INTENTS: (Refer App: Intents)

An intent is basically a mechanism by which we can switch from one activity to another. It simply serves as a connector between one activity and another. In this app, we shall create 2 variations of intents.

1. A basic intent app where we just enter one activity from another at the click of a button.
2. A somewhat advanced app, wherein we pass some information from one activity and then access it from another activity using Intents.
3. THE BASIC APP:

Let us first lay the foundation for the app. We first need to design the two activities and then glue them together using intents. We proceed in the usual manner- first manipulating the xml files for designing the look and feel of the app, and then giving brains to the app using the java files.

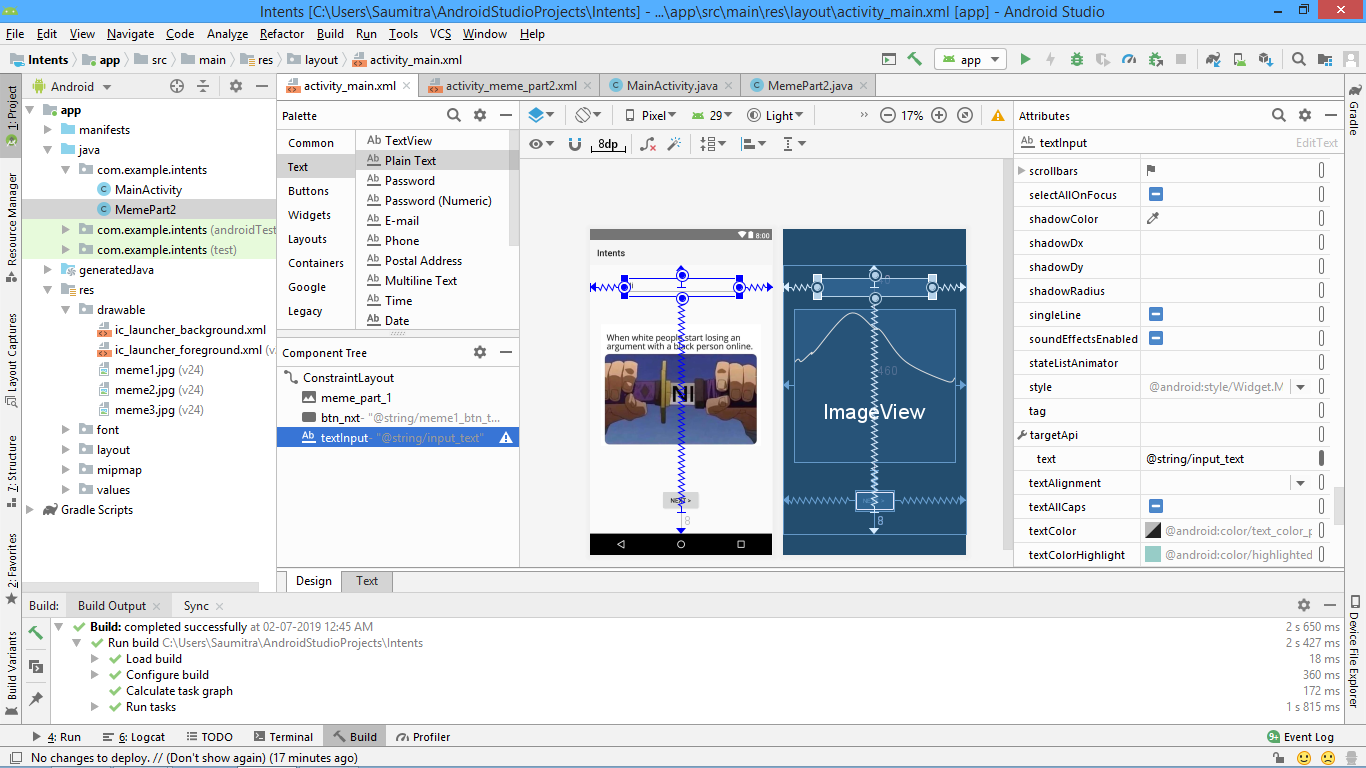
ACTIVITY\_MAIN.XML AND ACTIVITY\_MEME\_PART2.XML

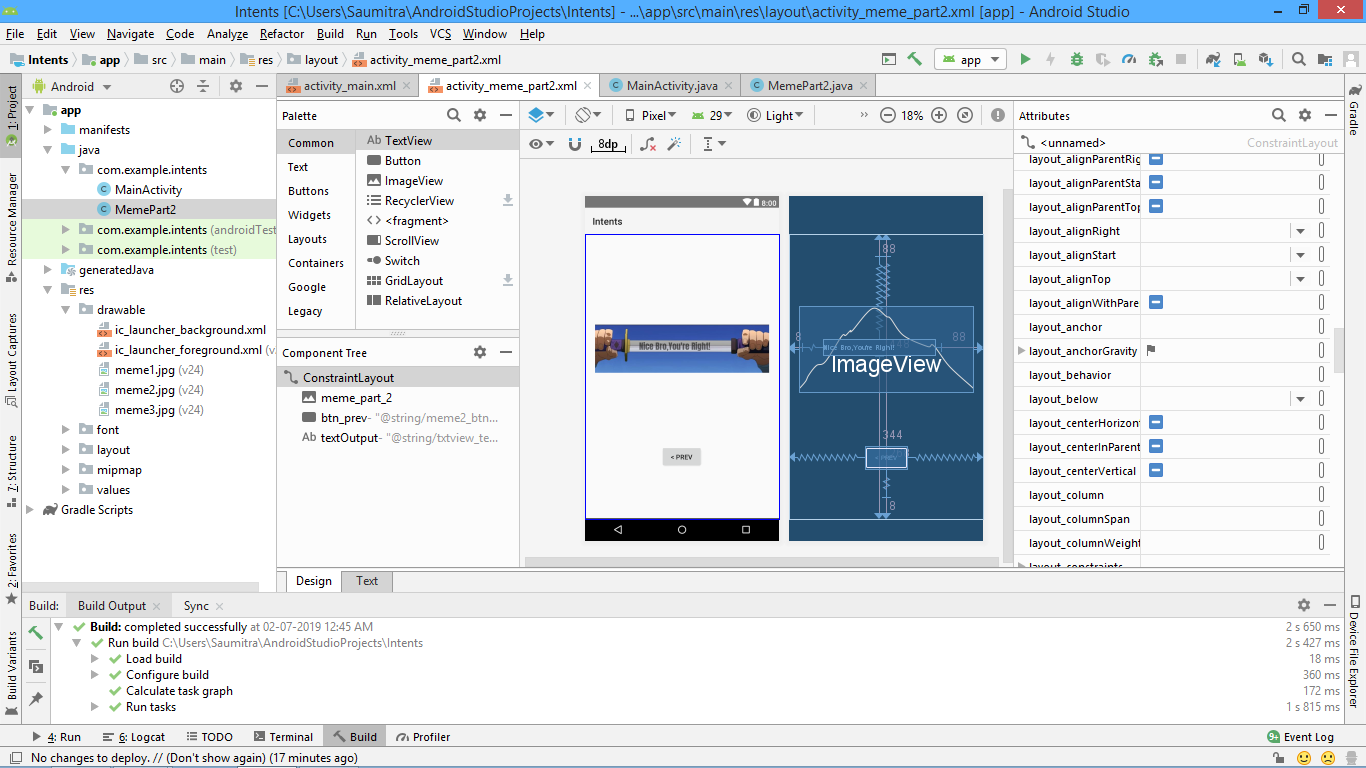
We shall be creating a simple meme where first part of the meme will be displayed in the first activity and then the second activity shall display the second part of the meme. We design the first part simply by using an imageview. Before we implement the imageView, we must copy the two meme image files in out res/drawable directory.

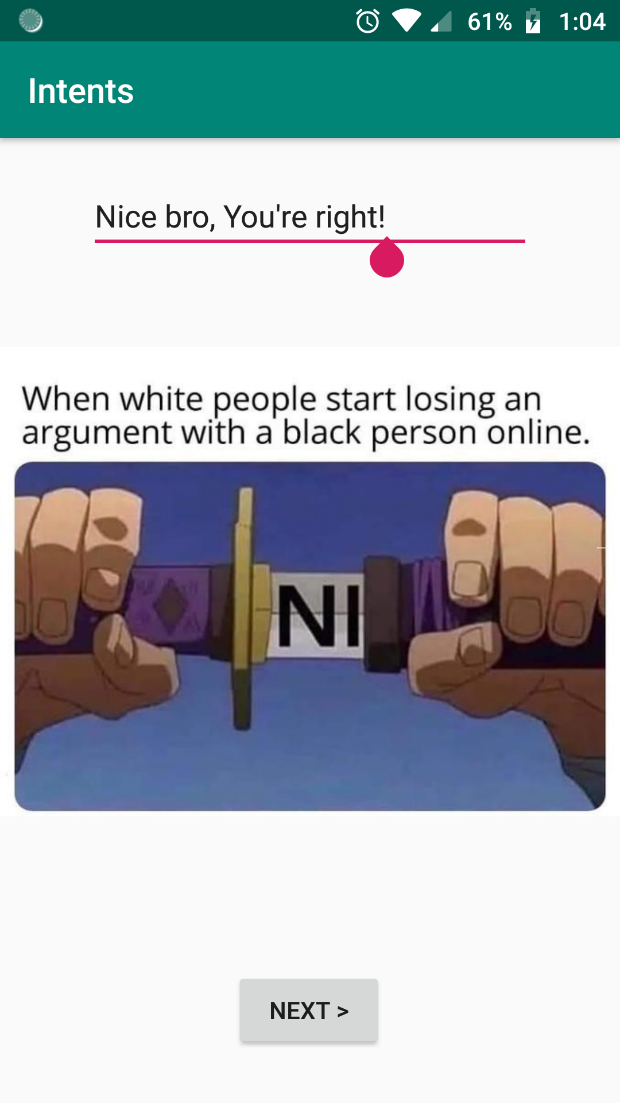
We also create a button which on clicking shall direct us to the second activity.

Now, we create a new activity by right clicking the package name of our app (under java/com.example.intents) and selecting new>activity>blank activity. We then design the second part of our meme using another imageview. We shall also include a button which shall again redirect us to the previous activity.

The final activities look like:





1. MAINACTIVITY.JAVA AND MEMEPART2.JAVA

In both the java files, we import the package android.content.intent. This allows us to use the functionality of the intents which bind different activities together.

**import** android.view.View;  
**import** android.widget.Button;  
**import** android.content.Intent;

In both the activities we create an onclicklistener for each of the buttons in the respective activities.

nextButton.setOnClickListener(

new Button.OnClickListener()

{

public void onClick(View view)

{

}

}

);

ACTUAL CODE:

1. MainActivity.java

Button nextbutton=findViewById(R.id.***btn\_nxt***);

nextbutton.setOnClickListener(  
 **new** Button.OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
  
 Intent myIntent;  
 myIntent = **new** Intent(getApplicationContext(),MemePart2.**class**);  
 startActivity(myIntent);  
  
  
 }  
 }  
);

1. MemePart2.java

Button prevbutton=findViewById(R.id.***btn\_prev***);

prevbutton.setOnClickListener(  
 **new** Button.OnClickListener()  
 {  
 **public void** onClick(View view)  
 {  
 Intent myIntent2=**new** Intent(getApplicationContext(),MainActivity.**class**);  
 startActivity(myIntent2);  
  
 }  
 }  
);

We see that we create a new instance of the Intent class and pass the current application context and the name of the .class file that the activity must switch to after clicking the button. Thus, for the next button, it is MemePart2.class and for the prevButton it is MainActivity.class.

We then call a method called startActivity which takes an intent as a parameter. This of course starts the activity that the intent is glued to.

There you have it! The program is ready, you can try it on your mobile device.

1. THE SLIGHTLY ADVANCED APP:

Here we shall make few tweaks to our already created program so as to make it a meme ‘creator’ rather that a meme ‘displayer’. We shall create an editText in our first activity and a TextView in our second activity. The text that we shall write in the first activity shall be displayed in the TextView in the second activity.

After positioning the UI elements in the xml files, we shall move on to the implementation of the intents. We first create a String variable and store the text that we record from the TextView into it. This should be written INSIDE THE ONCLICKLISTENER otherwise the program shall be pretty pointless.

**final** EditText inputText=findViewById(R.id.***textInput***);

nextbutton.setOnClickListener(  
 **new** Button.OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
  
 Intent myIntent;  
 myIntent = **new** Intent(getApplicationContext(),MemePart2.**class**);  
 **final** String userInput=inputText.getText().toString();  
 myIntent.putExtra(**"passedInput"**,userInput);  
 startActivity(myIntent);  
  
  
 }  
 }  
);

Before calling the startActivity method, we shall pass the accepted string into the intent. This is done with the help of the putExtra method. This method accepts 2 parameters: The keyword that the receiving activity shall refer to after receiving the passed data and the actual data (the string userInput in this case) which is to be passed.

ON THE RECEIVING END:

We now jump to the receiving end of the data. All the code here is written in the onCreate() method.Now we create an instance of the Bundle class.

SIDEBAR:

Refer: <https://www.journaldev.com/15872/android-bundle>

Android Bundle is used to pass data between activities. The values that are to be passed are mapped to String keys which are later used in the next activity to retrieve the values.

A Bundle is passed in the following way.

Intent intent = new Intent(this,SecondActivity.class);

Bundle bundle = new Bundle();

bundle.putString("key\_1", "MainActivity greeted you with a HI");

bundle.putBoolean("key\_2", true);

intent.putExtras(bundle);

startActivity(intent);

Data from a Bundle is retrieved in the SecondActivity.java in the following manner.

Bundle bundle = getIntent().getExtras();

String title = bundle.getString("key\_1");

boolean b = bundle.getBoolean("key\_2");

If the key doesn’t map to any value, it may lead to [NullPointerException](https://www.journaldev.com/14544/java-lang-nullpointerexception). Hence it’s recommended to add null checks for the Bundle as well as the retrieved values.

After creating the Bundle variable, we set it equal to getIntent().getExtras(); We also set a check to verify that it is not null. This check (although it doesn’t make much of a difference in our app) is used to ensure that the app doesn’t crash when it is entering THIS particular activity from any other activity where no data is passed.

Bundle meme1data=getIntent().getExtras();  
**if**(meme1data==**null**)  
{  
 **return**;  
}  
**else**{  
 String toDisplay=meme1data.getString(**"passedInput"**);  
 outputText.setText(toDisplay);  
}

We create an instance of the String class and set it equal to the string passed by the intent. This is done using the getString method which accepts 1 parameter – The keyword referring the sent string (“passedInput” in this case).

All that remains now is to display the received string in the TextView, accomplished easily by the setText method.

THREADS AND THREAD HANDLERS:

Oh yeah! We’re going there…

Now, manier times it might occur that your app begins executing something when you click on a button. If that particular calculation is time consuming, or requires a lot of query fetches and table insertion, or is very complex, then your app might appear to freeze in the foreground, as the app cannot execute any more button clicks WHILE it is processing data at the same time.

As the app is frozen, it might create an uncomfortable feeling to the user who naturally doesn’t understand the under-the hood processes that are actually working to provide the desired output.

In order to resolve this issue, we usually take the help of THREADS.

SIDEBAR:

A thread is an execution context, which is all the information a CPU needs to execute a stream of instructions.

Suppose you're reading a book, and you want to take a break right now, but you want to be able to come back and resume reading from the exact point where you stopped. One way to achieve that is by jotting down the page number, line number, and word number. So your execution context for reading a book is these 3 numbers.

If you have a roommate, and she's using the same technique, she can take the book while you're not using it, and resume reading from where she stopped. Then you can take it back, and resume it from where you were.

Threads work in the same way. A CPU is giving you the illusion that it's doing multiple computations at the same time. It does that by spending a bit of time on each computation. It can do that because it has an execution context for each computation. Just like you can share a book with your friend, many tasks can share a CPU.

On a more technical level, an execution context (therefore a thread) consists of the values of the CPU's registers.

Last: threads are different from processes. A thread is a context of execution, while a process is a bunch of resources associated with a computation. A process can have one or many threads.

A better analogy would equate person with CPU (both *do* something), and equate book with address-space (both just exist). That way, bookmarks in different books are like threads in different processes. A single book with more than one bookmark would be the analog of a multi-threaded process, which is what people usually mean when they say "threads." It works for a single processor machine, but it breaks down somewhat when you talk about multi-processing. Nobody cares which CPU executes function f(), but it *does* matter which person reads chapter 11.

Processes are like two people using two different computers, who use the network to share data when necessary. Threads are like two people using the same computer, who don't have to share data explicitly but must carefully take turns.

Conceptually, threads are just multiple worker bees buzzing around in the same address space. Each thread has its own stack, its own program counter, etc., but all threads in a process share the same memory. Imagine two programs running at the same time, but they both can access the same objects.

Contrast this with processes. Processes each have their own address space, meaning a pointer in one process cannot be used to refer to an object in another (unless you use shared memory).

I guess the key things to understand are:

* Both processes and threads can "run at the same time".
* Processes do not share memory (by default), but threads share all of their memory with other threads in the same process.
* Each thread in a process has its own stack and its own instruction pointer.

1. EXECUTION WITHOUT THREADS:

We now wish to demonstrate the difference between a multithreaded and a non-multithreaded process.

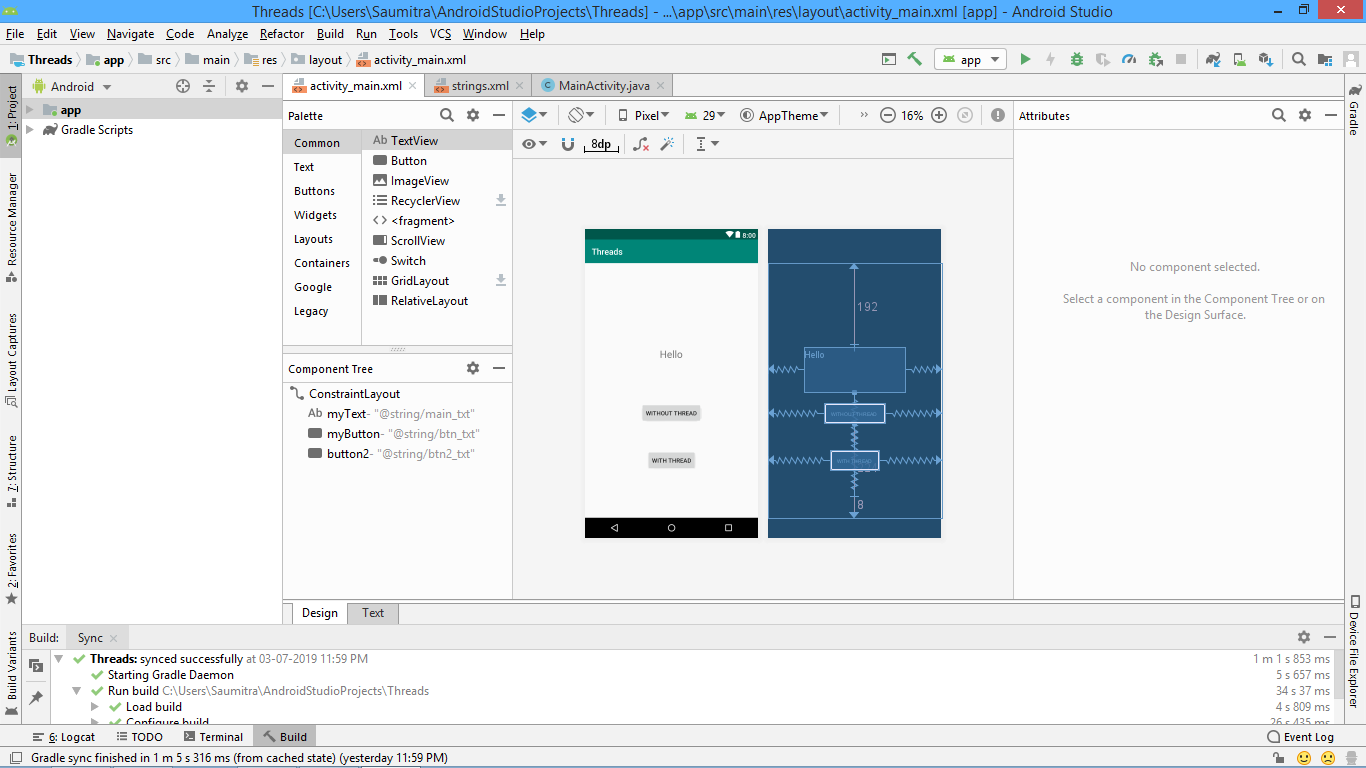
In order to simulate a complex calculation, we simply ask the app to wait for 10 seconds. We design a pretty simple UI with a single textview and a single button. On pressing the button, the app will wait for 10 seconds and then the message in the textview shall change. We also add an onClick attribute to the button in the xml file itself so that we shall not be required to write a separate onClickListener for the button.

In the java file, we shall write the code to wait for 10 seconds. We write this code in the onClick function that we had specified in the xml file. We define a long variable called futuretime and set it equal to current system time+10,000 milliseconds.(10 seconds)

We set a boolean variable to false and when the current time reaches futuretime, we set the variable to true and exit out of the while loop. Otherwise, we stay in the while loop until the variable is false and wait for futuretime-currenttime. (Of course this while loop will always run only once)

**long** futureTime = System.*currentTimeMillis*() + 10000;  
**boolean** test = **false**;  
**while** (test == **false**) {  
 **synchronized** (**this**) {  
 **try** {  
 wait(futureTime - System.*currentTimeMillis*());  
 test = **true**;  
 } **catch** (Exception e) {  
  
 }  
 }  
Then we set the textview to some other message.

mytext.setText(**"Thank you for wasting 10 seconds of your life!"**);



The synchronised keyword (although not required here) ensures that all the threads of the app run in a synchronised manner with no thread overlapping the other. The wait keyword requires a customary try-catch block and hence is included.

On executing the above app, we see that once the button is pressed, the app freezes and no UI element can be pressed again. The button and other UI elements are unresponsive. This gives a feeling of a ‘HANGED’ application.

1. EXECUTION WITH THREADS:

Let us now make this app responsive by dividing the work among the processor cores. We create an object of the Runnable class. The runnable class being an abstract class imposes on us to compulsorily override a method called run(). It is inside this method that we paste the same exact code where the app waits for 10seconds.

Runnable myrun = **new** Runnable() {  
 @Override  
 **public void** run() {  
 **long** futureTime = System.*currentTimeMillis*() + 10000;  
 **boolean** test = **false**;  
 **while** (test == **false**) {  
 **synchronized** (**this**) {  
 **try** {  
 wait(futureTime - System.*currentTimeMillis*());  
 test = **true**;  
 } **catch** (Exception e) {  
  
 }  
 }  
  
 }

Now, that we have our runnable process defined, let us allocate a thread to it. This thread ensures that some of the many cores of the processor will do the job of waiting 10secs and the front end of the app will be handled by the default thread of the activity.

In order to do so, we create an object of the Thread class. This object needs the runnable object as a parameter in the constructor. We then use the start() method to start the runnable process.

Thread myThread=**new** Thread(myrun);  
myThread.start();

Now, we currently have not written the code to change the text in our app. There is of course a reason for this. It is usually good programming practice to handle UI elements in a SEPARATE handler and NOT INSIDE A THREAD. This is because, if we have multiple threads in our app, and all the threads try to update the same UI element, the there is a possibility that our app might crash. Thus we write the code to handle the change of text in the textview in a separate handler.

We create an object of the Handler class. Inside the curly braces after the object declaration, we override a method called: handleMessage. (Use alt+insert to override methods)

Inside this handle message method, we write the code to change the UI.

Handler **myHandler**=**new** Handler()  
{  
 @Override  
 **public void** handleMessage(@NonNull Message msg) {  
  
 TextView mytext = findViewById(R.id.***myText***);  
 mytext.setText(**"Thank you for wasting 10 seconds of your life!"**);  
 }  
};

We call the method sendEmptyMessage at the end of the while loop to update the UI after 10 seconds have passed.

**myHandler**.sendEmptyMessage(0);

The entire java file is as follows:

**package** com.example.threads;  
  
**import** androidx.annotation.NonNull;  
**import** androidx.appcompat.app.AppCompatActivity;  
  
**import** android.os.Bundle;  
**import** android.os.Handler;  
**import** android.os.Message;  
**import** android.view.View;  
**import** android.widget.TextView;  
  
**public class** MainActivity **extends** AppCompatActivity {  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 }  
  
 **public void** onButtonClick(View view) {  
 TextView mytext = findViewById(R.id.***myText***);  
 mytext.setText(R.string.***main\_txt***);  
 **long** futureTime = System.*currentTimeMillis*() + 10000;  
 **boolean** test = **false**;  
 **while** (test == **false**) {  
 **synchronized** (**this**) {  
 **try** {  
 wait(futureTime - System.*currentTimeMillis*());  
 test = **true**;  
 } **catch** (Exception e) {  
  
 }  
 }  
  
 mytext.setText(**"Thank you for wasting 10 seconds of your life!"**);  
 }  
  
 }  
  
 Handler **myHandler**=**new** Handler()  
 {  
 @Override  
 **public void** handleMessage(@NonNull Message msg) {  
  
 TextView mytext = findViewById(R.id.***myText***);  
 mytext.setText(**"Thank you for wasting 10 seconds of your life!"**);  
 }  
 };  
  
 **public void** onButton2Click(View view) {  
 TextView mytext = findViewById(R.id.***myText***);  
 mytext.setText(R.string.***main\_txt***);  
 Runnable myrun = **new** Runnable() {  
 @Override  
 **public void** run() {  
 **long** futureTime = System.*currentTimeMillis*() + 10000;  
 **boolean** test = **false**;  
 **while** (test == **false**) {  
 **synchronized** (**this**) {  
 **try** {  
 wait(futureTime - System.*currentTimeMillis*());  
 test = **true**;  
 } **catch** (Exception e) {  
  
 }  
 }  
  
 }  
 **myHandler**.sendEmptyMessage(0);  
 }  
  
 };  
 Thread myThread=**new** Thread(myrun);  
 myThread.start();  
 }  
}

SERVICES:

Till now we have seen apps that have always had a UI. But, suppose we have an app which wants to periodically fetch data from a server or a database, then, we needn’t have a UI element for that. In this case, we do however need a procedure that would run in the background that would fetch the data from the database or server.

Such a UI less procedure is called a ‘Service’. A service can be periodically triggered at certain intervals of time, or can also be triggered whenever a particular activity is launched. Based on this classification, we have 2 different types of services:

* 1. Intent Service
  2. A normal service

INTENT SERVICE:

An intent service is triggered whenever a particular activity is launched. In order to create an Intent Service, we right click our package name in the java folder, and create a new class.

Here we shall recycle our old Intent app where we created a meme of a sword pull. The service will simply print a log message indicating that the service has started functioning. Now, in order to convert our class into a service, we need to satisfy 3 requirements:

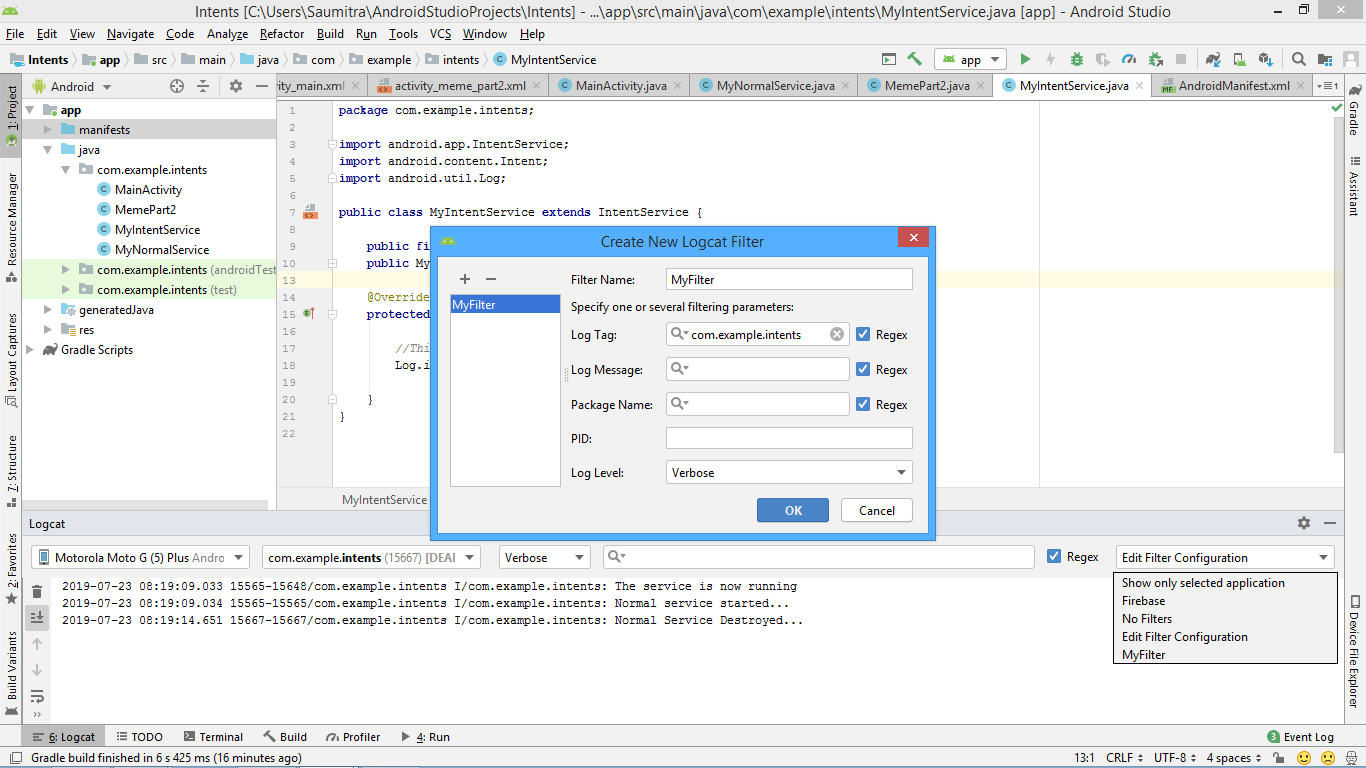
1. The class needs to extend the class IntentService
2. The class needs to have a constructor
3. The class needs to override a method called “OnHandleIntent”.

The constructor as well as the method can be easily overridden using alt+insert.

Now, inside this onHandleIntent method, we shall write the code which we want to execute whenever the required activity is launched. The java code is as follows:

**package** com.example.intents;  
  
**import** android.app.IntentService;  
**import** android.content.Intent;  
**import** android.util.Log;  
  
**public class** MyIntentService **extends** IntentService {  
  
 **public final** String **TAG**=**"com.example.intents"**;  
 **public** MyIntentService() {  
 **super**(**"MyIntentService"**);  
 }  
  
 @Override  
 **protected void** onHandleIntent(Intent intent) {  
  
 *//This is what the service does* Log.*i*(**TAG**,**"The service is now running"**);  
  
 }  
}

The string TAG is used as the identifier for the log messages. The method log.i is used to display the required log message. We also need to create a custom filter which shall only display our custom log messages.



We also need to specify the activity on whose launch the service will be started. This can be done by creating a new intent which shall start MyIntentService.class. This intent is created on the first meme page…

Intent meow=**new** Intent(getApplicationContext(),MyIntentService.**class**);  
startService(meow);

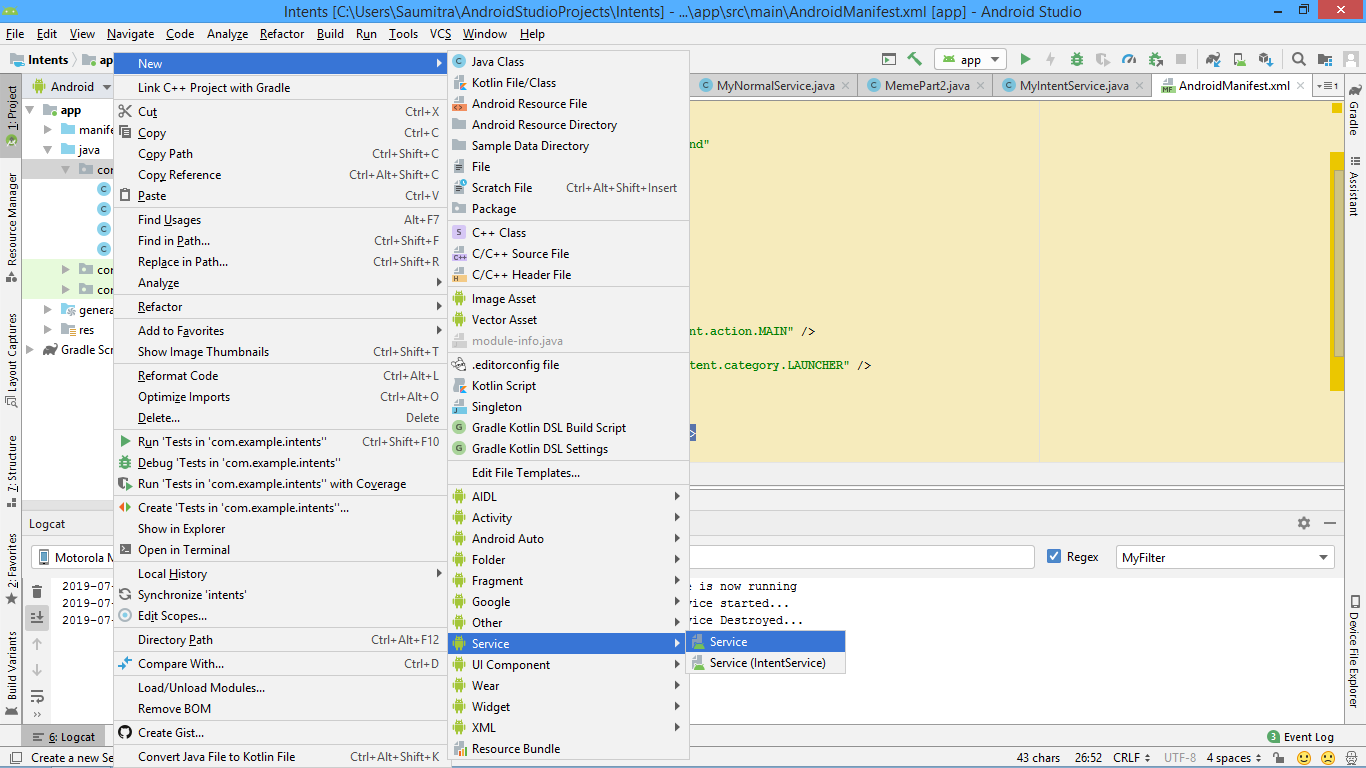
We also need to modify the manifest file to specify that we indeed are using an IntentService:

<**service android:name=".MyIntentService"** />

This tag should be added after the activity tag.

NORMAL SERVICE:

Now, there is yet another way by which we can create a Service. This is automatically created for us by Android Studio. If we right click our java package and select new-> service-> Service:



This will automatically create a new service for us…

Here we shall again print some log messages when the service is started. For this we shall have to override the method onStartCommand and (if needed) onDestroy…

It is also compulsory for us to write a constructor, but the constructor has already been automatically added to our class by android studio. Android studio also gives us another method called onBind. Currently we do not need this method, but we shall need this whenever we are dealing with Bound Services. So, currently we return null.

Here we need to manually create a new thread while in the previous example, Android automatically created a new thread for us. But here, we need to manually create a new thread. So, we come back to our approach of creating a runnable and overriding the method run(). Inside this, we shall simply insert a toast to signify that the service has indeed been started.

Outside the runnable, we create an object of the Thread class and pass the runnable into it. Finally, we start the thread. Using the start() method.

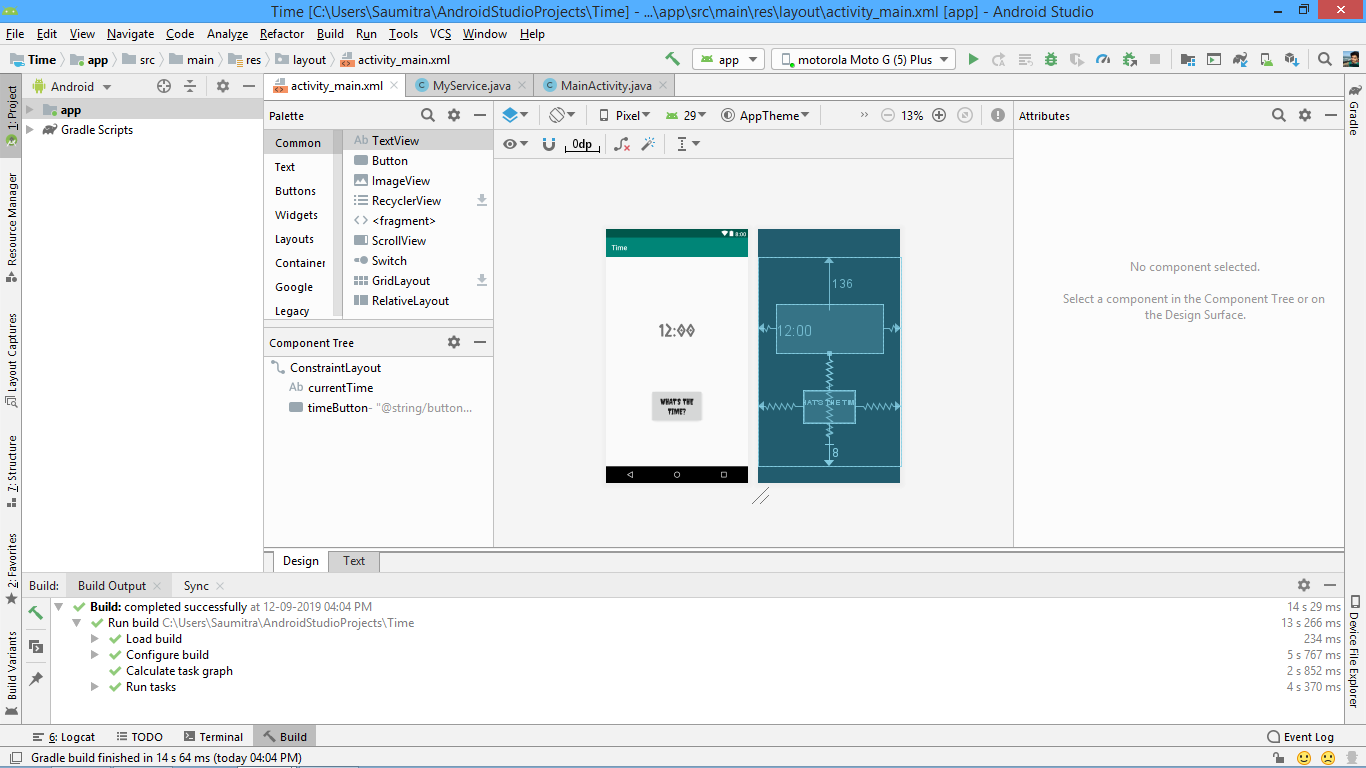
The method onStartCommand returns an int value. We in this case return the constant Service.START\_STICKY. This simply means that the service will be restarted, if by any chance it is disabled by the android OS. It immediately follows that if the service return Service.START\_NOT\_STICKY, then the service will not be restarted if it accidentally gets disabled by the android OS.

**package** com.example.intents;  
  
**import** android.app.Service;  
**import** android.content.Intent;  
**import** android.os.Handler;  
**import** android.os.IBinder;  
**import** android.os.Message;  
**import** android.util.Log;  
**import** android.widget.Toast;  
  
**import** androidx.annotation.NonNull;  
  
**public class** MyNormalService **extends** Service {  
  
 Handler **myHandler**=**new** Handler()  
 {  
 @Override  
 **public void** handleMessage(@NonNull Message msg) {  
  
 Toast mytoast=Toast.*makeText*(getApplicationContext(),**"Normal Service Started..."**,Toast.***LENGTH\_SHORT***);  
 mytoast.show();  
 *//super.handleMessage(msg);* }  
 };  
  
 **private static final** String ***TAG2***=**"com.example.intents"**;  
 **public** MyNormalService() {  
 }  
  
 @Override  
 **public int** onStartCommand(Intent intent, **int** flags, **int** startId) {  
 *//return super.onStartCommand(intent, flags, startId);* Log.*i*(***TAG2***,**"Normal service started..."**);  
  
 Runnable r=**new** Runnable() {  
 @Override  
 **public void** run() {  
  
 **myHandler**.sendEmptyMessage(0);  
  
 }  
 };  
  
 Thread myThread=**new** Thread(r);  
 myThread.start();  
 **return** Service.***START\_STICKY***;  
 }  
  
 @Override  
 **public void** onDestroy() {  
 *//super.onDestroy();* Log.*i*(***TAG2***,**"Normal Service Destroyed..."**);  
 }  
  
 @Override  
 **public** IBinder onBind(Intent intent) {  
 *//* ***TODO: Return the communication channel to the service.*** *//throw new UnsupportedOperationException("Not yet implemented");* **return null**;  
  
 }  
}

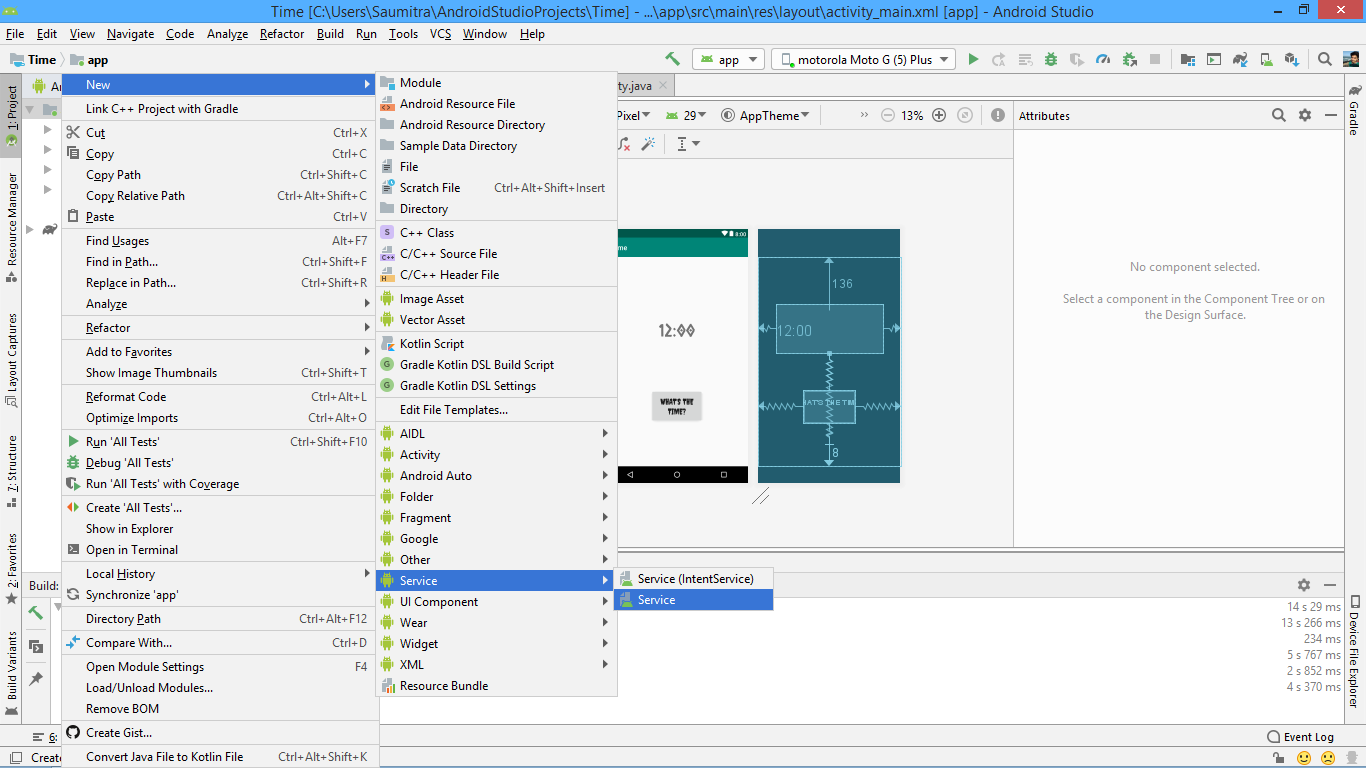
BOUND SERVICES:

Till now we have seen an intent service, and a normal service (which was just another manual way of creating an intent service). We shall now see another type of service called ‘Bound Service’. A bound service allows us to better communicate with the host app. It can also allow us to regularly schedule a service to run at specified intervals of time. In order to demonstrate this, we shall create a new app that shall display the current time whenever a button is pressed.

The UI for the app is pretty simple. It just contains a textview that shall update the current time, and a button that shall update the time whenever the user clicks it. Later, we will automate the service, so that the time is updated every second.



We shall now move on to create a new service, (A normal one) like we did in the previous example.



Here, we shall focus on the java file of the service. In order to communicate with the host program activity, we need to import android.os.Binder. We also need to import the utilities for accessing the current system time and displaying it in the appropriate format. This can be done by java.text.SimpleDateFormat , java.util,Date and java.util.Locale. The final imports list is as follows:

THESE ARE AUTOMATICALLY IMPORTED:

**import** android.app.Service;  
**import** android.content.Intent;  
**import** android.os.IBinder;

THESE ARE MANUALLY IMPORTED:  
**import** android.os.Binder;  
**import** java.util.Date;  
**import** java.text.SimpleDateFormat;  
**import** java.util.Locale;

Lets divide the functionality of this app into 3 sections:

1. SETTING UP THE SERVICE JAVA FILE (The confusing part)
2. SETTING UP THE MAIN ACTIVITY JAVA FILE (The fairly easy part)
3. AUTOMATING THE SERVICE (The moderately easy part)

So, crack your knuckles, and let’s begin:

***SETTING UP THE SERVICE JAVA FILE:***

Initally, we see only two methods in our file. One is a constructor to the MyService class, and the other is a method called onBind which returns an object of the IBinder class.

Inside the constructor, we shall create a new REFERENCE VARIABLE of the IBinder class and set it equal to an OBJECT of the MyLocalBinder class. We at this point need to remember 2 things:

1. Doing so WILL give you an error, as we DO NOT have any class called MyLocalBinder. We need to manually create it.
2. We also need to remember the basic java fact that REFERENCE VARIABLES OF THE SUPER CLASS CAN STORE REFERENCES TO OBJECT OF THEIR DERIVED CLASSES. A question now may arise… If we are manually creating the class called MyLocalBinder, then HOW CAN IT BE A SUPER CLASS OF **ANY** CLASS? In order to resolve this, we make MyLocalBinder **extend**  the Binder class.

Our file now looks like:

**public class** MyService **extends** Service {  
  
 **public final** IBinder **myBinder** =**new** MyLocalBinder();  
 **public** MyService() {  
 }  
  
 @Override  
 **public** IBinder onBind(Intent intent) {  
 *//* ***TODO: Return the communication channel to the service.*** *//throw new UnsupportedOperationException("Not yet implemented");*   
 }

KEY THING:

We shall now create the aforementioned class called MyLocalService. **BUT:**

**THE CLASS WILL BE A NESTED CLASS…**

Thus, we have a class called MyLocalBinder INSIDE the MyService class. This class shall have only one method, which will be return an object of the MyService class (The main class that we are currently writing our code into). Why do we do this? To answer this question, we need to remember another key java concept:

A NESTED CLASS CANNOT ACCESS THE METHODS OF THE OUTSIDE CLASS.

We don’t want our program to get stuck inside this class. We need the program to access the other methods in the MyService class. So, in our method, we return a reference to the class of the calling object (i.e. the MyService class) using the ‘this’ keyword.

**public class** MyLocalBinder **extends** Binder *//This is a nested class inside the MyService class*{  
 MyService getService()  
 {  
 **return** MyService.**this**;  
 }  
}

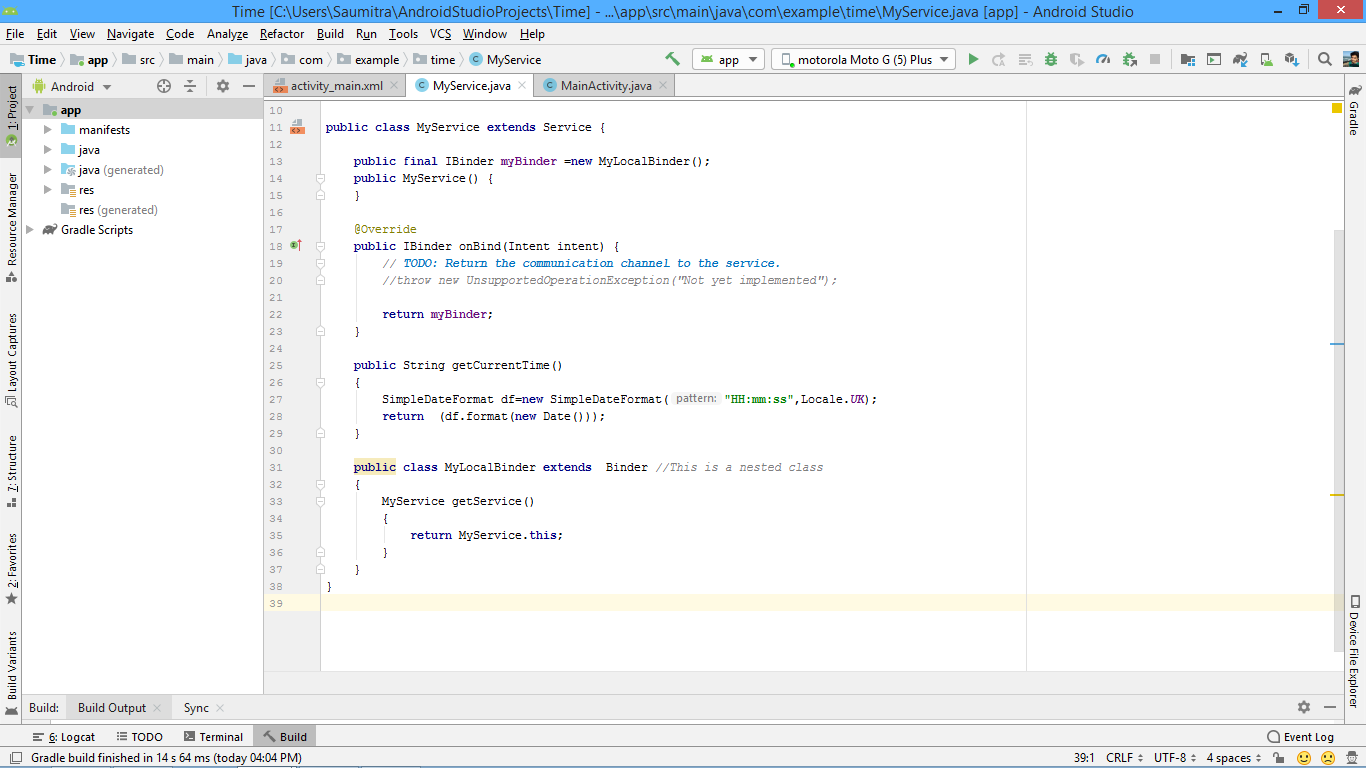
We also need our onBind method to return the myBinder object that we previously created. The flow of control will be clear in a moment:

ANOTHER KEY CONCEPT:

The reason were doing this is as follows:

Whenever the host app connects to our service, it looks for a method called onBind().

Then, it will be redirected to the superclass reference variable myBinder which stores an object of the MyLocalBinder class (The nested class). The class then contains a method which returns a reference to its outer class, thus giving access to all the methods inside the class.



4. Which will give it access to the entire class

3. Then inside the nested class

2. It will be directed here

1. The host program will arrive here

We now create a new method called getCurrentTime which will format our system time into HH:MM:SS format. This method returns a string. We create an object of the SimpleDateFormat class, and pass the time format and the UK locale to the object constructor. It returns a string that is in turn returned by a new Date object.

**public** String getCurrentTime()  
{  
 SimpleDateFormat df=**new** SimpleDateFormat(**"HH:mm:ss"**,Locale.***UK***);  
 **return** (df.format(**new** Date()));  
}

The entire java code looks like:

**package** com.example.time;  
  
**import** android.app.Service;  
**import** android.content.Intent;  
**import** android.os.IBinder;  
**import** android.os.Binder;  
**import** java.util.Date;  
**import** java.text.SimpleDateFormat;  
**import** java.util.Locale;  
  
**public class** MyService **extends** Service {  
  
 **public final** IBinder **myBinder** =**new** MyLocalBinder();  
 **public** MyService() {  
 }  
  
 @Override  
 **public** IBinder onBind(Intent intent) {  
 *//* ***TODO: Return the communication channel to the service.*** *//throw new UnsupportedOperationException("Not yet implemented");* **return myBinder**;  
 }  
  
 **public** String getCurrentTime()  
 {  
 SimpleDateFormat df=**new** SimpleDateFormat(**"HH:mm:ss"**,Locale.***UK***);  
 **return** (df.format(**new** Date()));  
 }  
  
 **public class** MyLocalBinder **extends** Binder *//This is a nested class* {  
 MyService getService()  
 {  
 **return** MyService.**this**;  
 }  
 }  
}

***SETTING UP THE MAIN ACTIVITY JAVA FILE:***

First, let us take care of the imports. These imports are essential, some of them are autoimported as we type the code.

**import** androidx.annotation.NonNull;  
**import** androidx.appcompat.app.AppCompatActivity;  
  
**import** android.content.ComponentName;  
**import** android.content.Context;  
**import** android.content.Intent;  
**import** android.content.ServiceConnection;  
**import** android.os.Handler;  
**import** android.os.IBinder;  
**import** android.os.Message;  
**import** android.widget.Button;  
**import** android.widget.TextView;  
**import** android.view.View;

We also need to import the inner nested class by explicitly writing the required import statement.

**import** com.example.time.MyService.MyLocalBinder;

Now, we need our host to connect to the service we just created. In order to do so, we need an object of the ServiceConnection class. We create such an object by using the concept of anonymous classes. This class defines 2 important methods. One which will decide what do we want to happen once we connect to the service, (onServiceConnected) and the other which will decide what to do, once we disconnect from the service (onServiceDisconnected). We also create a boolean variable called isBound and initially set it to false. This variables signifies whether the service is bound to the app or not. We also create an object of the myService class called lets say “myobject”.

MyService **myobject**;  
**boolean isBound**=**false**;

Inside the onServiceConnected, we want to get access to the class methods inside the service java file we created earlier. So, we create a reference of the MyLocalBinder class (the inner class) and assign it to the iBinder object parameter of the onServiceConnected method. Then, we assign the value returned by the getService method of the inner class method to the object of the MyService class that we created earlier (myobject). And what does the getService method return? That’s right! A reference to the MyService class that gives access to all the methods inside it. We also set the isBound Boolean variable to true.

In the onServiceDisconnected method, the only thing we need to do is to set the isBound variable to false.

**private** ServiceConnection **myconn**=**new** ServiceConnection() {  
 @Override  
 **public void** onServiceConnected(ComponentName componentName, IBinder iBinder) {  
  
 MyLocalBinder binder=(MyLocalBinder)iBinder;  
 **myobject**= binder.getService();  
 **isBound**=**true**;  
  
 }  
  
 @Override  
 **public void** onServiceDisconnected(ComponentName componentName) {  
  
 **isBound**=**false**;  
  
 }

We now create a new intent object in the onCreate method (as we did earlier). In the intent class constructor, we again pass getApplicationContext() and the classname of the service (MyService.class in this case). We did all this earlier too…

But, here comes the twist. Instead of calling the startService method, we call the bindService method. This method accepts 3 parameters.

1. The intent object.
2. The object of the ServiceConnection class (that we created earlier)
3. Flags… for which we write Context.***BIND\_AUTO\_CREATE***

See more here:

<https://developer.android.com/reference/android/content/ContextWrapper.html#bindService-android.content.Intent-android.content.ServiceConnection-int->

Intent myIntent=**new** Intent(getApplicationContext(),MyService.**class**);  
bindService(myIntent,**myconn**, Context.***BIND\_AUTO\_CREATE***);

Now comes the easy part, we create our button and textview objects and inflate them using our findviewbyID method. We then set an onclicklistener for the button. We call the getCurrentTime method from inside the service using the myobject which now has access to all the methods inside the MyService class thanks to the inner class mumbo jumbo we did.

As we know, the getCurrentTime method returns a string which will be used to update the text on the textView using the setText method.

Button gobtn=findViewById(R.id.***timeButton***);

**final** TextView disptxt=findViewById(R.id.***currentTime***);

gobtn.setOnClickListener(  
 **new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
 String currtime=**myobject**.getCurrentTime();  
 disptxt.setText(currtime);  
 }  
 }  
  
);

***AUTOMATING THE SERVICE:***

Our app works! The service fetches time whenever the button is clicked. But, you know what would be more fun? If our app automatically fetched the system time after regular 1 second intervals.

Java offers [ScheduledExecutorService](http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledExecutorService.html) to schedule and run periodic tasks or tasks with delay.

Documentation for scheduledExecutorservice is available here:

<https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledExecutorService.html>

An [ExecutorService](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html" \o "interface in java.util.concurrent) that can schedule commands to run after a given delay, or to execute periodically.

The schedule methods create tasks with various delays and return a task object that can be used to cancel or check execution. The scheduleAtFixedRate and scheduleWithFixedDelay methods create and execute tasks that run periodically until cancelled.

All schedule methods accept relative delays and periods as arguments, not absolute times or dates.

We use the scheduleAtFixedRate method to periodically call the service:

|  |  |
| --- | --- |
| [ScheduledFuture](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledFuture.html)<?> | [**scheduleAtFixedRate**](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledExecutorService.html#scheduleAtFixedRate(java.lang.Runnable,%20long,%20long,%20java.util.concurrent.TimeUnit))([Runnable](https://docs.oracle.com/javase/6/docs/api/java/lang/Runnable.html) command, long initialDelay, long period, [TimeUnit](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent) unit)           Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 \* period, and so on. |

### scheduleAtFixedRate

[ScheduledFuture](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledFuture.html)<?> **scheduleAtFixedRate**([Runnable](https://docs.oracle.com/javase/6/docs/api/java/lang/Runnable.html" \o "interface in java.lang) command,

long initialDelay,

long period,

[TimeUnit](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/TimeUnit.html) unit)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 \* period, and so on. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.

**Parameters:**

command - the task to execute

initialDelay - the time to delay first execution

period - the period between successive executions

unit - the time unit of the initialDelay and period parameters

**Returns:**

a ScheduledFuture representing pending completion of the task, and whose get() method will throw an exception upon cancellation

**Throws:**

[RejectedExecutionException](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/RejectedExecutionException.html) - if the task cannot be scheduled for execution

[NullPointerException](https://docs.oracle.com/javase/6/docs/api/java/lang/NullPointerException.html) - if command is null

[IllegalArgumentException](https://docs.oracle.com/javase/6/docs/api/java/lang/IllegalArgumentException.html) - if period less than or equal to zero

**Usage Example**

Here is a class with a method that sets up a ScheduledExecutorService to beep every ten seconds for an hour:

import static java.util.concurrent.TimeUnit.\*;

class BeeperControl {

private final ScheduledExecutorService scheduler =

Executors.newScheduledThreadPool(1);

public void beepForAnHour() {

final Runnable beeper = new Runnable() {

public void run() { System.out.println("beep"); }

};

final ScheduledFuture<?> beeperHandle =

scheduler.scheduleAtFixedRate(beeper, 10, 10, SECONDS);

scheduler.schedule(new Runnable() {

public void run() { beeperHandle.cancel(true); }

}, 60 \* 60, SECONDS);

}

}

We simply add this to our java file:

ScheduledExecutorService scheduler= Executors.*newScheduledThreadPool*(1);

A handler to update the text on the UI thread.

**final** Handler myHandler=**new** Handler()  
{  
 @Override  
 **public void** handleMessage(@NonNull Message msg) {  
 String currtime=**myobject**.getCurrentTime();  
 disptxt.setText(currtime);  
 }  
};

A runnable to pass the action that the handler performs.

Runnable r=**new** Runnable() {  
 @Override  
 **public void** run() {  
 myHandler.sendEmptyMessage(0);  
 }  
};

A scheduledExecutorService to schedule the execution of the runnable every 1 second…

**final** ScheduledFuture timechange=scheduler.scheduleAtFixedRate(r,1,1, TimeUnit.***SECONDS***);

The final Java file looks like:

**package** com.example.time;  
  
**import** androidx.annotation.NonNull;  
**import** androidx.appcompat.app.AppCompatActivity;  
  
**import** android.content.ComponentName;  
**import** android.content.Context;  
**import** android.content.Intent;  
**import** android.content.ServiceConnection;  
**import** android.os.Handler;  
**import** android.os.IBinder;  
**import** android.os.Message;  
**import** android.widget.Button;  
**import** android.widget.TextView;  
**import** android.view.View;  
**import** com.example.time.MyService.MyLocalBinder;  
  
**import** android.os.Bundle;  
  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.ScheduledExecutorService;  
**import** java.util.concurrent.ScheduledFuture;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** MainActivity **extends** AppCompatActivity {  
  
 MyService **myobject**;  
 **boolean isBound**=**false**;  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
  
 ScheduledExecutorService scheduler= Executors.*newScheduledThreadPool*(1);  
  
 Intent myIntent=**new** Intent(getApplicationContext(),MyService.**class**);  
 bindService(myIntent,**myconn**, Context.***BIND\_AUTO\_CREATE***);  
  
 Button gobtn=findViewById(R.id.***timeButton***);  
 **final** TextView disptxt=findViewById(R.id.***currentTime***);  
  
 **final** Handler myHandler=**new** Handler()  
 {  
 @Override  
 **public void** handleMessage(@NonNull Message msg) {  
 String currtime=**myobject**.getCurrentTime();  
 disptxt.setText(currtime);  
 }  
 };  
  
 gobtn.setOnClickListener(  
 **new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
 String currtime=**myobject**.getCurrentTime();  
 disptxt.setText(currtime);  
 }  
 }  
  
 );  
  
 Runnable r=**new** Runnable() {  
 @Override  
 **public void** run() {  
 myHandler.sendEmptyMessage(0);  
 }  
 };  
  
 **final** ScheduledFuture timechange=scheduler.scheduleAtFixedRate(r,1,1, TimeUnit.***SECONDS***);  
  
 *//CODE TO CANCEL TIME CHANGE AFTER 1 HOUR  
  
 /\*  
 scheduler.schedule(new Runnable() {  
 @Override  
 public void run() {  
 timechange.cancel(true);  
 }  
 }, 60 \* 60, TimeUnit.SECONDS);  
\*/* }  
  
  
  
 **private** ServiceConnection **myconn**=**new** ServiceConnection() {  
 @Override  
 **public void** onServiceConnected(ComponentName componentName, IBinder iBinder) {  
  
 MyLocalBinder binder=(MyLocalBinder)iBinder;  
 **myobject**= binder.getService();  
 **isBound**=**true**;  
  
 }  
  
 @Override  
 **public void** onServiceDisconnected(ComponentName componentName) {  
  
 **isBound**=**false**;  
  
 }  
 };  
}