurar las declaraciones](https://kotlinlang.org/docs/destructuring-declarations.html) (a destructuring declaration creates multiple variables at once) que comprometan a sus entradas en valores individuales. Ejemplo:

fun main() {

val countries = mapOf(

"Italy" to 0,

"Colombia" to 0,

"Mexico" to 1,

"France" to 2,

"EEUU" to 79,

"Egypt" to 82,

"Greece" to 27,

"Canada" to 14

)

for ((country, value) in countries) {

println("$country -> $value")

}

}

f. Funciones útiles para trabajar con mapas en Kotlin

**R/=**

* entries: retorna un tipo Set<Entry<K,V>> de solo lectura de todos los pares clave-valor
* keys: retorna un Set<K> de solo lectura de todas las claves
* size: retorna el número de entradas en el mapa
* values: retornar una Collection<V> de solo lectura con los valores en el mapa

6. Pares en Kotlin

a. ¿Qué es un par?

R/= Los pares en kotlin son una estructura que permite guardar dos valores relacionados.

b. Creación de pares en Kotlin

R/= Para crear pares se crean de la siguiente manera:

val par = Pair("Hola", 123)

c. Accediendo a los elementos de un par

R/= Para acceder se puede acceder de varias formas pero la más común es la siguiente:

println(par)

d. Modificando los elementos de un par

R/= Los pares en kotlin son una estructura de datos inmutable, lo que significa que no se puede modificar después de haber sido creado.

e. Recorriendo un par

R/= No se puede recorrer un par, ya que solo cuenta con dos valores.

f. Funciones útiles para trabajar con pares en Kotlin

R/= Algunas funciones útiles para trabajar con pares son:

* La función *first()* retorna el primer valor.
* La función *second()* retorna el primer valor.