**Лекция 13. Потоки и сервисы**

В этой лекции обсуждается компонент разработки Android, который не ***виден*** пользователям сразу: [**Services**](https://developer.android.com/guide/components/services.html)[40.](https://info448.github.io/services.html#fn40)*Services* — это своего рода Activities, которые не имеют пользовательского интерфейса или взаимодействия с пользователем, напрямую привязанных к ним. Services могут запускаться Activities или Applications, но затем выполнять свои собственные действия в фоновом режиме *даже после закрытия стартового компонента* . После запуска Service он продолжает работать до тех пор, пока не будет явно остановлен (Services могут быть уничтожены ОС для экономии памяти, аналогично Activity, но Services имеют более высокий [«приоритет»](https://developer.android.com/guide/topics/processes/process-lifecycle.html) и поэтому не так легко завершаются).

Некоторые распространенные варианты использования Услуги включают в себя:

* Загрузка или выгрузка данных из/в сеть в фоновом режиме, даже если приложение закрыто
* Сохранение данных в базе данных без сбоев, если пользователь покидает приложение
* Запуск какой-либо другой длительной «фоновой» задачи даже после закрытия приложения, например, воспроизведение музыки!

Наиболее распространенное использование служб — регулярное выполнение фоновой задачи таким образом, чтобы не блокировать основное взаимодействие с пользовательским интерфейсом. Таким образом, эта лекция начинается с обсуждения потоков и процессов в Android, а затем подробно описывается, как реализовать службу.

В этой лекции используется код, найденный по адресу <https://github.com/info448/lecture13-services> .

**13.1 Потоки и процессы**

**Параллелизм** — это процесс, при котором у нас есть несколько *процессов* (подумайте: методов), работающих одновременно. Это можно противопоставить процессам, которые работают **последовательно** или один за другим.

Например, если вы вызываете два метода один за другим, то выполнение второго метода будет «ждать» завершения первого:

*//java*

**public** void countUp() {

 **for**(int i=0; i<1000; i++){

 System.out.println(i);

 }

}

**public** void countDown() {

 **for**(int i=0; i> -1000; i--){

 System.out.println(i);

 }

}

countUp(); *//start counting up*

countDown(); *//start counting down (once finished going up)*

Компьютеры, как правило, выполняют только одну операцию за раз: ваш центральный процессор (ЦП) просто складывает два числа снова и снова, миллиарды раз в секунду.

* Стандартная мера *скорости* (сколько раз в секунду) — hertz(Гц). Таким образом, процессор с частотой 2 гигагерца (ГГц) может выполнять 2 миллиарда операций в секунду.

Однако мы не осознаем, что компьютеры делают только одну задачу за раз! Это потому, что компьютеры действительно хороши в *многозадачности* : они выполняют крошечную часть одной задачи, а затем переходят к другой задаче и делают немного ее, а затем переходят к другой задаче и делают немного ее, а затем возвращаются к первой задаче и так далее.



Схема переключения потоков (источник неизвестен)

Эти «задачи» делятся на два типа: **процессы** и **потоки** . ***Прочитайте***[***это краткое изложение***](https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html)***различий между ними*** .

Разбивая программу на потоки (которые «переплетаются»), мы можем фактически заставить компьютер выполнять две задачи одновременно. Это *особенно* полезно, если одна из «задач» может занять очень много времени — вместо того, чтобы **блокировать** приложение, мы можем позволить другим задачам также выполниться, пока мы ждем завершения долгой задачи.

**Потоки Java**

В Java мы создаем Поток, создавая класс, который является implementsинтерфейсом [**Runnable**](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html). Это представляет собой класс, который может быть «запущен» в отдельном потоке! run()Метод, требуемый интерфейсом, действует немного как «главный» метод для этого Потока: когда мы запускаем Поток, это метод, который будет вызван.

Если мы просто создадим экземпляр Runnable()и вызовем его run()метод, это фактически не выполнит метод в другом потоке (помните: интерфейс — это всего лишь «знак»; мы могли бы назвать интерфейс и метод как угодно, и он все равно скомпилируется). Вместо этого мы выполняем код в отдельном потоке, используя экземпляр класса [**Thread**](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html). Этот класс фактически выполняет работу по запуску кода в отдельном потоке.

Threadимеет [конструктор](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#Thread-java.lang.Runnable-) , который принимает Runnableэкземпляр в качестве параметра — вы передаете объект, представляющий «код для запуска» объекту Thread(это пример *шаблона «стратегия* »). Затем вы можете фактически **запустить**Thread , вызвав его метод .start()( *не* метод run!).

* Здесь будут полезны анонимные переменные; вам не нужно назначать имя переменной объектам Runnableили даже самим Threadобъектам, если вы просто используете их напрямую.

Например, выполнение указанных выше методов countUp()и countDown()в отдельных потоках может привести к тому, что выходные данные будут «переплетены», и некоторые положительные числа будут отображаться вместо некоторых отрицательных (вам может потребоваться увеличить количество или замедлить операцию, чтобы увидеть ее в действии, чтобы один поток не завершился до «переключения»).

**Android-темы**

Приложения Android по умолчанию запускаются в ***основном потоке*** (также называемом ***потоком пользовательского интерфейса*** ). Этот поток отвечает за все взаимодействия с пользователем — обработку нажатий кнопок, прокрутку, перетаскивание и т. д. — а также за *вывод* пользовательского интерфейса , например, рисование и отображение текста! Подробнее см. в [разделе Потоки Android .](https://developer.android.com/guide/components/processes-and-threads.html#Threads)

* Как упоминалось выше, поток — это часть программы, которая независимо запланирована процессором. Компьютеры выполняют только одну задачу за раз, но создают видимость того, что они выполняют множество задач одновременно, переключаясь между ними (т. е. между процессами) очень быстро. Потоки — это способ, которым мы можем разбить одно приложение или процесс на маленькие «подпроцессы», которые могут выполняться одновременно — периодически переключаясь вперед и назад, чтобы у каждого была возможность поработать

В пределах одного потока все вызовы методов **синхронны** — то есть один должен завершиться до того, как произойдет следующий. Вы не можете перейти к шагу 4, не завершив шаг 3. В системе, управляемой событиями, такой как Android, каждый вызов метода достаточно быстр, чтобы это не было проблемой (вы заканчиваете обработку одного щелчка к тому времени, как происходит следующий). Но длинные, затянутые процессы, такие как сетевой доступ, обработка битовых карт или доступ к базе данных, могут привести к тому, что другим задачам придется ждать. Это как пробка на дороге!

* Такие задачи, как сетевой доступ, **блокируют** вызовы методов, которые останавливают продолжение работы потока. Заблокированный *основной поток* приведет к печально известной ошибке **«Приложение не отвечает» (ANR)** !

Таким образом, нам нужно переместить любой «медленный» код (такой как сетевой доступ) *из* основного потока в **фоновый поток** , тем самым позволяя ему работать, не блокируя взаимодействие с пользователем, которое происходит в основном потоке. Чтобы сделать это в Android, мы используем класс под названием [41](https://info448.github.io/services.html#fn41) для выполнения задачи (такой как сетевой доступ) асинхронно — не дожидаясь других потоков.[ASyncTask](https://developer.android.com/reference/android/os/AsyncTask.html)

Изучение разработки Android включает в себя знание того, какие классы существуют и могут быть использованы для решения проблем, но как мы смогли узнать о существовании этого очень полезного (и специализированного) ASyncTaskкласса? Мы начали с официального API Guide on [Processes and Threads Guide](https://developer.android.com/guide/components/processes-and-threads.html)[42](https://info448.github.io/services.html#fn42) , который знакомит с этим классом! Таким образом, чтобы узнать о новых возможностях Android, *прочтите документацию* .

Обратите внимание, что ASyncTaskфоновый поток будет *привязан к жизненному циклу Activity* : если мы закроем Activity, сетевое соединение также прервется. Лучшим вариантом часто является использование Service, описанного ниже.

ASyncTaskможет быть довольно сложным, но является хорошим кандидатом для практики изучения документации API. Глядя на эту документацию, первое, что вы должны заметить (или заметили бы, если бы API был немного более читаемым), это то , ASyncTaskчто **abstract**, что означает, что вам нужно будет *создать подкласс* для его использования. Таким образом, вы можете создать подкласс как *внутренний* класс внутри Activity, который будет его использовать.

Вы также должны заметить, что ASyncTaskэто *универсальный* класс с тремя (3) общими параметрами: тип параметра для задачи, тип измерения прогресса, сообщаемого задачей, и тип результата задачи. Мы можем указать, какие типы параметров и результатов мы хотим получить от нашего асинхронного метода (например, взять Stringи вернуть a String[]). Вы можете использовать Voidтип для неопределенного типа, например, с измерением прогресса, если вы его не отслеживаете.

* На самом деле мы можем передавать несколько Stringаргументов, используя String... paramsсинтаксис оператора распространения (представляющий произвольное количество элементов этого типа). Подробности см. [здесь](https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html#varargs) . Значение, которое ASyncTaskметоды *фактически* получают, представляет собой массив аргументов.

Когда вы «запускаете» AsyncTask, он выполняет четыре (4) действия, представленные четырьмя методами:

1. onPreExecute()вызывается *в потоке пользовательского интерфейса* перед запуском задачи. Этот метод можно использовать для выполнения любой настройки для задачи.
2. doInBackground(Params...)вызывается *в фоновом потоке* для выполнения работы, которую вы хотите выполнить асинхронно. Вы **должны** переопределить этот метод (это abstract!) Параметры и возвращаемый тип для метода должны соответствовать ASyncTaskуниверсальным типам.
3. onProgressUpdate()может быть косвенно вызван *в потоке пользовательского интерфейса* , если мы хотим обновить наш прогресс (например, обновить полосу прогресса). Обратите внимание, что изменения пользовательского интерфейса могут быть сделаны **только** в потоке пользовательского интерфейса!
4. onPostExecute(Result)вызывается *в потоке пользовательского интерфейса* для обработки любых результатов задачи, которые передаются в качестве параметров этому методу по doInBackgroundзавершении.

Это doInBackground()то, что происходит в фоновом потоке (и является сутью задачи), поэтому вы можете поместить туда, например, вызовы методов доступа к сети или базе данных.

Затем мы можем *создать* новый ASyncTaskобъект в обратном вызове Activity onCreate()и вызвать его, ASyncTask#execute(params)чтобы запустить задачу, выполняющуюся в ее собственном потоке.

Чтобы вернуть результаты обратно в визуализированный View, вы используете doPostExecute()метод. Этот метод запускается в *потоке пользовательского интерфейса* , поэтому вы можете использовать его для обновления View (мы можем изменять View *только в потоке пользовательского интерфейса, чтобы избежать столкновений). Он также автоматически получает результаты , возвращаемые* переданным doInBackground()ему!

* Например, вы можете взять результат String[]и поместить его в Adapterfor ListView.

ASyncTaskэто самый простой, общий способ сделать какую-то работу в фоновом потоке. Однако у него есть несколько ограничений:

* Жизненный цикл ASyncTaskсвязан с жизненным циклом содержащей его Activity. Это означает, что если содержащая Activity уничтожена (например, finished()), то и ASyncTask тоже — вы потеряете все загружаемые данные или можете каким-то образом повредить свою базу данных.
* Для обработки множества различных фоновых задач вам придется заняться собственным управлением задачами.

Таким образом, повторяющиеся или длительные задачи (например, загрузка больших файлов) часто лучше выполнять с помощью **служб** .

**13.2 IntentServices**

Как упоминалось выше, Service — это компонент приложения (например, Activity), который работает в «фоновом режиме», даже если пользователь переключается на другое приложение. Services обычно не имеют связанного пользовательского интерфейса, а вместо этого просто группируют и управляют большими объемами обработки данных (например, для сетевых загрузок, обработки мультимедиа или доступа к базе данных).

* Важно отметить о службах: служба **не** является отдельным процессом; она выполняется в том же процессе, что и приложение, которое ее запускает (если не указано иное). Аналогично, служба **не** является потоком и на самом деле не должна выполняться вне потока пользовательского интерфейса! Однако мы довольно часто *хотим* запустить службу вне потока пользовательского интерфейса, и поэтому часто заставляем ее порождать и запускать новый Runnableпоток. Когда мы говорим, что служба выполняется «в фоновом режиме», мы имеем в виду с точки зрения пользователя, а не обязательно в фоновом потоке.

Чтобы создать Службу, мы собираемся создать подкласс [Service](https://developer.android.com/reference/android/app/Service.html)и переопределить некоторые *обратные вызовы жизненного цикла* , как мы это делали с Activity, Fragment, BroadcastReceiver, и большинством других компонентов Android. В этом смысле Службы фактически реализованы как Действия, которые работают без пользовательского интерфейса («в фоновом режиме»).

Поскольку Services будут выполнять дополнительные фоновые вычисления, важно также создать отдельный фоновый поток, чтобы не блокировать Main Thread. Поскольку создание Service, который выполняет некоторую (конкретную) задачу в фоновом потоке, является настолько распространенным, Android включает подкласс, который Serviceмы можем использовать для выполнения именно этой работы. Этот класс называется [IntentService](https://developer.android.com/reference/android/app/IntentService.html)—a service, которая отвечает Intentsи выполняет некоторую работу в ответ на них.

* IntentService выполняет ту же работу, что и Activity ASyncTask, но с тем преимуществом, что он продолжит выполнять эту работу даже после уничтожения Activity!
* IntentServiceбудет прослушивать любые входящие «запросы» ( Intents) и «ставить их в очередь», обрабатывая каждый из них по очереди. Как только у службы не останется задач, она завершит работу, чтобы сэкономить память (хотя она перезапустится, если Intentsей будут отправлены новые запросы). По сути, она сама выполняет большую часть настройки и очистки, связанных с использованием службы!
* (Эта лекция начнется с IntentServiceболее простой версии Services, а затем перейдет к более общей и сложной версии Services).

Мы создаем IntentService, определяя новый класс (например, CountingService), который является подклассом IntentService.

* Реализуйте *конструктор* по умолчанию , который может вызывать super(String nameForDebugging). Это позволяет создавать экземпляр класса (фреймворком Android; опять же, как Activity).
* Также реализуйте onHandleIntent(Intent)метод. Входящие Intentsбудут ждать своей очереди в очереди, а затем каждый будет доставлен этому методу по очереди. Обратите внимание, что вся эта работа (доставка и выполнение) будет происходить в **фоновом потоке,** предоставленном IntentService.

Например, мы можем заставить Службу (при запуске) выходить из системы по счетчику, делая паузу в несколько секунд между ними. Это будет представлять собой «дорогостоящую» логику для выполнения, а-ля доступ к сети или базе данных.

*//java*

**for**(int count=0; count<=10; count++){

 Log.v(TAG, "Count: "+count);

 **try** {

 Thread.sleep(2000); *//sleep for 2 seconds*

 } **catch** (InterruptedException e) {

 Thread.currentThread().interrupt();

 }

}

*//kotlin*

**for** (count **in** 0..10) {

 Log.v(TAG, "Count: $count")

 **try** {

 Thread.sleep(2000) *//sleep for 2 seconds*

 } **catch** (e: InterruptedException) {

 Thread.currentThread().interrupt()

 }

}

Как и в случае с Activity, нам также необходимо объявить <service>в Manifest, как дочерний элемент <application>:

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

Наконец, мы можем отправить Intentв Службу, используя startService()метод. Это похоже на startActivity(), но для Служб! Мы можем использовать явные намерения, как и в случае с Действиями, и даже можем включить Дополнительные, если хотим пометить и отслеживать конкретные Намерения, отправленные в Службу.

* Когда Служба запускается, мы можем видеть, что она начинает отсчет (но без блокировки потока пользовательского интерфейса). Мы также можем уничтожить Activity и увидеть, что Служба продолжает работать.

Если мы хотим, чтобы Служба взаимодействовала с пользовательским интерфейсом (например, отображала тост), нам нужно убедиться, что это происходит в потоке пользовательского интерфейса (вы не можете изменить пользовательский интерфейс в отдельном потоке). Для этого требуется *межпотоковое взаимодействие* : нам нужно получить сообщение (вызов функции) из фонового потока в поток пользовательского интерфейса.

* Именно это и делают для нас различные ASyncTaskметоды потоков пользовательского интерфейса (например, onProgressUpdate()).

Мы можем осуществлять эту коммуникацию с помощью [Handler](https://developer.android.com/reference/android/os/Handler.html), который представляет собой объект, используемый для передачи сообщений между потоками — он «обрабатывает» сообщения!

* Мы создаем экземпляр new Handler()объекта (например, в обратном вызове Service onCreate()), вызывая метод этого объекта, когда мы хотим «отправить» сообщение. Самый простой способ отправить сообщение — использовать функцию post(), которая принимает Runnable()метод, который будет выполнен *в Main Thread* :
* *//java*
* mHandler.post(**new** Runnable() {
* @Override
* **public** void run() {
* Toast.makeText(CountingService.this, "Count: " + count, Toast.LENGTH\_SHORT).show();
* }

});

*//kotlin*

mHandler.post {

 Toast.makeText(this@CountingService, "Count: $count", Toast.LENGTH\_SHORT).show()

 Log.v(TAG, "" + count)

}

**Жизненный цикл услуги**

Продемонстрировав базовое использование сервиса, рассмотрим, что происходит «под капотом» — начиная с [жизненного цикла сервиса](https://developer.android.com/guide/components/services.html#LifecycleCallbacks) . На самом деле существует два разных «типа» сервисов с разными вариациями жизненного цикла. **Запущенные сервисы** (или «непривязанные» сервисы) — это те, которые инициируются через startService()функцию, как в приведенном выше примере. Другой вариант, **привязанные сервисы** , — это сервисы, которые имеют привязанные к ним «клиентские» действия для взаимодействия; подробности см. ниже.



Диаграмма жизненного цикла услуги от Google [43](https://info448.github.io/services.html#fn43)

* Как и в случае с Activity, у Services есть onCreate()метод, который вызывается при первом создании Service. Поскольку у Services нет пользовательского интерфейса для настройки, мы нечасто делаем здесь много.
	+ IntentService уже переопределяет это, чтобы настроить «очередь сообщений», чтобы он мог выстраивать в очередь намерения (задачи) для запуска по одному за раз.
* Самый важный обратный вызов для *запущенной службы* называется onStartCommand(). Этот метод вызывается, когда служба **получает команду** от другого компонента (через Intent). Важно, onStartCommand()что он вызывается не только при первом запуске службы, но и всякий раз, когда служба получает Intent для запуска (даже если служба уже запущена)! Эти Intent отправляются через startService().
	+ Обратите внимание, что при работе с *IntentService*конкретным, onStartCommand()будет «ставить в очередь» любые входящие Intent. Когда очередь конкретного Intent будет «запущена», этот Intent автоматически передается методу onHandleIntent(), который выполняется в *фоновом потоке* (аналогично AsyncTask#doInBackground()). Этот обратный вызов не является частью обычного жизненного цикла Service, но является специальным вспомогательным методом, используемым IntentService — аналогично тому, как onCreateDialog()специальный метод используется DialogFragments.
* Обратные вызовы onBind()и onUnbind()используются для связанных служб и обсуждаются ниже.
	+ IntentService имеет onBind()реализацию по умолчанию, которая возвращает null.
* Наконец, у служб есть onDestroy()обратный вызов, который снова эквивалентен обратному вызову активности.
	+ В общем, Службам нужно вручную сказать **остановиться** . Мы останавливаем Службу, используя stopService(Intent)для отправки этой Службе намерения «стоп». Служба также может остановить себя, вызвав stopSelf().
	+ *Важно* : Когда указано остановиться, an IntentServiceзавершит обработку любых Intents, которые в данный момент «выполняются», но любые другие Intents, которые «стоят в очереди», будут удалены. Как только в очереди больше не будет IntentService, IntentService вызовет stopSelf(), тем самым вызвав onDestroy()выполнение обратного вызова.

***Практика:*** заполните функции обратного вызова сообщениями Log или Toast, чтобы увидеть, как и когда они выполняются. Например:

*//java*

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

 Toast.makeText(**this**, "Intent received", Toast.LENGTH\_SHORT).show();

 **return** **super**.onStartCommand(intent, flags, startId);

}

В качестве последнего замечания о жизненном цикле сервиса рассмотрим intвозвращаемое onStartCommand(). Это intфлаг, который указывает, как сервис должен [себя вести](https://developer.android.com/guide/components/services.html#ExtendingService)[44](https://info448.github.io/services.html#fn44) при его «перезапуске» после уничтожения:

* [START\_NOT\_STICKY](https://developer.android.com/reference/android/app/Service.html#START_NOT_STICKY)указывает, что если Служба уничтожена системой, ее не следует воссоздавать. Это «самый безопасный вариант» для избежания посторонних запусков служб; вместо этого просто перезапустите Службу приложением.
* [START\_STICKY](https://developer.android.com/reference/android/app/Service.html#START_STICKY)указывает, что если служба уничтожена системой, ее следует воссоздать, когда это возможно. В этот момент onStartCommand()будет вызван путем доставки nullIntent (если только не было других начальных Intent, ожидающих доставки, в этом случае они используются). Этот параметр хорошо подходит для медиаплееров или подобных служб, которые работают неопределенно долго (а не выполняют определенные команды).
* [START\_REDELIVER\_INTENT](https://developer.android.com/reference/android/app/Service.html#START_REDELIVER_INTENT)указывает, что если Служба уничтожена системой, она должна быть воссоздана, когда это возможно. В этот момент onStartCommand()будет вызван с *последним* Intent, который был доставлен Службе (и любые другие Intent доставляются в свою очередь). Этот параметр хорош для Служб, которые активно выполняют задания, которые необходимо возобновить, например, загрузку файла.

Другими словами: службы могут быть убиты, но мы можем указать, как они будут воскрешены! И, конечно, мы можем и должны возвращать разные значения для разных стартовых команд (Intent): так, Intent для загрузки музыкального файла может вернуть START\_REDELIVER\_INTENT, но Intent для воспроизведения музыкального файла может вернуть START\_STICKY.

**13.3 Пример: Музыкальный сервис**

Одно из классических применений фоновой службы — воспроизведение музыки, поэтому мы будем использовать это в качестве примера. Можно воспроизводить музыку непосредственно из Activity, и музыка будет продолжать воспроизводиться, пока Activity активна. Но помните, что Activity хрупки и могут быть уничтожены в любой момент (будь то нами или системой для экономии ресурсов). Поэтому для того, чтобы наша музыка продолжала воспроизводиться, даже когда мы занимаемся другими задачами, мы должны [использовать Service](http://developer.android.com/guide/topics/media/mediaplayer.html#mpandservices) . Services имеют более высокий «приоритет» в глазах системы Android, и поэтому не жертвуются ресурсами так же легко, как обычные Activities.

**Медиаплеер**

Для того, чтобы сделать музыкальный сервис, нам нужно кратко объяснить, как воспроизводить музыку с помощью [MediaPlayer](https://developer.android.com/guide/topics/media/mediaplayer.html). Это основной API для воспроизведения звука и видео (например, если вы хотите воспроизводить .mp3файлы).

Android на самом деле имеет в общей сложности три (3) аудио API! [SoundPool](https://developer.android.com/reference/android/media/SoundPool.html)API отлично подходит для коротких звуковых эффектов, которые воспроизводятся одновременно (хотя вам нужно выполнить дополнительную работу, чтобы загрузить эти клипы заранее), например, для простых игр. API [AudioTrack](http://developer.android.com/reference/android/media/AudioTrack.html)позволяет вам воспроизводить аудио на очень низком уровне (например, «проталкивая» байты в поток). Это полезно для сгенерированного аудио (например, MIDI-музыки) или если вы пытаетесь сделать что-то еще на низком уровне.

MediaPlayerочень прост в использовании, особенно при воспроизведении локально определенного ресурса (например, чего-то в res/raw/). Вы просто используете фабрику для создания нового MediaPlayerобъекта, а затем вызываете .play()ее:

MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.my\_sound\_file);

mediaPlayer.start(); *// no need to call prepare(); create() does that for you*

Мы также можем нажать кнопку, .pause()чтобы приостановить воспроизведение музыки, .seekTo()перейти к определенной миллисекунде и .stop()остановить воспроизведение музыки.

* Обратите внимание, что когда мы stop(), нам также необходимо освободить все ресурсы, используемые MediaPlayer(чтобы освободить память):

mediaPlayer.release();

mediaPlayer = **null**;

Мы также можем реализовать и зарегистрировать событие MediaPlayer.OnCompletionListener, которое будет выполнять какие-либо действия после окончания воспроизведения песни.

Наконец, его можно использовать MediaPlayerдля воспроизведения файлов из ContentProviderИнтернета или даже за его пределами!

*//java*

String url = "http://........"; *// your URL here*

MediaPlayer mediaPlayer = **new** MediaPlayer();

mediaPlayer.setAudioStreamType(AudioManager.STREAM\_MUSIC);

mediaPlayer.setDataSource(url);

mediaPlayer.prepareAsync(); *//prepare media in the background (buffering, etc)*

 *//.prepare() for synchronous buffering*

mediaPlayer.setOnPreparedListener(**this**); *//handle when file is buffered*

mediaPlayer.start();

**Создание услуги**

Чтобы сделать наш музыкальный сервис, мы собираемся создать подкласс Serviceсамого класса ( *не забудьте включить Service вManifest* ) и вручную настроить все части. В частности, мы заполним обратные вызовы жизненного цикла:

* onCreate()мы можем включить, хотя это не обязательно.
* onStartCommand()следует создать и запустить наш MediaPlayer. Мы можем вернуться, START\_NOT\_STICKYчтобы музыка не запускалась снова случайным образом, если системе понадобится уничтожить Службу (так как она не будет создана заново).

*Важно:* Обычно с помощью Service мы создаем фоновый поток для обработки любой внешней работы (например, подготовки музыки к воспроизведению, как это делается с помощью create()метода). Однако это может быть необязательно для MediaPlayer при загрузке ресурсов.

* onDestroy()может останавливать, освобождать и очищать MediaPlayer. Мы можем реализовать это в отдельной вспомогательной функции для повторного использования.
	+ Если мы хотим управлять паузой, мы можем указать это в Intent, который мы отправляем в службу (например, через пользовательское ACTION или Extra). Но более эффективный подход — использовать Service Binding; подробности см. ниже.

Теперь мы можем иметь нашу Activity startService()и stopService()воспроизводить нашу музыку в «фоновом режиме», даже если мы выйдем и вернемся к Activity!

**13.4 Службы переднего плана**

Службы обычно являются «фоновыми» задачами, которые выполняются без какого-либо пользовательского интерфейса и о которых пользователь не знает (например, для загрузки или выгрузки данных и т. д.). Но воспроизведение музыки — это определенно то, о чем пользователь *знает* , и на самом деле может захотеть с этим взаимодействовать! Поэтому мы хотели бы, чтобы у Службы был какой-то пользовательский интерфейс, но мы хотели бы по-прежнему сохранять эту Службу отдельно от Действия (чтобы она могла работать при активном Действии).

Для этого мы используем то, что называется [**Foreground Service**](https://developer.android.com/guide/components/services.html#Foreground) . Foreground Services представляют собой Services, которые отделены от Activities (в конце концов, они *являются* Services), но пользователь знает о них — и, соответственно, имеют еще более высокий [приоритет](https://developer.android.com/guide/topics/processes/process-lifecycle.html) выживания , если ОС испытывает нехватку памяти!

Для служб переднего плана требуется **Уведомление** в строке состояния, похожее на Уведомления, которые мы создали ранее. Это Уведомление будет эффективно действовать как «пользовательский интерфейс» для Службы и позволит пользователю видеть и быть в курсе ее выполнения!

Мы создаем это уведомление внутри метода Службы onStartCommand(), затем передаем его методу startForeground(), чтобы перевести нашу Службу на передний план:

*//java, pre-Oreo*

String songName = "The Entertainer";

PendingIntent pendingIntent = PendingIntent.getActivity(getApplicationContext(), 0,

 **new** Intent(getApplicationContext(), MainActivity.class), PendingIntent.FLAG\_UPDATE\_CURRENT);

Notification notification = **new** NotificationCompat.Builder(**this**)

 .setSmallIcon(android.R.drawable.ic\_media\_play)

 .setContentTitle("Music Player")

 .setContentText("Now playing: "+songName)

 .setContentIntent(pendingIntent)

 .setOngoing(**true**) *//cannot be dismissed by the user*

 .build();

startForeground(NOTIFICATION\_ID, notification); *//make this a foreground service!*

*//kotlin*

**val** songName = "The Entertainer"

**val** pendingIntent = PendingIntent.getActivity(applicationContext, 0,

 Intent(applicationContext, MainActivity::**class**.java), PendingIntent.FLAG\_UPDATE\_CURRENT)

if (android.os.Build.VERSION.SDK\_INT >= android.os.Build.VERSION\_CODES.O) {

 **val** channel = NotificationChannel(NOTIFICATION\_CHANNEL\_ID, "Demo channel", NotificationManager.IMPORTANCE\_LOW)

 **val** notificationManager = getSystemService(Context.NOTIFICATION\_SERVICE) **as** NotificationManager

 notificationManager.createNotificationChannel(channel)

}

**val** notification = NotificationCompat.Builder(**this**, NOTIFICATION\_CHANNEL\_ID)

 .setSmallIcon(android.R.drawable.ic\_media\_play)

 .setContentTitle("Music Player")

 .setContentText("Now playing: $songName")

 .setContentIntent(pendingIntent)

 .setOngoing(**true**) *//cannot be dismissed by the user*

 .build()

startForeground(NOTIFICATION\_ID, notification) *//make this a foreground service!*

Некоторые подробности об этом уведомлении:

* Создаем и устанавливаем иконку, заголовок и текст, как в предыдущей лекции.
* Мы даем Notification PendingIntentдля запуска при выборе. Этот PendingIntent может просто открыть наш MainActivity, позволяя нам управлять Player. (В качестве альтернативы мы могли бы использовать [Notification Actions](https://developer.android.com/guide/topics/ui/notifiers/notifications.html#controllingMedia) для управления музыкой напрямую).
* *Мы устанавливаем постоянное* уведомление , чтобы гарантировать, что пользователь не сможет его отклонить.

*Важно* , как только мы закончим работу на переднем плане (например, воспроизведение музыки), мы должны вызвать stopForeground(true), чтобы избавиться от службы переднего плана. Это хорошая вещь, которую можно сделать в нашем stopMusic()помощнике (вызывается из onDestroy()).

Есть еще несколько деталей, которые вам следует учесть, если вы создаете полноценное приложение для музыкального проигрывателя, в том числе: предотвращение перехода телефона в спящий режим, одновременное воспроизведение других аудиофайлов (например, звуков уведомлений, мелодий звонка), переключение на внешние динамики и т. д. Более подробную информацию см. [в руководстве .](https://developer.android.com/guide/topics/media/mediaplayer.html)

**13.5 Связанные услуги**

Как упоминалось выше, существует два «типа» служб: **запущенные службы** (службы, запущенные с помощью startService()) и **привязанные службы** . [Привязанная служба](https://developer.android.com/guide/components/bound-services.html) — это служба, которая действует как «сервер» в клиент-серверной настройке: она позволяет клиентским Activity «подключаться» к ней ( ***привязывать ее*** ) и затем обмениваться с ней сообщениями — в первую очередь, вызывая методы в Service. Эти сообщения могут быть даже *между процессами* , что позволяет осуществлять [межпроцессное взаимодействие](https://en.wikipedia.org/wiki/Inter-process_communication) ! Это полезно, когда вы хотите каким-либо образом взаимодействовать со службой из Activity (например, если мы хотим слушать pause()нашу музыку) или если мы хотим сделать возможности службы доступными для других приложений.

Мы создаем привязанную службу, заставляя службу реализовывать и использовать **onBind()**обратный вызов. Этот метод возвращает IBinderобъект ( Iуказывает, что это *Интерфейс* , следуя соглашению Java, распространенному в корпоративном программном обеспечении). Когда Действия подключаются к этой службе (используя метод bindService()), этот IBinderобъект передается им, и они могут использовать его для получения доступа к процессу службы для вызова ее методов. Фактически, процесс привязки создает объект *,* представляющий эту службу, поэтому Действия могут вызывать методы службы без необходимости отправлять ей намерения!

В качестве примера добавим возможность «приостанавливать» воспроизведение музыки как привязанную службу:

Первое, что нам нужно сделать, это заставить нашу Службу реализовать onBind()метод. Для этого нам понадобится IBinderобъект для возврата (то есть объект класса, реализующего IBinderинтерфейс). Для «локальных служб» (например, Служб, которые запускаются *в том же процессе* ) самый простой способ получить IBinder— расширить Binderкласс:

*//kotlin*

**public** **class** MyBinder **extends** Binder { *//implements IBinder*

 *//binder class methods will go here!*

}

**private** final IBinder mBinder = **new** MyBinder(); *//singleton instance variable*

*//kotlin*

**class** LocalBinder : Binder() {

 *//binder class methods will go here!*

}

**private** **val** mLocalBinder = LocalBinder() *//singleton instance variable*

* Наша локальная версия пока «пустая»; подробности мы добавим ниже.
* Мы просто вернем этот объект из onBind().

Поскольку Activity получает копию этого MyBinderобъекта, этот класс может быть разработан для поддержки взаимодействия Activity со Service несколькими различными способами:

1. The IBinderможет предоставлять publicметоды для вызова Activity. Затем эти методы могут получать доступ к переменным экземпляра Service (поскольку MyBinderявляется вложенным классом). Это заставляет его IBinderбуквально действовать как «публичный интерфейс» для Service!
2. *//in MyBinder*
3. **public** String getSongName() {
4. **return** songName; *//access Service instance variable*

}

1. Может IBinderпредоставлять доступ к самой Службе (через геттер, который возвращает Serviceобъект). Затем Activity может вызывать любые публичные методы, предоставляемые этим Serviceклассом.
2. *//in MyBinder*
3. **public** MusicService getService() {
4. *// Return this instance of this Service so clients can call public methods on it!*
5. **return** MusicService.this;

}

1. Может IBinderпредоставлять доступ к какому-либо другому объекту, «принадлежащему» Службе, методы которого затем может вызывать Activity (например, MediaPlayer). Это своего рода компромисс между первыми двумя вариантами: вам не нужно реализовывать определенный публичный интерфейс на IBinder, но вам также не нужно предоставлять полный доступ к вашему объекту Службы! Это хорошо для раскрытия только части Службы.

В Activity нам нужно сделать немного больше работы, чтобы взаимодействовать с привязанной Service. Мы привязываемся к Service с помощью [bindService(Intent, ServiceConnection, flag)](http://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)). Мы можем сделать это, когда захотим, чтобы Service был доступен ( onStart()это хороший вариант, поэтому он доступен, когда Activity активен).

* Параметр Intent должен быть адресован службе, которую мы хотим привязать.
* Параметр ServiceConnection— это ссылка на объект, который реализует ServiceConnectionинтерфейс, предоставляя обратные вызовы, которые могут быть выполнены, когда Activity подключается к Service. Мы можем заставить Activity реализовать интерфейс или создать отдельный анонимный класс:
* *//java*
* */\*\** Defines callbacks for service binding*,* passed to bindService*()* *\*/*
* */\*\** From Android documentation *\*/*
* **private** ServiceConnection mConnection = **new** ServiceConnection() {
* **public** void onServiceConnected(ComponentName className, IBinder service) {
* *// We've bound to LocalService, cast the IBinder and get LocalService instance*
* MyBinder binder = (MyBinder) service;
* mService = binder.getService();
* mBound = **true**;
* }
* **public** void onServiceDisconnected(ComponentName arg0) {
* mBound = **false**;
* }

};

*//kotlin*

*/\*\* Defines callbacks for service binding, passed to bindService() \*/*

*/\*\* From Android documentation \*/*

**private** **val** mConnection = **object** : ServiceConnection {

 **override** **fun** onServiceConnected(className: ComponentName, service: IBinder) {

 *// We've bound to LocalService, cast the IBinder and get LocalService instance*

 **val** binder = service **as** LocalService.LocalBinder

 mService = binder.getService()

 mBound = **true**

 }

 **override** **fun** onServiceDisconnected(arg0: ComponentName) {

 mBound = **false**

 }

}

Обратный onServiceConnected()вызов обрабатывает фактическое соединение, и это то, где мы получаем доступ к этому IBinder. Мы можем использовать это, IBinderчтобы получить объект Service для вызова методов, сохраняя его как переменную экземпляра, чтобы делать с ним что-то! Мы также будем отслеживать, была ли привязана Service, чтобы избежать любых nullошибок.

* Наконец, flagпараметр указывает некоторые параметры того, как должна быть связана служба (например, для указания приоритета выживания). Conetxt.BIND\_AUTO\_CREATEявляется хорошим значением по умолчанию, указывающим, что привязка должна создать объект службы, если это необходимо.

После привязки службы мы можем вызывать методы этой службы (которая должна существовать!), что позволяет нам поддерживать кнопку «пауза».

Мы также должны вызвать функцию unbindService(), когда активность останавливается, чтобы освободить ресурсы:

**protected** void onStop() {

 **if** (mBound) {

 unbindService(**this**);

 }

 **super**.onStop();

}

Помните, что привязка Службы **НЕ** вызывает onStartCommand(), а просто создает Службу и дает нам доступ к ней! Привязанные Службы по умолчанию не считаются «запущенными» и обычно сохраняются ОС до тех пор, пока они привязаны. Если мы «запускаем» Привязанную Службу (например, с помощью startService()), нам нужно будет не забыть остановить ее позже!

Этот пример предназначен только для *локальных служб* (доступ к которым осуществляется в рамках одного процесса — в рамках одного приложения). Если мы хотим, чтобы служба была доступна другим процессам (т. е. другим приложениям), нам нужно проделать больше работы. В частности, мы используем Handlerи Messengerобъекты для передачи сообщений между этими процессами (аналогично примеру, который мы сделали для передачи сообщений между потоками); для получения более подробной информации см. [пример в руководстве](https://developer.android.com/guide/components/bound-services.html#Messenger) , а также полные примеры классов [MessengerService](https://github.com/android/platform_development/blob/master/samples/ApiDemos/src/com/example/android/apis/app/MessengerService.java)и [MessengerServiceActivities](https://github.com/android/platform_development/blob/master/samples/ApiDemos/src/com/example/android/apis/app/MessengerServiceActivities.java).

1. [https://developer.android.com/guide/comComponents/services.html](https://developer.android.com/guide/components/services.html)[↩](https://info448.github.io/services.html#fnref40)
2. <https://developer.android.com/reference/android/os/AsyncTask.html>[↩](https://info448.github.io/services.html#fnref41)
3. [https://developer.android.com/guide/comComponents/processes-and-threads.html](https://developer.android.com/guide/components/processes-and-threads.html)[↩](https://info448.github.io/services.html#fnref42)
4. <https://developer.android.com/images/service_lifecycle.png>[↩](https://info448.github.io/services.html#fnref43)
5. [https://developer.android.com/guide/comComponents/services.html#ExtendingService](https://developer.android.com/guide/components/services.html#ExtendingService)[↩](https://info448.github.io/services.html#fnref44)