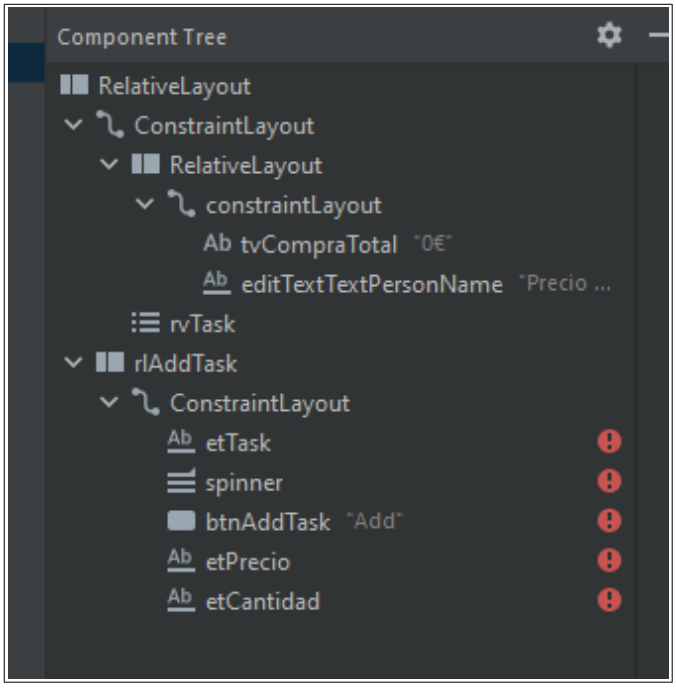
DAO DB ROOM

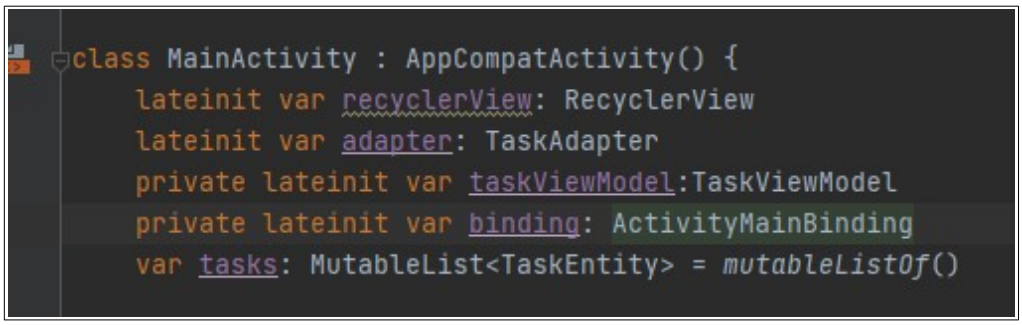
1-Lo primero como siempre es tener las dependencias agregadas y el viewBinding activado

```
    buildFeatures{

        viewBinding true
    }
}

dependencies {

    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.4.0'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.2'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'

    implementation "com.google.android.material:material:$material_version"

    //RecycleView
    implementation("androidx.recyclerview:recyclerview:$recycler_version")

    //Room
    implementation "androidx.room:room-runtime:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
    implementation "androidx.room:room-ktx:$room_version"

    // Lifecycle libraries
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
}
```

2-Ahora vamos a empezar por el MainActivity,pero primero copia su XML



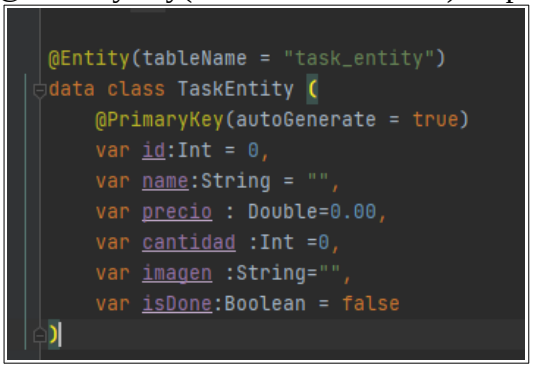
3-Ahora vamos a empezar con la declaracion de las variables donde tenemos lo de siempre:
-RecyclerView
-Adapter RecyclerView
-ViewModel
-Binding
-Lista Mutable de Tareas(compras)



```
class MainActivity : AppCompatActivity() {
    lateinit var recyclerView: RecyclerView
    lateinit var adapter: TaskAdapter
    private lateinit var taskViewModel:TaskViewModel
    private lateinit var binding: ActivityMainBinding
    var tasks: MutableList<TaskEntity> = mutableListOf()
```

4-Este ultimo es una DataClass un tanto particular,ya que es una data class orientada a ser utilizada en una base de datos ya que tiene dos cosas diferenciales:

-@Entity(tableName = "task_entity") → Nombre tabla en DB
-@PrimaryKey(autoGenerate = true) → primary key autogenerada en DB



```
@Entity(tableName = "task_entity")
data class TaskEntity (
    @PrimaryKey(autoGenerate = true)
    var id:Int = 0,
    var name:String = "",
    var precio : Double=0.00,
    var cantidad :Int =0,
    var imagen :String="",
    var isDone:Boolean = false
)
```

5-Ahora aquí vamos a tener una parte del OnCreate ,si que es cierto que es MUY largo pero nosotros ahora vamos a ver un trozito y el cual infla el binding y crea el ViewModel.
Vamos a seguir el hilo de las operaciones.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(ActivityMainBinding.inflate(layoutInflater).also {  it: ActivityMainBinding
        binding = it
    }.root)

    taskViewModel = ViewModelProvider( owner: this)[TaskViewModel::class.java]
```

6-Aquí en el viewModel tenemos lo que tenemos asi siemppre:
-Contexto
-DAO
-Variables Mutables que observan desde el MainActivity

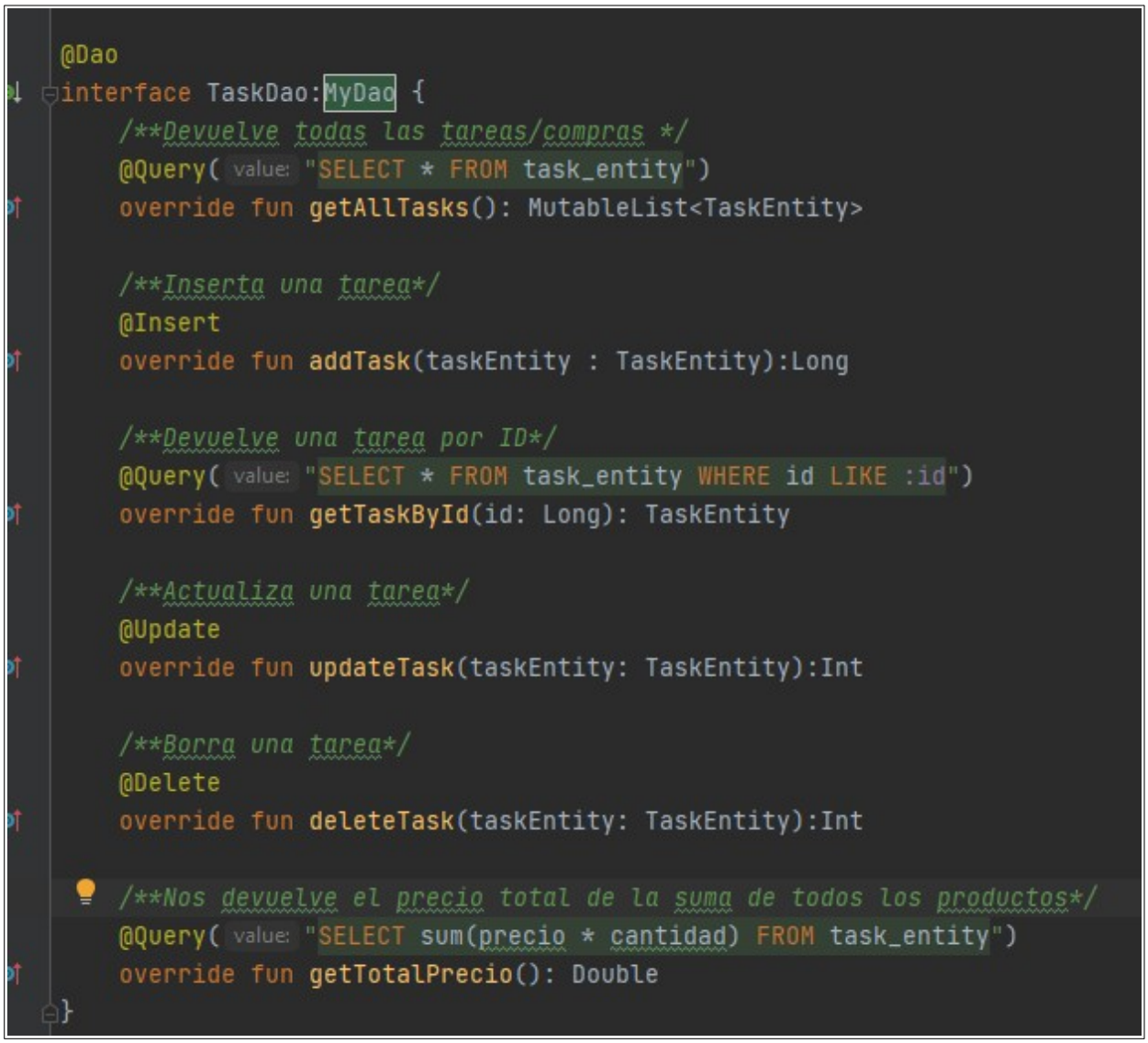
este myDao no es un DaoFile ni un daoShared, es un DB.

```
    val context = application
    var myDao: MyDao = TasksDatabase.getInstance(context)

    val taskListLD:MutableLiveData<MutableList<TaskEntity>> = MutableLiveData()
    val updateTaskLD:MutableLiveData<TaskEntity?> = MutableLiveData()
    val deleteTaskLD:MutableLiveData<Int> = MutableLiveData()
    val insertTaskLD:MutableLiveData<TaskEntity> = MutableLiveData()
    val sumarTotal:MutableLiveData<String> = MutableLiveData<String>()
```

7-Ahora vamos a ver en mas profundidad el TaskDatabase comentado,este .kt como el propio nombre indica es la DB

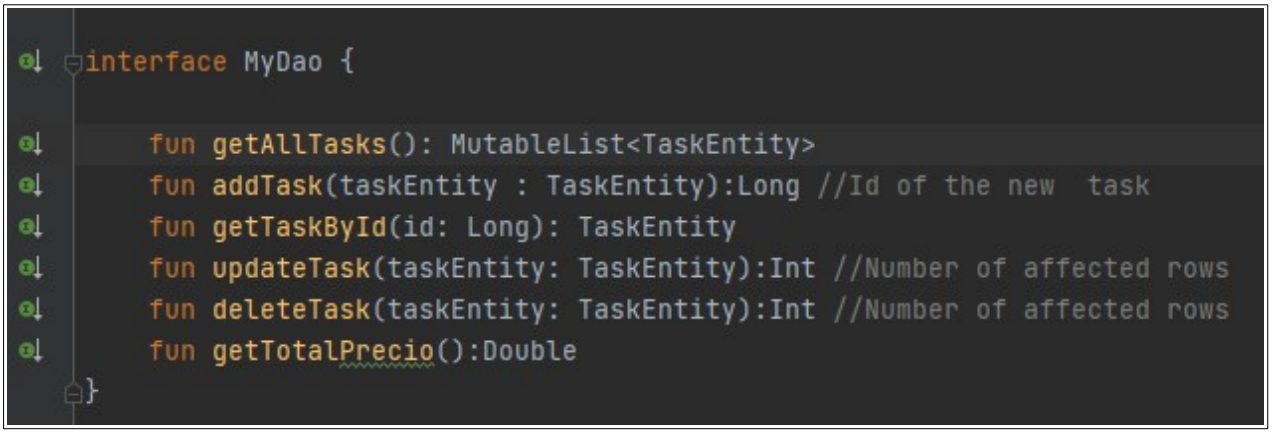
```
/**DATABASE en la que le indicamos que su unica tabla se llamara entities que es un array of de TaskEntity,la clase anteriormente enseñada que
 * era una tabla*/
@Database(entities = arrayOf(TaskEntity::class), version = 1)
abstract class TasksDatabase : RoomDatabase() {

    /**Creamos una funcion de la interfaz que contiene las consultas/funciones tipicas de un DB*/
    abstract fun taskDao(): TaskDao

    /**Companion object que trata de conseguir el contexto de la interfaz/funcion que hemos creado arriba*/
    companion object{ //Singleton Pattern
        private var instance:TaskDao? = null

        fun getInstance(context: Context):TaskDao{
            return instance ?: Room.databaseBuilder(context, TasksDatabase::class.java,  name: "tasks-db").build().taskDao().also { instance = it }
        }
    }


}
```

8-Y aquí tenemos la joya de la corona,todas las funciones para trabajar con las bases de datos,estan comentadas para entender que hace cada una



```
@Dao
interface TaskDao:MyDao {
    /**Devuelve todas las tareas/compras */
    @Query( value: "SELECT * FROM task_entity")
    override fun getAllTasks(): MutableList<TaskEntity>

    /**Inserta una tarea*/
    @Insert
    override fun addTask(taskEntity : TaskEntity):Long

    /**Devuelve una tarea por ID*/
    @Query( value: "SELECT * FROM task_entity WHERE id LIKE :id")
    override fun getTaskById(id: Long): TaskEntity

    /**Actualiza una tarea*/
    @Update
    override fun updateTask(taskEntity: TaskEntity):Int

    /**Borra una tarea*/
    @Delete
    override fun deleteTask(taskEntity: TaskEntity):Int

    /**Nos devuelve el precio total de la suma de todos los productos*/
    @Query( value: "SELECT sum(precio * cantidad) FROM task_entity")
    override fun getTotalPrecio(): Double
}
```

9-Antes en la interfaz creada utilizabamos otra interfaz para coger esas funciones , esa interfaz que cogiamos se llamaba MyDao,aquí la tenemos



```
interface MyDao {

    fun getAllTasks(): MutableList<TaskEntity>
    fun addTask(taskEntity : TaskEntity):Long //Id of the new  task
    fun getTaskById(id: Long): TaskEntity
    fun updateTask(taskEntity: TaskEntity):Int //Number of affected rows
    fun deleteTask(taskEntity: TaskEntity):Int //Number of affected rows
    fun getTotalPrecio():Double
}
```

10-Ahora vamos a seguir con el ViewModel que esta tal cual como lo dejamos en el paso 6,recuerda que solo vimo esto

```
val context = application
var myDao: MyDao = TasksDatabase.getInstance(context)

val taskListLD:MutableLiveData<MutableList<TaskEntity>> = MutableLiveData()
val updateTaskLD:MutableLiveData<TaskEntity?> = MutableLiveData()
val deleteTaskLD:MutableLiveData<Int> = MutableLiveData()
val insertTaskLD:MutableLiveData<TaskEntity> = MutableLiveData()
val sumarTotal:MutableLiveData<String> = MutableLiveData<String>()
```

La parte del código que queda en si se encarga de realizar/utilizar todas las funciones que hemos visto en los anteriores ficheros de DB,para que sea mas simple voy a enseñar función a función explicando aunque quede más extenso.

getAllTasks → coge todas las tareas de la DB y las guarda en la lista de TaskEntity que creamos anteriormente.
La cual era mutable,o sea que cuando invoquemos esta función habrá un observe en MainActivity que hara x.

```
fun getAllTasks(){
    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        taskListLD.postValue(myDao.getAllTasks())
    }
```

Add → Básicamente añade una tarea/compra ,tiene los siguientes atributos :

task → nombre de lo que has comprado
cantidad → cantidades que compras
precio → precio de lo que cuesta 1 unidad de lo que compras
sniper →item seleccionado en un sniper(carne,verdura,muebles…)

```
fun add(task:String,cantidad:Int,precio:Double,sniper:String) {
    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        val id = myDao.addTask(TaskEntity(name = task , precio = precio, cantidad = cantidad, imagen = sniper))
        val recoveryTask = myDao.getTaskById(id)
        insertTaskLD.postValue(recoveryTask)
    }
```

Lo que haces en las 3 lineas marcadas de rojo es lo siguiente.

1-Almacenas la ID de una tarea que creas en la DB
2-Coges esa tarea mediante una funcion de la DB que hemos visto anteriormente
3-le haces un postValue a insert para que desde el MainActivity reaccione al estar observando y te cree de nuevo el RecyclerView con esta tarea añadida

Delete → Esta función tiene dos “utilidades” ,ayuda a actualizar lo que viene a ser la tarjeta de compra o borrarla.
Pongo un ejemplo:

-Compras 3 zanahorias a 2 € → has gastado 6€
-Pero cuando tu le das a la papelera que ejecuta el delete si hay 3 zanarias SOLO deja una,por lo tanto el precio total de la tarjeta de compra deberia ser 4€ → 2 zanahorias * 2 €

Y cuando solo te queda una zanahoria y la eliminas,debe eliminar la tarjeta.
Para verlo más claro estará explicado en código:

```
fun delete(task:TaskEntity){
    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope

        /**Conseguimos el precio total de una tarjeta*/
        var precioTotal =myDao.getTotalPrecio()
        /**A este precio total le quitamos el precio de 1 unidad ya que quitamos una*/
        var xd= (precioTotal-task.precio).toString()

        /**Ese precio lo posteamos en la MutableLiveData que esta observando el MainAcitivty con el precio total*/
        sumarTotal.postValue( value: xd  +"€")
        /**en el MainActivity se le ha quitado a esta tarea 1 de cantidad entonces tenemos que realizar el update para cuando
         * hagamos el get del precioTotal este actualizado*/
        myDao.updateTask(task)


        /**Si la cantidad es 0 lo borra*/
        if(task.cantidad==0) {
            val res = myDao.deleteTask(task)
            if (res > 0)
                deleteTaskLD.postValue(task.id)
            else {
                deleteTaskLD.postValue( value: -1)
            }

        }

    }
}
```

Update → En teoria es para actualizar el Recycleviewer pero no entiendo un carajo ,solo copialo xd

```
fun update(task: TaskEntity){
    CoroutineScope(Dispatchers.IO).launch {  this
        task.isDone = !task.isDone
        val res = myDao.updateTask(task)
        if(res>0)
            updateTaskLD.postValue(task)
        else
            updateTaskLD.postValue( value: null)
    }
}
```

getSuma → Se encarga d epostear en la observable del precio total para que así se cambie debajo.es muy simple

```
fun getSuma(){

    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        sumarTotal.postValue( value: myDao.getTotalPrecio().toString() +"€")
    }
}
```

11-Ahora vamos a volver al MainActivity ,recordar que hemos hecho todo este rollo antes por que nos quedabamos en la parte que creaba el viewModel

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(ActivityMainBinding.inflate(layoutInflater).also {  it: ActivityMainBinding
        binding = it
    }.root)

    taskViewModel = ViewModelProvider( owner: this)[TaskViewModel::class.java]
```

12-A partir de aquí voy a ir mostrando partes de funciones de viewmodel,observables o funciones del MainActivity ya documentadas en el propio ejercicio

```
/**Invocamos a la funcion del viewmodel encargada de guardar task en una lista mutable*/
taskViewModel.getAllTasks()
/**Invocamos a la funcion del viewmodel encargada de guardar la suma en una lista mutable*/
taskViewModel.getSuma()

/**Nos quedamos observando la MutableList donde se guardan las tareas y cuando se poste algo*/
taskViewModel.taskListLD.observe( owner: this){  it: MutableList<TaskEntity>!
    tasks.clear() /**Limpiamos*/
    tasks.addAll(it)/**Añadimos todo*/
    recyclerView.adapter?.notifyDataSetChanged() /**Notificamos al adaptador del RecycleViewer que actualice*/
}

/**Esta atento del update y si esta a null cuandoo actualice lanza error,no es de mucha utilidad aqui*/
taskViewModel.updateTaskLD.observe( owner: this){ taskUpdated ->
    if(taskUpdated == null){
        showMessage( s: "Error updating task")
    }
}

/**Aqui estamos observamdo la LiveData del precio total de todo y cuando se poostee lo cambiamos en el xml*/
taskViewModel.sumarTotal.observe( owner: this){  it: String!
    binding.tvCompraTotal.text=it.toString()
}
```

```
/**Observamos el delete*/
taskViewModel.deleteTaskLD.observe( owner: this){ id ->

    /**Actualizamos el precio de la suma mediante el metodo getSume*/
    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        //Actualizamos precio
        taskViewModel.getSuma()

    }
    /**Si va bien borra,si no,muestra error*/
    if(id != -1){
        val task = tasks.filter {  it: TaskEntity
            it.id == id

        }[0]
        val pos = tasks.indexOf(task)
        tasks.removeAt(pos)
        recyclerView.adapter?.notifyItemRemoved(pos)
    }else{
        showMessage( s: "Error deleting task")
    }
}

/**Añade la task a la lista que tenemos aquí y notifica el recycleviewer*/
taskViewModel.insertTaskLD.observe( owner: this){  it: TaskEntity!
    tasks.add(it)
    recyclerView.adapter?.notifyItemInserted(tasks.size)

    /**Calcula las variable necesarias para actualizar el precio de compra TOTAL*/
    val precioTotal = it.precio * it.cantidad
    val precioQueYaHay = binding.tvCompraTotal.text.toString().replace( oldValue: "€", newValue: "").toDouble()
    //Actualizamos precio que se ve
    binding.tvCompraTotal.text=String.format("%.2f", precioTotal+precioQueYaHay).toString() +"€"

    CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
        /**Actualizamos el precio en la DB*/
        taskViewModel.getSuma()

    }
}
```

Boton encargado de añadir tareas con filtros

```
    //Boton añadir
    binding.btnAddTask.setOnClickListener {  it: View!
        var buleano:Boolean = true
        if (binding.etPrecio.text.contains( other: ".")){
            //Realizamos un split para coger los decimales
        val strs = binding.etPrecio.text.toString().split( ...delimiters: ".").toTypedArray()
            //Si la longitud de despues del punto es > 2
        if (strs[1].length > 2) {
            //Muestra el mensaje
            showMessage( s: "Oye ,que maximo son 2 decimales")
            buleano=false

        }
        }
        //Lo mismo que lo anterior pero con el nombre
         if (binding.etTask.text.toString().length > 10) {
            showMessage( s: "Oye ,que maximo son 10 caracteres")
            buleano=false

        }
            //Si no introduce nombre
            if (binding.etTask.text.toString().length == 0) {
                //Mensaje de advertencia
                showMessage( s: "Oye ,Tienes que escribir un nombre")
                buleano=false
        }



            //Si la cantidad esta vacia
            if (binding.etCantidad.text.toString().equals("")) {
                //mensaje
                showMessage( s: "Oye ,que no has puesto cantidad")
                buleano=false
                //Si no es un int
            }else if(isInteger(binding.etCantidad.text.toString())){}else{
                //mensaje
                showMessage( s: "Oye ,que has puesto una LETRA en CANTIDAD burro")
                buleano=false}
        //Si el precio esta vacio
        if (binding.etPrecio.text.toString().equals("")) {
            //mensaje
            showMessage( s: "Oye ,que no has puesto ningun precio")
            buleano=false
```

```
    //Si te has fijado,si daba algun fallo poniamos a buleano false, para que aquí no creara la tarea
    if(buleano==true){ addTask()}

}
```

Y al final del OnCreate tenemos la llamada a la funcion que invoca el recycleviewer

```
        setUpRecyclerView()
    }
```

13-Ahora aquí tenemos unas funciones del MainActivity que ya hemos visto muchas veces y el nombre es muy descriptivo como apra explicarlo

```
private fun showMessage(s: String) {
    Toast.makeText( context: this, s, Toast.LENGTH_SHORT).show()
}

private fun addTask() {
    taskViewModel.add(binding.etTask.text.toString(),binding.etCantidad.text.toString().toInt(),binding.etPrecio.text.toString().toDouble(),binding.spinner.selectedItem.toString())
    clearFocus()
    hideKeyboard()
}

fun setUpRecyclerView() {
    adapter = TaskAdapter(tasks, { taskEntity ->  updateTask(taskEntity) }, { taskEntity -> deleteTask(taskEntity) })
    recyclerView = binding.rvTask
    recyclerView.setHasFixedSize(true)
    recyclerView.layoutManager = LinearLayoutManager( context: this)
    recyclerView.adapter = adapter
}

private fun updateTask(taskEntity: TaskEntity) {
    taskViewModel.update(taskEntity)
}

private fun deleteTask(taskEntity: TaskEntity) {
    taskViewModel.delete(taskEntity)

}

private fun clearFocus(){
    binding.etTask.setText("")
}

private fun Context.hideKeyboard() {
    val inputMethodManager = getSystemService(Activity.INPUT_METHOD_SERVICE) as InputMethodManager
    inputMethodManager.hideSoftInputFromWindow(currentFocus?.windowToken,  flags: 0)
}

fun isInteger(str: String?) = str?.toIntOrNull()?.let { true } ?: false
```

14-Lo último que me queda que me queda es explicar las clases del recycleviewer(adapter + viewholder) las cuales estan en el mismo archivo esta vez

Lo primero que tenemos es el adapter el cual en onCreateViewHolder crea un viewHolder con el item_task.xml que luego mostrare.

Después en el metodo onBindViewHolder se encarga de pasarle los datos y el getItemCount es obvio para que es

```
class TaskAdapter(
    val tasks: List<TaskEntity>,
    val checkTask: (TaskEntity) -> Unit,
    val deleteTask: (TaskEntity) -> Unit) : RecyclerView.Adapter<TaskAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.item_task, parent, attachToRoot: false))
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = tasks[position]
        holder.bind(item, checkTask, deleteTask)
    }


    override fun getItemCount() = tasks.size
```

15-Por penultimo tenemos el viewHolder el cual asigna valores al xml que se reitera y tiene un listener que pasa la tarea que hay que borrar al deleteTask

```
class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {

    val tvTask = view.findViewById<TextView>(R.id.tvTask)
    val tvPrecio =view.findViewById<TextView>(R.id.tvPrecio)
    val tvCantidad =view.findViewById<TextView>(R.id.tvCantidad)
    val tvTotal = view.findViewById<TextView>(R.id.tvTotal)
    val buton = view.findViewById<ImageButton>(R.id.imgBtn)
    val imagen = view.findViewById<ImageView>(R.id.iv1)

    fun bind(task: TaskEntity, checkTask: (TaskEntity) -> Unit, deleteTask: (TaskEntity) -> Unit) {
        //Asignamos los datos del item que se repite
        tvTask.text = task.name
        tvPrecio.text= task.precio.toString()
        tvCantidad.text = task.cantidad.toString()
        tvTotal.text=   String.format("%.2f", task.precio*task.cantidad).toString() +"€" //Le damos formato al precio total de dos decimales solo

        //Cogemos los datos seleccionados en el Spiner y dependiendo de este asignamos una imagen
        if(task.imagen.equals("Carne")){imagen.setImageResource(R.drawable.imgcarne)}
        else if (task.imagen.equals("Verdura")){imagen.setImageResource(R.drawable.imgverdura)}
        else if (task.imagen.equals("Congelado")){imagen.setImageResource(R.drawable.imgcongelado)}
        else if (task.imagen.equals("Muebles")){imagen.setImageResource(R.drawable.imgmuebles)}
        else if (task.imagen.equals("Medicinas")){imagen.setImageResource(R.drawable.imgmedicinas)}
        else if (task.imagen.equals("Herramientas")){imagen.setImageResource(R.drawable.imherramientas)}



        buton.setOnClickListener {  it: View!
            task.cantidad--

            tvCantidad.text=task.cantidad.toString()

            tvTotal.text=   String.format("%.2f", task.precio*task.cantidad).toString() +"€"

            deleteTask(task) }
    }
```

16-Estructura de XML que se repite