1. Уровни архитектуры Android

1. Linux Kernel – ядро Linux, управляющее аппаратным обеспечением (память, процессы, сети).
2. Hardware Abstraction Layer (HAL) – интерфейс между драйверами оборудования и Android.
3. Native Libraries & Android Runtime – обеспечивает доступ к важным библиотекам (например, для графики, баз данных). Android Runtime (ART) отвечает за выполнение приложений.
4. Application Framework – инструменты для взаимодействия с функциональностью устройства.
5. Applications – пользовательские приложения (например, мессенджеры, игры).

Пример: простой код, взаимодействующий с уровнем *Application Framework* (например, получение данных о батарее через BatteryManager):

import android.content.Context;

import android.os.BatteryManager;

public class BatteryInfo {

public static int getBatteryPercentage(Context context) {

BatteryManager batteryManager = (BatteryManager) context.getSystemService(Context.BATTERY\_SERVICE);

return batteryManager != null ? batteryManager.getIntProperty(BatteryManager.BATTERY\_PROPERTY\_CAPACITY) : -1;

}

}

2. Архитектура приложения, основные компоненты

1. Activity – экран приложения.
2. Service – выполнение фоновых задач.
3. Broadcast Receiver – реагирует на системные или пользовательские события.
4. Content Provider – предоставляет доступ к данным между приложениями.

Пример кода:

// Activity

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

}

// Service

public class MyService extends Service {

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

// Фоновая задача

return START\_STICKY;

}

@Override

public IBinder onBind(Intent intent) {

return null;

}

}

// Broadcast Receiver

public class MyReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

// Реакция на событие

}

}

// Content Provider

public class MyContentProvider extends ContentProvider {

@Override

public boolean onCreate() {

return true;

}

@Override

public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {

return null;

}

// Другие методы (insert, delete, update)

}

3. Манифест приложения

Манифест – это файл AndroidManifest.xml, который описывает ключевые характеристики приложения.

Структура манифеста:

1. <manifest>: корневой элемент.
2. <application>: описание приложения.
3. <activity>, <service>, <receiver>, <provider>: компоненты приложения.
4. <uses-permission>: разрешения.
5. <uses-sdk>: минимальная и целевая версии SDK.

Пример манифеста:

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.myapp">

<!-- Разрешения -->

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE" />

<!-- Минимальная и целевая версия SDK -->

<uses-sdk

android:minSdkVersion="21"

android:targetSdkVersion="33" />

<application

android:allowBackup="true"

android:label="@string/app\_name"

android:theme="@style/Theme.MyApp">

<!-- Активность -->

<activity android:name=".MainActivity">

4. Ресурсы приложения

Ресурсы приложения – это внешние данные, используемые приложением (строки, изображения, стили, макеты и т.д.). Хранятся в папке res.

Типы ресурсов:

1. Строки (res/values/strings.xml)
2. Цвета (res/values/colors.xml)
3. Макеты (res/layout/)
4. Изображения (res/drawable/)
5. Стили (res/values/styles.xml)

Пример ресурсов: res/values/strings.xml

<resources>

<string name="app\_name">MyApp</string>

<string name="welcome\_message">Добро пожаловать!</string>

</resources>

res/values/colors.xml

xml

Копировать код

<resources>

<color name="primary\_color">#6200EE</color>

5. Активность (Activity). Жизненный цикл активности

Жизненный цикл активности:

1. onCreate() – создание активности.
2. onStart() – активность становится видимой.
3. onResume() – активность на переднем плане.
4. onPause() – активность уходит на задний план.
5. onStop() – активность скрыта.
6. onDestroy() – активность уничтожена.

Пример кода:

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Log.d("Lifecycle", "onCreate");

}

@Override

protected void onStart() {

super.onStart();

Log.d("Lifecycle", "onStart");

}

@Override

protected void onResume() {

super.onResume();

Log.d("Lifecycle", "onResume");

}

6. Службы (Сервисы). Жизненный цикл. Способы запуска. Методы.

Жизненный цикл службы:

1. onCreate() – создание службы.
2. onStartCommand() – запуск службы.
3. onBind() – связь с компонентом (только для привязанных служб).
4. onDestroy() – завершение службы.

Способы запуска:

1. Start Service – через метод startService().
2. Bind Service – через метод bindService().

Пример кода:

public class MyService extends Service {

@Override

public void onCreate() {

super.onCreate();

Log.d("Service", "onCreate");

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

Log.d("Service", "onStartCommand");

return START\_STICKY;

}

7. Android. Контент-провайдеры

Контент-провайдеры предоставляют доступ к данным между приложениями. Основные методы:

1. query() – получение данных.
2. insert() – добавление данных.
3. update() – обновление данных.
4. delete() – удаление данных.

Пример реализации контент-провайдера:

public class MyContentProvider extends ContentProvider {

private static final String AUTHORITY = "com.example.myprovider";

private static final Uri CONTENT\_URI = Uri.parse("content://" + AUTHORITY + "/data");

@Override

public boolean onCreate() {

return true;

}

@Override

public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {

// Логика получения данных

return null;

}

8. Приемники широковещательных сообщений (Broadcast Receivers)

Приемники (Broadcast Receivers) реагируют на широковещательные сообщения от системы или приложений.

Пример статического приемника:  
AndroidManifest.xml

<receiver android:name=".MyBroadcastReceiver">

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED" />

</intent-filter>

</receiver>

MyBroadcastReceiver.java

java

Копировать код

public class MyBroadcastReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

Log.d("BroadcastReceiver", "Сообщение получено: " + intent.getAction());

}

}

Пример динамического приемника:

BroadcastReceiver receiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

Log.d("BroadcastReceiver", "Динамическое сообщение: " + intent.getAction());

}

};

IntentFilter filter = new IntentFilter("com.example.MY\_ACTION");

registerReceiver(receiver, filter);

9. Фрагменты. Жизненный цикл фрагмента

Жизненный цикл фрагмента:

1. onAttach() – фрагмент прикрепляется к активности.
2. onCreate() – инициализация фрагмента.
3. onCreateView() – создание представления фрагмента.
4. onViewCreated() – завершение создания представления.
5. onStart() – фрагмент становится видимым.
6. onResume() – фрагмент на переднем плане.
7. onPause() – фрагмент уходит с переднего плана.
8. onStop() – фрагмент скрыт.
9. onDestroyView() – уничтожение представления.
10. onDestroy() – освобождение ресурсов.
11. onDetach() – фрагмент отсоединяется от активности.

Пример кода:

public class MyFragment extends Fragment {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

Log.d("FragmentLifecycle", "onCreate");

}

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

Log.d("FragmentLifecycle", "onCreateView");

return inflater.inflate(R.layout.fragment\_my, container, false);

}

10. Статическое и динамическое использование фрагментов

Статическое использование фрагментов:  
Фрагмент указывается в XML-разметке активности:

<FrameLayout

android:id="@+id/fragment\_container"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

<fragment

android:id="@+id/static\_fragment"

android:name="com.example.MyFragment"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" />

Динамическое использование фрагментов:  
Фрагменты добавляются или заменяются программно во время выполнения:

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction transaction = fragmentManager.beginTransaction();

MyFragment myFragment = new MyFragment();

transaction.add(R.id.fragment\_container, myFragment);

transaction.commit();

11. Списки и адаптеры. Разновидности стандартых адаптеров.  
Списки в Android чаще всего отображаются с использованием ListView или RecyclerView, а для их заполнения используются адаптеры.

Пример использования ArrayAdapter:

ListView listView = findViewById(R.id.list\_view);

String[] data = {"Элемент 1", "Элемент 2", "Элемент 3"};

ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple\_list\_item\_1, data);

listView.setAdapter(adapter);

Разновидности стандартных адаптеров:

1. ArrayAdapter – для работы с массивами данных.
2. SimpleAdapter – для отображения данных в формате List<Map<String, Object>>.
3. CursorAdapter – для работы с данными из базы данных (SQLite).

12. Кастомные адаптеры для списков.

Кастомные адаптеры используются, когда стандартные адаптеры не подходят для отображения сложных элементов списка с кастомной разметкой.

Пример создания кастомного адаптера:

public class CustomAdapter extends ArrayAdapter<Item> {

public CustomAdapter(Context context, List<Item> items) {

super(context, 0, items);

}

@Override

public View getView(int position, View convertView, ViewGroup parent) {

return convertView;

}

}

13. Пользовательские компоненты ( custom views ) и составные (compound\composite component).

public class CustomCircleView extends View {

public CustomCircleView(Context context) {

super(context);

}

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

Paint paint = new Paint();

paint.setColor(Color.RED);

canvas.drawCircle(getWidth() / 2, getHeight() / 2, 100, paint);

}

}

Cоздание пользовательского составного элемента на основе Layout класса:

public class CustomCompoundView extends LinearLayout {

public CustomCompoundView(Context context, AttributeSet attrs)

} }

<com.example.CustomCompoundView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

app:layout\_constraintTop\_toTopOf="parent"

app:layout\_constraintStart\_toStartOf="parent" />

15. Многопоточность. Использование Thread + Handler

Многопоточность в Android часто применяется для выполнения задач, которые могут заблокировать основной поток (UI). Комбинация Thread и Handler позволяет выполнять фоновые операции и возвращать результаты в основной поток.

Пример использования Thread + Handler:

1. Создание Handler в главном потоке:

Handler handler = new Handler(Looper.getMainLooper()) {

@Override

public void handleMessage(@NonNull Message msg) {

// Обработка результата на UI

String result = (String) msg.obj;

TextView textView = findViewById(R.id.textView);

textView.setText(result);

}

};

1. Запуск фонового потока:

new Thread(() -> {

// Выполнение фоновой задачи

String result = performBackgroundTask();

// Отправка результата обратно через Handler

Message message = handler.obtainMessage();

message.obj = result;

handler.sendMessage(message);

}).start();

16. Многопоточность. Использование AsyncTask

AsyncTask – устаревший инструмент для работы с многопоточностью в Android (с Android 11 использовать его не рекомендуется). Однако, он позволяет выполнять фоновые операции и обновлять интерфейс без явного управления потоками.

### Пример использования AsyncTask:

1. **Класс** MyAsyncTask**:**

private class MyAsyncTask extends AsyncTask<Void, Integer, String> {

protected String doInBackground // Фоновая задача

protected void onProgressUpdate// Обновление UI во время выполнения задачи

protected void onPostExecute // Выполняется после завершения фоновой задач

new MyAsyncTask().execute(); // Запуск AsyncTask

* onPreExecute() – выполняется в основном потоке перед началом работы.
* doInBackground() – выполняется в фоновом потоке.
* onProgressUpdate() – обновляет интерфейс (вызов publishProgress()).
* onPostExecute() – обновляет интерфейс по завершении работы.

17. Хранение данных. Preferences.

**Preferences** – это способ хранения простых данных (например, настроек пользователя) в виде ключ-значение. В Android для этого используется SharedPreferences.

### Пример работы с SharedPreferences:

1. **Сохранение данных:**

SharedPreferences sharedPreferences = getSharedPreferences("MyPrefs", MODE\_PRIVATE);

SharedPreferences.Editor editor = sharedPreferences.edit();

editor.putString("username", "JohnDoe");

editor.putInt("age", 25);

editor.apply(); // Сохраняет изменения асинхронно

### Области использования:

* **MODE\_PRIVATE**: Данные доступны только текущему приложению.
* **MODE\_MULTI\_PROCESS** (устарело): Для доступа из нескольких процессов.
* **apply()** – сохраняет данные асинхронно (рекомендуется).
* **commit()** – сохраняет данные синхронно (используется реже).

SharedPreferences идеально подходят для хранения настроек, состояния приложения или другой простой информации.

18. Хранение данных. Файлы.

**Хранение данных в файлах** в Android позволяет сохранять данные в виде текстовых или бинарных файлов. Файлы могут быть сохранены во внутреннем или внешнем хранилище.

### **Запись данных в файл (внутреннее хранилище):**

String filename = "example.txt";

String fileContents = "Привет, мир!";

try (FileOutputStream fos = openFileOutput(filename, MODE\_PRIVATE)) {

fos.write(fileContents.getBytes());

} catch (IOException e) {

e.printStackTrace();

}

boolean deleted = deleteFile("example.txt"); if (deleted) { Log.d("File", "Файл успешно удалён."); }

### **Расположение файлов:**

* **Внутреннее хранилище:**  
  /data/data/<package\_name>/files
* **Внешнее хранилище (папка приложения):**  
  /storage/emulated/0/Android/data/<package\_name>/files

19. Хранение данных. SQLite.

SQLite — это встроенная реляционная база данных, часто используемая для хранения данных в мобильных и десктопных приложениях.

public class DBHelper extends SQLiteOpenHelper {

public DBHelper(Context context) {

super(context, "myDatabase", null, 1);

}

@Override

public void onCreate(SQLiteDatabase db) {

db.execSQL("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)");

}

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

db.execSQL("DROP TABLE IF EXISTS users");

onCreate(db);

}

}

// Вставка данных

DBHelper dbHelper = new DBHelper(context);

SQLiteDatabase db = dbHelper.getWritableDatabase();

ContentValues values = new ContentValues();

values.put("name", "John");

db.insert("users", null, values);

db.close();

20. Классы SQLiteDatabase и SQLiteOpenHelper

SQLiteDatabase — интерфейс для работы с базой данных SQLite (выполнение запросов: INSERT, SELECT, UPDATE, DELETE).

SQLiteOpenHelper — вспомогательный класс для управления созданием и обновлением базы данных.

import android.content.Context;

import android.database.sqlite.SQLiteDatabase;

import android.database.sqlite.SQLiteOpenHelper;

public class MyDatabaseHelper extends SQLiteOpenHelper

// Использование

MyDatabaseHelper dbHelper = new MyDatabaseHelper(context);

SQLiteDatabase db = dbHelper.getWritableDatabase();

db.execSQL("INSERT INTO users (name, age) VALUES ('Alice', 30);");

21. Базы данных. Использование класса Cursor

Cursor — класс для чтения данных из базы SQLite построчно.

Пример:

Cursor cursor = db.rawQuery("SELECT \* FROM users", null);

while (cursor.moveToNext()) {

int id = cursor.getInt(cursor.getColumnIndex("id"));

String name = cursor.getString(cursor.getColumnIndex("name"));

int age = cursor.getInt(cursor.getColumnIndex("age"));

System.out.println(id + ": " + name + ", " + age);

}

cursor.close();

22. ORM ( Object-relational mapping). Принцип работы. Преимущества и недостатки

ORM (Object-Relational Mapping)

ORM — это технология, которая позволяет работать с базой данных через объектно-ориентированный подход, вместо написания SQL-запросов.

Принцип работы:

Таблицы базы данных отображаются в виде классов.

Строки таблиц становятся объектами этих классов.

SQL-запросы автоматически формируются ORM-библиотекой, основываясь на операциях с объектами.

Преимущества:

Удобство: нет необходимости писать SQL-код вручную.

Поддержка кросс-платформенности: легко менять базы данных (например, SQLite на PostgreSQL).

Поддержание читаемости кода: работа с объектами вместо строк SQL.

Недостатки:

Снижение производительности: сложные запросы могут быть медленнее.

Ограниченность функциональности: не все возможности SQL доступны через ORM.

Порог вхождения: изучение самой ORM может занять время.

23. Описание и инициализация базы данных ORMLite или Room

Добавление зависимостей

Создание сущности (Entity)

Определение DAO (Data Access Object)

Создание базы данных (Database)

Инициализация базы данных

import android.content.Context;

import androidx.room.Room;

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

UserDao userDao = db.userDao();

24. Описание и инициализация базы данных в GreenDAO

Добавление зависимостей

Создание сущности (Entity)

Определение DAO (Data Access Object)

Создание базы данных (Database)

Инициализация базы данных

import org.greenrobot.greendao.database.Database;

DaoMaster.DevOpenHelper helper = new DaoMaster.DevOpenHelper(context, "users-db");

Database db = helper.getWritableDb();

DaoSession daoSession = new DaoMaster(db).newSession();

UserDao userDao = daoSession.getUserDao();

// Вставка данных

User user = new User();

user.setName("Alice");

user.setAge(30);

userDao.insert(user);

// Получение данных

List<User> users = userDao.loadAll();

25. Воспроизведение звука. MediaPlayer. Диаграмма состояний.

MediaPlayer проходит через несколько состояний в процессе воспроизведения звука или видео. Вот основные состояния:

1. Idle: Инициализация объекта.
2. Initialized: После вызова setDataSource().
3. Prepared: После вызова prepare() или prepareAsync().
4. Started: Воспроизведение началось после вызова start().
5. Paused: Воспроизведение приостановлено с помощью pause().
6. Stopped: Воспроизведение остановлено вызовом stop().
7. End: MediaPlayer больше нельзя использовать.

### **Пример работы с MediaPlayer:**

MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sample\_audio);

// Запуск воспроизведения

mediaPlayer.start();

// Приостановка

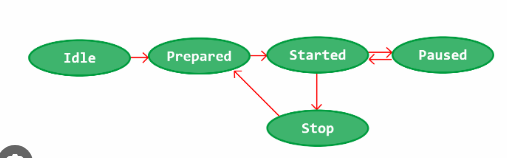
mediaPlayer.pause();

// Остановка

mediaPlayer.stop();

// Освобождение ресурсов

mediaPlayer.release();



26. Воспроизведение звука. SoundPool.

SoundPool — это класс в Android, предназначенный для управления воспроизведением коротких аудиоэффектов. Он позволяет эффективно загружать и воспроизводить аудиофайлы, что особенно полезно для игр или приложений, где нужно воспроизводить звуки с низкой задержкой.

// Создание объекта SoundPool

soundPool = SoundPool.Builder().setMaxStreams(1).build()

// Загрузка звукового файла

soundId = soundPool.load(this, R.raw.sound\_effect, 1)

// Воспроизведение звука

soundPool.play(soundId, 1f, 1f, 0, 0, 1f)

27. Запись звука с помощью AudioRecorder и MediaRecorder

MediaRecorder используется для записи аудио и видео в Android-приложениях. В приведенном примере создается объект MediaRecorder, который настраивается для записи аудио с микрофона и сохраняет файл в формате 3GP с использованием кодека AMR-NB.

// Создание объекта MediaRecorder

val mediaRecorder = MediaRecorder()

// Настройка источника и формата записи

mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC)

mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE\_GPP)

mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR\_NB)

mediaRecorder.setOutputFile(filePath)

// Начало записи

mediaRecorder.prepare()

mediaRecorder.start()

// Остановка записи

mediaRecorder.stop()

mediaRecorder.release()

28. Android NDK. Подготовка интерфейса на стороне Java для работы с нативным С кодом

Создание нативного метода в Java:

В Java создается метод с ключевым словом external, который будет вызываться из C или C++ кода.

Этот метод будет использоваться для связи с нативной библиотекой.

Загрузка нативной библиотеки:

В Java с помощью System.loadLibrary() загружается библиотека, содержащая нативный код, написанный на C или C++.

Реализация нативного метода на C/C++:

В C/C++ файле реализуется метод, который будет выполняться при вызове из Java.

Для связи между Java и C используется JNI (Java Native Interface).

public class MainActivity extends AppCompatActivity {

// Загрузка нативной библиотеки

static {

System.loadLibrary("native-lib");

}

// Нативный метод, который будет реализован на стороне C

public native String stringFromJNI();

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// Вызов нативного метода

TextView textView = findViewById(R.id.sample\_text);

textView.setText(stringFromJNI());

}

}

cmake\_minimum\_required(VERSION 3.4.1)

# Укажите путь к исходным файлам

add\_library(native-lib

SHARED

src/main/cpp/native-lib.cpp)

find\_library(log-lib

log)

target\_link\_libraries(native-lib

${log-lib})

27. Android NDK.Передача и конвертация параметров. JNI ( java native interface)

// В Java передаем строку и число в нативный метод

String input = "Hello from Java";

int number = 42;

String result = stringFromJNI(input, number); // Вызов нативного метода с параметрами

extern "C" JNIEXPORT jstring JNICALL

Java\_com\_example\_myapplication\_MainActivity\_stringFromJNI(JNIEnv\* env, jobject /\* this \*/, jstring inputString, jint inputInt) {

// Конвертация jstring в std::string

const char\* str = env->GetStringUTFChars(inputString, 0);

std::string strInput(str);

// Конвертация jint в std::string

std::string result = strInput + " and the number is " + std::to\_string(inputInt);

// Освобождение памяти

env->ReleaseStringUTFChars(inputString, str);

// Возврат результата как jstring

return env->NewStringUTF(result.c\_str());

}

30. Android NDK. Android.mk Application.mk – файлы проекта

В проекте Android с использованием NDK, файлы Android.mk и Application.mk используются для конфигурации сборки нативного кода. Эти файлы описывают, как компилировать и связывать нативные библиотеки с приложением.

### **1. Android.mk:**

Файл Android.mk используется для указания параметров сборки для нативных библиотек. Он описывает, какие исходные файлы компилировать и как собрать библиотеку.

Пример Android.mk:

makefile

Копировать код

# Указываем имя библиотеки (будет создана как libnative-lib.so)

LOCAL\_PATH := $(call my-dir)

include $(CLEAR\_VARS)

# Имя нативной библиотеки

LOCAL\_MODULE := native-lib

# Источник исходного кода (файл .cpp)

LOCAL\_SRC\_FILES := native-lib.cpp

# Указываем, какие библиотеки необходимо использовать (например, для логирования)

LOCAL\_LDLIBS := -llog

include $(BUILD\_SHARED\_LIBRARY)

### Объяснение:

* LOCAL\_PATH указывает на текущую директорию, где находится Android.mk.
* LOCAL\_MODULE указывает имя библиотеки, которая будет собрана (в данном случае native-lib).
* LOCAL\_SRC\_FILES перечисляет исходные файлы (например, native-lib.cpp), которые будут скомпилированы.
* LOCAL\_LDLIBS добавляет библиотеки, которые требуются для компиляции (например, -llog для логирования).
* include $(BUILD\_SHARED\_LIBRARY) указывает, что необходимо собрать динамическую библиотеку (shared library).

### **2. Application.mk:**

Файл Application.mk используется для конфигурации глобальных параметров сборки проекта. В нем можно указать, какие архитектуры поддерживаются, какие флаги компилятора использовать, и другие глобальные настройки.

Пример Application.mk:

makefile

Копировать код

# Указываем архитектуры, для которых собирается приложение

APP\_ABI := armeabi-v7a arm64-v8a x86

# Указываем минимальный уровень SDK

APP\_PLATFORM := android-21

# Определение флагов компиляции

APP\_CPPFLAGS := -std=c++11

### Объяснение:

* APP\_ABI указывает, для каких архитектур процессоров нужно собрать приложение (например, armeabi-v7a, arm64-v8a, x86).
* APP\_PLATFORM указывает минимальный уровень API, с которым будет совместимо приложение.
* APP\_CPPFLAGS позволяет указать флаги компилятора, в данном случае для использования стандарта C++11.

### Как это работает:

1. Android.mk: Этот файл используется для указания всех исходных файлов, зависимостей и параметров сборки для нативной библиотеки. Он собирает код C/C++ в динамическую библиотеку.
2. Application.mk: Этот файл настраивает глобальные параметры проекта, такие как целевые архитектуры и минимальный уровень API, на которых будет работать приложение.

31. Android NDK. Компиляция С-кода и сборка Android проекта

### 1. **Добавление исходных файлов C/C++ в проект**

Создайте директорию jni в корневой папке вашего проекта (рядом с src и res), где будут находиться исходные файлы C/C++

### **2. Создание файла** Android.mk

Файл Android.mk используется для описания исходных файлов и параметров сборки для нативной библиотеки.

### 3. **Создание файла** Application.mk

Файл Application.mk задает глобальные параметры, такие как архитектуры и минимальный уровень SDK.

### 4. **Запуск сборки с использованием NDK**

1. Откройте терминал и перейдите в корневую директорию проекта.
2. Введите команду для компиляции с использованием ndk-build

В вашем проекте Android Studio, в файле build.gradle, укажите использование NDK, чтобы приложение могло использовать нативную библиотеку:

android {

defaultConfig {

externalNativeBuild {

ndkBuild {

path 'jni/Android.mk' // Путь к файлу Android.mk

}

}

}

externalNativeBuild {

ndkBuild {

path 'jni/Android.mk'

}

}

32. Датчики и сенсоры современных мобильных устройств [76]

// Инициализация датчика освещенности

SensorManager sensorManager = (SensorManager) getSystemService(SENSOR\_SERVICE);

Sensor lightSensor = sensorManager.getDefaultSensor(Sensor.TYPE\_LIGHT);

// Регистрация слушателя для датчика освещенности

sensorManager.registerListener(sensorEventListener, lightSensor, SensorManager.SENSOR\_DELAY\_UI);

// Обработчик событий сенсоров

SensorEventListener sensorEventListener = new SensorEventListener() {

@Override

public void onSensorChanged(SensorEvent event) {

if (event.sensor.getType() == Sensor.TYPE\_LIGHT) {

// Обработка данных датчика освещенности

float lightLevel = event.values[0];

// Использование данных о уровне освещенности

}

}

};

### 33. Классы Sensor & SensorManager **SensorManager**

SensorManager — это системный сервис, который управляет сенсорами устройства. Он позволяет получать доступ к доступным датчикам, а также регистрировать слушателей событий для этих сенсоров.

### Основные методы:

* **getSystemService(SENSOR\_SERVICE)** — получает экземпляр SensorManager.
* **getDefaultSensor(int sensorType)** — возвращает объект Sensor для заданного типа сенсора.
* **registerListener(SensorEventListener listener, Sensor sensor, int samplingPeriodUs)** — регистрирует слушателя для определенного сенсора.
* **unregisterListener(SensorEventListener listener)** — отменяет регистрацию слушателя.

### **Sensor**

Sensor — это объект, представляющий один сенсор устройства, такой как акселерометр, гироскоп, датчик освещенности и т.д.

### Основные методы:

* **getType()** — возвращает тип сенсора (например, Sensor.TYPE\_ACCELEROMETER).
* **getName()** — возвращает имя сенсора.
* **getVendor()** — возвращает производителя сенсора.

// Получаем экземпляр SensorManager

SensorManager sensorManager = (SensorManager) getSystemService(SENSOR\_SERVICE);

// Получаем доступ к акселерометру

Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE\_ACCELEROMETER);

// Регистрация слушателя

sensorManager.registerListener(sensorEventListener, accelerometer, SensorManager.SENSOR\_DELAY\_UI);

// Обработчик событий сенсора

SensorEventListener sensorEventListener = new SensorEventListener() {

@Override

public void onSensorChanged(SensorEvent event) {

if (event.sensor.getType() == Sensor.TYPE\_ACCELEROMETER) {

// Получаем данные о движении устройства

float x = event.values[0];

}

}

};

34. Сенсоры. Обработка событий

В Android для обработки событий сенсоров используется интерфейс **SensorEventListener**, который позволяет отслеживать изменения значений сенсоров в реальном времени.

### Основные шаги для обработки событий сенсоров:

1. **Получение доступа к сенсору** через SensorManager.
2. **Регистрация слушателя** событий с помощью метода registerListener.
3. **Обработка событий** в методе onSensorChanged интерфейса SensorEventListener.
4. **Отмена регистрации слушателя** с помощью unregisterListener.

// Инициализация SensorManager

SensorManager sensorManager = (SensorManager) getSystemService(SENSOR\_SERVICE);

// Получение сенсора (например, акселерометра)

Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE\_ACCELEROMETER);

// Создание слушателя для обработки событий сенсора

SensorEventListener sensorEventListener = new SensorEventListener() {

@Override

public void onSensorChanged(SensorEvent event) {

// Проверяем, что событие пришло от акселерометра

if (event.sensor.getType() == Sensor.TYPE\_ACCELEROMETER) {

// Обработка данных акселерометра

float x = event.values[0];

}

}

};

// Регистрация слушателя

sensorManager.registerListener(sensorEventListener, accelerometer, SensorManager.SENSOR\_DELAY\_UI);

// Отмена регистрации слушателя (например, в методе onPause)

sensorManager.unregisterListener(sensorEventListener);