## EJERCICIO 1

1.a.i) Para crear objetos es suficiente escribir “object <nombre> {cuerpo}”, donde nombre es como se llamara el objeto y en cuerpo se definen las variables internas, metodos, etc.

Para modificar objetos, suponiendo que existen variables internas, solo es necesario escribir lo siguiente: “<nombre>.<variable> = <nuevo valor>”, donde variable es el nombre de la variable a modificar y nuevo valor es el valor a asignar a la variable.

Por otra parte, no podemos modificar los métodos definidos para un objeto, pero si sobreescribirlo para un objeto hijo anónimo que definamos de la siguiente manera:

```
var obj = object : constructorObjetoPadre() {
        override fun nombreDelMetodoAModificar(){
                super.nombreDelMetodoAModificar()
        }
}
```

Donde “object : constructorObjetoPadre()” define al objeto como hijo del objeto padre.

Luego tenemos las clases, para las cuales definimos de la siguiente manera:

<prefijos> class <nombre> (<argumentos>) : <superclase> {<cuerpo>}

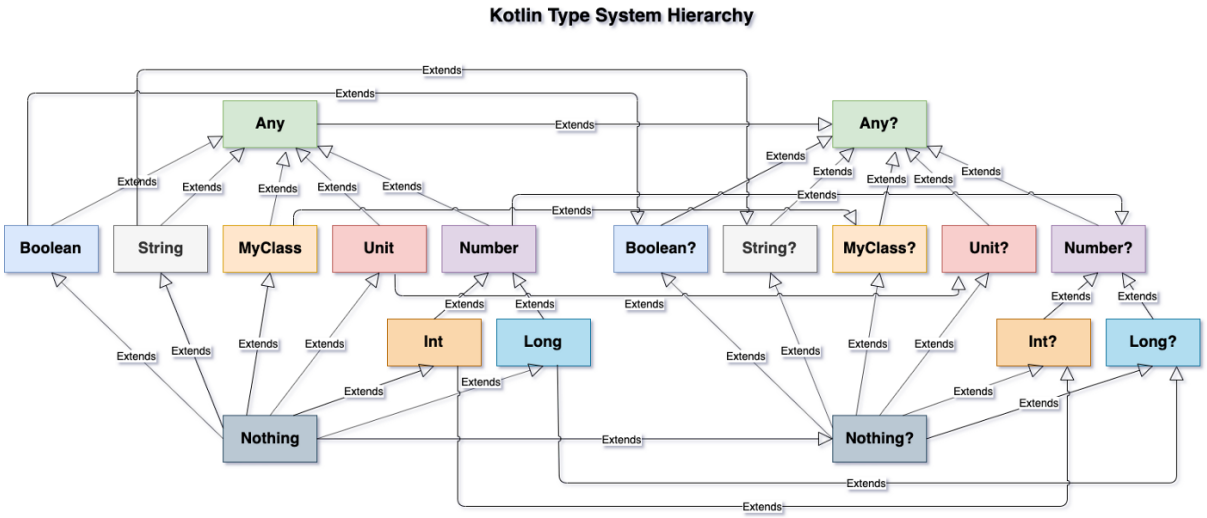
Donde los prefijos son una lista de palabras clave como public, abstract, open, etc; los argumentos son una secuencia de “<var/val> <nombre>: <tipo>” separados por comas y el cuerpo contiene todas las declaraciones de variables y métodos internos de la clase.

No es necesario el “: <superclase> si no se busca heredar métodos de una superclase.

1.a.ii) El manejo de memoria en Kotlin se hace de manera automática. El lenguaje utiliza el método de Garbage Collection por “deferred reference counting”, de manera que las se revisan las referencias a un objeto desde otros objetos y se pasan por alto las referencias a variables del programa; si el número de referencias es igual a 0, entonces el objeto pasa a ser basura.

1.a.iii) Kotlin posee asociación dinámica de métodos, con la posibilidad de definir métodos estáticos con la funcionalidad “companion”, la cual funciona como el “static” en java pero Kotlin la utiliza de manera que el código mantenga mayor nivel de legibilidad, puesto que solo hace falta declarar el objeto como “companion”, todos los métodos dentro del objeto como estáticos (En java tendría que declarar cada método como static para el mismo objetivo).

1.a.iv) Jerarquia de tipos:




Kotlin no tiene soporte para herencia múltiple, sin embargo, además de extender a su padre, una clase puede implementar varias interfaces.

El lenguaje no posee polimorfismo paramétrico, siendo muy sencillo verlo en el funcionamiento de “println()”, puesto que si el argumento es un string, se imprimirá el string en pantalla, pero si se pasa como argumento un entero, tambien se imprimira en pantalla sin necesidad de hacer una conversión **explícita** a string; sin embargo, todos los tipos en Kotlin tienen un método toString(), lo cual es utilizado para imprimir los elementos de un tipo determinado. Si una clase se define sin un método toString() y se intenta imprimir una instancia de la clase no se obtendrá el resultado esperado.

Por último, Kotlin permite la varianza, dado que es posible definir métodos de igual nombre en una clase, siempre y cuando no tengan exactamente la misma definición de parámetros. Si los parámetros para la primera clase son Int y String, para la segunda no pueden ser los mismos en ese mismo orden, podría utilizar más argumentos (o menos) o simplemente cambiar el orden en que aparecen en la lista de argumentos.

Mas graficamente:

Este ejemplo no funciona:

```
public class clase {
        fun hola(a: String, b: Int){
        }

        fun hola(z: String, q: Int){
        }
}
```

Este ejemplo si funciona:

```
public class clase {
        fun hola(a: String, b: Int){
        }

        fun hola(q: Int, z: String){
        }
}
```

## EJERCICIO 2

**Lenguaje utilizado: Scala (Diego)**

2a.i) Capacidades nativas para concurrencia:

Scala tiene propiedades nativas para concurrencia, puesto que posee tipos como los Futuros y las promesas, los cuales funcionan para crear objetos futuros de manera asíncrona (en otra tarea). Posee también métodos para el tipo de sincronía que se requiera, ya sean métodos síncronos con bloqueos o sin ellos o asíncronos.

2.a.ii) Creación/manejo de tareas concurrentes:

En Scala, la creación de tareas concurrentes está definida por el uso de los futuros o promesas, que son funcionalidades del lenguaje para indicar que determinada tarea se quiere realizar en un hilo distinto sin bloquear el principal, dejando un espacio para un valor que puede o no existir aún en determinado momento de la ejecución del hilo principal.

Para manejar la memoria compartida en el lenguaje es importante utilizar locks, los cuales son provistos por el lenguaje, es decir, hay que proteger las regiones críticas manualmente.

2.a.iii) Mecanismo de sincronización:

El lenguaje permite tareas multihilos de manera sincrona y asincrona. Depende de la manera en que se llame a los métodos a utilizar; por ejemplo, si se utiliza Futuros el proceso se realizará de manera asíncrona.