Table of Contents

[FRONT END – XML 3](#_Toc127931416)

[STYLES 3](#_Toc127931417)

[Set the status bar to be part of the screen in Android Studio. 3](#_Toc127931418)

[SHORT NECESSARRY LINES OF CODES 4](#_Toc127931419)

[Picasso image loader 4](#_Toc127931420)

[A button that takes the user to a whatsapp chat for a specific number: 4](#_Toc127931421)

[Glide Image Loader 5](#_Toc127931422)

[Randomizing array list 6](#_Toc127931423)

[Setting Date and Time 6](#_Toc127931424)

[SPLASH SCREEN 7](#_Toc127931425)

[Xml Files 7](#_Toc127931426)

[Layout 7](#_Toc127931427)

[Value 7](#_Toc127931428)

[Android Manifest File 7](#_Toc127931429)

[Java Activity File 8](#_Toc127931430)

[ANDROID MANIFEST 9](#_Toc127931431)

[How to create a button that takes the user to the phone dialer to call a specific number: 9](#_Toc127931432)

[ANDROID CLASS ADAPTER 10](#_Toc127931433)

[Example of how to use Recycle View – ViewHolder – AdapterClass 10](#_Toc127931434)

[GRADLE DEPENDANCIES 12](#_Toc127931435)

[INTENT 13](#_Toc127931436)

[Set the intent to go to the previous activity 13](#_Toc127931437)

[ONCLICKLISTENER INTERFACE 14](#_Toc127931438)

[FIREBASE 15](#_Toc127931439)

[FIREBASE PHONE AUTHENICATION 15](#_Toc127931440)

[Sending Verification Code to Number 15](#_Toc127931441)

[FIREBASE DATABASE 18](#_Toc127931442)

[Normalization in Firebase 18](#_Toc127931443)

[Firebase data schema 19](#_Toc127931444)

[Saving Data on Realtime FirbaseDatabase on the Application Code Way 1 20](#_Toc127931445)

[Saving Data on Realtime FirbaseDatabase on the Application Code Way 2 21](#_Toc127931446)

[Update Fields in The Firebase Realtime Database and Have Those Changes Automatically Reflected in The Application. 23](#_Toc127931447)

[Method I 23](#_Toc127931448)

[Method II 23](#_Toc127931449)

[Retrieving data from the database 24](#_Toc127931450)

[ROOM DATABASE 25](#_Toc127931451)

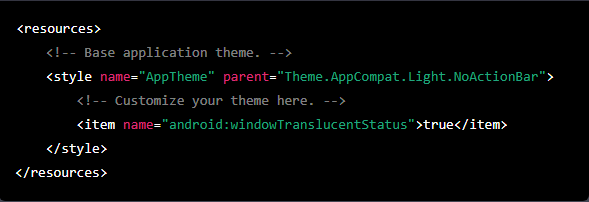
# FRONT END – XML

# STYLES

## Set the status bar to be part of the screen in Android Studio.

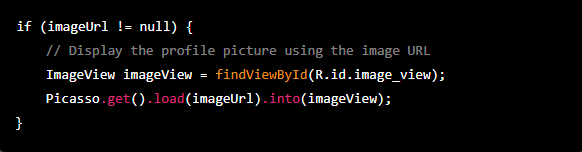
1. Use the **android:windowTranslucentStatus** attribute

In your styles.xml file, you can add the **android:windowTranslucentStatus** attribute to your application theme. This attribute makes the status bar translucent and allows your layout to be laid out behind it. Here's an example:



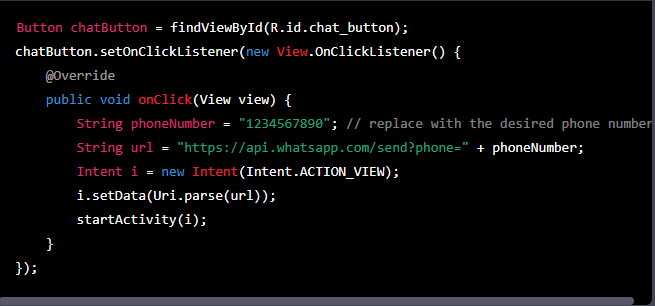
# SHORT NECESSARRY LINES OF CODES

## Picasso image loader



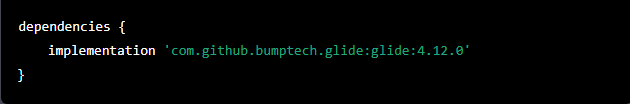
This code uses the Picasso library to display the profile picture. The findViewById method is used to find the ImageView in the layout file, and the load method is used to load the image from the URL into the ImageView. The into method is used to set the image into the ImageView.

## A button that takes the user to a whatsapp chat for a specific number:



## Glide Image Loader

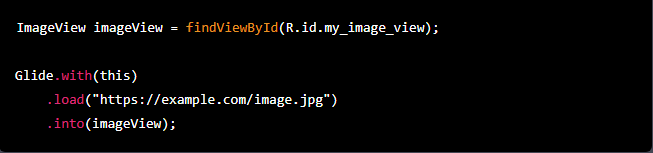
1. Minimum SDK version - Glide supports Android SDK version 14 (Ice Cream Sandwich) and higher.
2. Dependency - Add the following dependency to your app-level build.gradle file:

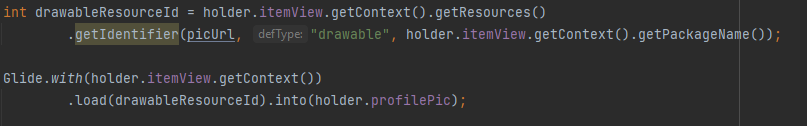


1. Permissions - If you are loading images from external sources, you may need to add internet permission to your app's manifest file:



1. Initialization - Initialize the Glide library in your Application class's onCreate method or another appropriate initialization method.
2. Once you've met these requirements, you can use Glide to load images from various sources into your app's ImageView, Drawable, and Bitmap objects.





## Randomizing array list

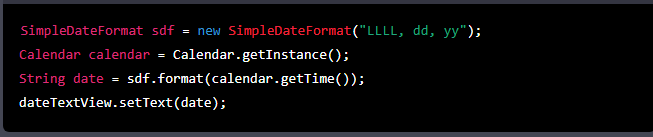


To randomize the items in the ArrayList, you can use the collections.shuffle() method.

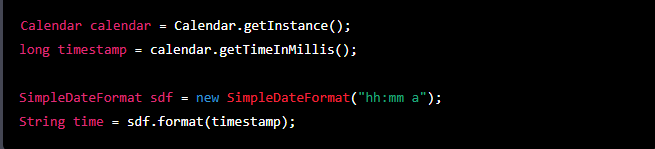
This will shuffle the items in the ArrayList using the "r" Random object. If you want to display the shuffled items, you can use a for loop to iterate through the list and display the properties of each FoodDomain object.

## Setting Date and Time

This code creates a SimpleDateFormat object with the LLLL, dd, yy pattern, which specifies a date format with the full month name, day, and year.



This code gets the current time in milliseconds using the Calendar class, and then creates a SimpleDateFormat object with the "hh:mm a" pattern, which specifies a time format with the hour and minute in a 12-hour clock format.



# SPLASH SCREEN

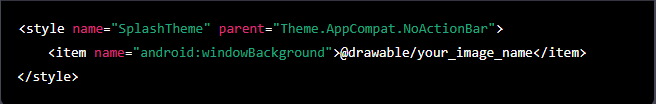
## Xml Files

### Layout

1. In Android Studio, go to the "res" folder and create a new folder called "drawable."
2. In the "drawable" folder, add your image that you want to use for the splash screen.

### Value

1. Go to the "values" folder and create a new file called "styles.xml."
2. In the "styles.xml" file, create a new style for the splash screen by adding the following code:

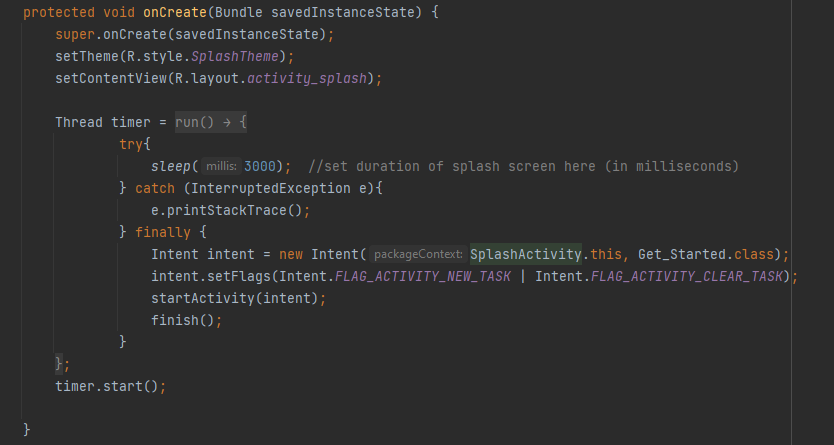


## Android Manifest File

1. In the "AndroidManifest.xml" file, set the theme for the launcher activity to the splash theme by adding the following line within the activity tag:



## Java Activity File



# ANDROID MANIFEST

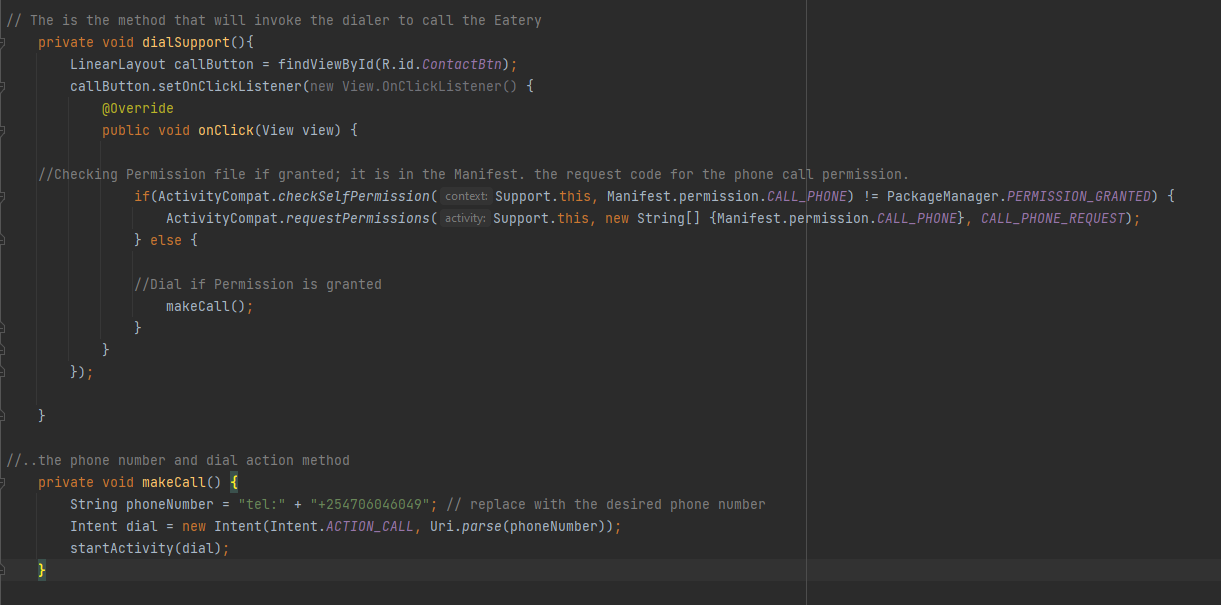
## How to create a button that takes the user to the phone dialer to call a specific number:

1. It's important to add the CALL\_PHONE permission in the AndroidManifest.xml file:



1. On Your Activity you need to define the CALL\_PHONE\_REQUEST variable before using it. It's a constant integer that you can create to store the request code for the phone call permission. It's important to note that you need to set it to a unique integer value and should be used consistently throughout your activity to identify the request.





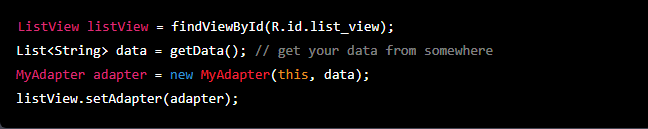
# ANDROID CLASS ADAPTER

In Android, the Adapter class is used to connect a set of data to a ListView, GridView, or any other type of AdapterView that displays a scrollable list of items. An Adapter acts as a bridge between the data source and the view that displays the data, providing the data to the view in a format that can be easily rendered.

The Adapter class is used extensively in Android to display lists of items, such as contacts, messages, and images. It is a powerful tool for connecting data to views and making your app more dynamic and responsive.

## Example of how to use Recycle View – ViewHolder – AdapterClass

1. Set Recycle View in Activity Class with the Adapters

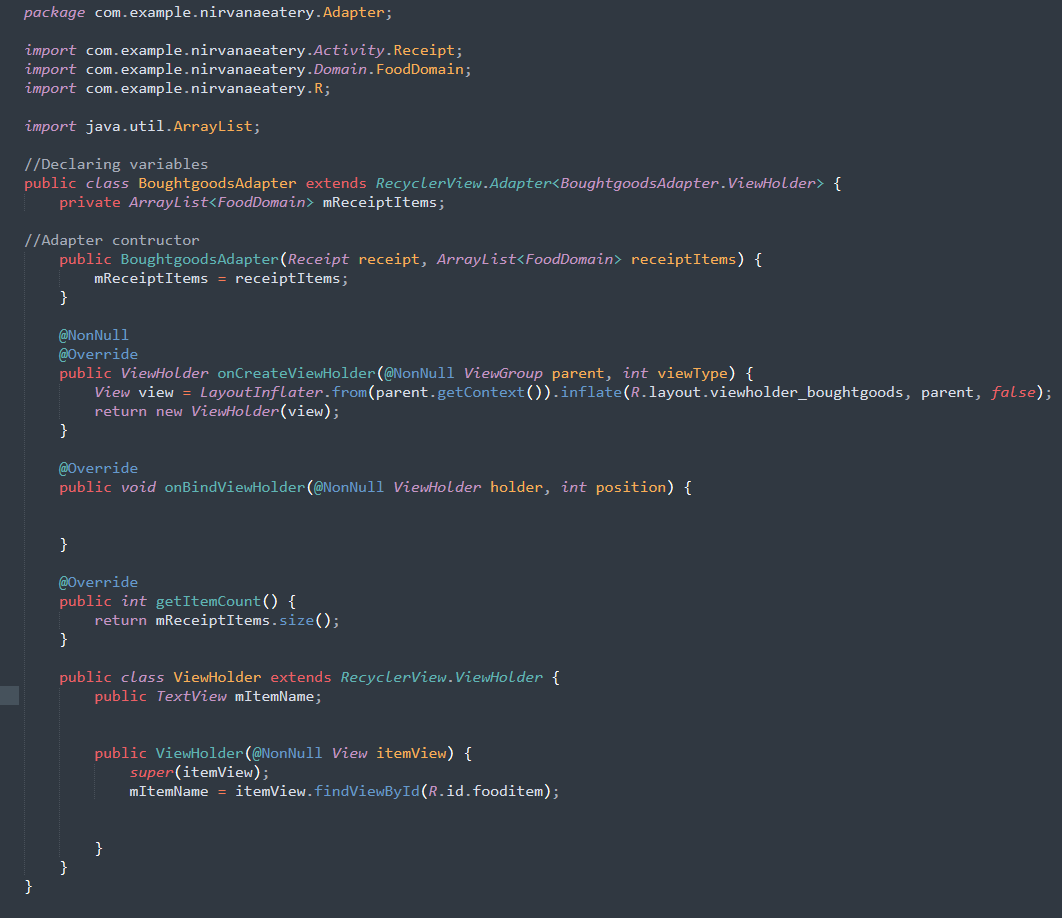


1. Create an XML Viewholder file of how you want the items to UI display top look like.
2. Create an Adapater Class.

The CategoryAdapter takes an ArrayList<CategoryDomain> as a constructor parameter, and stores it as an instance variable. The onCreateViewHolder() method inflates the viewholder\_category layout file and returns a new ViewHolder instance. The onBindViewHolder() method sets the CategoryName and CategoryImage views with data from the corresponding CategoryDomain object, and uses the Glide library to load the image resource corresponding to the category.

The ViewHolder class holds references to the layout elements that will be used to display the category data.

It looks like this adapter below is designed to be used with a RecyclerView to display a list of categories, with each category consisting of a name and an image.



# GRADLE DEPENDANCIES

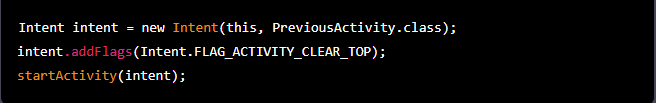
dependencies **{** implementation 'com.squareup.picasso:picasso:2.8'  
  
 implementation 'com.karumi:dexter:6.2.3'  
  
  
 implementation 'com.androidstudy.daraja:daraja:2.0.1'  
  
  
 //Added for Mobile country picker  
 implementation 'com.hbb20:ccp:2.6.1'  
  
 //Adding Glide capabilities for Adapter  
 implementation 'com.github.bumptech.glide:glide:4.13.2'  
 implementation 'com.google.code.gson:gson:2.9.1'  
  
 //Adding RoundedImageView  
 implementation 'com.makeramen:roundedimageview:2.3.0'  
  
 implementation 'androidx.appcompat:appcompat:1.4.2'  
 implementation 'com.google.android.material:material:1.6.1'  
 implementation 'androidx.constraintlayout:constraintlayout:2.1.4'  
  
 implementation 'androidx.navigation:navigation-fragment:2.5.1'  
 implementation 'androidx.navigation:navigation-ui:2.5.1'  
  
 implementation 'com.google.android.gms:play-services-location:20.0.0'  
 implementation 'com.google.android.gms:play-services-maps:18.1.0'  
  
 implementation 'com.google.firebase:firebase-analytics:21.1.0'  
 implementation 'com.google.firebase:firebase-core:21.1.0'  
 implementation 'com.google.firebase:firebase-firestore:24.2.2'  
 implementation 'com.google.firebase:firebase-auth:21.0.7'  
 implementation 'com.google.firebase:firebase-database:20.0.5'  
 implementation 'com.google.firebase:firebase-storage:20.0.1'  
 implementation 'androidx.room:room-common:2.4.3'  
 implementation 'androidx.room:room-runtime:2.4.3'  
  
  
 testImplementation 'junit:junit:4.13.2'  
 androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
 androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
 annotationProcessor 'androidx.room:room-compiler:2.4.3'  
**}**

# INTENT

## Set the intent to go to the previous activity

You can set the intent to go to the previous activity using the FLAG\_ACTIVITY\_CLEAR\_TOP flag. This flag will clear the current activity stack and bring the previous activity to the top.

Here is an example of how to set the intent to go to the previous activity:



Or

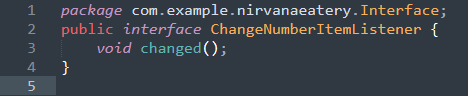
Alternatively, you can also use the finish() method to close the current activity and return to the previous activity.



This method, **however, doesn't take any input and returns to the previous activity without checking where you were before.**

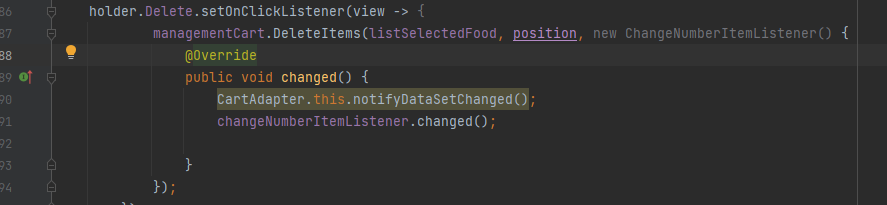
# ONCLICKLISTENER INTERFACE

Example of Interface from Nirvana Eatery Application



This is an interface in Java that defines a callback method for notifying listeners when the number of items in a list has changed. The interface has only one method, "changed", which is called to notify the listeners that the number of items has changed. Classes that implement this interface can register as listeners and will be notified whenever the number of items in the list changes. This is useful for updating the UI when the list changes, as well as for performing any other actions that are necessary when the list changes.

Example of Interface implemented in class from Nirvana Eatery Application



This code defines an onClickListener for the "Delete" button in the view holder. When the button is clicked, the "DeleteItems" method of the "managementCart" object is called, passing in the following parameters:

1. **listSelectedFood** - an ArrayList of selected FoodDomain objects
2. **position** - the position of the FoodDomain object in the ArrayList
3. **ChangeNumberItemListener** - an interface that has a single method, "changed".

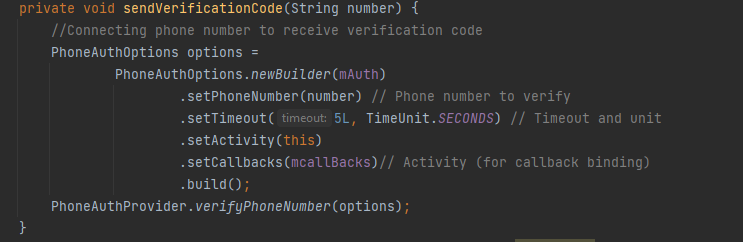
The "DeleteItems" method is responsible for removing the FoodDomain object at the specified position in the "listSelectedFood" ArrayList.

After the item is removed, the "notifyDataSetChanged" method of the adapter is called to refresh the view, and the "changed" method of the "changeNumberItemListener" interface is called to notify any listeners that the number of items has changed.

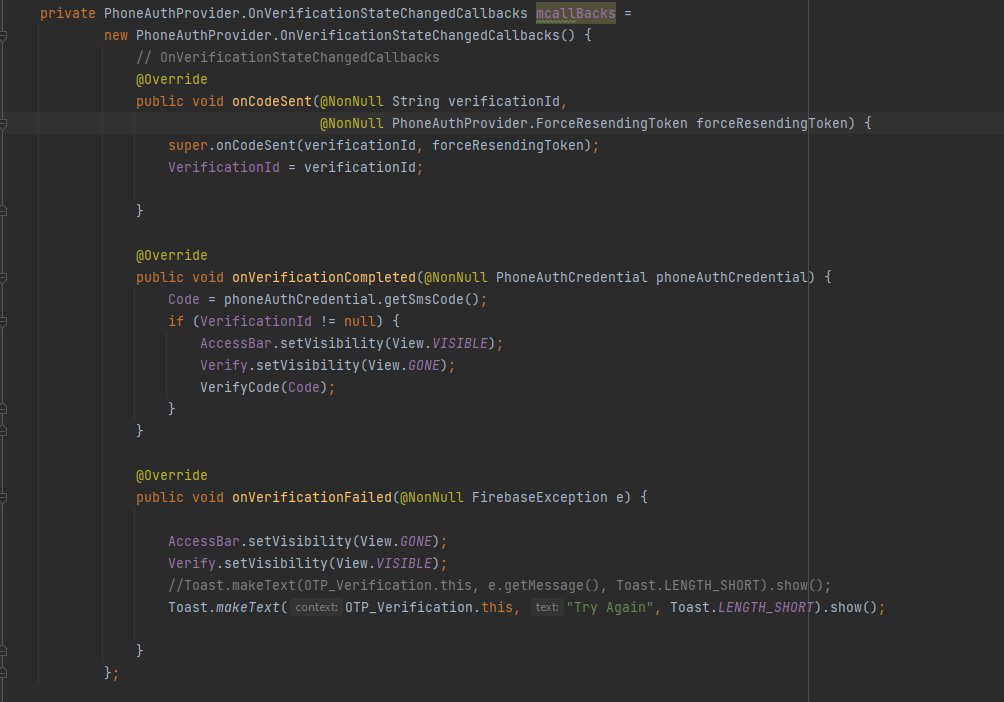
# FIREBASE

# FIREBASE PHONE AUTHENICATION

## Sending Verification Code to Number

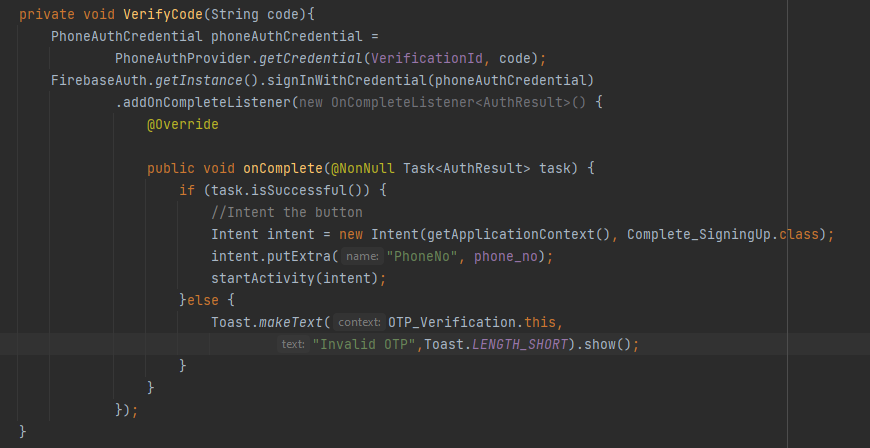


This is a Java method that sends a verification code to a phone number using Firebase Authentication's PhoneAuthProvider. The method takes in a phone number as a parameter, and uses it to build a PhoneAuthOptions object. The options include the phone number to verify, a timeout of 5 seconds, the current activity (for callback binding), and a callback object (mcallBacks) for handling the verification process. The method then calls the verifyPhoneNumber() method on the PhoneAuthProvider class with the options, which sends the verification code to the specified phone number.



This is a Java class that defines a callback object for handling the phone number verification process using Firebase Authentication's PhoneAuthProvider. The class implements the PhoneAuthProvider.OnVerificationStateChangedCallbacks interface, which provides three methods for handling different states of the verification process:

1. onCodeSent(): This method is called when the verification code has been sent to the specified phone number. It takes in two parameters: a verification ID and a ForceResendingToken. The method saves the verification ID to a class variable (VerificationId) so that it can be used later for verifying the code.
2. onVerificationCompleted(): This method is called when the verification process has been completed, either automatically (if the app can automatically read the verification code from the SMS) or manually (if the user entered the code manually). It takes in a PhoneAuthCredential object, which contains the verification code. The method gets the code from the PhoneAuthCredential object, and if the verification ID is not null, it calls the VerifyCode() method and passes in the code.
3. onVerificationFailed(): This method is called when the verification process has failed. It takes in a FirebaseException object, which contains information about the failure. The method show the message 'Try Again' as a toast message.

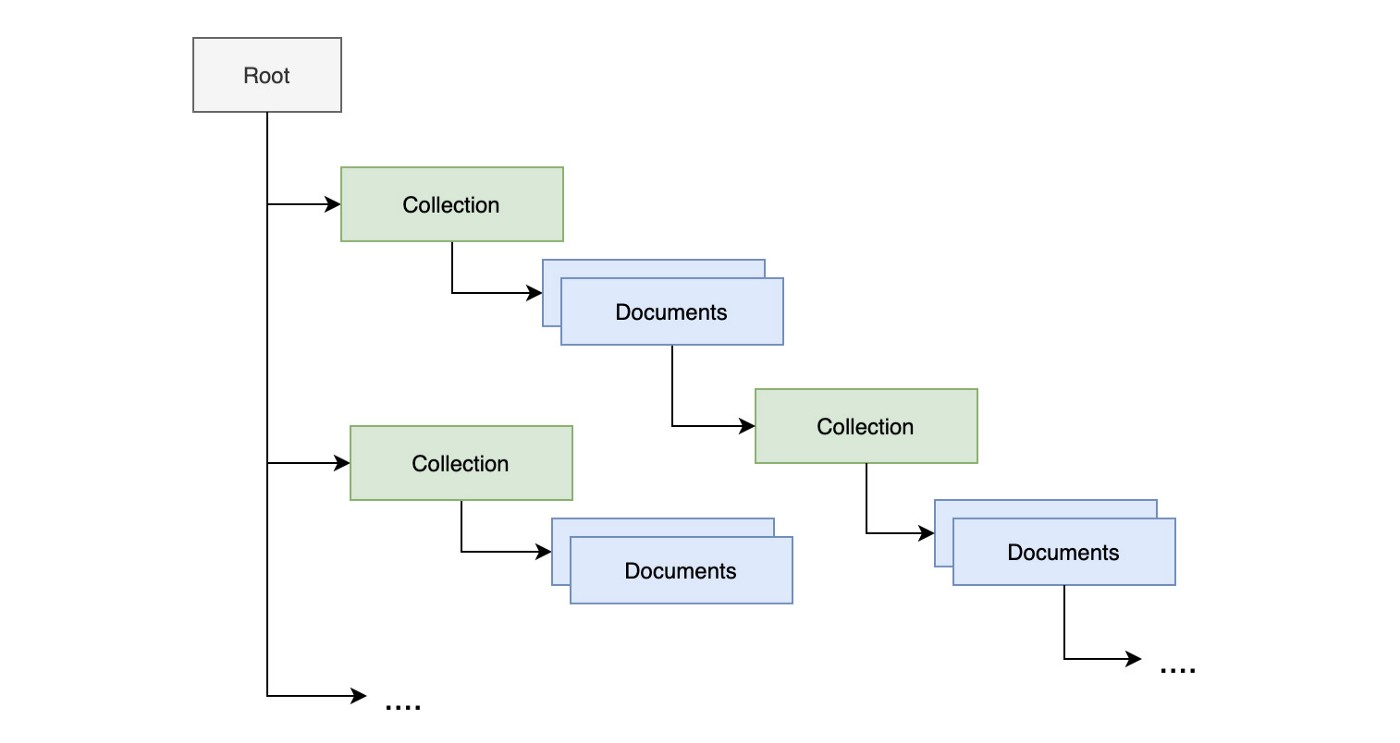


# FIREBASE DATABASE

Firebase uses NoSQL databases to store data in the cloud. Unlike traditional relational databases, NoSQL databases are designed to handle semi-structured or unstructured data, making it easier to store complex data structures in a single document. However, this ease of use can come at the cost of increased data redundancy and decreased data integrity. To mitigate these issues, it's important to normalize your data in Firebase.

## Normalization in Firebase

Normalization in Firebase involves breaking down complex data structures into smaller, more manageable components, and organizing those components in a way that reduces redundancy and improves data integrity. This is typically done by organizing data into collections and documents, which can be nested to form a hierarchical structure. For example, you might have a collection of users, each represented by a document containing information about the user. Within each user document, you could then have another collection of posts, each represented by a document containing information about a specific post.



Another common technique for normalizing data in Firebase is embedding. This involves storing related data directly within a single document, rather than breaking it out into separate collections and documents. For example, you might have a document representing a user, and within that document, you could embed data about the user's posts and comments. While embedding can make it easier to retrieve related data in a single query, it can also increase the size of your documents, which can have an impact on performance and cost.

Finally, denormalization is another technique that can be used to normalize data in Firebase. Denormalization involves duplicating data across multiple documents, in order to make it easier to retrieve and update data. For example, you might have a document representing a post, and within that document, you could store information about the user who wrote the post, rather than storing that information in a separate user document. Denormalization can improve query performance, but it also increases the risk of data inconsistency, as updates to the duplicated data may need to be made in multiple places.

In conclusion, data normalization in Firebase involves organizing your data in a way that reduces redundancy and improves data integrity. The specific approach you take will depend on your particular use case, the size and complexity of your data, and your performance requirements.

## Firebase data schema

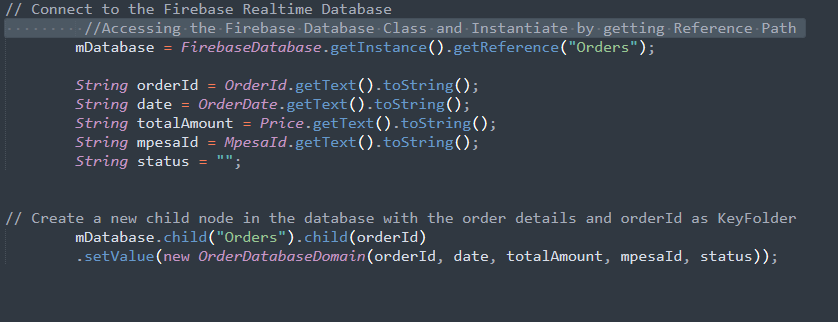
Firebase data schema refers to the structure and organization of data within a Firebase database. It includes the arrangement of collections, documents, and fields, and how they relate to each other. The data schema determines how data is stored, accessed, and updated within a Firebase database.

Firebase supports a flexible data schema, allowing developers to define their data structures as they see fit, without having to worry about setting up tables or columns in advance. This makes it easier to work with semi-structured or unstructured data, but it also means that careful planning is needed to ensure that data is stored in a way that supports efficient querying and updates.

A good data schema in Firebase should take into account the size and complexity of the data, the performance requirements of the application, and the expected volume of reads and writes. It should also consider data normalization techniques, such as embedding and denormalization, to reduce data redundancy and improve data integrity.

When designing a Firebase data schema, it's important to think about the relationships between collections, documents, and fields, and to choose the appropriate data modeling techniques to represent those relationships. This includes considering how data will be retrieved and updated, and how changes to the data will impact the rest of the application. With a well-designed data schema, Firebase can provide fast, efficient access to your data, and ensure that your application can scale as your data grows.

## Saving Data on Realtime FirbaseDatabase on the Application Code Way 1

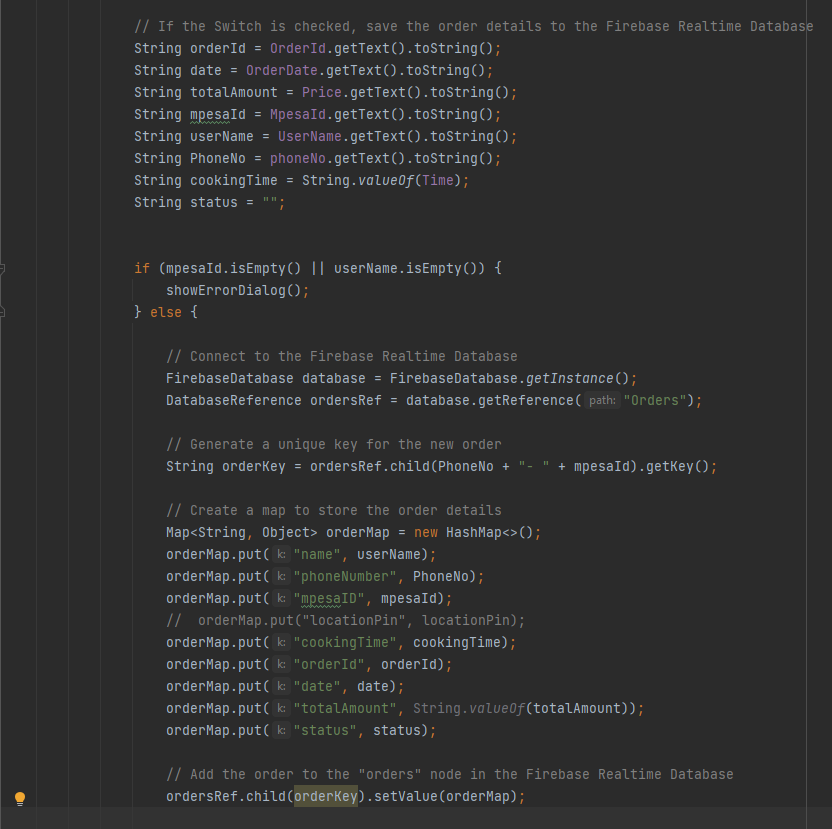


**Explanation**

The Firebase Database class is used to get an instance of the database, and then a reference is obtained to the "Orders" node in the database.

It then creates a new child node in the "Orders" node with the orderId as the key and stores the OrderDatabaseDomain object as its value. This OrderDatabaseDomain object is created with the orderId, date, totalAmount, mpesaId, and status.

## Saving Data on Realtime FirbaseDatabase on the Application Code Way 2

****

**Explanation**

First, the code initializes several variables with the values entered by the user in the respective fields. These variables include the order ID, order date, total amount, Mpesa ID, user name, phone number, cooking time, and order status.

The code connects to the Firebase Realtime Database and retrieves the reference to the "Orders" node. It then generates a unique key for the new order by calling the getKey() method on the child node that combines the phone number and Mpesa ID.

After that, the code creates a Map object to store the order details, which include the user's name, phone number, Mpesa ID, cooking time, order ID, order date, total amount, and status. The put() method is used to add each of these details to the Map object.

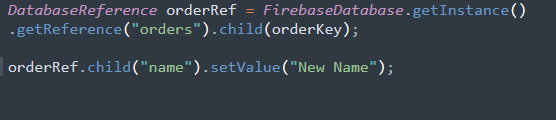
Finally, the code adds the order to the "Orders" node in the Firebase Realtime Database by calling the setValue() method on the ordersRef object and passing in the orderKey and orderMap objects as parameters. This saves the order details to the Firebase Realtime Database.

## Update Fields in The Firebase Realtime Database and Have Those Changes Automatically Reflected in The Application.

This is one of the key features of Firebase Realtime Database, as it allows for real-time data synchronization between the database and connected clients.

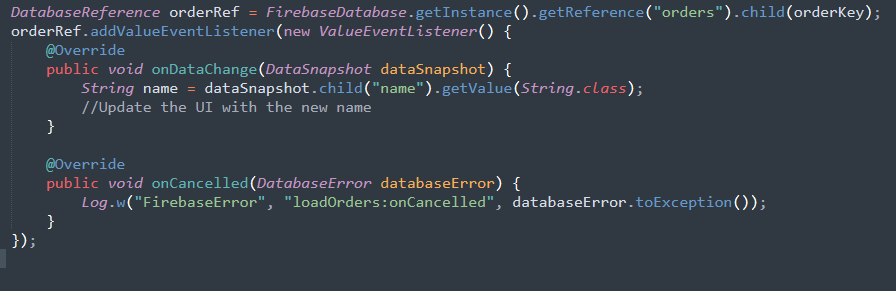
### Method I

To update a field in the Firebase Realtime Database, you can use the setValue() or updateChildren() method on a DatabaseReference object. For example, to update the name of an order, you can use the following code:



### Method II

You can also retrieve data from the Firebase Realtime Database in real-time using a ValueEventListener or ChildEventListener. These listeners allow you to receive updates whenever the data at a specific location in the database changes. For example, you can use the following code to listen for changes to the order's name:

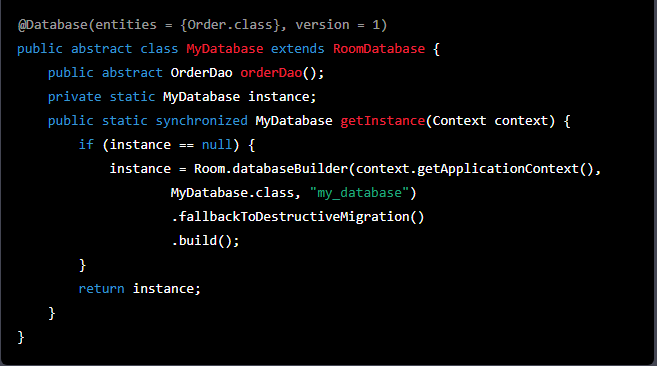


## Retrieving data from the database



# ROOM DATABASE

1. First, create a database class that extends **RoomDatabase** and annotate it with **@Database**. Inside this class, define an abstract method for each DAO (Data Access Object) that you want to use in your database, as well as a static field for each entity that you want to store in the database:



In this example, we have defined a static **getInstance** method that creates an instance of the **MyDatabase** class using the **Room.databaseBuilder()** method, which takes three parameters:

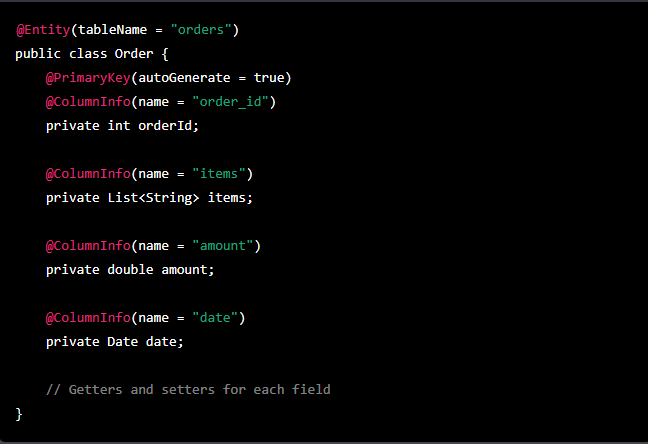
1. The **Context** of the application.
2. The class of the database.
3. The name of the database.

We have also specified the version of the database using the **version** parameter in the **@Database** annotation, and set the **fallbackToDestructiveMigration** option to **true** to allow Room to recreate the database if the version number changes.

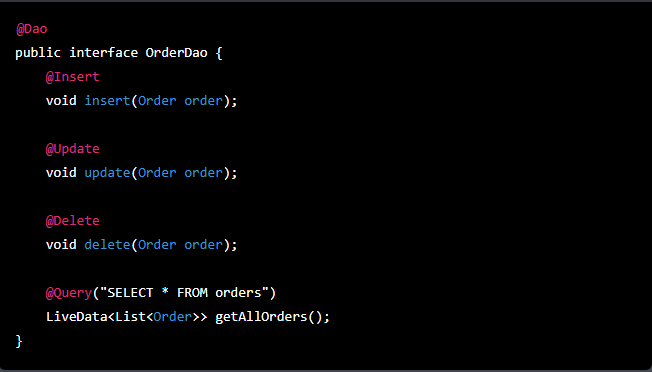
The **OrderDao** abstract method is defined to allow access to the **Order** entity and its methods. In this case, we have only defined one DAO for the **Order** entity, but you can define multiple DAOs for multiple entities if needed.

Note that the **instance** field is declared as **volatile** to ensure that all threads see the same value of the instance. The **synchronized** keyword is used to ensure that only one thread at a time can access the **getInstance()** method. This is important because creating a database instance is an expensive operation and we want to avoid creating multiple instances of the database.

1. Next, create an entity class for the **Order** table. An entity class represents a table in your database, and each field in the class represents a column in the table. Annotate the class with **@Entity** and specify the table name as a parameter. Annotate each field with **@ColumnInfo** and specify the name of the column that the field should be mapped to. You can also use the **@PrimaryKey** annotation to indicate that a field is the primary key for the table:



1. Create a DAO (Data Access Object) interface for the **Order** entity. A DAO is an interface that defines the methods that you want to use to access the data in your database. Annotate the interface with **@Dao** and define methods for inserting, updating, deleting, and querying for **Order** objects:



1. To use the database, you will need to create a **RoomDatabase.Callback** class and override the **onCreate** method. Inside the **onCreate** method, you can insert some initial data into the database if you want. In Activity.

