Universidad Simón Bolívar.

Departamento de Computación y Tecnología de la Información.

CI3641 – Lenguajes de Programación I.

Abril – Junio 2022.

Estudiante: Keyber Yosnar Sequera Avendaño.

Carnet: 16-11120.

Examen I (25 puntos).

Respuesta 1:

Lenguaje de alto nivel seleccionado: Kotlin.

a) Descripción del lenguaje: Kotlin es un lenguaje de programación estático de código abierto orientado a objetos, con características de lenguajes orientados a la programación funcional, este lenguaje se ejecuta en la máquina virtual de Java (JVM) y es interoperable en JavaScript. Comenzó a ser desarrollado en el año 2010 por la empresa JetBrains, en el año 2012 pasa a ser de código abierto, su popularidad creció rápidamente tras ser comparado con lenguajes de alto nivel como Java y C++ y adoptado por Google en el año 2017, cuando fue declarado por la empresa como lenguaje oficial en Android. Kotlin surge con la idea de compensar las carencias de Java, poder ejecutarse en la JVM, ser completamente interoperable con Java para que la migración de Java a Kotlin no sea tan brusca, tener unos tiempos de compilación cercanos a los de Java y brindar potencia y facilidad de uso al usuario.

i) Tipo de alcance y asociaciones de Kotlin: Kotlin tiene asociación profunda y alcance estático. Los bloques de código en Kotlin están delimitados por llaves "{...}" al igual que en Java, lo cual permite tener una idea intuitiva de los alcances que tienen ciertos objetos. El que Kotlin use alcance estático le brinda la oportunidad al programador de razonar sobre referencias de objetos como parámetros, variables, constantes, tipos, funciones, etc, como simples sustituciones de nombres, lo cual hace más fácil hacer código modular y razonar sobre él, pues la estructura de nombre local se puede mantener de forma aislada, lo cual es una gran ventaja frente al alcance dinámico, el cual obliga al programador a anticipar todos los contextos de ejecución posibles en los que se puede invocar el código del módulo. Una desventaja de este tipo de alcance frente al dinámico radica en su uso en funciones fuertemente anidadas, pues la búsqueda de variables puede volverse un poco ineficiente, sin embargo, puesto que Kotlin se crea con el objetivo de ser un programa sencillo e intuitivo para el usuario, es ideal el uso del alcance estático para cumplir con esto.

ii) Módulos en Kotlin: este lenguaje contiene varios módulos que se importan por defecto, entre ellos contamos con el módulo kotlin (contiene todas las funciones y tipos, disponibles en todas las plataformas soportadas por el lenguaje), kotlin.annotation (librería de soporte para las anotaciones de Kotlin), kotlin.collections (librería que contiene todas estructuras de colección de objetos que vienen por defecto en el lenguaje), kotlin.comparisons (contiene las funciones que permiten crear instancias comparadoras), kotlin.io (API IO que permite trabajar con archivos e hilos), kotlin.ranges (contiene funciones de rangos, progresiones y extensiones relacionados con objetos de alto nivel), kotlin.sequences (contiene el tipo de dato Sequence) y kotlin.text (contiene a las funciones para trabajar con textos y con expresiones regulares), además, al ser un lenguaje basado en Java y diseñado con el objetivo de ser interoperable con este lenguaje puede hacer uso de las librerías principales de este último.

Aparte de las importaciones de paquetes que se hacen por defecto, kotlin permite la creación de paquetes (módulos) propios, para declarar un paquete en kotlin, basta con que un archivo fuente inicie con la declaración "package", por ejemplo:

```
package keyber.ejemplo

fun mostrarMensaje(argumentos opcionales) {...} // Contiene una función que muestra algún mensaje
```

```
class ClaseDePaquete(argumentos opcionales) {...} // Contiene una clase propia del paquete
```

Con esta declaración, todo el contenido del paquete está disponible y puede ser accedido desde otros programas que deseen ejecutar alguna función, clase o incluso variable implementada en el paquete.

Para importar este paquete creado en un programa, se puede hacer de varias maneras, la primera es haciendo un llamado al nombre del paquete seguido por un punto y de lo que sea que queramos extraer de él, esto es:

```
keyber.ejemplo.mostrarMensaje(argumentos declarados) // Llamará a la función mostrar mensaje del paquete

var variable = keyber.ejemplo.ClaseDePaquete(argumentos declarados) // Creará un objeto de la clase del paquete.
```

Ahora, existe otra manera mucho más sencilla de importar paquetes que evita el estar copiando el nombre del paquete más lo que sea que se quiera de él, esto se hace con el comando “import” escrito al inicio del programa seguido del nombre del paquete, esto es:

```
import keyber.ejemplo.ClaseDePaquete // Importará la clase ClaseDelPaquete del paquete en cuestión

import keyber.ejemplo.* // Importará todo el contenido del paquete tanto variables como funciones y clases.

import keyber.ejemplo.ClaseDePaquete as otroNombre // Permite importar la clase ClaseDePaquete usando el nombre
. // otroNombre para referirse a ella
```

También el uso de “as” permite evitar colisiones de nombres al importar paquetes con clases o funciones con el mismo nombre.

Tener en cuenta además que no todas las declaraciones hechas en un paquete pueden ser usadas, pues algunas de ellas pueden tener el prefijo “private”, lo cual hace que la declaración solo sea accesible dentro del mismo paquete.

iii) Alias en Kotlin: Kotlin permite la creación de alias de tipo, veamos que un ejemplo de esto se dio más arriba al importar una clase de un paquete con otro nombre, este tipo de alias, no modifica las propiedades del tipo de la clase, solo permite acceder a esta clase con el uso de otro nombre, otra forma de alias que se usa en Kotlin son los alias de tipo o “Type aliases”, que permiten darles nombres alternativos a nombres de tipos existentes, esto se suele usar cuando los nombres de tipo suelen ser muy largos, veamos, por ejemplo:

```
typealias ParDeListas = Pair<MutableList<Int>, MutableList<Double>>

var par = ParDeListas(mutableListOf(1,2), mutableListOf(1.0, 2.0))
```

Veamos entonces que un objeto de tipo ParDeListas funcionará como un objeto de tipo Pair<MutableList<Int>,MutableList<Double>>, lo cuál simplifica el tipo en cuestión.

Entre otras de las funciones de la declaración typealias es la de proveer alias para tipos de funciones, por ejemplo:

```
typealias NombreFuncion<T> = (T) -> Boolean
```

También permite crear nuevos nombres para clases internas y anidadas, esto es:

```
class A {
   inner class Inner
}

typealias AInner = A.Inner // Lo que nos permite referirnos a la clase interna con el nombre AInner.
```

Sobrecarga en Kotlin: Kotlin es un lenguaje de programación que admite la sobrecarga de funciones, por lo que se pueden declarar varias funciones con el mismo nombre pero distinto número o tipo de parámetros, esto es:

```
fun suma(n1: Int, n2: Int) {
   println(n1 + n2)
}
fun suma(n1: Int, n2: Int, n3: Int) {
   println(n1 + n2 +n3)
}
suma(1,2)  // Se imprime 3, concuerda con la función suma con dos argumentos
suma(1,2,3) // Se imprime 6, concuerda con la función suma con tres argumentos enteros
```

También es posible definir una función con un número variable de argumentos con la forma:

```
fun suma(vararg numeros: Int) {
   var sumaTotal: Int = 0
   for (numero in numeros) {
      sumaTotal += 1
   }
   return sumaTotal
}
println(suma(1,2,3,4,5,6)) // Se obtiene como resultado 21, se suman 6 argumentos
println(suma(3,4)) // Se obtiene como resultado 7, se suman 2 argumentos
```

Otro tipo de sobrecarga perfectamente válido en Kotlin es la sobrecarga en el constructor de una clase, esto es:

```
class Objeto() {
   constructor (nombreArgumento: TipoArgumento, ...){
      ... Acciones con el argumento dado ....
   }
   constructor (nombreArgumento2: TipoArgumento2, ...) {
      ... Acciones con el argumento dado ....
   }
}
```

Tener en cuenta que al igual que la sobrecarga de funciones los nombres de los argumentos, cantidad y tipo deben ser distintos para poder crear la sobrecarga de constructor.

Otro tipo de sobrecarga permitido es la sobrecarga de operadores, esta funciona con un conjunto predefinido de operadores que tengan una representación simbólica fija y precedencia fija. Para implementar una sobrecarga de operador se debe proporcionar una función miembro o una función de extensión con un nombre fijo, para el tipo correspondiente, esto es, el tipo del lado izquierdo para las operaciones binarias y el tipo de argumento para las unarias. Las funciones que sobrecargan a los operadores deben declararse con el modificador “operator”.

Supongamos que queremos hacer la modificación de la resta unaria, para un objeto Objeto(val x: Int, val y: Int) y queremos que su resta unaria vuelva negativos los números pasados como parámetros, entonces se hace:

```
data class Objeto(x: Int, y: Int) {}

operator fun Objeto.unaryMinus() = Objeto(-x, -y)

val instanciaDeObjeto = Objeto(1, 2)

fun main() {
   println(- instanciaDeObjeto) // Devuelve Objeto(x = -1, y = -2)
}
```

Por lo que es posible hacer sobrecarga de operadores y definir estas nuevas operaciones a gusto, considerando que cada operación tiene una llamada a método distinta, así que eso es algo que se debe tener en cuenta.

Polimorfismo en Kotlin: al ser un lenguaje orientado a objetos es posible crear polimorfismo en las clases de Kotlin con el objetivo de reutilizar código, Kotlin permite que esto se implemente de varias maneras, supongamos que queremos crear una estructura para definir los lados de un grafo, tenemos que un lado en un grafo va a tener distintos nombres dependiendo del tipo de grafo, si el grafo es dirigido su lado se llamará arco, si es no dirigido su lado se llamará arista, partiendo de esto supongamos que queremos crear una estructura Lado(elemento1: Int, elemento2: Int) que sea heredad a las clases Arco(elemento1: Int, elemento2: Int) y Arista(elemento1: Int, elemento2: Int), pero no se desea crear instancias de la clase grafo, entonces se hace:

```
abstract class Lado(val elemento1: Int, val elemento2: Int) {     // Se declara la clase Lado de forma abstracta por lo que
   ... métodos y variables locales de la clase Lado ...       // no se pueden crear instancias de ella, pero al ser abstracta
}                                                             // puede ser heredada por otras clases.

class Arco(val elemento1: Int, val elemento2: Int): Lado(elemento1, elemento2) {
   ... métodos y variables locales de la clase lado ...
}
```

Ver que para heredar de una super clase es necesario después de declarar el constructor de la clase colocar dos puntos ":" y añadir la clase o interface que se desea heredar, la clase a heredar debe ser declarada como open o abstract, el prefijo abstract impide crear instancias de la clase padre pero el prefijo open no impide la creación de instancias de esta clase padre, ahora es posible sobreescribir tanto métodos como variables de la clase padre y esto se puede hacer usando el prefijo "override", por ejemplo:

```
// supongamos que existe una función en la clase Lado llamada mostrarLado y queremos usarla en una clase Arista.

class Arista(val elemento1: Int, val elemento2: Int): Lado(elemento1, elemento2) {
   ... métodos y variables de la clase Arista ...
   override fun mostrarLado(): String {
      return "($elemento1 <-> $elemento2)"
   }
}
```

De esta manera se sobrescribe el método mostrarLado y se adecúa a la clase Arista.

También existe otra clase importante en la herencia de Kotlin, estas son las interfaces, que son creadas cuando con la intención de que una acción tenga diferentes efectos dependiendo de en la clase que se use, este tipo de estructura no pueden tener constructor a diferencia de las clases abstractas, ni ser instanciadas, son muy generales, pero pueden ser declarados dentro de ellas atributos y funciones sin definir, una clase puede heredar de varias interfaces.

iv) Compiladores de Kotlin: al ser un lenguaje de programación fuertemente inspirado en Java, hace uso de la JVM, por lo que su proceso de compilación es similar al de los programas de Java, además al ser un lenguaje de tipado estático, brinda todas las ventajas que traen consigo los lenguajes compilados, entre las que se cuentan, rapidez de ejecución y detección de algunos errores de tipo a la hora de realizar la compilación.

Frameworks de Kotlin: al ser un lenguaje tan popular y siendo considerado un lenguaje a la par con Java, Kotlin cuenta con una gran variedad de Frameworks, algunos de los cuales comparte, con Java, algunos de ellos son: Ktor (un framework para la rápida creación de aplicaciones web con esfuerzo mínimo), Kweb (un framework de creación de sitios web en Kotlin), Javalin (un framework web ligero, para Kotlin y Java que soporta WebSockets, HTTP2 y peticiones async), Spark (para la creación de aplicaciones web en Kotlin y Java), Spring Boot, Vaadin-On-Kotlin (es un framework de aplicaciones web que incluye todo lo necesario para crear bases de datos para aplicaciones web), Jooby (framewrok de Java y Kotlin para desarrollo usando Maven o Gradle), Vert.x for Kotlin, Tekniw y HexaGon (un framework de microservicio que cuida de HTTP, serialización y almacenamiento).

Entornos de desarrollo de Kotlin: La gran mayoría de los editores de texto tienen soporte para Kotlin y muchos paquetes que mejoran la experiencia al programar en este lenguaje, sin embargo, algunos de los entornos de desarrollo de este lenguaje por todas las funcionalidades que aportan son: IntelliJ IDEA (creado por la empresa JetBrains, desarrolladora de Kotlin, contiene herramientas de debugging y profiling), Eclipse (contiene soporte para Kotlin al estar basado en Java) y Visual Studio Code (que contiene muchas extensiones y complementos para este lenguaje de programación).