Proyecto Tienda Bedu: Documentación

Equipo 2

Integrantes:

- Alan Sandoval León (sandoval.leon.alan@gmail.com)
- Luis Fernando Pedraza Estañol (luisfedranol@gmail.com)
- Marco Antonio Mojica Martinez (mojicamarc@gmail.com)
- Carlos Armando Morales Bautista (carloamoraleb@gmail.com)

Índice

Introducción

Como parte de la evaluación del módulo ***Kotlin Fundamentals*** se desarrolló un proyecto que lleva de nombre **Tienda Bedu**, el cual tiene como objetivo poner en práctica todos los temas vistos durante las distintas sesiones impartidas en esta fase 2 del curso, y aplicarlas a un caso real.

Los temas abarcan desde declaración de variables hasta programación asíncrona, los cuales dan pauta para el desarrollo de una aplicación de una tienda online móvil que permite vender productos de tipo ropa, calzado y hogar.

La aplicación se basa en un ambiente de tipo consola, con menús de tipo funciones que llaman a los diferentes objetos del programa, como son: orden, producto, usuario, inventario e impuestos.

Definición del problema

Un negocio se encuentra con la necesidad de un programa que permita a los clientes navegar entre menús, seleccionar artículos disponibles en su inventario y poder realizar sus compras, todo ello que esté al alcance de sus manos. Además, que dicho programa también permita a los empleados del negocio poder monitorear y actualizar la lista de productos con los que cuenta, incluyendo reabastecer stock, agregar productos, modificarlos o incluso eliminarlos.

Dadas las necesidades, tanto de clientes como empleados, se requiere desarrollar un programa que pueda realizar diferentes tareas y que tenga en cuenta las siguientes reglas del negocio:

- Los usuarios que usen el programa deben ser capaces de registrar una cuenta haciendo uso de sus datos personales, y con ella permitirles acceder a sus características mediante un inicio de sesión.

- El usuario debe de tener la capacidad de navegar entre las diferentes funciones del programa, y poder acceder a ella si así lo requiere.

- Todo usuario de la aplicación encargada de las tareas relacionadas con el inventario de productos deberá ser capaz de acceder a su información, el cual debe de accederse con facilidad mediante un identificador y también debe de tener la capacidad de agregar, modificar o eliminar cualquier producto.

- Los clientes deben de poder agregar productos a un carrito de compras, el programa deberá mostrarles el total a pagar por dichos productos más el Impuesto al Valor Agregado (IVA), y en base a toda esa información, generar una orden de venta e imprimir un ticket de compra.

Descripción detallada de componentes

De acuerdo a los requerimientos mencionados, se plantea esta arquitectura en un diagrama de clases:

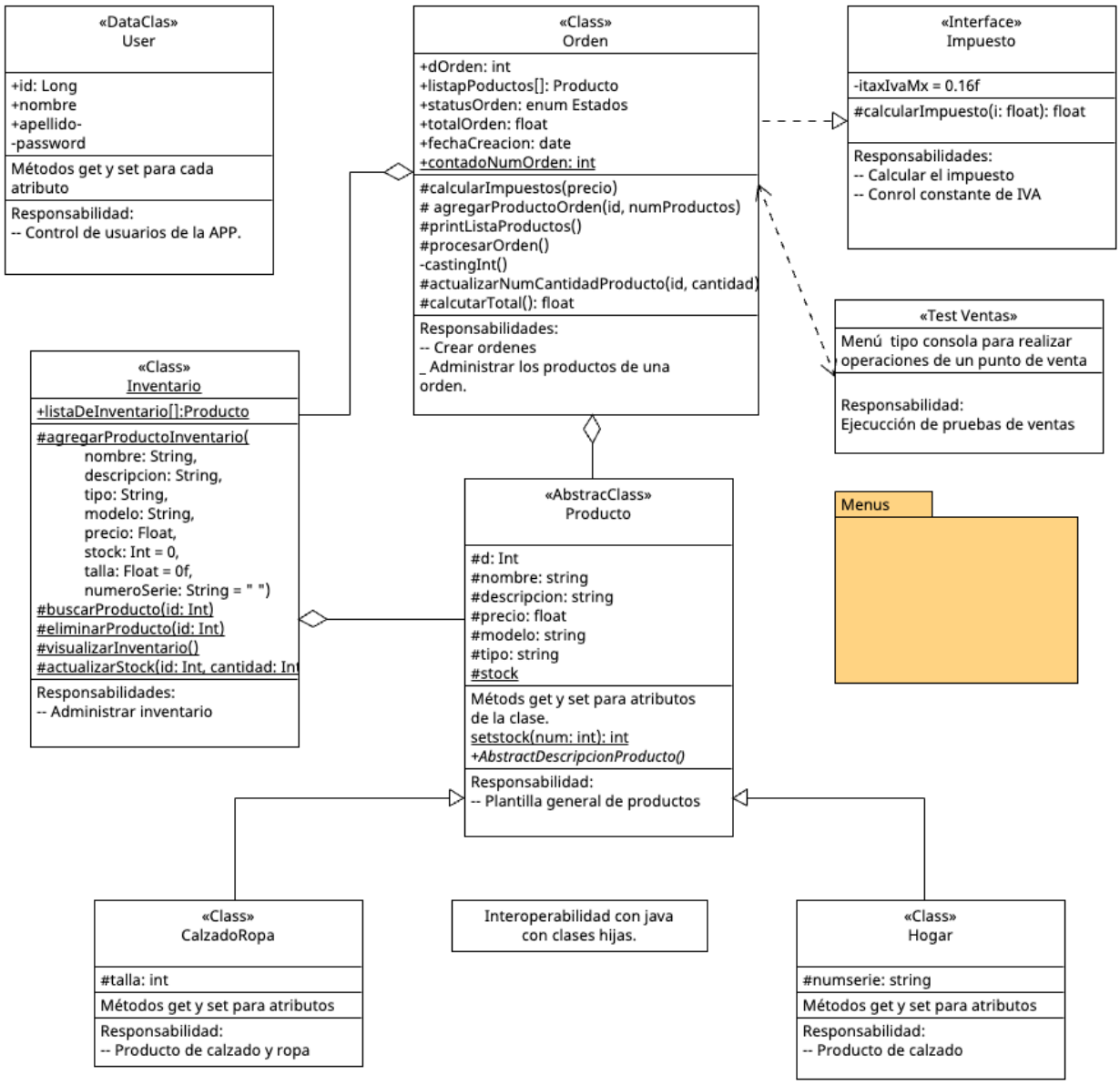

Figura 1: Diagrama de clases del proyecto.

Descripción de las clases

Clase usuario: Nos permite administrar los usuarios que entran a la aplicación mediante un ID y un password. Esta es una clase de tipo “data class”.

Clase producto: Esta es una clase de tipo abstracta, cuya función principal es definir las características principales de un producto, las cuales son heredadas a la clase *calzadoRopa* y *Hogar*.

Clase calzadoRopa: Es una clase en la que se aplica herencia y cuya propiedad se diferencia en la talla.

Clase hogar: Es una clase en la que se aplica herencia y cuya propiedad se diferencia en el número de serie.

Clase inventario: Su función principal es almacenar y administrar los productos disponibles en la tienda. Esta clase es de tipo *static*.

Clase orden: Administra las órdenes de venta de producto, permite agregar, procesar y calcular el costo total del despacho de productos; así como generar el ticket de venta correspondiente.

Clase impuesto: Esta clase de tipo *interface* en el cual tiene una constante de IVA y un método para calcular impuestos que hereda a la clase *orden*.

Desarrollo

Variables y tipos de datos

En la imagen número 2 se observa la declaración de varios tipos de de datos como flotante y string la cuales son inmutables con la palabra reservada val.

```
var tallaProducto = 0f
val serieProducto = " "
print("Nombre del producto:")
val nombreProducto: String = readln()
print("Descripción del producto:")
val descripcionProducto: String = readln()
print("Modelo del producto:")
val modeloProducto: String = readln()
```

Figura 2: Declaración de variables del proyecto.

Funciones, condicionales, ciclos y estructura de datos

Se hace uso de de condicionales de tipo selección en la figura 3 con if y else if con de identificar si es un objeto de tipo CalzadoRopa o Hogar.

.

```
if (tipoProducto == "calzado" || tipoProducto == "ropa") {
    print("Talla de producto:")
    tallaProducto = readln().toFloat()
} else if (tipoProducto == "hogar") {
    print("Número de serie:")
    val serieProducto: String = readln()
}
```

Figura 3: Condicionales utilizados en el proyecto.

```
do {
    println("1. Agregar producto")
    println("2. Visualizar inventario")
    println("3. Eliminar producto")
    println("4. Actualizar Stock")
    println("5. Salir")
    print("Ingrese una opción: ")
    try {
        opc = readLine()?.toInt() as Int
        when (opc) {
            1 -> agregarProducto()
            2 -> visulizarInventario()
            3 -> eliminarProducto()
            4 -> actualizarInventario()
            5 -> println("Saliendo del menu")
            else -> {
                println("Opción no válida")
            }
        }
    } catch (exception: NumberFormatException) {
        opc = 0
        println("Solo acepta valores del 1..4")

    }
} while (opc != 5)
```

Figura 4: Uso del ciclo do while.

En la figura anterior número 4 se aplica un do while para poder hacer un menú de tipo consola en la aplicación.

```
fun eliminarProducto() {
    print("Identificador del producto:")
    val identificadorProducto: Int = readln().toInt()

    Inventario.eliminarProducto(id = identificadorProducto)
    println("Producto eliminado")

}
```

Figura 5: Uso de una función para el método estático.

En la *figura 5* se puede observar el ejemplo de uso de una función la cual permite usar un método estático de la clase inventario.

La estructura de datos utilizada en el proyecto consiste en un conjunto de listas y diccionarios, las cuales son mutables y nos permiten almacenar los datos de los usuarios y de los productos. En las *figuras 6 y 7* podemos observar que tanto *arregloUser* como *listaProducto* están declarados como listas linkeadas.

```
var arregloUser = LinkedList<User>()
```

Figura 6: Declaración de una lista de tipo LinkedList.

```
private var objetoProducto: MutableMap<String, Any> = mutableMapOf()
var listaProducto = LinkedList<MutableMap<String, Any>>()
```

Figura 7: Declaración de listaProducto y de objetoProducto.

Para agregar un elemento, por ejemplo un objeto de tipo usuario a lista se usa el método addLast al final de lista, como se observa en la *figura 8*.

```
arregloUser.addLast(usuario)
```

Figura 8: Agregando un elemento a arregloUser, usando el método addLast().

Clases, objetos y constructores

En las siguientes imágenes se muestra el uso de la clase orden, la cual hereda de la interfaz Impuesto y se muestra la declaración de atributos en su constructor. En la *figura 9* podemos ver la creación de la clase *Orden*, mientras que en la *figura 10* se realiza una instancia de dicha clase.

```
class Orden(
    var noOrden: Int = 0,
) : Impuesto {

    private var objetoProducto: MutableMap<String, Any> = mutableMapOf()
    var listaProducto = LinkedList<MutableMap<String, Any>>()
    var statusOrden = Estados.PENDIENTE
    private var totalOrden = 0f
```

Figura 9: Declaración de la clase Orden con su correspondiente constructor.

```
val orden = Orden()
```

Figura 10: Instancia de la clase Orden.

Getters, setters y modificadores de acceso

Debido a que en kotlin los getters y setters están declarados de manera implícita, mostramos cómo se implementan desde la clase CalzadoRopa desarrollada en java. En la *figura 11* se hace uso de modificadores de acceso *public*.

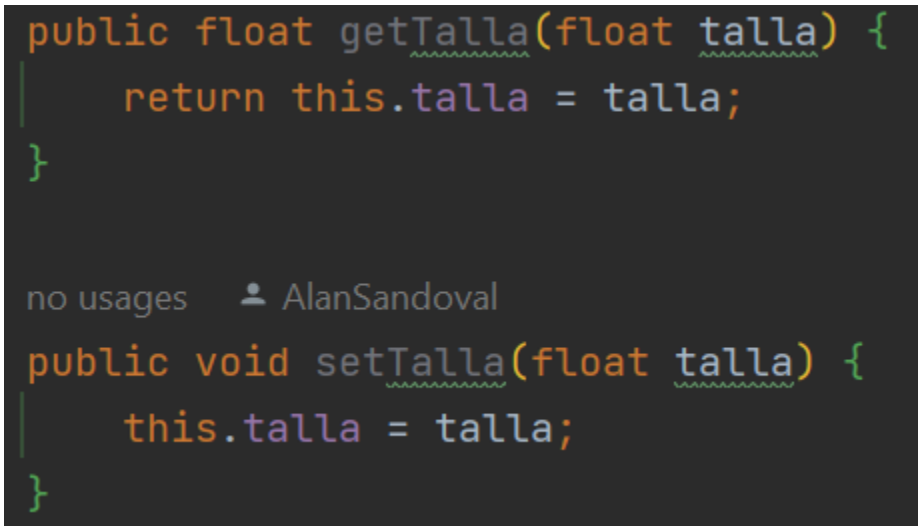

Figura 11: Getter y setter de un atributo de la clase CalzadoRopa.

Herencia, polimorfismo, clases abstractas e interfaces, data class y companion object

Parte de la programación orientada a objetos consiste en aplicar los pilares de dicho paradigma, los cuales son la encapsulación, la herencia, el polimorfismo y la abstracción, este último concepto se aplica en el proyecto creando la clase *Producto.*

Como tal, un producto contiene un identificador, un nombre, una descripción, un precio, entre otras características, pero “producto” es un concepto muy general que engloba a todo artículo que se pueda vender. Por lo tanto, es ideal considerar “producto” como un objeto abstracto, que se puede interpretar de muchas maneras.

Basándonos en ese pensamiento, se ha optado por crear una clase *Producto* en el proyecto, la cual será una clase abstracta, en la *figura 12* se puede observar la declaración de dicha clase con sus respectivos atributos.

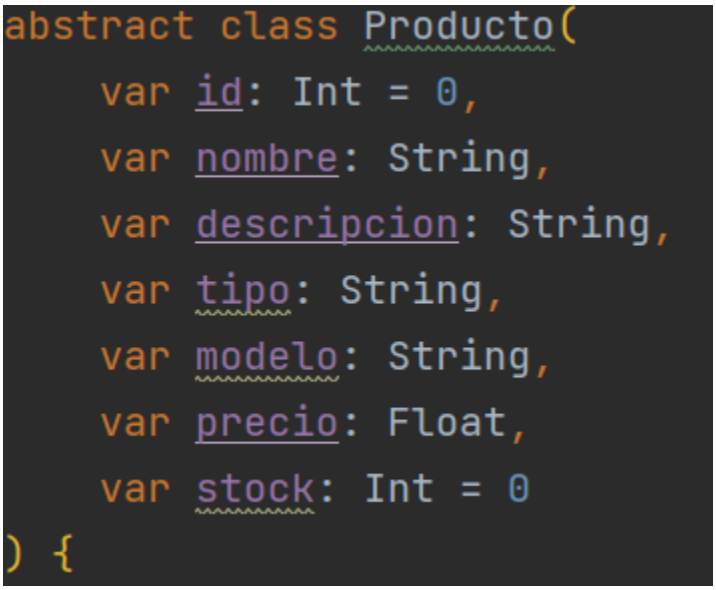

Figura 12: Creación de la clase abstracta Producto.

Una vez que se tenga creada la clase Producto, ahora es posible considerar los diferentes tipos de producto que existan y se puedan vender en la aplicación, para el caso de este proyecto, se dará un enfoque a los siguientes tipos de productos: productos de calzado y ropa y productos para el hogar.

En base a ello, se crea la clase *CalzadoRopa* y la clase *Hogar*, la cual heredan de la clase abstracta *Producto*. En la *figura 13* se puede ver la declaración de la clase *CalzadoRopa*.

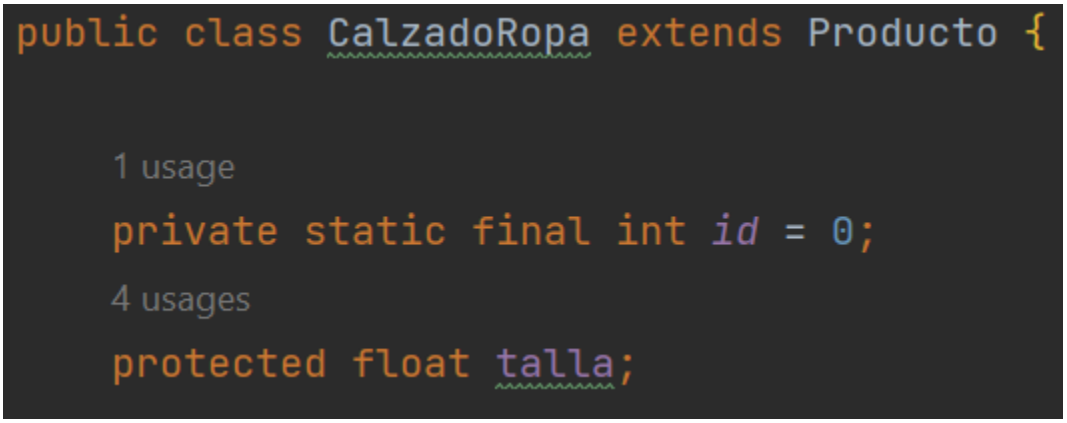

Figura 13: Creación de la clase CalzadoRopa, que hereda de Producto.

Una vez creada la clase, ahora se procede a sobreescribir el método *descripcionProducto*, la cual fue creada como una función abstracta en la clase *Producto*.

Se aplican cambios del comportamiento al método descripciónProducto en las clases hijas *CalzadoRopa* y *Hogar*, como se puede ver de ejemplo en la *figura 14*.

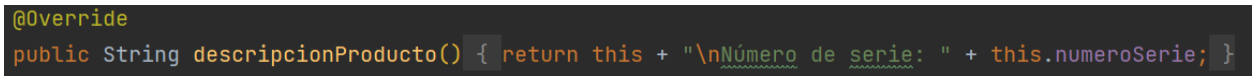

Figura 14: Ejemplo de polimorfismo, donde se reescribe el método descripcionProducto().

Ya que se terminaron de definir las clases relacionadas con el producto, ahora se procede a crear una clase *Inventario*, la cual permite almacenar todos los productos. En la *figura 15* se ve la declaración de la clase *Inventario*, junto con un companion object que define una variable de tipo lista que lleva el nombre de *listaDeInventario*.

El companion object permite crear una lista de inventario sin necesidad de instanciar la clase *Inventario*.

```
class Inventario {
    mojicamarcumplo +1 *
    companion object {
        var listaDeInventario = LinkedList<Producto>()

        mojicamarcumplo
        fun agregarInventario(producto: Producto) {
            listaDeInventario.addLast(producto)
        }
```

Figura 15: Creación de la clase Inventario, con su correspondiente companion object.

En la definición del problema también se hizo mención de la necesidad de la creación de usuarios, y que estos se puedan almacenar. Como en este caso la creación de usuarios no requiere de realizar métodos ajenos a los getters y setters, se optó por hacer de la clase *User* una data class, la cual sólo contendrá la información típica de un usuario: el nombre, el correo, la contraseña, entre otros. En la *figura 16* se ve la creación de la data class.

Dentro de la data class se encuentra un companion object que sirve principalmente como un contador de usuarios, esto con el objetivo de llevar un control con el número de usuarios. En la *figura 17* se muestra esa fracción de código.

```
data class User(
    var id: Int = 0,
    var nombre: String,
    var apellido: String,
    var email: String,
    var password: String
) {
```

Figura 16: Data class User.

Figura 17: Companion object de la data class User.

Dentro del proyecto se ha definido que para el cálculo del total de un pedido de compra este debe incluir el Impuesto al Valor Agregado, el cual representa un 16% adicional del valor del producto.

Para simplificar la aplicación del impuesto se ha optado por colocarlo dentro de una interface, dentro de ella se encuentra el valor del impuesto, la cual este permanecerá constante dentro de un companion object, y un método abstracto para calcular el impuesto, el cual recibe como parámetro el valor del producto. En la *figura 18* se encuentra la fracción de código relacionado a la interfaz.

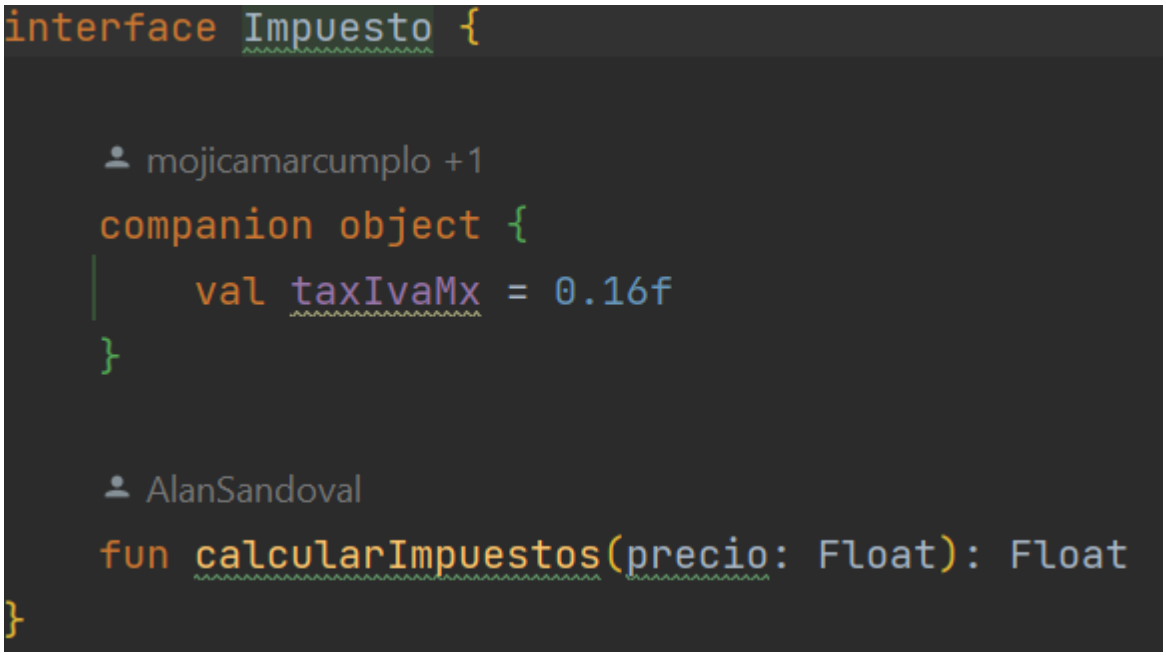

Figura 18: Interface Impuesto.

Programación funcional

Se han aplicado los conocimientos de la programación funcional de nuestro proyecto, todo esto a través del uso de lambdas y funciones como forEach, para iterar entre los objetos almacenados en una lista y filter, para la búsqueda de objetos con los Id´s de cada producto. En la *figura 19* se muestra el ejemplo de uso del forEach, en la *figura 20* se ha hecho uso de una función inline en la función buscarProducto y en la *figura 21* para la función buscarOrden.

```
listaProducto.forEach { it: MutableMap<String, Any>
    val idProducto : Int? = it["_id"]?.let { it1 -> castingInt(it1) }
    val cantidadProducto : Int? = it["Cantidad"]?.let { it2 -> castingInt(it2) }
```

Figura 19: Uso de forEach y funciones lambda.

```
fun buscarProducto(id: Int): List<Producto> = listaDeInventario.filter { Producto -> Producto.id == id }
```

Figura 20: Función inline.

```
fun buscarOrden(noOrden: Int): List<Orden> {
    return arregloOrden.filter { Orden -> Orden.noOrden == noOrden }
}
```

Figura 21: Uso de función inline para el filtro de ordenes en buscarOrden.

Interoperabilidad Kotlin-Java

Retomando la clase Producto, previamente se han creado dos clases que hereden de esta misma: *Hogar* y *CalzadoRopa*. Dichas clases han sido escrita en el lenguaje Java, demostrando la capacidad que tienen estos lenguajes Interoperar entre ellos. En la *figura 22* se muestra la clase Hogar, que contiene sintaxis del lenguaje Java, y en la *figura 23* se puede ver cómo un archivo de Kotlin puede instanciar sin problemas dicha clase.

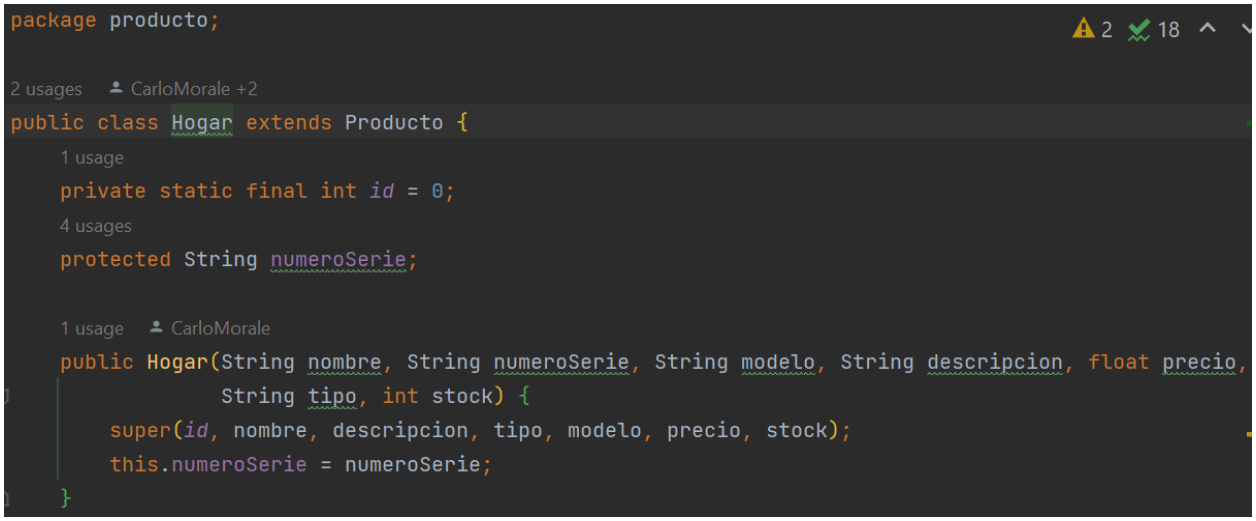

Figura 22: clase Hogar escrita en Java.

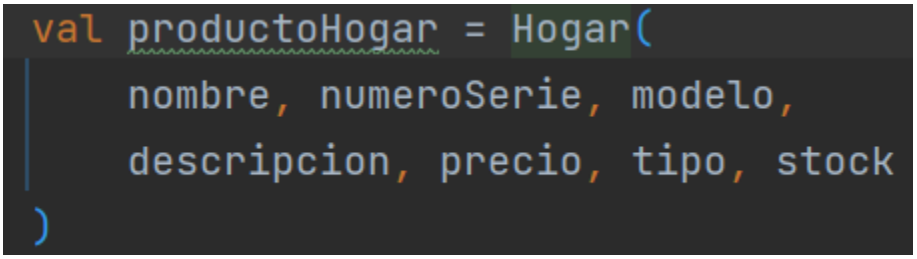

Figura 23: instancia de clase hogar en un archivo Kotlin.

Manejo de errores

Todo programa que se desarrolla está propenso a tener errores, ya sea en tiempo de compilación o de ejecución, y muchas veces el programa no termina de compilar correctamente porque se cierta parte del código no termina de ejecutar bien y termina lanzándose una excepción, nuestro trabajo es prevenir en la mayor medida posible que sucedan este tipo de errores.

Parte del trabajo en el manejo de errores consiste en que los desarrolladores tienen que prever casos en los que el programa puede llegar a fallar, y tras ello ver una posible solución para solucionar o mitigar el error.

Para el manejo de errores en la aplicación se hace uso de *throw* para lanzar una exception personalizada llamada Error cuando la existencia en el stock de un valor negativo. En la figura 24 se observa dicha declaración.

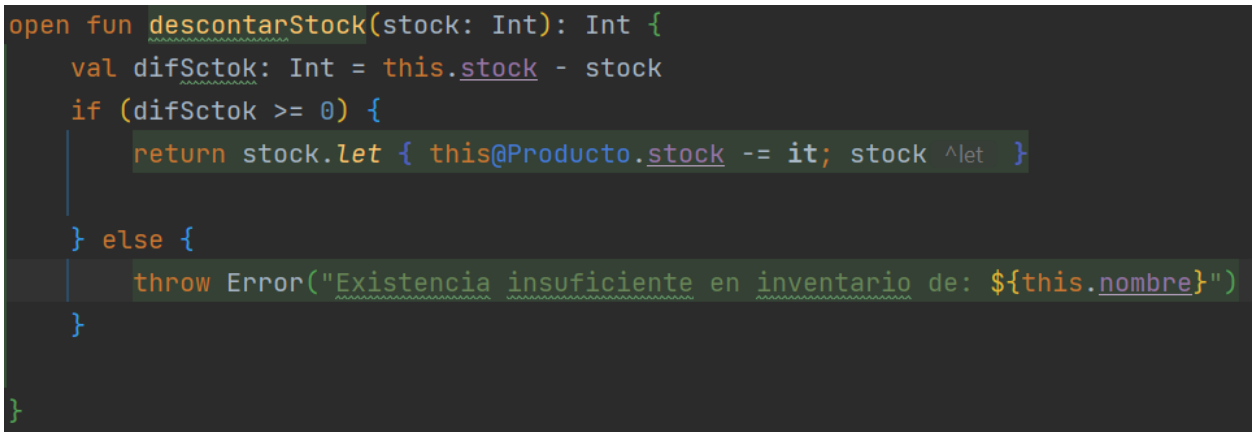

Figura 24: Declaración para lanzamiento de excepción en descontarStock.

De igual manera, se puede prevenir la entrada de datos nulos haciendo uso del *null safety*, en la *figura 25* se puede ver un fragmento de código donde el operador **'?'** se usa para prevenir la entrada de datos nulos.

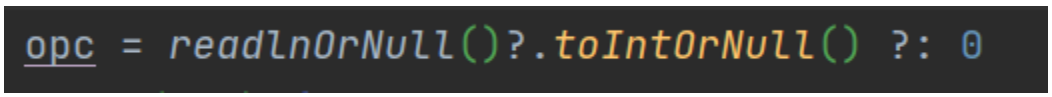

Figura 25: Uso del Null Safety.

Otro de los métodos con los que se cuenta para tratar con errores es utilizar los bloques *try-catch*, este bloque permite intentar ejecutar una parte del código, y en caso de que no se obtenga el resultado esperado, se ejecutará un bloque donde se “atrapa” el error y se lanza una excepción, y ejecuta una porción de código para tratar con dicho problema. En la *figura 26* se hace uso de dicho bloque para prevenir posibles errores al momento de intentar realizar un cambio a los datos del usuario.

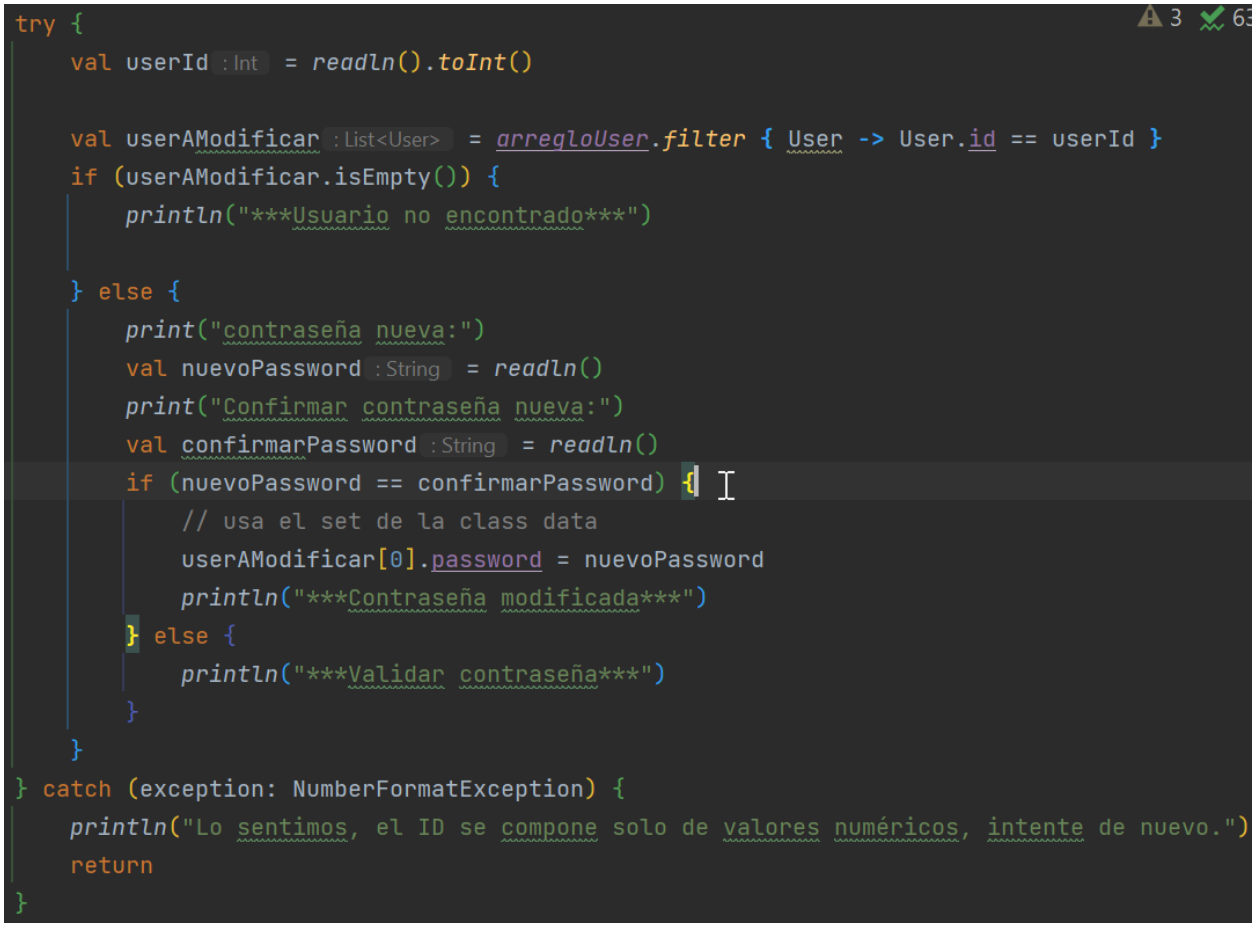

Figura 26: Uso del bloque try-catch para la modificación de datos de usuario.

Programación asincrona

En la figura 27 se puede ver que se utilizó RunBlocking para bloquear nuestro hilo principal de la aplicación y simular el procesamiento de impresión del ticket de la orden de la aplicación.

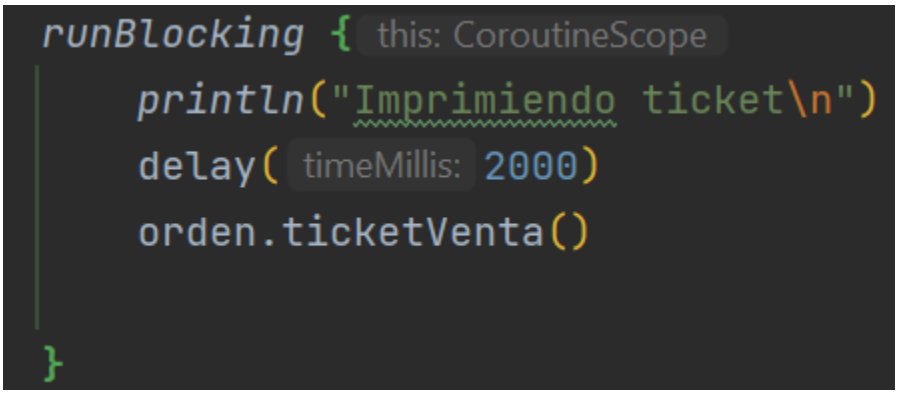

Figura 27: Uso de runBlocking.

Conclusiones

Más allá de la programación este proyecto ayudó a ver más allá de lo que implica desarrollar un programa, desde la planificación, definición de objetivos, alcance del proyecto, hasta la parte de la codificación y las pruebas. El practicar todos estos elementos nos ayuda a dar una resolución del problema proponiendo soluciones que se ajusten y satisfagan las demandas con nuestro producto, el cual sería el software.

La realización de este proyecto fomentó la comunicación y el trabajo en equipo entre nuestros compañeros, ayudando a desarrollar esa soft skill que es de mucha importancia al momento de realizar cualquier tipo de proyecto.

En la parte técnica este proyecto ha podido ayudar en la práctica de las hard skills de programación en Kotlin, aplicando todo lo que se ha aprendido durante este módulo del curso, desde cómo declarar variables y funciones hasta aplicar conceptos como la programación funcional, el manejo de errores y la programación asíncrona para adaptar una mejor solución a la problemática previamente planteada, facilitando también el flujo de trabajo.