Taller Estructuras de Datos en Kotlin

El objetivo de este taller es que los aprendices sean capaces de comprender y utilizar las principales estructuras de datos en Kotlin, incluyendo arreglos, listas, conjuntos, mapas y pares.

1. Introducción a las estructuras de datos en Kotlin

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

R/ Las estructuras de datos son formas de organizar y almacenar datos de manera eficiente, se utiliza para gestionar grandes cantidades de datos, almacenar y recuperar información de manera rápida y es una herramienta fundamental para la manipulación de datos.

b. Ventajas de utilizar estructuras de datos en Kotlin

R/ El uso de estructuras de datos en Kotlin mejora la eficiencia, organización y flexibilidad de un programa, al tiempo que facilita la reutilización de código y el mantenimiento a largo plazo.

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

R/ Una de las diferencias entre Kotlin y C# son las Propiedades y campos: las propiedades se definen utilizando la palabra clave "val" para propiedades de solo lectura y "var" para propiedades que pueden ser modificadas. En C#, las propiedades se definen utilizando las palabras clave "get" y "set". Además, en C# Se pueden definir campos públicos, mientras que en Kotlin todas las propiedades deben tener un getter y un setter.

Otra de las diferencias son los Tipos anulables: En Kotlin, todas las variables son anulables por defecto, lo que significa que pueden ser nulas. En C#, todas las variables son no anulables por defecto, pero se pueden hacer anulables mediante el uso del operador "?".

2. Arreglos en Kotlin

a. ¿Qué es un arreglo?

R/ Un arreglo es una estructura de datos que permite almacenar diferentes datos de un mismo tipo en una única variable. Se define utilizando la palabra clave "arrayOf" seguida de los elementos a almacenar. Los arreglos son inmutables por defecto, pero también existe la variante "Array" que se puede modificar después de su creación.

b. Creación de arreglos en Kotlin

R/ Con la Función arrayOf(): Para declarar un arreglo en Kotlin se usa la función (arrayof) que de igual forma se puede indicar que tipo de dato es. Después se indican los elementos del arreglo como argumentos a la función.

val arreglo = arrayOf(element1, element2, ...)

val nombres = arrayOf("Sandra", "Andrea", "aytana", …)

c. Accediendo a los elementos de un arreglo

R/ Para acceder a los elementos de un arreglo se utiliza el operador de índice que utiliza los corchetes junto con el índice de elemento.

val numbers = arrayOf(1, 2, 3, 4, 5)

índice del elemento es [0] quiere decir que mostrara 1

val firstNumber = numbers [0]

La otra forma es con la función get()

val firstNumber = numbers .get(0)

d. Modificando los elementos de un arreglo

R/ Para modificar un elemento de un array se accede al índice o posición del elemento que deseas modificar seguido del nuevo valor que deseas actualizar.

val numbers = arrayOf(1, 2, 3, 4, 5)

accedemos al índice y asignamos un nuevo valor

numeros[1] = 10

La otra forma es llamando a la funcion set()

numbers.set(3, 13)

e. Recorriendo un arreglo

R/ Se puede recorrer los elementos con diferentes tipos de bucles o funciones como el bucle “for” y el bucle “for each” y funciones como el “forEach”

para recorrer con un bucle for, debemos usar como base los índices de sus elementos.Esto quiere decir que usamos en la declaración de variable el índice y luego expresamos su existencia sobre la propiedad rango entero índices.

val numbers= arrayOf(1, 2, 3, 4, 5)

for (i in numbers.indices) {

println(numbers[i])

}

Y con la función forEach y si le pasa como argumento una función lambda (Función anónima)

val numbers= arrayOf(1, 2, 3, 4, 5)

numbers.forEach { numero -> println(numero) }

f. Funciones útiles para trabajar con arreglos en Kotlin

Alguna de las funciones útiles son:

1. size: devuelve el tamaño del arreglo.

val number = arrayOf(1, 2, 3, 4, 5)

val size = number.size // 5

2. filter: devuelve un nuevo arreglo con los elementos que cumplen con una condición.

val numeros = arrayOf(1, 2, 3, 4, 5)

val numerosPares = numeros.filter { numero -> numero % 2 == 0 }

3. sorted: devuelve un nuevo arreglo ordenado.

val numeros = arrayOf(5, 2, 4, 1, 3)

val numerosOrdenados = numeros.sorted()

3. Listas en Kotlin

a. ¿Qué es una lista?

R/ Las listas son una colección ordenada de elementos que pueden contener duplicados. Se puede acceder a los elementos de la lista por su índice y se puede agregar, eliminar y modificar elementos de la lista. Hay dos tipos principales de listas en Kotlin: la lista inmutable (List) y la lista mutable (MutableList). La lista inmutable es de sólo lectura, mientras que la lista mutable se puede modificar después de que se crea.

b. Creación de listas en Kotlin

R/ Crea una lista vacia y se puede agregar el tipo de dato

val listaVacia = emptyList<String>()

crea una lista de elementos especificos, la funcion list es para las listas inmutables

val lista = listOf("manzana", "naranja", "plátano")

la funcion mutableList es para las listas mutables

val listaMutable = mutableListOf("manzana", "naranja", "plátano")

Crea una lista utilizando un rango de valores

val numeros = (1..10).toList() // crea una lista de números del 1 al 10

Crea una lista utilizando una expresión lambda

val numeros = List(10) { it \* 2 } // crea una lista de números pares del 2 al 20

c. Accediendo a los elementos de una lista

R/ Se pueden acceder a una elemento de una lista por su indece (posicion) por ejemplo:

val frutas = listOf("manzana", "naranja", "plátano")

Acceder al primer elemento de la lista

val primeraFruta = frutas[0] // "manzana"

Acceder al segundo elemento de la lista

val segundaFruta = frutas[1] // "naranja"

Acceder al último elemento de la lista

val ultimaFruta = frutas[frutas.size - 1] // "plátano"

la funcion last accede al ultimo elemento de la lista

val ultimaFruta = frutas.last() // "plátano"

la funcion get(indice) accede a los elementos de la lista con su indice

val fruta = frutas.get(3) // devuelve null

d. Modificando los elementos de una lista

R/En Kotlin, se puede modificar los elementos de una lista si se utiliza una lista mutable ( **MutableList** ). Si se utiliza una lista inmutable ( List ), los elementos de la lista no se pueden modificar después de que se crea.

val numeros = mutableListOf(1, 2, 3, 4, 5)

Para modificar se tiene en cuenta el índice que se va a modificar y el valor nuevo

Modificar el primer elemento de la lista

numeros[0] = 6 // ahora la lista es [6, 2, 3, 4, 5]

Agregar un nuevo elemento al final de la lista

numeros.add(8) // ahora la lista es [6, 3, 7, 4, 5, 8]

Eliminar un elemento de la lista

numeros.removeAt(0) // ahora la lista es [3, 7, 4, 5, 8]

e. Recorriendo una lista

R/ 1. Utilizando un bucle for para iterar sobre los elementos de la lista.

val numeros = listOf(1, 2, 3, 4, 5)

for (numero in numeros) {

println(numero)

}

2. Utilizando la propiedad indices de la lista y un bucle for para acceder a cada elemento de la lista mediante su índice.

val numeros = listOf(1, 2, 3, 4, 5)

for (i in numeros.indices) {

println(numeros[i])

}

3. Utilizando la función forEach() para ejecutar una función para cada elemento de la lista.

val numeros = listOf(1, 2, 3, 4, 5)

numeros.forEach {

println(it)

}

4. Utilizando la función forEachIndexed para ejecutar una función para cada elemento de la lista y acceder tanto al índice como al valor de cada elemento.

val numeros = listOf(1, 2, 3, 4, 5)

numeros.forEachIndexed { index, value ->

println("El elemento en la posición $index es $value")

}

f. Funciones útiles para trabajar con listas en Kotlin

R/

1. filter( ): devuelve una nueva lista que contiene solo los elementos que cumplen con una determinada condición.

val numeros = listOf(1, 2, 3, 4, 5)

val numerosPares = numeros.filter { it % 2 == 0 }

Output: [2, 4]

1. map( ): devuelve una nueva lista que contiene los elementos resultantes de aplicar una determinada operación a cada elemento de la lista original.

val numeros = listOf(1, 2, 3, 4, 5)

val cuadrados = numeros.map { it \* it }

Output: [1, 4, 9, 16, 25]

1. reduce( ): combina los elementos de la lista utilizando una operación específica y devuelve un único resultado.

val numeros = listOf(1, 2, 3, 4, 5)

val suma = numeros.reduce { acc, numero -> acc + numero }

Output: 15

4. Conjuntos en Kotlin

a. ¿Qué es un conjunto?

R/Un conjunto es una colección de elementos sin orden y sin elementos duplicados. Son útiles en situaciones en las que se necesita almacenar un conjunto de elementos únicos y no importa el orden en que se almacenen.

b. Creación de conjuntos en Kotlin

R/

1. Usando la función setOf() para crear un conjunto inmutable con los elementos especificados:

val set1 = setOf(1, 2, 3, 4, 5)

1. Usando la función mutableSetOf() para crear un conjunto mutable con los elementos especificados:

val set2 = mutableSetOf("apple", "banana", "orange")

c. Accediendo a los elementos de un conjunto

R/ En los conjuntos no se puede acceder al elmento por el indice como las listas o los array, pero hay un metodo contains() para verificar si un elemento está presente en el conjunto que devuelve un valor Booleano.

val mySet = setOf("apple", "banana", "orange")

val containsBanana = mySet.contains("banana")

val containsPear = mySet.contains("pear")

println(containsBanana) // Output: true

println(containsPear) // Output: false

Para acceder a los elementos de un conjunto a través de la iteración, puedes utilizar un bucle for-each o un método forEach. Por ejemplo:

val mySet = setOf("apple", "banana", "orange")

for (element in mySet) {

println(element)

}

Output: apple, banana, orange

mySet.forEach { element ->

println(element)

}

Output: apple, banana, orange

d. Modificando los elementos de un conjunto

R/ los conjuntos set son estructuras inmutables por lo que no se puede modificar los elementos, por eso para modificar debes crear otro conjunto con los elementos actualizados.

Crear un nuevo conjunto con elementos actualizados

val originalSet = setOf(1, 2, 3)

En este ejemplo usamos map se utiliza para multiplicar cada elemento original \* 2 y toSet() para crear el nuevo conjunto

val updatedSet = originalSet.map { it \* 2 }.toSet() // [2, 4, 6]

Crear un nuevo conjunto sin un elemento específico

val originalSet = setOf("apple", "banana", "orange")

usamos filter para crear un nuevo conjunto con los elementos originales y eliminamos un elemento y toSet() para crear el conjunto nuevo con su cambio

val updatedSet = originalSet.filter { it != "banana" }.toSet() // ["apple", "orange"]

e. Recorriendo un conjunto

R/ Para recorrer un conjunto (Set) utilizamos un ciclo for o forEach(). El ciclo for te permite recorrer cada elemento del conjunto de manera secuencial, mientras que forEach() es una función de orden superior que toma una función lambda como argumento y la aplica a cada elemento del conjunto.

Recorriendo un conjunto con ciclo for

val set = setOf("apple", "banana", "orange")

for (item in set) {

println(item)

}

Output: "apple", "banana", "orange"

Recorriendo un conjunto con forEach()

val set = setOf("apple", "banana", "orange")

set.forEach { item ->

println(item)

}

Output: "apple", "banana", "orange"

f. Funciones útiles para trabajar con conjuntos en Kotlin

1. union(): esta función crea un nuevo conjunto que contiene todos los elementos de dos conjuntos. Los elementos duplicados se eliminan automáticamente.

val set1 = setOf("apple", "banana", "orange")

val set2 = setOf("banana", "pear")

val unionSet = set1.union(set2)

println(unionSet) // Output: ["apple", "banana", "orange", "pear"]

2. intersect(): esta función crea un nuevo conjunto que contiene solo los elementos que se encuentran en ambos conjuntos.

val set1 = setOf("apple", "banana", "orange")

val set2 = setOf("banana", "pear")

val intersectSet = set1.intersect(set2)

println(intersectSet) // Output: ["banana"]

3. max() y min(): estas funciones devuelven el elemento máximo y mínimo del conjunto, respectivamente

val set = setOf(3, 6, 2, 9, 4)

val maxElement = set.max()

val minElement = set.min()

println(maxElement) // Output: 9

println(minElement) // Output: 2

5. Mapas en Kotlin

a. ¿Qué es un mapa?

R/ Son estructura de datos que permite asociar pares clave-valor, donde cada clave es única en el mapa y se utiliza para acceder al valor correspondiente. Se pueden crear mapas inmutables con la función mapOf() y mapas mutables con mutableMapOf(). Los valores en un mapa pueden ser de cualquier tipo, siempre y cuando la clave sea única

b. Creación de mapas en Kotlin

R/ Para crear mapas utilizamos la función mapOf O mutableMapOf.

La función mapOf() se utiliza para crear mapas inmutables. Toma una lista de pares clave-valor y devuelve un mapa inmutable. Por ejemplo

val map = mapOf("key1" to "value1", "key2" to "value2", "key3" to "value3")

La función mutableMapOf() se utiliza para crear mapas mutables. Toma una lista de pares clave-valor y devuelve un mapa mutable. Por ejemplo:

val mutableMap = mutableMapOf("key1" to "value1", "key2" to "value2", "key3" to "value3")

c. Accediendo a los elementos de un mapa

R/ Para acceder a un mapa utilizamos la clave asociada a cada valor se utilizan la notación de corchetes para los map inmutables y para los mutables se utilizan los corchetes, función get() y función put().

map inmutables

val map = mapOf("key1" to "value1", "key2" to "value2")

println(map["key1"]) // Output: "value1"

println(map["key2"]) // Output: "value2"

map mutables

val mutableMap = mutableMapOf("key1" to "value1", "key2" to "value2")

println(mutableMap["key1"]) // Output: "value1"

println(mutableMap.get("key2")) // Output: "value2"

println(mutableMap.put("key3", "value3")) // Output: "value3"

d. Modificando los elementos de un mapa

R/

1. Para modificar un valor existente, se puede acceder al elemento a través de su clave y asignarle un nuevo valor:

val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)

map["b"] = 4 // modifica el valor asociado a la clave "b" a 4

2. Para agregar un nuevo elemento al mapa, se puede asignar un nuevo valor a una clave que no existía previamente:

val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)

map["d"] = 4 // agrega un nuevo elemento con la clave "d" y el valor 4

3.Para eliminar un elemento del mapa, se puede utilizar el método remove y especificar la clave del elemento que se desea eliminar:

val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)

map.remove("b") // elimina el elemento con la clave "b"

e. Recorriendo un mapa

R/

1. Recorrido por pares clave-valor utilizando un bucle for:

val map = mapOf("a" to 1, "b" to 2, "c" to 3)

for ((key, value) in map) {

println("La clave es $key y el valor es $value")

}

1. Recorrido utilizando la función forEach:

val map = mapOf("a" to 1, "b" to 2, "c" to 3)

map.forEach { (key, value) ->

println("La clave es $key y el valor es $value")

}

f. Funciones útiles para trabajar con mapas en Kotlin

R/

1. filter: función que devuelve un nuevo mapa que contiene sólo los elementos que cumplen con una condición especificada en una función lambda.

val map = mapOf("a" to 1, "b" to 2, "c" to 3)

val filteredMap = map.filter { (key, value) -> key != "b" && value > 1 }

devuelve {"c" to 3}

1. mapOf: función que crea un mapa inmutable a partir de pares clave-valor.

val map = mapOf("a" to 1, "b" to 2, "c" to 3)

6. Pares en Kotlin

a. ¿Qué es un par?

R/ Un par es un objeto que contiene dos elementos y se utiliza para empaquetar dos valores relacionados. Se puede crear un par utilizando la función to, y se puede acceder a sus elementos utilizando las propiedades first y second. Los pares son útiles cuando se necesita trabajar con dos valores relacionados juntos en un contexto donde no tiene sentido definir una nueva clase para encapsular estos valores.

b. Creación de pares en Kotlin

R/Se pueden crear pares utilizando la función to, que toma dos valores y los convierte en un objeto Pair.

Se puede crear un par que contenga una cadena de texto y un número entero.

val pair = "clave" to 10

también se puede crear utilizando la función Pair que requiere de los dos valor para crear el objeto pair

val pair = Pair("clave", 10)

c. Accediendo a los elementos de un par

R/

val miPar = Pair(42, "Hola mundo")

primero accede al primer valor de Pair

val primerElemento = miPar.first // devuelve 42

segundo accede al segundo valor de Pair

val segundoElemento = miPar.second (devuelve "Hola mundo")

También se pueden asignar los elementos del par a variables individuales usando la sintaxis de desestructuración:

val (primerElemento, segundoElemento) = miPar

d. Modificando los elementos de un par

R/ Los pares son estructuras de datos inmutables, lo que significa que no se puede modificar.

Sin embargo si deseas modificar 1 o ambos datos utilizas la función copy() lo que hace es crear un nuevo par con los datos originales o actualizados de un par.

val miPar = Pair(42, "Hola mundo")

val miNuevoPar = miPar.copy(second = "Hola Kotlin")

e. Recorriendo un par

R/ Se puede recorrer un par utilizando la función forEach y una función lambda con dos parámetros para acceder a los elementos del par.

val miPar = Pair(42, "Hola mundo")

miPar.forEach { (primerElemento, segundoElemento) ->

println("El primer elemento es $primerElemento y el segundo elemento es $segundoElemento")

}

de otra forma se puede acceder a los elementos de un par utilizando el first(que accede al primer valor) y second(que accede al segundo valor).

val miPar = Pair(42, "Hola mundo")

println("El primer elemento es ${miPar.first}")

println("el segundo elemento es ${miPar.second}")

f. Funciones útiles para trabajar con pares en Kotlin

R/

to(): La función to() permite crear un par a partir de dos valores. Por ejemplo, si se quiere crear un par que contenga un entero y una cadena, se puede utilizar la siguiente sintaxis: val miPar = 1 to "Hola".

first y second: Las propiedades first y second permiten acceder a los valores individuales de un par. Por ejemplo, si se tiene un par llamado miPar, se puede acceder al primer valor del par utilizando la propiedad miPar.first y al segundo valor utilizando la propiedad miPar.second.

swap(): La función swap() permite intercambiar los valores de un par. Por ejemplo, si se tiene un par llamado miPar que contiene los valores "Hola" y 1, se puede intercambiar los valores utilizando la siguiente sintaxis: miPar = miPar.swap(), y el valor de miPar sería (1, "Hola").

7. Prácticas de estructuras de datos en Kotlin

a. Ejercicios prácticos para aplicar los conceptos aprendidos

1. Dado un arreglo de números enteros obtener la suma de todos los elementos, el promedio, y encontrar el elemento mayor y menor.

1. Dado una lista de nombres filtra los nombres que comienzan con "M", ordenarlos alfabéticamente, contar cuántos nombres tienen más de 4 letras, y convertir todos los nombres a mayúsculas.

1. Dado dos conjuntos de números realiza algunas operaciones, como unirlos en un nuevo conjunto, obtener la intersección de los dos conjuntos, obtener la diferencia entre el conjunto A y el conjunto B, y verificar si un elemento está en el conjunto A.

1. Dado un mapa de nombres y edades y realiza algunas operaciones, obtener la edad de una persona en particular, agregar una nueva persona al mapa de edades, y recorrer el mapa en orden alfabético e imprimir sus valores.

1. Dado un par de valores y realiza alguna operación, imprime los valores del par, y luego crea una lista de pares y realiza filtro de los pares que tienen una edad mayor a 25 años y ordenarlos alfabéticamente por nombre.

b. Solución a los ejercicios prácticos

1. fun main() {

// Creamos un arreglo de números enteros

val numeros = intArrayOf(5, 10, 15, 20, 25)

// Imprimimos los valores del arreglo

println("Los valores del arreglo son: ${numeros.contentToString()}")

// Obtenemos la suma de todos los elementos del arreglo

var suma = 0

for (numero in numeros) {

suma += numero

}

println("La suma de los elementos del arreglo es: $suma")

// Obtenemos el promedio de los elementos del arreglo

val promedio = suma / numeros.size.toDouble()

println("El promedio de los elementos del arreglo es: $promedio")

// Buscamos el elemento mayor y menor del arreglo

var mayor = numeros[0]

var menor = numeros[0]

for (numero in numeros) {

if (numero > mayor) {

mayor = numero

}

if (numero < menor) {

menor = numero

}

}

println("El elemento mayor del arreglo es: $mayor")

println("El elemento menor del arreglo es: $menor")

}

1. fun main() {

// Creamos una lista de nombres

val nombres = listOf("Juan", "Pedro", "María", "Ana", "Luisa")

// Imprimimos la lista de nombres

println("Los nombres en la lista son: $nombres")

// Filtramos los nombres que comienzan con "M"

val nombresConM = nombres.filter { it.startsWith("M") }

println("Los nombres que comienzan con M son: $nombresConM")

// Ordenamos los nombres alfabéticamente

val nombresOrdenados = nombres.sorted()

println("Los nombres ordenados alfabéticamente son: $nombresOrdenados")

// Contamos cuántos nombres tienen más de 4 letras

val cantidadNombresLargos = nombres.count { it.length > 4 }

println("Hay $cantidadNombresLargos nombres con más de 4 letras")

// Convertimos la lista de nombres a mayúsculas

val nombresMayusculas = nombres.map { it.toUpperCase() }

println("Los nombres en mayúsculas son: $nombresMayusculas")

}

1. fun main() {

// Creamos dos conjuntos de números

val conjuntoA = setOf(1, 2, 3, 4, 5)

val conjuntoB = setOf(3, 4, 5, 6, 7)

// Imprimimos los conjuntos

println("El conjunto A es: $conjuntoA")

println("El conjunto B es: $conjuntoB")

// Unimos los dos conjuntos en un nuevo conjunto

val conjuntoUnion = conjuntoA.union(conjuntoB)

println("La unión de los conjuntos es: $conjuntoUnion")

// Obtenemos la intersección de los dos conjuntos

val conjuntoInterseccion = conjuntoA.intersect(conjuntoB)

println("La intersección de los conjuntos es: $conjuntoInterseccion")

// Obtenemos la diferencia entre el conjunto A y el conjunto B

val conjuntoDiferencia = conjuntoA.subtract(conjuntoB)

println("La diferencia entre el conjunto A y el conjunto B es: $conjuntoDiferencia")

// Verificamos si un elemento está en el conjunto A

val elemento = 3

if (elemento in conjuntoA) {

println("El elemento $elemento está en el conjunto A")

} else {

println("El elemento $elemento no está en el conjunto A")

}

}

1. fun main() {

// Creamos un mapa mutable de nombres y edades

val edades = mutableMapOf("Juan" to 25, "Pedro" to 30, "María" to 28, "Ana" to 20, "Luisa" to 35)

// Imprimimos el mapa de edades

println("El mapa de edades es: $edades")

// Obtenemos la edad de una persona en particular

val persona = "María"

val edad = edades[persona]

if (edad != null) {

println("La edad de $persona es $edad años")

} else {

println("No se encontró la edad de $persona en el mapa")

}

// Agregamos una nueva persona al mapa de edades

val nuevaPersona = "Carlos"

val nuevaEdad = 40

edades.put(nuevaPersona, nuevaEdad)

println("El mapa de edades actualizado es: $edades")

// Recorremos el mapa de edades y lo imprimimos en orden alfabético

println("El mapa de edades en orden alfabético es:")

edades.toSortedMap().forEach { (nombre, edad) ->

println("$nombre tiene $edad años")

}

}

1. fun main() {

// Creamos un par de valores

val par = Pair("Hola", 123)

// Imprimimos los valores del par

println("El par es: $par")

println("El primer valor del par es: ${par.first}")

println("El segundo valor del par es: ${par.second}")

// Creamos una lista de pares

val listaPares = listOf(

Pair("Juan", 25),

Pair("Pedro", 30),

Pair("María", 28),

Pair("Ana", 20),

Pair("Luisa", 35)

)

// Imprimimos la lista de pares

println("La lista de pares es: $listaPares")

// Filtramos los pares que tienen una edad mayor a 25 años

val paresMayores25 = listaPares.filter { it.second > 25 }

println("Los pares con edad mayor a 25 son: $paresMayores25")

// Ordenamos los pares alfabéticamente por nombre

val paresOrdenados = listaPares.sortedBy { it.first }

println("Los pares ordenados alfabéticamente son: $paresOrdenados")

}

Recursos adicionales:

Documentación oficial de Kotlin: <https://kotlinlang.org/docs/reference/>

Entrega.

Se deberá realizar la entrega de un informe con la solución de los puntos anteriores, el aprendiz acompañará la investigación con ejemplos prácticos de cada estructura y deberá publicar el código fuente en un repositorio en GitHub