# Android RecyclerView: How to Insert, Update and Delete Item

October 6, 2020 Techenum

RecyclerView is possibly most used View in Android development for showing list of items. There was something called ListView the use of which is now outdated. It is not deprecated but RecyclerView does much better task of managing list of thousands of items. It is because of the View Holder pattern. In this blog post we will see how we can use Android RecyclerView to insert, update and delete items in it.

Here we will be adding updating and removing some Product. So everything will be based around hypothetical *Store*. To follow along you must have a basic grasp of Android. With that being said let us start now.

Quick Navigation

## Why use RecyclerView in the first place ?

If you have just started out with Android Development, this might be a general question on your mind.

Why ?

It is because RecyclerView does a much better memory management out of the box. No matter the amount of items you have in your Adapter. RecyclerView will show only the items which the user is currently looking at.

Also it is worth noting RecyclerView is really much more straight froward once you grasp the basic idea behind it.

## How to use RecyclerView

Let us first see what are the steps that we must go through in order to implement RecyclerView. Let us first see what we will be needing to make RecyclerView display items in list.

- First, we must create a RecyclerView instance in the XML layout
- Then, we will need to create a `RecyclerView.Adapter<*>` implementation
- `RecyclerView.Adapter<*>` requires a `RecyclerView.ViewHolder` to isolate each row and better mange them
- Then, we will define an Interface to add update and remove item values
- Finally, we will need our item as a POJO or any form you like

Let us look at how to implement these things one by one. But beware this is a really simple and basic `RecyclerView` implementation.

Also Read : Learn EventBus, The LocalBroadcastManager Alternative

*TECHENUM*

## Adding recyclerview dependency

We have to add the following dependency in our `app > build.gradle` file before we can use `RecyclerView`.

```
implementation 'androidx.recyclerview:recyclerview:1.1.0'
```
Code language: Groovy (groovy)

This will enable us to use RecyclerView in our application. I have created a simple UI like below to add Product and it’s price.

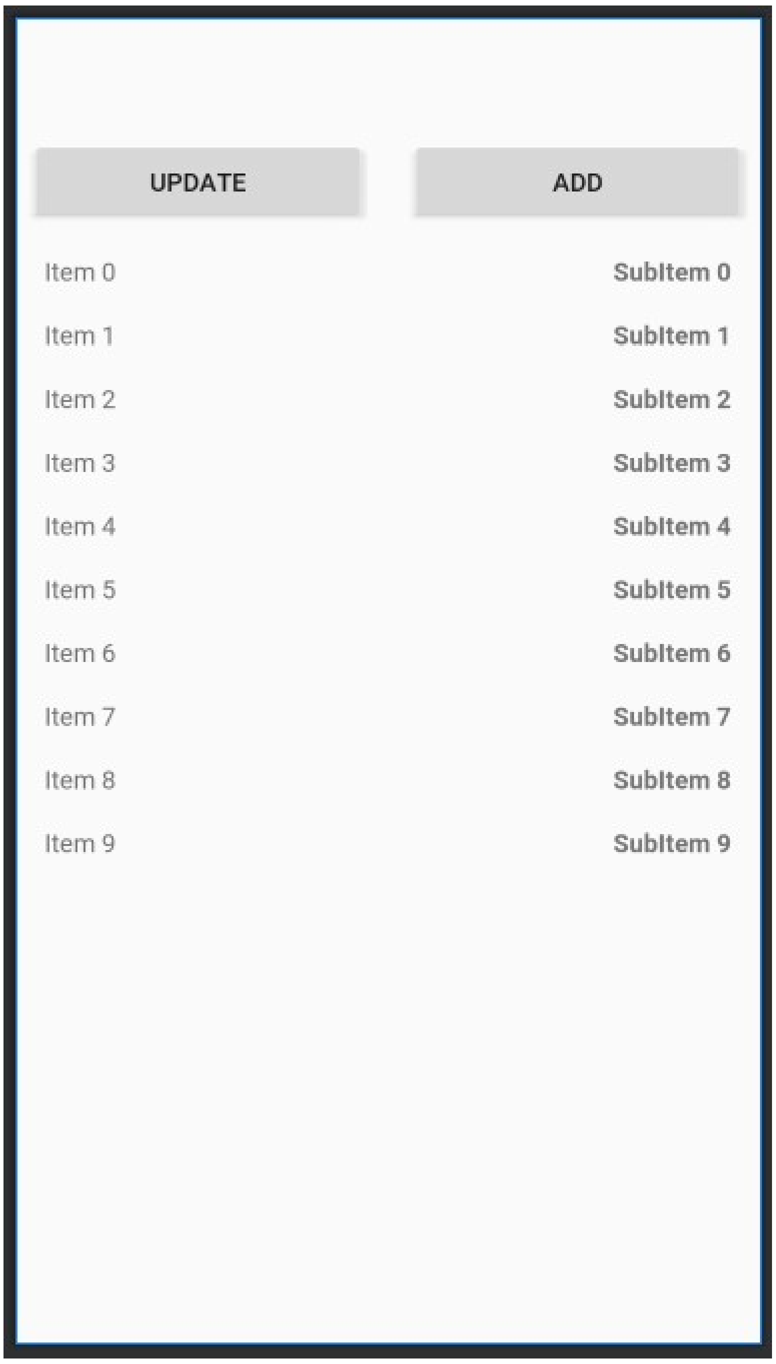
## Creating layout for RecyclerView

See the code for `activity_main.xml` below:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <EditText
```

```xml
        android:id="@+id/product_name"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:hint="Product Name"
        android:inputType="textPersonName"
        app:layout_constraintEnd_toStartOf="@+id/product_price"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/product_price"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        android:hint="Price"
        android:inputType="number"
        app:layout_constraintBottom_toBottomOf="@+id/product_name"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/product_name" />

    <Button
        android:id="@+id/add_product"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="8dp"
        android:text="Add"
        app:layout_constraintBottom_toBottomOf="@+id/update_product"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/update_product"
        app:layout_constraintTop_toTopOf="@+id/update_product" />

    <Button
        android:id="@+id/update_product"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="16dp"
        android:text="Update"
        app:layout_constraintEnd_toStartOf="@+id/add_product"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/product_name" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/product_list_recycler_view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/add_product"
        app:layout_constraintVertical_bias="0.0" />
</androidx.constraintlayout.widget.ConstraintLayout>
```
Code language: HTML, XML (xml)

The UI from the above code is something like below.



And map the views to activity like this :

```
class MainActivity : AppCompatActivity() {

    private lateinit var addProduct: Button
    private lateinit var updateProduct: Button
    private lateinit var productName: EditText
    private lateinit var productUnit: EditText
    private lateinit var productList: RecyclerView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // initialize the recycler view
        productList = findViewById(R.id.product_list_recycler_view)
        productList.layoutManager = LinearLayoutManager(this)
        productList.setHasFixedSize(true)
        // we will be adding adapter here later

        productName = findViewById(R.id.product_name)
        productUnit = findViewById(R.id.product_price)

        updateProduct = findViewById(R.id.update_product)
        updateProduct.setOnClickListener {
        }

        addProduct = findViewById(R.id.add_product)
        addProduct.setOnClickListener {
        }

    }

}
Code language: Kotlin (kotlin)
```

Focus on the highlighted lines as you need to map RecyclerView and also add some other properties.

`.setHasFixedSize(true)` tells the system that we will not be manipulating row height and width dynamically. It will make the operation a little bit faster.

We have to have a LayoutManager for our RecyclerView to display items in a list. Before we can use Android RecyclerView to insert, update and delete items. Let us look at what are the available layout managers.

## LayoutManager in RecyclerView

If you are quite new to Android development you might not know that there are 3 layout managers built in for us to use with RecyclerView. Which are:

- **`LinearLayoutManager`**, when we want to show items as a simple list
- **`GridLayoutManager`**, when we want to show items in a grid
- **`StaggeredGridLayoutManager`**, when we want to show uneven grids

We will look at these separately but for this post we will be using only the **`LinearLayoutManager`**. And it's usage is pretty straight forward.

We create an instance by simply doing : `LinearLayoutManager(this)`. Which we have already done in the highlighted text in above section.

## Preparing a ViewHolder

Let us first prepare a ViewHolder before we move forward to our Adapter. It is really very simple.

Create a layout file in your `res > layout` folder and name it `item_product`.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?selectableItemBackground"
    android:clickable="true"
    android:focusable="true">

    <TextView
        android:id="@+id/product_name"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:padding="8dp"
        app:layout_constraintEnd_toStartOf="@+id/product_price"
        app:layout_constraintHorizontal_bias="1.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/product_price"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="8dp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="@+id/product_name"
        app:layout_constraintEnd_toStartOf="@+id/delete_product"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/product_name" />

    <ImageView
        android:id="@+id/delete_product"
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:padding="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/ic_delete_24" />

</androidx.constraintlayout.widget.ConstraintLayout>
```
Code language: HTML, XML (xml)

Eveything is straightforward except for maybe the `app:srcCompat="@drawable/ic_delete_24"` portion. You can use the image of your choice.

Or just use the available vector image from android drawable. Right click on `res > drawable` then `new > vector asset` and follow.

For the code portion create a class named `ProductListAdapter` and copy the code below.

```
class ProductListAdapter() :
RecyclerView.Adapter<ProductListAdapter.ProductViewHolder>() {

    /**
     * ViewHolder implementation for holding the mapped views.
     */
    inner class ProductViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        val productName: TextView =
itemView.findViewById(R.id.product_name)
        val productPrice: TextView =
itemView.findViewById(R.id.product_price)
        val productDelete: ImageView =
itemView.findViewById(R.id.delete_product)
    }

}
Code language: Kotlin (kotlin)
```

You will receive errors but don't worry. It's because of me explaining it in portions. We will remove the errors step by step.

The ViewHolder is the `inner class ProductViewHolder` and not the `class ProductListAdapter`.

I like to nest my `ViewHolder` class but you can extract it into a separate file in your other projects.

Also, `itemView: View` is the root view for each row which we will be using later. For now everything is mapped just like in the `MainActivity` class.

Also Read : Kotlin Coroutine in Android : Understanding the Basics

*TECHENUM*

## Creating Product Model

To store our simple data we first need to define a simple model class. It is just the POJO if you are familiar with java. For the sake of simplicity I am calling it model in this post.

Create a data class with the following code.

```
data class ProductModel(
    var id: Int = 0,
    var name: String = "",
    var price: String = ""
)
Code language: Kotlin (kotlin)
```

We will be using `id` to uniquely identify each item in the list. This will be useful when updating item later. Alongside we will be displaying `name` and `price`.

## Creating interface for CUD

C = Create, U = Update, D = Delete

We will be creating an interface to communicate between Adapter and Activity. As it is generally a bad idea to put each and every logic inside the adapter. Let us first create an interface named `OnProductClickListener`.

```
interface OnProductClickListener {

    /**
     * When the user clicks on each row this method will be invoked.
     */
    fun onUpdate(position: Int, model: ProductModel)

    /**
     * when the user clicks on delete icon this method will be invoked to
remove item at position.
     */
    fun onDelete(model: ProductModel)

}
Code language: Kotlin (kotlin)
```

We will be using this to show new items in the list. Let us move on with our Adapter.

Those were the prerequisite for use of Android RecyclerView to insert, update and delete items in list.

Learn More : [Interface in OOP: Guide to Polymorphism and Callbacks]

*TECHENUM*

## Creating RecyclerView Adapter

Now that we have created our ViewHolder and almost everything for adapter to use let us implement the adapter.

We have created the class `ProductListAdapter` let us continue there. We need to implement certain methods before we can actually code any further.

Lets see below I have left out the ViewHolder class for the sake of simplicity.

```
class ProductListAdapter() :
RecyclerView.Adapter<ProductListAdapter.ProductViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ProductViewHolder {
    }

    override fun onBindViewHolder(holder: ProductViewHolder, position: Int)
{
    }

    /**
     * Returns the total number of items in the list to be displayed.
     * this will refresh when we call notifyDataSetChanged() or other
related methods.
     */
    override fun getItemCount(): Int {
    }
}
Code language: Kotlin (kotlin)
```

Let me explain each method in brief.

**`onCreateViewHolder()`** is where we create a new instance of ViewHolder and return it.

**`onBindViewHolder()`** is where we will map the data to the views as it is called each time the user does something on the screen.

Do not initialize any listeners here, generally considered a bad practice as it will harm the performance.

Finally, **`getItemCount()`** is where we return the number of items the Adapter will hold at any given time. Will most likely be called when we do invoke `notifyDataSetChanged()` or related methods.

Let us fill in the body for each one by one. Let us start by

## Declaring List<*> to hold item detail

You must modify the `ProductListAdapter`'s constructor and change it as following.

We will be using `mContext` to create the view we have defined above in the view holder i.e. `item_product.xml`.

And we will be storing all our data in `mProductList` variable.

The modified constructor of the class will look something like.

```
class ProductListAdapter(
    private val mContext: Context,
    private val mOnProductClickListener: OnProductClickListener,
    private val mProductList: ArrayList<ProductModel> = ArrayList()
) : RecyclerView.Adapter<ProductListAdapter.ProductViewHolder>() {
    // code here is intentionally left out
}
Code language: Kotlin (kotlin)
```

`mOnProductClickListener` will be used when we want to perform some actions in each row i.e. update or delete. More on this later as you follow along.

Learn more About `ArrayList` here : Understanding ArrayList in Java With Example

*TECHENUM*

## getItemCount()

This method is called by the adapter to determine total items whenever a new data set is passed in.

We will return the total size of the `mProductList` in this method. Change the above method to make it look like below

```
override fun getItemCount(): Int {
    return mProductList.size
}
Code language: Kotlin (kotlin)
```

## onBindViewHolder()

Here, the data is bind to the view. As the working mechanism of RecyclerView is optimized. It will create only the visible views to user.

So every time the user scrolls or does something. It will intelligently perform binding.

You must change the code of the method to this.

```
@SuppressLint("SetTextI18n")
override fun onBindViewHolder(holder: ProductViewHolder, position: Int) {

    // data will be set here whenever the system thinks it's required

    // get the product at position
    val product = mProductList[position]

    holder.productName.text = product.name
    holder.productPrice.text = "Rs. ${product.price}"
}
Code language: Kotlin (kotlin)
```

Remember the `ViewHolder` we created above ? We will be using `.setText()` to set the current position's value. The value is in `product`.

## onCreateViewHolder()

We will do a little bit of tasks here. Add some click listeners and add fairly simple logic.

Replace the method above with the code below. I will provide explanation based on line number so keep looking at code for reference.

`Line 2` will create an `inflater` object. Which we use at `Line 3` to inflate the view. And finally the `ViewHolder` which we defined earlier will be created at `Line 4`.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ProductViewHolder {
    val inflater = LayoutInflater.from(parent.context)
    val view = inflater.inflate(R.layout.item_product, parent, false)
    val holder = ProductViewHolder(view)

    // item view is the root view for each row
    holder.itemView.setOnClickListener {

        // adapterPosition give the actual position of the item in the
RecyclerView
        val position = holder.adapterPosition
        val model = mProductList[position]
        // remove the Rs. prefix before sending the model for editing
        model.price = model.price.substringAfterLast(" ")
        mOnProductClickListener.onUpdate(position, model)
    }

    // to delete the item in recycler view
    holder.productDelete.setOnClickListener {
        val position = holder.adapterPosition
        val model = mProductList[position]
        mOnProductClickListener.onDelete(model)
    }

    return holder
}
```
Code language: Kotlin (kotlin)

`Line 7` is where we have set `click listener` to the root view. Because when we click the row, we should pass data back to the input box.

`Line 10` is where we will get the actual position of the view holder. This position is different from the `onBindViewHolder()`‘s `position` because in `onBindViewHolder()` a relative `position` is passed in.

At Line 14 we have used `.substringAfterLast()` to remove the **Rs.** prefix from the string. Which we have set in **onBindViewHolder()** above.

`Line 15` is where the actual magic happens. We invoke the `onUpdate()` method and pass in the actual position with the `model`. We will see to use in in later sections.

`Line 19` to `Line 23` is for deletion of item from the adapter. It is similar to update section described above.

We need to add certain methods in the Adapter before continuing with using the adapter.

## Add item

We will need to add each items the code for looks like below. Add the method below in your adapter class.

```
fun addProduct(model: ProductModel) {
    mProductList.add(model)
    // notifyDataSetChanged() // this method is costly I avoid it whenever
possible
    notifyItemInserted(mProductList.size)
}
```
Code language: Kotlin (kotlin)

We have used `notifyItemInserted(mProductList.size)` instead of the `notifyDataSetChanged()` method. It is because we want to update only one item and not the whole data set.

The performance difference will be noticeable when there are 1000’s of items in the list.

## Update item

To update item we will be using similar method but we will be using `notifyItemChanged()` method instead of `notifyItemInserted(mProductList.size)`.

Let us look at the code below

```
fun updateProduct(model: ProductModel?) {

    if (model == null) return // we cannot update the value because it is
null

    for (item in mProductList) {
        // search by id
        if (item.id == model.id) {
            val position = mProductList.indexOf(model)
            mProductList[position] = model
            notifyItemChanged(position)
            break // we don't need to continue any more iterations
        }
    }
}
Code language: Kotlin (kotlin)
```

We loop through each item and compare the id of it. If the item is found we replace the model at that position and notify that item has changed.

## Remove item

Similarly we remove item with the method below. Add it too to the adapter class.

```
fun removeProduct(model: ProductModel) {
    val position = mProductList.indexOf(model)
    mProductList.remove(model)
    notifyItemRemoved(position)
}
Code language: Kotlin (kotlin)
```

First we get the position and then remove the item from the list. And then we will notify that the item has been removed at that position with `notifyItemRemoved()`.

Finally we only have one more method to auto increment and get the id for each item.

Also Read : Consume an API with Retrofit in Android

*TECHENUM*

## Auto generating ID

It's tiring to create one id manually each time so let us automate the process. This ID will be used to uniquely identify each item in the list. Which will help us in updating the values.

```
fun getNextItemId(): Int {
    var id = 1
    if (mProductList.isNotEmpty()) {
        // .last is equivalent to .size() - 1
        // we want to add 1 to that id and return it
        id = mProductList.last().id + 1
    }
    return id
}
```
Code language: Kotlin (kotlin)

The logic is pretty straight forward. We will check if the list is empty. If it is not get the last item’s id and return it after doing `+1`.

## Summary

Let us take a look at final code of our `RecyclerView.Adapter` class including the `ViewHolder`.

```
class ProductListAdapter(
    private val mContext: Context,
    private val mOnProductClickListener: OnProductClickListener,
    private val mProductList: ArrayList<ProductModel> = ArrayList()
) : RecyclerView.Adapter<ProductListAdapter.ProductViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ProductViewHolder {
        val inflater = LayoutInflater.from(mContext)
        val view = inflater.inflate(R.layout.item_product, parent, false)
        val holder = ProductViewHolder(view)

        // item view is the root view for each row
        holder.itemView.setOnClickListener {

            // adapterPosition give the actual position of the item in the
RecyclerView
            val position = holder.adapterPosition
            val model = mProductList[position]

            // remove the Rs. prefix before sending the model for editing
            model.price = model.price.substringAfterLast(" ")
            mOnProductClickListener.onUpdate(position, model)
        }

        // to delete the item in recycler view
        holder.productDelete.setOnClickListener {
            val position = holder.adapterPosition
            val model = mProductList[position]
            mOnProductClickListener.onDelete(model)
        }

        return holder
    }

    @SuppressLint("SetTextI18n")
    override fun onBindViewHolder(holder: ProductViewHolder, position: Int)
{

        // data will be set here whenever the system thinks it's required

        // get the product at position
        val product = mProductList[position]

        holder.productName.text = product.name
        holder.productPrice.text = "Rs. ${product.price}"
    }
    /**
     * Returns the total number of items in the list to be displayed.
     * this will refresh when we call notifyDataSetChanged() or other
related methods.
     */
    override fun getItemCount(): Int {
        return mProductList.size
    }

    /**
     * Adds each item to list for recycler view.
     */
    fun addProduct(model: ProductModel) {
        mProductList.add(model)
        // notifyDataSetChanged() // this method is costly I avoid it
whenever possible
        notifyItemInserted(mProductList.size)
    }

    /**
     * Updates the existing product at specific position of the list.
     */
    fun updateProduct(model: ProductModel?) {

        if (model == null) return // we cannot update the value because it
is null

        for (item in mProductList) {
            // search by id
            if (item.id == model.id) {
                val position = mProductList.indexOf(model)
                mProductList[position] = model
                notifyItemChanged(position)
                break // we don't need to continue anymore
            }
        }
    }

    /**
     * Removes the specified product from the list.
     *
     * @param model to be removed
     */
    fun removeProduct(model: ProductModel) {
        val position = mProductList.indexOf(model)
        mProductList.remove(model)
        notifyItemRemoved(position)
    }

    fun getNextItemId(): Int {
        var id = 1
        if (mProductList.isNotEmpty()) {
            // .last is equivalent to .size() - 1
            // we want to add 1 to that id and return it
            id = mProductList.last().id + 1
        }
        return id
    }

    /**
     * ViewHolder implementation for holding the mapped views.
     */
    inner class ProductViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        val productName: TextView =
itemView.findViewById(R.id.product_name)
        val productPrice: TextView =
itemView.findViewById(R.id.product_price)
        val productDelete: ImageView =
itemView.findViewById(R.id.delete_product)
    }

}
```
Code language: Kotlin (kotlin)

Now that we have created the adapter. Let us use it in our activity. We are only left with final step in using Android RecyclerView for insert, update and delete.

## Using the RecyclerView Adapter

Finally we have passed the most tiring bit of code and come to the almost end.

Let us see how we can use the `ProductListAdapter` in our application. I will break it down for you and there is a complete code below if you just want to see the code.

### Creating necessary variables

Declare the following variables in the MainActivity class

```
/**
 * The adapter which we have prepared.
 */
private lateinit var mProductListAdapter: ProductListAdapter

/**
 * To hold the reference to the items to be updated as a stack.
 * We can just remove and get the item with [Stack] in one shot.
 */
private var modelToBeUpdated: Stack<ProductModel> = Stack()

/**
 * The listener which we have defined in [OnProductClickListener]. Will be
added to the adapter
 * which constructing the adapter
 */
private val mOnProductClickListener = object : OnProductClickListener {
    override fun onUpdate(position: Int, model: ProductModel) {

        // store this model that we want to update
        // we will .pop() it when we want to update
        // the item in the adapter
        modelToBeUpdated.add(model)

        // set the value of the clicked item in the edit text
        productName.setText(model.name)
        productUnit.setText(model.price)
    }

    override fun onDelete(model: ProductModel) {

        // just remove the item from list
        mProductListAdapter.removeProduct(model)
    }
}
```
Code language: Kotlin (kotlin)

Them `mOnProductClickListener` is the listener we had defined earlier. I have added comments above for you to easily understand it.

We also create and use a new instance of the adapter in the `onCreate()` method like below.

```
mProductListAdapter = ProductListAdapter(this, mOnProductClickListener)
productList.adapter = mProductListAdapter
```
Code language: Kotlin (kotlin)

## Creating the item

Let us first create the item and add it to the adapter.

Add the following click listener to the `addProduct()` variable in the `MainActivity`.

```
addProduct.setOnClickListener {

    val name = productName.text.toString()
    val price = productUnit.text.toString()

    if (!name.isBlank() && !price.isBlank()) {

        // prepare id on incremental basis
        val id = mProductListAdapter.getNextItemId()

        // prepare model for use
        val model = ProductModel(id, name, price)

        // add model to the adapter
        mProductListAdapter.addProduct(model)

        // reset the input
        productName.setText("")
        productUnit.setText("")
    }
}
```
Code language: Kotlin (kotlin)

We have only done basic validation and got the next id for the list. Then we have created a new object of the `ProductModel` by passing in `id`, `name` and `price`.

And then called the `addProduct()` method of the adapter. Finally, we have cleared the input with `.setText("")`.

Let us see the click listener for the update portion.

## Updating the item

Add a listener to the variable `updateProduct()` in MainActivity.

```
updateProduct.setOnClickListener {
    // we have nothing to update
    if (modelToBeUpdated.isEmpty()) return@setOnClickListener

    val name = productName.text.toString()
    val price = productUnit.text.toString()

    if (!name.isBlank() && !price.isBlank()) {
        val model = modelToBeUpdated.pop()
        model.name = name
        model.price = price
        mProductListAdapter.updateProduct(model)

        // reset the input
        productName.setText("")
        productUnit.setText("")
    }
}
Code language: Kotlin (kotlin)
```

Most of the things are same from the create item. The only difference is in the model variable. Instead of creating the new variable we have taken it from the `modelToBeUpdated` variable we have defined earlier.

Once we have new values set we passed the model to the adapter by calling `updateProduct()` method.

## Summary

We have the final code for our MainActivity as below cross verify if you have missed anything at all.

```
class MainActivity : AppCompatActivity() {

    private lateinit var addProduct: Button
    private lateinit var updateProduct: Button
    private lateinit var productName: EditText
    private lateinit var productUnit: EditText

    private lateinit var productList: RecyclerView

    /**
     * The adapter which we have prepared.
     */
    private lateinit var mProductListAdapter: ProductListAdapter

    /**
     * To hold the reference to the items to be updated as a stack.
     * We can just remove and get the item with [Stack] in one shot.
     */
    private var modelToBeUpdated: Stack<ProductModel> = Stack()

    /**
     * The listener which we have defined in [OnProductClickListener]. Will
be added to the adapter
     * which constructing the adapter
     */
    private val mOnProductClickListener = object : OnProductClickListener {
        override fun onUpdate(position: Int, model: ProductModel) {

            // we want to update
```

```
                modelToBeUpdated.add(model)

                // set the value of the clicked item in the edit text
                productName.setText(model.name)
                productUnit.setText(model.price)
            }

            override fun onDelete(model: ProductModel) {

                mProductListAdapter.removeProduct(model)
            }
        }

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)

            // initialize the recycler view
            productList = findViewById(R.id.product_list_recycler_view)
            productList.layoutManager = LinearLayoutManager(this)
            productList.setHasFixedSize(true)

            mProductListAdapter = ProductListAdapter(this,
    mOnProductClickListener)
            productList.adapter = mProductListAdapter

            productName = findViewById(R.id.product_name)
            productUnit = findViewById(R.id.product_price)

            updateProduct = findViewById(R.id.update_product)
            updateProduct.setOnClickListener {

                // we have nothing to update
                if (modelToBeUpdated.isEmpty()) return@setOnClickListener

                val name = productName.text.toString()
                val price = productUnit.text.toString()

                if (!name.isBlank() && !price.isBlank()) {
                    val model = modelToBeUpdated.pop()
                    model.name = name
                    model.price = price
                    mProductListAdapter.updateProduct(model)

                    // reset the input
                    productName.setText("")
                    productUnit.setText("")
                }
            }

            addProduct = findViewById(R.id.add_product)
            addProduct.setOnClickListener {

                val name = productName.text.toString()
                val price = productUnit.text.toString()

                if (!name.isBlank() && !price.isBlank()) {

                    // prepare id on incremental basis
                    val id = mProductListAdapter.getNextItemId()

                    // prepare model for use
                    val model = ProductModel(id, name, price)
                    // add model to the adapter
                    mProductListAdapter.addProduct(model)

                    // reset the input
                    productName.setText("")
                    productUnit.setText("")
                }
            }

        }

    }
```
Code language: Kotlin (kotlin)

That is how you use the adapter after creating it. If you want to look at the full source code visit this GitHub repository.

With this we have completed using Android RecyclerView for insert, update and delete item from it.

Feel free to comment below if you have any issue with this blog post or the code in the repository about android recyclerview .

Also Read : Install Scrcpy: Mirror Your Android Device Wired or Wirelessly

*TECHENUM*