PROYECTO GESTIÓN DE DIETAS.

1. **Objetivo de la aplicación**. Gestionar la entrada, edición y consulta de datos referidos a elementos nutricionales. Partiremos de elementos básicos (simples o procesados) a partir de la información de la etiqueta y crearemos recetas, menús y dietas, de manera que tengamos una información global de todos ellos referidos a gramos de proteínas, Hidratos de carbono, lípidos (grasas) , cantidades y Kcalorías.
2. **Objetivos del proyecto.**
    1. Estudiar la manera más adecuada de implementar algoritmos, estructuras de datos y P.O.O en Kotlin.
    2. Persistencia en archivos binarios, bases de datos con Room y conexión a BD no SQL con Firebase/Firestore. Analizando la mejor manera de estructurar la información
    3. Crear una interfaz de usuario en Android con JetPack Compose.
    4. Analizar los patrones de diseño y arquitecturas necesarias para hacer el proyecto facilmente escalable

FASE 1. Estudio de dos implementaciones distintas para el modelo de datos.

Una característica importante de este proyecto es que los datos se relacionan entre si no solo de una manera estática, con asociaciones entre ellos. También hay una relación dinámica entre los mismos, ya que los cálculos de unos se basan o dependen de otros que tienen asociados.

A partir de ahora tendremos lo que llamamos un **Componente de la dieta** como un elemento que se relaciona con otros (puede ser un alimento simple/procesado, receta, menú o dieta). La relacion de un Componente Dieta (C.D) se hace a traves de lo que llamaremos **Ingrediente** que tiene asociado una cantidad.
Por tanto un C.D tendrá otros C.D como ingredientes con una cierta cantidad (llamaremos hijos) y un C.D a su vez es ingrediente de otros C.D que llamaremos padres. Los alimentos simples/procesados no tienen elementos hijos o ingredientes.

Dicho ésto, podremos establecer, dos maneras de abordar el modelo:

1. **Modelo 1. Basado en el recursividad para la actualización de los cálculos.** Esto implica que cuando un C.D se actualiza, se hará una actualización recursiva hacia atrás, es decir, sus padres mediante la función backUpdate(). Esto implica dos cosas:
    1. Todos los C.D guardan sus datos como atributos.
    2. Los cambios siempre se guardan actualizados.
2. **Modelo 2. Basado en el concepto de propiedades derivadas.** En este modelo, los valores nutricionales no se guardarán de forma explícita, sino que se calcularán cuando se soliciten a través de metodos get(). Se actualizarán los padres a través de los datos de los hijos en cascada y tendremos los datos actualizados pero **No se guardarán.**

Sin entrar en detalles, diremos que el modelo1 es más intuitivo y tradicional cuando se espera que estén todos los valores calculado y se utilizaría siempre que queramos guardar toda la información actualizada para ser leída de manera pasiva: Una página HTML estática, un documento de texto, etc. En cambio el modelo 2 sería más útil para una aplicación dinámica que va mostrando la información según se va haciendo las peticiones al sistema.
En el caso de querer guardar la información en BD en principio podríamos pensar que necesitaríamos guardar toda la información en campos de la BD, por lo que necesitaríamos el modelo 1. Sin embargo como ya veremos, podremos usar BD reactivas que permiten hacer los cálculos necesarios para actualizar los datos en tiempo real.
Por todo ello, a partir de ahora, seguiremos desarrollando el proyecto usando el **modelo 2**

FASE 2: Implementación de la Gestión de dietas con Ficheros. Aspectos a tener en cuenta:

1. Se trabaja en el proyecto solo con ficheros, por lo que no se mezclará con otras tecnologías de persistencia.
2. El código de manejo con ficheros cambia ligeramente si es en Android o en una aplicación en PC. Pero la lógica es la misma.
3. Se ha adoptado el enfoque del **modelo$_2$** como ya se dijo. Y el tratamiento que se va a dar a las entidades es enteramente como objetos y listas de objetos, ya que solo es un proyecto para probar la lógica de negocio y el interfaz de usuario que se diseñe.
4. La arquitectura tendrá las siguientes capas:
    1. Clase BDFichero que es el que se encarga en última instancia de guardar y recuperar la lista de ComponentesDieta.
    2. DaoCD y DaoIngrediente son los encargados de implementar el CRUD tanto para ComponentesDieta como Ingredientes que actuará sobre una lista de ComponentesDieta.
    3. De momento no existen Repositorios o algo parecido. Los repositorios son una capa de abstracción por encima de los DAO que permiten hacer las funcionalidades de los DAO pero sin preocuparse por la tecnología subyacente. (Ficheros, Room, FireBase, ApiRest…) En este caso, como solo tenemos Ficheros, no es necesaria.
    4. Capa de servicio, usa Daos (o repositorios si los hubiese) para añadir lógica de negocio de la aplicación. En el proyecto de prueba del modeloDietasFicheros, dicha capa de servicio no existe como tal. Solo un método: **fun mostrar(daoCD:IDaoCD)** hace una función de Servicio.