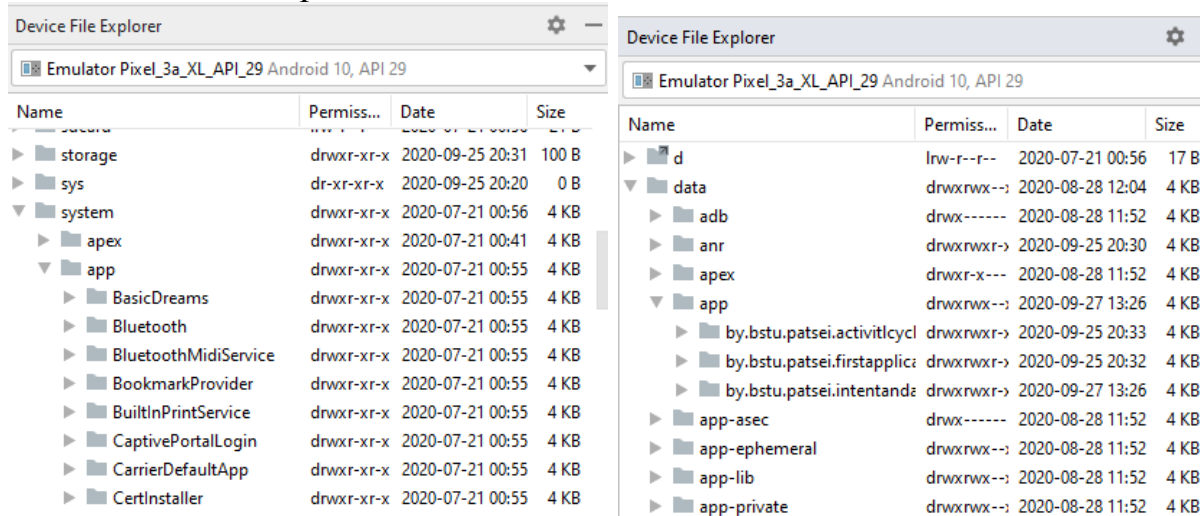


6. РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ

Ядро Linux, используемое в Android, в той или иной степени поддерживает большое количество самых разных файловых систем от используемых в Windows FAT и NTFS, до сетевой 9pfs из Plan 9. Есть и много «родных» для Linux файловых систем – например, Btrfs и семейство ext. Стандартом де-факто для Linux уже долгое время является **ext4**, используемая по умолчанию большинством популярных дистрибутивов Linux. В некоторых сборках также используется **F2FS** (Flash-Friendly File System), оптимизированная специально для flash-памяти. **Samsung RFS** – файловая система разработанная корейской компанией Samsung для устройств на базе ОС Linux.

Приложения, их APK, файлы odex (скомпилированный ahead-of-time Java-код) и ELF-библиотеки устанавливаются в `/system/app` (для приложений, поставляемых с системой) или в `/data/app` (для установленных пользователем приложений). Каждому предустановленному приложению при создании сборки Android выделяется папка с именем вида `/system/app/Terminal`, а для устанавливаемых пользователем приложений при установке создаются папки, имена которых начинаются с их имени пакета.



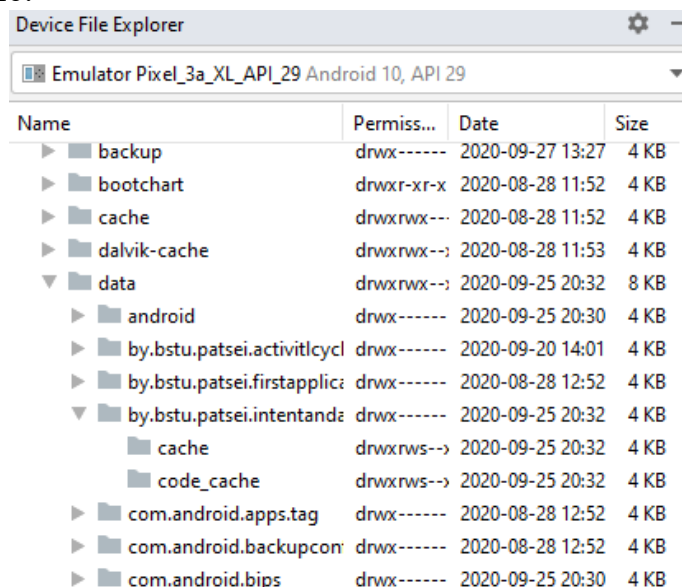
Name	Permiss...	Date	Size
storage	drwxr-xr-x	2020-09-25 20:31	100 B
sys	dr-xr-xr-x	2020-09-25 20:20	0 B
system	drwxr-xr-x	2020-07-21 00:56	4 KB
apex	drwxr-xr-x	2020-07-21 00:41	4 KB
app	drwxr-xr-x	2020-07-21 00:55	4 KB
BasicDreams	drwxr-xr-x	2020-07-21 00:55	4 KB
Bluetooth	drwxr-xr-x	2020-07-21 00:55	4 KB
BluetoothMidiService	drwxr-xr-x	2020-07-21 00:55	4 KB
BookmarkProvider	drwxr-xr-x	2020-07-21 00:55	4 KB
BuiltInPrintService	drwxr-xr-x	2020-07-21 00:55	4 KB
CaptivePortalLogin	drwxr-xr-x	2020-07-21 00:55	4 KB
CarrierDefaultApp	drwxr-xr-x	2020-07-21 00:55	4 KB
CertInstaller	drwxr-xr-x	2020-07-21 00:55	4 KB

Name	Permiss...	Date	Size
d	lrw-r--r--	2020-07-21 00:56	17 B
data	drwxrwx--	2020-08-28 12:04	4 KB
adb	drwx-----	2020-08-28 11:52	4 KB
anr	drwxrwxr-x	2020-09-25 20:30	4 KB
apex	drwxr-x---	2020-08-28 11:52	4 KB
app	drwxrwx--	2020-09-27 13:26	4 KB
by.bstu.patsei.activitlycycl	drwxrwxr-x	2020-09-25 20:33	4 KB
by.bstu.patsei.firstapplic	drwxrwxr-x	2020-09-25 20:32	4 KB
by.bstu.patsei.intentand	drwxrwxr-x	2020-09-27 13:26	4 KB
app-asec	drwx-----	2020-08-28 11:52	4 KB
app-ephemeral	drwxrwx--	2020-08-28 11:52	4 KB
app-lib	drwxrwx--	2020-08-28 11:52	4 KB
app-private	drwxrwx--	2020-08-28 11:52	4 KB

Содержимое `/system` описывает систему и содержит большинство составляющих её файлов. `/system` располагается на отдельном разделе flash-памяти, который по умолчанию монтируется в режиме read-only; обычно данные на нём изменяются только при обновлении системы. `/data` также располагается на отдельном разделе и описывает изменяемое состояние конкретного устройства, в том числе пользовательские настройки, установленные приложения и их данные, кэши и т. п. Очистка всех пользовательских данных заключается просто в очистке содержимого раздела `data`; нетронутая система остаётся установлена в разделе `system`.

Для хранения изменяемых данных каждому приложению выделяется папка в `/data/data`. Доступ к этой папке есть только у самого приложения – то есть только у UID, под которым запускается это приложение (если приложение использует несколько UID, или несколько приложений используют общий UID). В этой папке приложения сохраняют настройки,

кэш (в подпапках *shared_prefs* и *cache* соответственно) и любые другие нужные им данные:



Name	Permiss...	Date	Size
▶ backup	drwx-----	2020-09-27 13:27	4 KB
▶ bootchart	drwxr-xr-x	2020-08-28 11:52	4 KB
▶ cache	drwxrwx---	2020-08-28 11:52	4 KB
▶ dalvik-cache	drwxrwx---	2020-08-28 11:53	4 KB
▼ data	drwxrwx---	2020-09-25 20:32	8 KB
▶ android	drwx-----	2020-09-25 20:30	4 KB
▶ by.bstu.patsei.activitl cycl	drwx-----	2020-09-20 14:01	4 KB
▶ by.bstu.patsei.firstapplici	drwx-----	2020-08-28 12:52	4 KB
▼ by.bstu.patsei.intentand	drwx-----	2020-09-25 20:32	4 KB
cache	drwxrws-->	2020-09-25 20:32	4 KB
code_cache	drwxrws-->	2020-09-25 20:32	4 KB
▶ com.android.apps.tag	drwx-----	2020-08-28 12:52	4 KB
▶ com.android.backupcon	drwx-----	2020-08-28 12:52	4 KB
▶ com.android.bips	drwx-----	2020-09-25 20:30	4 KB

Система знает о существовании папки *cache* и может очищать её самостоятельно при нехватке места. При удалении приложения вся папка этого приложения полностью удаляется, и приложение не оставляет за собой следов. Альтернативно, и то, и другое пользователь может явно сделать в настройках.

Итак

/data - хранит изменяемые данные,

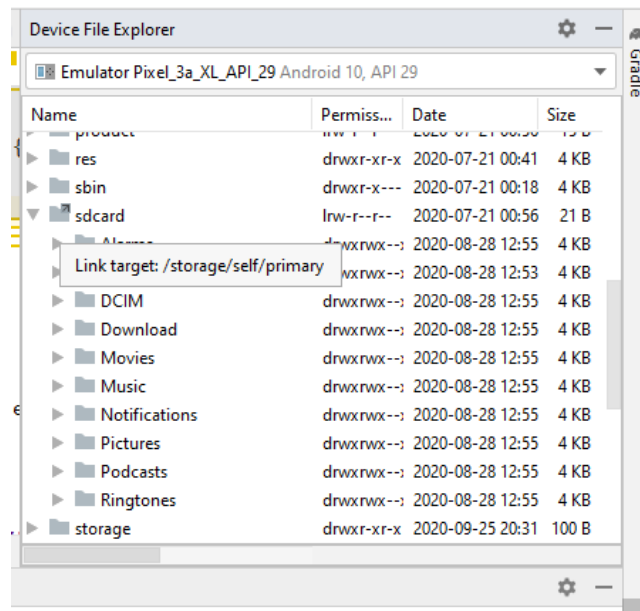
/system - хранит систему,

/vendor - предназначен для файлов, специфичных для этой сборки Android, а не входящих в «стандартный» Android,

/dev - хранит файлы устройств и другие специальные файлы.

Все устройства Android имеют две области хранения файлов: *внутреннюю память* и *внешние хранилища*. Эти области появились в первые годы существования Android, когда на большинстве устройств имелись встроенная память (внутреннее хранилище) и карты памяти, например micro SD, внешнее хранилище. Некоторые устройства делят встроенную память на внутренний и внешний разделы, так что даже без съемных носителей в системе имеется две области хранения файлов, и API-интерфейс работает одинаково вне зависимости от типа внешнего хранилища.

Внешнее хранилище играет роль домашней папки пользователя — именно там располагаются такие папки, как *Documents*, *Download*, *Music* и *Pictures*.



В отличие от внутреннего хранилища, разделённого на папки отдельных приложений, внешнее хранилище представляет собой «общую зону». К нему есть полный доступ у любого приложения, получившего соответствующее разрешение от пользователя.

Исходно предполагалось, что внешнее хранилище действительно будет располагаться на внешней SD-карте, поскольку в то время объём SD-карт значительно превышал объём встраиваемой в телефоны памяти. С тех пор условия изменились; в современных телефонах часто нет слота для SD-карты, зато устанавливается огромное количество встроенной памяти.

Поэтому в современном Android практически всегда и внутреннее, и внешнее хранилища располагаются во **встроенной памяти**. Настоящий путь, по которому располагается внешнее хранилище в файловой системе, имеет форму `/data/media/0` (для каждого пользователя устройства создаётся отдельное внешнее хранилище). В целях совместимости до внешнего хранилища также можно добраться по путям `/sdcard`, `/mnt/sdcard`, `/storage/self/primary`, `/storage/emulated/0`, несколькими путями, начинающимся с `/mnt/runtime/` и другим.

С другой стороны, у многих устройств все-таки есть слот для SD-карты. Вставленную в Android-устройство SD-карту можно использовать как обычный внешний диск. Кроме того, Android позволяет «заимствовать» SD-карту и разместить внутреннее и внешнее хранилище на ней (это называется заимствованным хранилищем – *adopted storage*). При этом система переформатирует SD-карту и шифрует содержимое – хранящиеся на ней данные невозможно прочесть, подключив ее к другому устройству.

6.1. Internal Storage (внутренняя память)

Внутренняя память имеет следующие характеристики:

- энергонезависимая;
- x

- сохраненные данные в памяти позволяют читать и записывать файлы;
- файлы могут быть доступны только данному приложению, а не другим приложениям. Приложение *всегда имеет разрешение на чтение и запись файлов в свой внутренний каталог* на internal storage. Не требует разрешений для обращения;
- файлы хранятся до тех пор, пока приложение остается на устройстве. Когда оно удаляется, все связанные файлы автоматически удалятся (рис. 6.1);
- файлы хранятся в разделе *data/data/* за которым следует имя пакета приложения;
- пользователь может явно разрешить другим приложениям доступ к файлам;
- чтобы сделать данные конфиденциальными, можно использовать режим *MODE_PRIVATE*;
- данные хранятся в файле в битовом формате, поэтому требуется преобразовать их перед добавлением в файл или при извлечении из него;
- для непосредственного чтения и записи файлов применяются стандартные классы Java из пакета *java.io*.

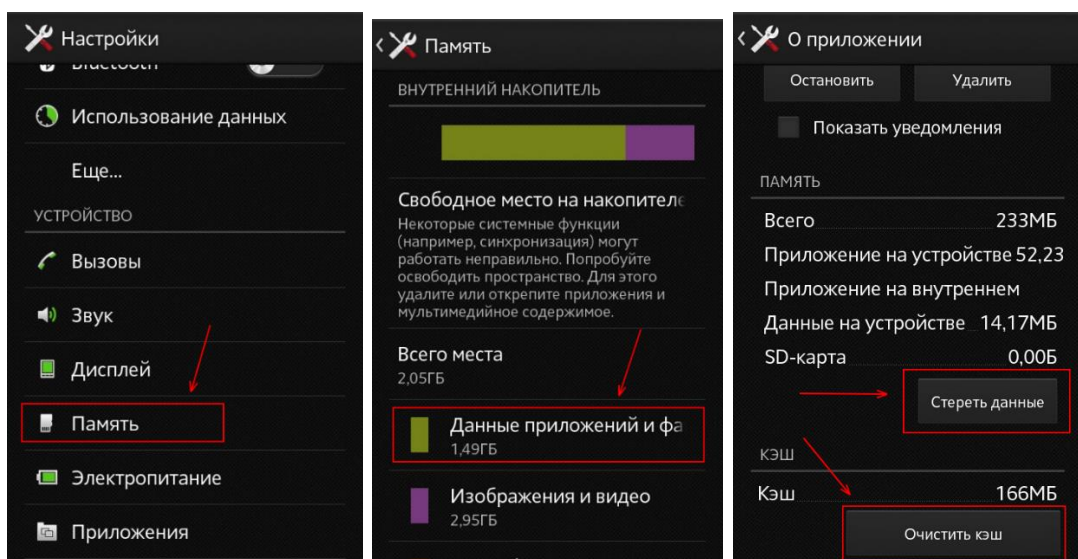


Рис. 6.1. Доступ к внутренней памяти устройства

Внутренняя память лучше всего подходит для ситуаций, когда надо быть уверенным, что ни пользователь, ни другие приложения не смогут получить доступ к файлам. Есть следующие режимы:

- *MODE_PRIVATE*: в этом режиме данные, сохраненные ранее, всегда переопределяются текущими данными, т. е. каждый раз, когда добавляется новая запись в файл, предыдущий контент удаляется или переопределяется (режим по умолчанию);
- *MODE_APPEND*: в этом режиме данные добавляются к существующему контенту.

Запись данных в файл во внутреннем хранилище может быть выполнена следующим образом:

```
String File_Name= "Demo.txt"; // Имя
String Data="Hello!!"; // Данные

FileOutputStream fileobj = null;
try {
    fileobj = openFileOutput(File_Name, Context.MODE_PRIVATE);

    byte[] ByteArray = Data.getBytes(); //Конвертируем в bytes stream
    fileobj.write(ByteArray); //Записываем в файл
    fileobj.close(); // Закрываем файл
}
catch (FileNotFoundException e ) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

Для того чтобы проверить записанные данные, справа в Android Studio есть вкладка *Device File Explorer*. В открывшейся панели можно найти записанный файл (рис. 6.2). Для этого следует перейти в *data/data/«имена пакетов»/files/Demo.txt*. Также есть возможность открыть и посмотреть содержимое файла.

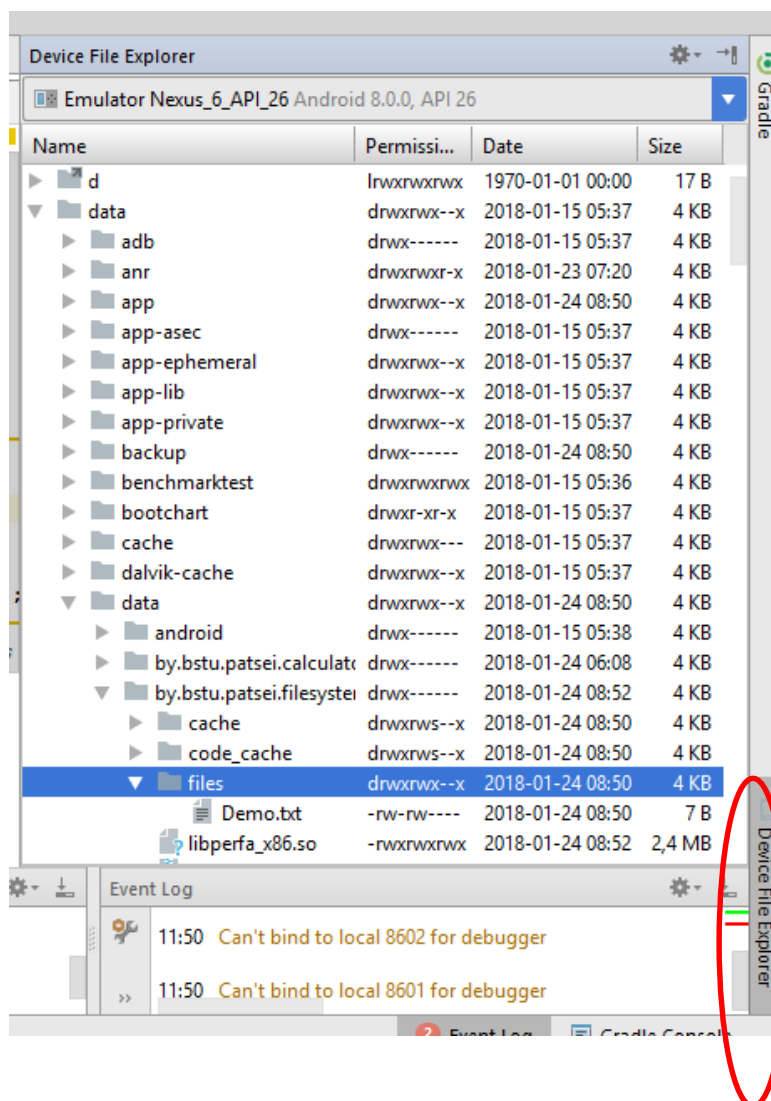


Рис.6.2 Панель просмотра файлов устройства

Ниже рассмотрены основные операции с файлами.

Функция создания файла:

```
public static void createFile(String filesDir,String fileName){
    try {
        File file = new File(filesDir, fileName);
        file.createNewFile();
        Log.d("Log_2", "Файл " + fileName + " создан");
    } catch (IOException e){
        Log.d("Log_2", "Файл " + fileName + " не создан");
    }
}
```

Функция записи в файл:

```
public static void writeToFile(String filesDir, String fileName,String
string){
    if(!isExistFile(filesDir,fileName)){
        createFile(filesDir,fileName);
    }
    File file = new File(filesDir,fileName);
    BufferedWriter bw;
    try{
        FileWriter fw = new FileWriter(file,true);
        bw = new BufferedWriter(fw);
        bw.write(string);
        bw.close();
        Log.d("Log_2", "Файл " + fileName + " записан");
    }
    catch (IOException e){
        Log.d("Log_2", "Файл " + fileName + " не открыт " +e.getMessage()
);
    }
}
```

Функция чтения файла:

```
public static String readFromFile(Context context,String fileName){

    String line="";
    try {
        FileInputStream fis = context.openFileInput(fileName);
        InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
        BufferedReader bufferedReader = new BufferedReader(isr);
        StringBuilder sb = new StringBuilder();
        while ((line = bufferedReader.readLine()) != null) {
            sb.append(line).append("\n");
        }
        line = sb.toString();
    } catch (FileNotFoundException e) {
    } catch (UnsupportedEncodingException e) {
    } catch (IOException e) {
    }
    return line;
}
```


Функция записи в файл произвольного доступа:

```
public static void writeToRandomAccessFile(String filesDir, String
fileName,String string, long point){

    if(!isExistFile(filesDir,fileName)){
        createFile(filesDir,fileName);
    }
    File file = new File(filesDir,fileName);
    try {
        RandomAccessFile randomAccessFile = new RandomAccessFile(file,
"rw"); // r - read, rw - read and write
        randomAccessFile.seek(point);
        randomAccessFile.writeChars(string);
        Log.d("Log_2", "Файл " + fileName + " записан");
    }
    catch (FileNotFoundException e){
        Log.d("Log_2", "Файл " + fileName + " не найден " +e.getMessage()
);
    }
    catch (IOException e){
        Log.d("Log_2", "Файл " + fileName + " не открыт " +e.getMessage()
);
    }
}
}
```

Функция чтения из файла произвольного доступа:

```
public static String readFromRandomAccessFile(String filesDir, String
fileName,long point, int length){

    File file = new File(filesDir, fileName);
    byte [] buffer = new byte[length];
    String data="";
    try{
        RandomAccessFile randomAccessFile = new RandomAccessFile(file,"r");
        randomAccessFile.seek(point);
        randomAccessFile.read(buffer);
        data = new String(buffer,"utf-16");
    }
    catch (FileNotFoundException e){
        Log.d("Log_2", "Файл " + fileName + " не найден " +e.getMessage() );
    }
    catch (IOException e){
        Log.d("Log_2", "Файл " + fileName + " не открыт " +e.getMessage());
    }
    return data;
}
}
```

Функция удаления файла:

```
public static void deleteFile(String filesDir,String fileName){

    File file = new File(filesDir,fileName);
    file.delete();

}
}
```

Если нужно временно сохранить некоторые данные, можно использовать специальный кэш-каталог. Когда пользователь удаляет приложение, эти файлы также удаляются.

6.2. External Storage (внешняя память)

Внешняя память имеет следующие характеристики:

- энергонезависимая;
- может быть в собственной памяти устройства (эмуляция) или на внешнем носителе (SD-карте);
- доступна не всегда, пользователь может подключать и отключать хранилища, например USB-накопители;
- для доступа к External-памяти требуется разрешение, устанавливаемое в файле манифеста приложения;
- хранилища доступны для чтения везде, поэтому невозможно контролировать чтение сохраненных в них данных;
- при удалении пользователем приложения система Android удаляет из внешних хранилищ файлы этого приложения, только если они сохраняются в директории из `getExternalFilesDirectory()`;
- доступность памяти должна проверяться;
- могут содержать *public*- и *private*-файлы.

Внешнее хранилище лучше всего подходит для файлов без ограничений доступа и файлов, которые нужно сделать доступными другим приложениям или пользователю.

Для записи во внешнее хранилище следует указать запрос разрешения `WRITE_EXTERNAL_STORAGE` в файле манифеста:

```
<manifest ...>
  <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  ...
</manifest>
```

Если приложению потребуется считать данные из внешнего хранилища (но не записать их), необходимо будет декларировать разрешение `READ_EXTERNAL_STORAGE`:

```
<manifest ...>
  <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
  ...
</manifest>
```

Однако если приложение использует разрешение `WRITE_EXTERNAL_STORAGE`, оно косвенно получает разрешение на чтение данных из внешнего хранилища.

Чтобы избежать сбоя приложения, необходимо проверить, доступна ли SD-карта хранения для операций чтения и записи. Метод

`getExternalStorageState()` используется для определения состояния носителя данных:

```
boolean isAvailable = false;
boolean isWritable = false;
boolean isReadable = false;
String state = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
    // Read- и write-операции доступны
    isAvailable = true;
    isWritable = true;
    isReadable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // Read-операция доступна
    isAvailable = true;
    isWritable = false;
    isReadable = true;
} else {
    // SD-карта не смонтирована
    isAvailable = false;
    isWritable = false;
    isReadable = false; }
```

Хотя внешнее хранилище может быть изменено пользователем и другими приложениями, существует две категории файлов, которые в нем можно сохранять, это общедоступные (*public*) и личные (*private*) файлы:

6.2.1. Общедоступные файлы

Это файлы, которые должны быть доступны другим приложениям и пользователю. Когда пользователь удаляет приложение, эти файлы должны оставаться доступны пользователю. Например, в эту категорию входят снимки, сделанные с помощью вашего приложения, а также другие загружаемые файлы.

6.2.2. Личные файлы

Это файлы, которые принадлежат приложению. Они должны удаляться при удалении приложения. Хотя технически эти файлы доступны для других приложений, поскольку находятся во внешнем хранилище, они не имеют никакой ценности для пользователей вне приложения. К этой категории относятся дополнительные ресурсы, загруженные приложением, и временные мультимедийные файлы.

Существуют следующие методы для работы с файлами:

- `getExternalStorageDirectory()` – старый способ доступа к памяти, в настоящее время не рекомендуется. Получает прямую ссылку на *root* директорию внешней памяти;
- `getExternalFilesDir(String type)` – рекомендованный способ создания *private* файлов;

- `getExternalStoragePublicDirectory()` – рекомендованный способ получения доступа к *public*-файлам. При деинсталляции приложения файлы не удаляются;
- `getFreeSpace()` и `getTotalSpace()` – методы позволяют узнать текущее доступное пространство и общее пространство в хранилище. Эта информация также позволяет избежать переполнения объема хранилища сверх определенного уровня.

```
TextView showText;
```

```
public void getPublic(Textview view) {  
    File folder =  
    Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_ALARMS  
); // Имя каталога  
    File myFile = new File(folder, "PublicData.txt"); // Имя файла  
    String text = getdata(myFile);  
    if (text != null) {  
        showText.setText(text);  
    } else {  
        showText.setText("No Data");  
    }  
}  
  
public void getPrivate(Textview view) {  
    File folder = getExternalFilesDir("FSAndroid"); // Имя каталога  
    File myFile = new File(folder, "PrivateData.txt"); // Имя файла  
    String text = getdata(myFile);  
    if (text != null) {  
        showText.setText(text);  
    } else {  
        showText.setText("No Data");  
    }  
}  
  
private String getdata(File myfile) {  
    FileInputStream fileInputStream = null;  
    try {  
        fileInputStream = new FileInputStream(myfile);  
        int i = -1;  
        StringBuffer buffer = new StringBuffer();  
        while ((i = fileInputStream.read()) != -1) {  
            buffer.append((char) i);  
        }  
        return buffer.toString();  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        if (fileInputStream != null) {  
            try {  
                fileInputStream.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
    return null;  
}
```

6.3. SharedPreferences

SharedPreferences (общие настройки) – тип файла, который позволяет сохранять информацию в формате ключ - значение для примитивных типов данных: *boolean*, *float*, *int*, *long* и *string*. Часто используется для сохранения настроек приложения.

Пары ключ - значение записываются в файлы XML, которые сохраняются в течение сеансов, даже если приложение закрыто. Можно вручную указать имя файла для сохранения.

6.3.1. Получение доступа

Для получения доступа к настройкам в коде приложения используются три метода:

- *getPreferences()* – вызывается внутри активности, чтобы обратиться к определенному для нее предпочтению; можно вызвать метод без указания названия настроек. Доступ к возвращенному ассоциативному массиву настроек ограничен активностью, из которой он был вызван. Каждая активность поддерживает только один безымянный объект *SharedPreferences*;

- *getSharedPreferences()* – вызывается внутри активности, чтобы обратиться к предпочтению на уровне приложения;

- *getDefaultSharedPreferences()* – вызывается из объекта *PreferencesManager*, чтобы получить общедоступную настройку, предоставляемую Android.

Все эти методы возвращают экземпляр класса *SharedPreferences*, из которого можно получить соответствующую настройку с помощью следующих методов:

- *getBoolean(String key, boolean defValue);*
- *getFloat(String key, float defValue);*
- *getInt(String key, int defValue);*
- *getLong(String key, long defValue);*
- *getString(String key, String defValue).*

Чтобы создать или изменить *SharedPreferences*, нужно вызвать метод *getSharedPreferences* в контексте приложения, передав имя общих настроек (имя файла). Например, следующий код обращается к файлу *SharedPreferences*, который идентифицируется строкой ресурса *R.string.preference_file_key* и открывает его с помощью *private mode*. Поэтому файл доступен только вашему приложению:

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
    getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

При именовании файлов *SharedPreferences* необходимо использовать имя, уникально идентифицируемое для приложения. Простой способ

сделать это — давать префикс имени файла с идентификатором приложения. Например:

```
by.belstu.myapplication.PREFERENCE_FILE_KEY
```

В качестве альтернативы, если нужен только один файл для *Activity*, можно использовать метод *getPreferences()*:

```
SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);
```

Файлы настроек хранятся в каталоге */data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml*. Поэтому в отладочных целях, если нужно сбросить настройки в эмуляторе, необходимо используя файловый менеджер зайти в нужную папку, удалить файл настроек и перезапустить эмулятор. На устройстве можно удалить программу и установить ее заново, то же самое можно сделать и на эмуляторе, что бывает проще, чем удалять файл настроек вручную и перезапускать эмулятор.

Если открыть файл настроек текстовым редактором, то можно увидеть приблизительно следующее:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="0005">|000000|000000|000000|0</string>
<string name="0004">|000000|000000|000000|1</string>
<string name="0006">|000000|000000|000000|0</string>
</map>
```

В данном случае в настройках хранятся только строковые значения.

6.3.2. Сохранение значений параметров

Для удобства следует создать константу для имени файла *SharedPreferences*, например:

```
// Это будет именем файла настроек
public static final String APP_PREFERENCES = "mysettings";
```

Далее нужно создать параметры для сохранения в настройках. Удобнее их сделать константами:

```
public static final String APP_PREFERENCES_NAME = "Nickname"; // Имя
public static final String APP_PREFERENCES_AGE = "Age"; // Возраст
```

Когда названия параметров определены, можно сохранять любые значения этих параметров. Для этого создается переменная, представляющая экземпляр класса *SharedPreferences*:

```
SharedPreferences mSettings = getSharedPreferences(APP_PREFERENCES,
Context.MODE_PRIVATE);
```

В указанный метод передается название файла (он будет создан автоматически) и стандартное разрешение, дающее доступ только компонентам приложения.

Чтобы внести изменения в настройки (редактировать), нужно использовать класс *SharedPreferences.Editor*. Получить объект *Editor* можно через вызов метода *edit* объекта *SharedPreferences*, который нужно изменить. После того как внесены все необходимые изменения, необходимо вызвать метод *commit()* или *apply()* объекта *Editor*, чтобы изменения вступили в силу. Метод *apply()* работает в асинхронном режиме, что является более предпочтительным вариантом. Метод *commit()* приходится использовать для старых версий и кроме того, он возвращает значение *true* в случае успеха и *false* в случае ошибки.

Предположим, что имеется два текстовых поля, где пользователь должен ввести имя и возраст. Чтобы сохранить параметр, нужно получить текст, который ввел пользователь. Получив нужный текст, нужно сохранить его через метод *putString()* (есть также *putLong()*, *putBoolean()* и т. п.):

```
SharedPreferences.Editor editor = mSettings.edit();
editor.putString(APP_PREFERENCES_NAME, "Nikita");
editor.apply();
```

Как правило, параметры в методах активности *onPause()* или *onStop()* сохраняют в тех случаях, когда между запусками приложения данные должны сохраняться. Но могут быть и другие сценарии.

6.3.3. Чтение значений параметров

Для считывания данных при загрузке приложения обычно используют методы *onCreate()* или *onResume()*. Чтобы получить доступ к настройкам программы и проверить, есть ли среди них нужный нам параметр, нужно найти ключ *Nickname* и загрузить его значение в текстовое поле.

```
if(mSettings.contains(APP_PREFERENCES_NAME)) {
    nicknameText.setText(mSettings.getString(APP_PREFERENCES_NAME, ""));
}
```

После проверки существования параметра *APP_PREFERENCES_NAME* и получения его значения через *getString()*, следует передать ключ и значение по умолчанию (используется в том случае, если для данного ключа пока что не сохранено никакое значение). Остается только загрузить полученный результат в текстовое поле.

Можно получить ассоциативный массив со всеми ключами и значениями через метод *getAll()*. После этого можно проверить наличие конкретного ключа с помощью метода *contains()*:

```
Map<String, ?> allPreferences = mSettings.getAll();
boolean containsNickName = mSettings.contains(APP_PREFERENCES_NAME);
```

6.3.4. Очистка значений

Для очистки значений используйте методы `SharedPreferences.Editor.remove(String key)` и `SharedPreferences.Editor.clear()`.

Начиная с API 11, у класса `SharedPreferences` появился новый метод `getStringSet()`, а у класса `SharedPreferences.Editor` родственный ему метод `putStringSet()`. Данные методы позволяют работать с наборами строк, что бывает удобно при большом количестве настроек, которые нужно сразу записать:

```
Set<String> names = new HashSet<String>();
names.add("Olga");
names.add("Nikita");
names.add("Andrei");
editor = mSettings.edit();
editor.putStringSet("strSetKey", names);
editor.apply();
```

ИЛИ СЧИТАТЬ:

```
Set<String> ret = mSettings.getStringSet("strSetKey", new
HashSet<String>());
for (String r : ret) {
    Log.i("Share", "Имя: " + r);
}
```

6.3.5. Удаление файла

Как упоминалось ранее, файл настроек хранится в `/data/data/имя_пакета/shared_prefs/имя_файла_настроек.xml`. Можно удалить его программно, например:

```
File file = new File("/data/data/.../shared_prefs/файл.xml");
file.delete();
```

Данные могут оставаться в памяти и временном файле `*.bak`. Поэтому даже после удаления файла, он может заново воссоздаться. Файл автоматически удалится при удалении программы.

6.3.6. Сохранение состояния активности

Если необходимо сохранить информацию, которая принадлежит активности и не должна быть доступна другим компонентам (например, переменным экземпляра класса), можно вызвать метод `Activity.getPreferences()` без указания названия. Доступ к возвращенному ассоциативному массиву ограничен активностью, из которой он был вызван. Каждая активность поддерживает только один безымянный объект `SharedPreferences`:


```

SharedPreferences activityPreferences =
    getPreferences (Activity.MODE_PRIVATE);
// Извлекаем редактор, чтобы изменить Общие настройки.
SharedPreferences.Editor editor = activityPreferences.edit();
// Записываем новые значения примитивных типов в объект Общих настроек.
editor.putString("currentTextValue", "new text");
// Сохраняем изменения.
editor.commit();

```

Система сама сгенерирует имя файла из имени пакета с добавлением слова *_preferences*.

```

SharedPreferences sharedPreferences =
    PreferenceManager.getDefaultSharedPreferences(this);
SharedPreferences.Editor edit = sharedPreferences.edit();
edit.putBoolean("isDone", false);
edit.commit();

```

6.4 DataStore

<https://developer.android.com/topic/libraries/architecture/datastore>

В последних версиях произошла миграция от SharedPreferences к DataStore.

Недостатки SharedPreferences:

- вручную необходимо переключиться на фоновый поток не может сигнализировать об ошибках, IOException
- не защищены от исключения времени выполнения, может вызвать исключение синтаксического анализа.
- нет транзакционного API

Jetpack DataStore – решение, которое позволяет хранить пары ключ-значение или типизированные объекты с [protocol buffers](#). DataStore позволяет хранить асинхронно, согласованно и транзакционно.

[Protocol buffers](#) - это не зависящий от языка и платформы механизм Google для сериализации структурированных данных – похож на XML, но меньше, быстрее и проще. Структурированные данные определяются один раз и используя различные потоки считываются и записываются.

```

message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
}

```

DataStore предоставляет две разные реализации: **Preferences DataStore** и **Proto DataStore**.

Preferences DataStore хранит данные и получает доступ с помощью ключей. Эта реализация не требует предопределенной схемы и не обеспечивает безопасность типов.

Proto DataStore хранит данные как экземпляры настраиваемого типа. Эта реализация требует, чтобы вы определяли схему с использованием protocol buffers, и обеспечивает безопасность типов.

Feature	SharedPreferences	Preferences DataStore	Proto DataStore
Async API	✓ (only for reading changed values, via listener)	✓ (via Flow)	✓ (via Flow)
Synchronous API	✓ (but not safe to call on UI thread)	✗	✗
Safe to call on UI thread	✗*	✓ (work is moved to Dispatchers.IO under the hood)	✓ (work is moved to Dispatchers.IO under the hood)
Can signal errors	✗	✓	✓
Safe from runtime exceptions	✗**	✓	✓
Has a transactional API with strong consistency guarantees	✗	✓	✓
Handles data migration	✗	✓ (from SharedPreferences)	✓ (from SharedPreferences)
Type safety	✗	✗	✓ with Protocol Buffers

<https://howtodoandroid.com/datastore-android-jetpack/>

<https://developer.android.com/topic/libraries/architecture/datastore>

<https://developer.android.com/codelabs/android-preferences-datastore#5>

Вы можете использовать DataStore только в RxJava.

```

/PreferenceDataStoreB
RxDataStore<Preferences> datastore =
    new RxPreferenceDataStoreBuilder(this, /*name=*/
"settings").build();

//write
Single<Preferences> updateResult =
RxDataStore.updateDataAsync(dataStore,
    prefsIn -> {
        MutablePreferences mutablePreferences =
prefsIn.toMutablePreferences();
        Integer currentInt = prefsIn.get(INTEGER_KEY);
        mutablePreferences.set(INTEGER_KEY, currentInt != null
? currentInt + 1 : 1);
    });

```

```
        return Single.just(mutablePreferences);
    });
//read
    Preferences.Key<Integer> EXAMPLE_COUNTER =
    PreferencesKeys.int("example_counter");

    Flowable<Integer> exampleCounterFlow =
        RxDataStore.data(dataStore).map(prefs ->
    prefs.get(EXAMPLE_COUNTER));
```