BLUETOOTH:

**BlueTooth:**

Which allows to wirelessly Exchange data b/w devices

**Features:**

(1) Setting up bluetooth

(2) Find devices that are either paired or available in local area.

(3) Connecting devices

(4) Transfer data b/w devices

**PErmission**

android.permission.BLUETOOTH (For bluetooth communicaiton)

android.permission.BLUETOOTH\_ADMIN (If ur app want to make any changes to bluetooth settings or discover devices)

android.permission.ACCESS\_COARSE\_LOCATION (bluetooth wants to discover location of user)

**BluetoothAdapter:**

BluetoothAdapeter is used to check whether device supports bluetooth or not, enable/disable bluetooth settings

(1)BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

if mbluetoothADapter is null, device does not support bluetooth

**to Enable blue tooth**

Intent enableBIntent = new Intent(BluetoothAdapter.ACTION\_REQUEST\_ENABLE)

startActivityresult(enableBItent,REQUEST\_ENABLE\_BT);

RESULT\_OK - onActivityResult()

RESULT\_CANCELLED –

**Query Paired Devices:**

**In order to query paired devices:**

**Set<BluetoothDevice> pairedDevices = BlueToothAdapter.getBondedDevices()**

Traversing this set , we can information about device name ,and device addresss

**Discover Devices:**

startDiscovery() called to discover devices. Runs asynchronously to discover devices

Need to register broadcast receiver to

IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION\_FOUND);

registerReceiver(mReceiver, filter);

**private final BroadcastReceiver mReceiver = new BroadcastReceiver() {**

**public void onReceive(Context context, Intent intent) {**

**String action = intent.getAction();**

**if (BluetoothDevice.ACTION\_FOUND.equals(action)) {**

**// Discovery has found a device. Get the BluetoothDevice**

**// object and its info from the Intent.**

**BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA\_DEVICE);**

**String deviceName = device.getName();**

**String deviceHardwareAddress = device.getAddress(); // MAC address**

**}**

**}**

**};**

**Room:**

**Room provides an abstraction layer over SQLite , for fluent access of sQLite and utilizing full functionality of SQLite , reducing boiler plate code.**

**There are 3 important components in room**

1. **DataBase**
2. **Entity**
3. **DAO**

**DataBase: IS an abstract class with annotation “Database” extends RoomDataBase.**

**The annotated database contains list of entities**

**IT contains an abstract method with no argument that returns DataAccessObject class annotated with DAO**

**Can be access instance of roomdatabase as Room.dataBaseBuilder()**

**Entity: Entity specifies a class annotated with entity that specifies table name and columns**

**DAO : Contains methods to access DB like insert,update,select,delete and is annotated with @dao**

**Advantages:**

1. **Compile time verification of SQLite queries**
2. **Less boiler plate code**
3. **Full integration with android components**

**Example:**

@Entity  
public class User {  
    @PrimaryKey  
    private int uid;  
  
    @ColumnInfo(name = "first\_name")  
    private String firstName;  
  
    @ColumnInfo(name = "last\_name")  
    private String lastName;  
  
    // Getters and setters are ignored for brevity,  
    // but they're required for Room to work.  
}

@Dao  
public interface UserDao {  
    @Query("SELECT \* FROM user")  
    List<User> getAll();  
  
    @Query("SELECT \* FROM user WHERE uid IN (:userIds)")  
    List<User> loadAllByIds(int[] userIds);  
  
    @Query("SELECT \* FROM user WHERE first\_name LIKE :first AND "  
           + "last\_name LIKE :last LIMIT 1")  
    User findByName(String first, String last);  
  
    @Insert  
    void insertAll(User... users);  
  
    @Delete  
    void delete(User user);  
}

@Database(entities = {User.class}, version = 1)  
public abstract class AppDatabase extends RoomDatabase {  
    public abstract UserDao userDao();  
}

AppDatabase db = Room.databaseBuilder(getApplicationContext(),  
        AppDatabase.class, "database-name").build();

**Memory Leaks:**

**Scenarios:**

**(1 ) UnRegisterListeners:**

**There are many situations where you register for a listener in Android/Fragment and forgot to unregister it. This leads to memory leaks**

**Once you register your listener, be sure to unregister when its job is done or unregister in onDestroy() of Activity.**

**Android Monitor tool -🡪 Memory Tab -> CLick Dump Java heap icon 🡪hprof file which Android Studio opens 🡪 Click Analyzer Tasks 🡪 check for leaked activities.**

**(2) Inner classes:**

**Inner classes are most widely in Android app development.**

**A familiar is the Async Task to perform back ground operation.**

**Public class performBG extends Aysctask<void,void,void> {**

**doInBackGround(){**

**//Where we perform Bg operation**

**}**

**onPostExecute(){**

**//Is where we update the UI**

**}**

**}**

**The Above class is an inner class that holds the reference if enclosing class Activity. When an orientation changes the activity is killed and re created , but this class still holds the destroyed activity reference which will lead to memory leak.**

**Solution:**

**The solution is to create a private static class as static class does not hold the reference of its enclosing class. But a private static class can’t access the non static variables from static class.So pass them as arguments to constructor and create a weak reference for them.**

**Context:**

**Context is most common thing used in Android apps. There can be many contexts like Application context,Activity context. Misuse of this may lead to memory leak. Its important to understand the situation in which we should use context. If we use an Activity context in wrong place ,it can hold activity reference and can lead to potential memory leak.**