Сервисы

Сервис – это компонент приложения, который может выполнять длительные операции в фоновом режиме, даже некоторое время после того как пользователь переключился на другое приложение. Сервис не предоставляет пользовательский интерфейс.

Существует три вида сервисов:

- *Foreground*, сервис переднего плана – такой сервис выполняет операции, заметные для пользователя. Например, аудиопроигрыватель будет использовать сервис переднего плана для воспроизведения звуковой дорожки. Такие сервисы продолжают работать, даже если пользователь не взаимодействует с приложением. Сервис переднего плана обязан отображать уведомление, чтобы пользователи были в курсе, что сервис работает – это уведомление нельзя отклонить пока сервис не остановлен или не удален с переднего плана.
- *Background*, фоновый сервис – выполняет операции, которые пользователь непосредственно не замечает, например, оптимизацию базы данных. Система налагает ограничения на работу фоновых сервисов, если само приложение не находится на переднем плане – например, не даёт им доступ к местоположению пользователя.
- *Bound*, привязываемый сервис – компонент приложения привязывается к сервису и может взаимодействовать с ним, отправлять запросы и получать результаты. Привязанный сервис работает только до тех пор, пока к нему привязан другой компонент приложения.

Сервис может сочетать сразу несколько характеристик, например, быть фоновым, и одновременно привязываемым.

Сервис – это просто компонент для работы в фоновом режиме, и важно понимать когда следует использовать его, а когда поток или корутину. Если работа должна выполняться за пределами основного потока, но только пока пользователь взаимодействует с приложением – достаточно создать новый поток или корутину. Если работа должна продолжаться даже после закрытия приложения – разумно использовать сервис. Например, если требуется воспроизводить звуки в игре, то нужно использовать поток, ведь после закрытия игры звуки должны прекращаться. Напротив, аудиопроигрыватель может быть закрыт, но воспроизведение трека должно продолжаться – в этом случае используется сервис.

Следует помнить, что по умолчанию сервис работает в основном потоке приложения, поэтому для использования блокирующих операций внутри сервиса необходимо создать новый поток.

Создание сервиса

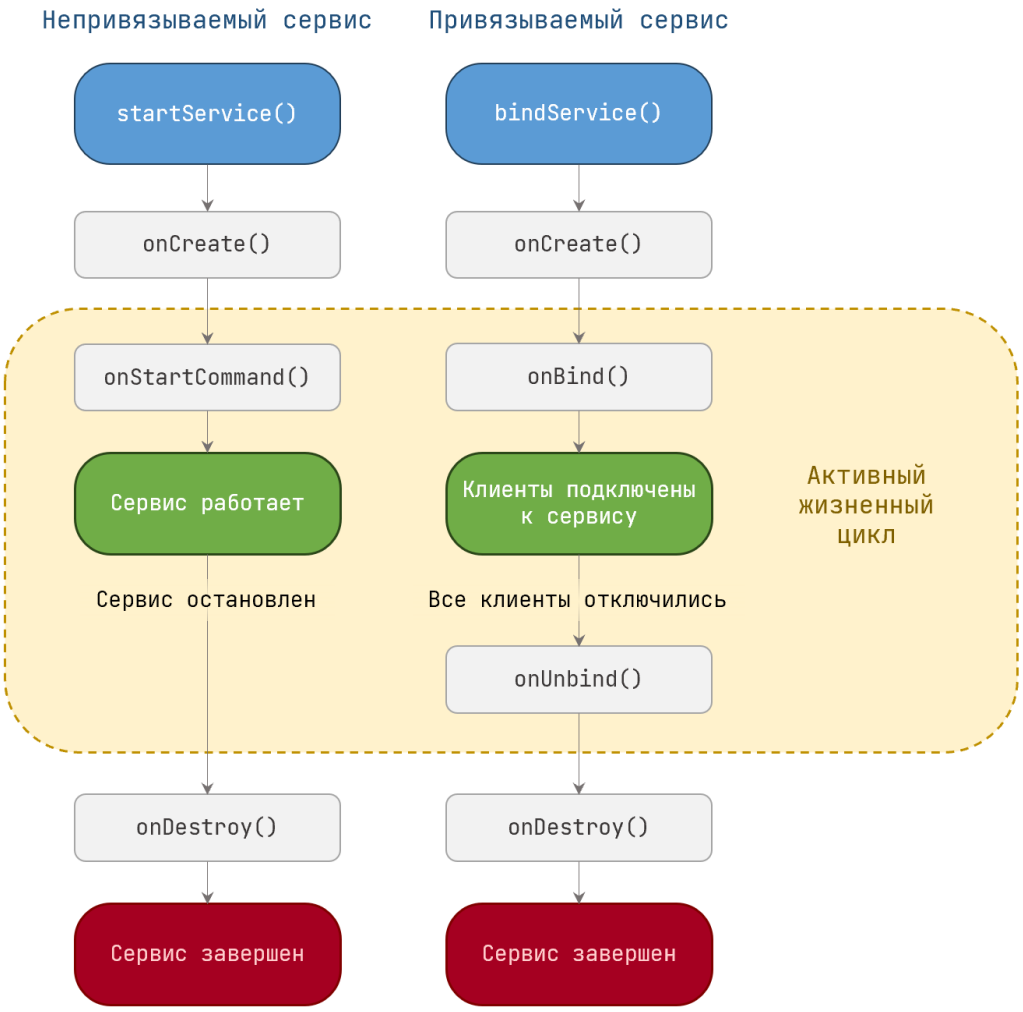
Для создания сервиса нужно сделать дочерний класс для класса `Service` или одного из его подклассов (например, `LifecycleService`, если используются объекты классов вроде `LiveData` и пр.), и переопределить в нём ряд методов:

- `onStartCommand()` Этот метод система вызывает при запуске сервиса с помощью `startService()`. Сервис запускается и может работать в фоновом режиме

неограниченное время, его нужно остановить, вызвав `stopSelf()` или `stopService()`. Если сервис будет только привязываемым, то этот метод не нужно реализовывать.

- `onBind()` Этот метод система вызывает при привязке к сервису с помощью `bindService()`. В этом методе нужно предоставить интерфейс `IBinder`, который клиенты будут использовать для связи с сервисом. Этот метод всегда должен переопределяться, но если сервис не будет привязываемым, то нужно вернуть `null`.
- `onCreate()` Этот метод вызывается однократно при первоначальном запуске сервиса, для выполнения инициализации, до вызовов `onStartCommand()` или `onBind()`. Если сервис уже запущен, этот метод не вызывается.
- `onDestroy()` Этот метод будет вызван если сервис больше не используется и будет уничтожен. Он должен быть реализован для очистки ресурсов, отключения приемников широковещательных сообщений и т.д.

Жизненный цикл сервиса выглядит следующим образом:



Android останавливает сервис только при нехватке памяти, если системные ресурсы требуются для той деятельности, которой сейчас занят пользователь. Если привязываемый сервис связан с активностью на экране – вероятность его закрытия невысока; у сервиса переднего плана шанс закрытия минимален. Однако если сервис запущен и работает продолжительное время, система со временем понижает его позицию в списке фоновых задач, и сервис становится уязвим к уничтожению, поэтому нужно спроектировать сервис так, чтобы он корректно обрабатывал перезапуск системой. Если система уничтожает сервис, она перезапускает его, как только появляются свободные ресурсы.

Сервис обязательно должен быть заявлен в манифесте, в атрибуте `name` нужно указать имя класса сервиса:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application ...>
        <service
            android:name=".ExampleService"
            android:exported="true"/>
        ...
    </application>
</manifest>
```

Если атрибут `exported` имеет значение `true`, то другие приложения на смартфоне также смогут запускать этот сервис.

Запускаемый сервис

Запускаемый сервис – это сервис, который другой компонент запускает путем вызова `startService()` или `startForegroundService()`, что приводит к вызову метода `onStartCommand()`. В сервис при этом можно передать объект `Intent` с нужными данными.

После запуска сервиса его жизненный цикл не зависит от компонента, который его запустил: сервис может работать даже если запустивший его компонент был уничтожен. Сервис должен либо остановиться сам после завершения своего задания, вызвав `stopSelf()`, или же он может быть остановлен извне с помощью `stopService()`.

Метод `onStartCommand()` должен возвращать целое число, которое описывает как система поступит с сервисом после вынужденного завершения его работы:

- `START_NOT_STICKY` – сервис не будет создан заново, если очередь намерений пуста.
- `START_STICKY` – сервис будет создан заново, но последнее намерение не будет передано в сервис ещё раз, только новые намерения из очереди.
- `START_REDELIVER_INTENT` – сервис будет создан заново, и ему будет повторно передано последнее намерение, а затем другие намерения из очереди.

Если поступает несколько запросов на запуск сервиса, то каждый раз вызывается метод `onStartCommand()`, который получает намерение из соответствующего запроса на запуск. А для остановки сервиса достаточно только одного запрос на остановку `stopSelf()` или `stopService()`. Чтобы случайно не остановить сервис, если во время обработки одного запроса поступили новые запросы, лучше вызывать метод `stopSelf()` с числовым параметром `startId`, который был получен в `onStartCommand()`. Если сервис выполнил работу и хочет остановить себя, а в это время поступил новый запрос с новым `startId`, то идентификаторы не совпадут и сервис не остановится.

Чтобы не тратить системные ресурсы и не расходовать заряд батареи, следует всегда останавливать сервисы после того как они выполнили свою работу!

Запускаемые сервисы переднего плана

Сервисы переднего плана рекомендуется использовать только если задача, выполняемая сервисом, должна быть заметна для пользователя, даже если он не взаимодействует с приложением напрямую, например:

- музыкальный проигрыватель, воспроизводящий музыку даже если пользователь находится в другом приложении: в уведомлении может отображаться воспроизводимый в данный момент трек;
- фитнес-приложение, которое записывает пробежку пользователя: в уведомлении может отображаться расстояние текущей тренировки или количество шагов.

Сервис переднего плана нужно не просто объявить в манифесте, но и указать какую работу он выполняет, иначе при его запуске будет выброшено исключение `MissingForegroundServiceTypeException`:

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PLAYBACK"/>
<application ...>
    <service
        android:name=".MyMediaPlaybackService"
        android:foregroundServiceType="mediaPlayback"
        android:exported="false">
    </service>
    ...
</application>
```

Помимо этого, соответствующую константу необходимо передать и в метод `startForeground`: (все константы начинаются с `FOREGROUND_SERVICE_TYPE_`, но в таблице ниже приведено только окончание названия, чтобы сэкономить место), а также объявить дополнительные разрешения в манифесте (все разрешения начинаются с `FOREGROUND_SERVICE_…`):

`foregroundServiceType` в манифесте	Разрешения в манифесте, константа для метода `startForeground`	Зачем используется
`camera`	`…_CAMERA`	Постоянный доступ к камере, например, для видеочатов
`connectedDevice`	`…_CONNECTED_DEVICE`	Взаимодействие с устройствами через Bluetooth, NFC, IR, USB или сеть
`dataSync`	`…_DATA_SYNC`	Передача или прием данных, резервное копирование, обработка данных из файла, …
`health`	`…_HEALTH`	Запись хода упражнений для фитнес-трекеров, …
`location`	`…_LOCATION`	Навигация, запись трека перемещений
`mediaPlayback`	`…_MEDIA_PLAYBACK`	Воспроизведение аудио или видео в фоновом режиме

`mediaProjection`	`…_MEDIA_PROJECTION`	Отображение контента на неосновном экране или внешнем устройстве
`microphone`	`…_MICROPHONE`	Постоянная запись голоса для диктофонов или коммуникационных приложений
`phoneCall`	`…_PHONE_CALL`	Исходящий звонок с использованием методов `ConnectionServce`
`remoteMessaging`	`…_REMOTE_MESSAGING`	Передача текста на другое устройство, в т. ч. если пользователь переключается на другое устройство
`shortService`	`…_SHORT_SERVICE`	Быстрое завершение работы, которая не может быть прервана или отложена. До 3 минут максимум, нет перезапуска, и ряд других ограничений
`specialUse`	`…_SPECIAL_USE`	Всё что не вписывается в другие типы. Требует подробностей в манифесте.

К некоторым типам сервисов могут применяться дополнительные требования, у других может быть альтернативный вариант без использования сервисов – больше подробностей можно посмотреть в документации.

Помимо разрешения `FOREGROUND_SERVICE` и разрешения из таблицы выше, необходимо также получить и разрешения, которые могут быть нужны для выполнения соответствующих действий. Например, если сервис должен работать с камерой, то список разрешений в манифесте должен быть таким:

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE "/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_CAMERA"/>
<uses-permission android:name="android.permission.CAMERA"/>
```

И если разрешения типа `FOREGROUND_SERVICE_…` система предоставляет автоматически, просто по факту что они заявлены в манифесте, то разрешения типа `CAMERA` могут потребовать явного запроса у пользователя. Работе с разрешениями посвящена отдельная тема данного курса.

При запуске сервиса переднего плана нужно сначала запустить сам сервис:

```
val intent = Intent(...)
context.startForegroundService(intent)
```

После этого сервис в методе `onStartCommand()` должен сообщить что он желает работать на переднем плане, для этого используется метод `ServiceCompat.startForeground()`. В этот метод передаётся ссылка на сам сервис, числовой идентификатор для уведомлений в статусной строке, сам объект-уведомление, а также константа типа сервиса из таблицы выше. Например, если в манифесте заявлены типы сервиса `FOREGROUND_SERVICE_TYPE_HEALTH`

и `FOREGROUND_SERVICE_TYPE_MEDIA_PLAYBACK`, то когда пользователь начинает пробежку – можно запустить сервис с типом `…_HEALTH`, а если он при этом включил опцию для прослушивания музыки, то можно сложить обе константы.

Сервис переднего плана может быть запущен только если приложение само находится на переднем плане. Если пользователь переключился на другое приложение, то запустить сервис переднего плана уже не получится (хотя уже запущенные сервисы будут работать как обычно).

Чтобы убрать сервис с переднего плана используется метод `stopForeground()`, сервис при этом продолжит работать. Если же остановить сервис, пока он работает на переднем плане, его уведомление будет удалено.

Привязываемые сервисы

Привязываемый сервис позволяет компонентам приложения привязываться к себе путем вызова `bindService()`. Обычно его нельзя запустить с помощью `startService()`.

Чтобы сделать сервис привязываемым нужно реализовать метод обратного вызова `onBind()`, который будет возвращать объект `IBinder` для связи с сервисом. Другие компоненты приложения будут вызывать метод `bindService()`, получать интерфейс, и с его помощью вызывать методы сервиса. Когда связь больше не нужна, клиент вызывает `unbindService()` и привязка уничтожается.

Пример сервиса:

```
class LocalService : Service() {

    private val binder = LocalBinder()

    public fun getRandomNumber() : Int {
       return Random.nextInt(100)
    }

    inner class LocalBinder : Binder() {
        fun getService(): LocalService = this@LocalService
    }

    override fun onBind(intent: Intent): IBinder {
        return binder
    }
}
```

Здесь определяется минимальный внутренний класс `LocalBinder`, который имеет один-единственный метод `getService()` для передачи ссылки на сервис в основное приложение. Метод `onBind()` получает намерение (и может извлечь из него какую-то информацию), и возвращает экземпляр класса `LocalBinder`. Также для примера добавлен метод `getRandomNumber()`, который возвращает случайное число – на его примере будет показано обращение из приложения к сервису.

Чтобы работать с этим сервисом из приложения, нужно создать *привязку* – связь с сервисом, для этого используется метод `bindService()`. Механизм привязки является

асинхронным и `bindService()` возвращается немедленно, не возвращая `IBinder`, поэтому нужно создать специальный объект класса `ServiceConnection` – туда будет поступать информация о том, что сервис привязан или отвязан. Можно реализовать его в виде синглтона:

```
val connection = object : ServiceConnection {

    override fun onServiceConnected(className: ComponentName, service: IBinder) {
        val binder = service as LocalService.LocalBinder
        svc = binder.getService()
        isBound = true
    }

    override fun onServiceDisconnected(className: ComponentName) {
        isBound = false
    }
}
```

Метод `onServiceConnected()` вызывается при подключении сервиса – именно он получает ссылку на `IBinder`. Ссылка на сервис сохраняется для последующего использования, и для удобства устанавливается переменная-флаг `isBound`, чтобы случайно не обратиться к отключенному сервису. Метод `onServiceDisconnected()` вызывается если соединение с сервисом неожиданно теряется, например, если сервис упал или был закрыт (но не если была вызвана функция отвязки).

Сам сервис создаётся таким образом:

```
val intent = Intent(this, LocalService::class.java)
bindService(intent, connection, BIND_AUTO_CREATE)
```

После этого приложение по необходимости может обращаться к сервису – например, вызвать метод `getRandomNumber()` из примера выше и отобразить это число на экране:

```
if (isBound) {
    val num: Int = svc.getRandomNumber()
    Toast.makeText(this, "Число: $num", Toast.LENGTH_SHORT).show()
}
```

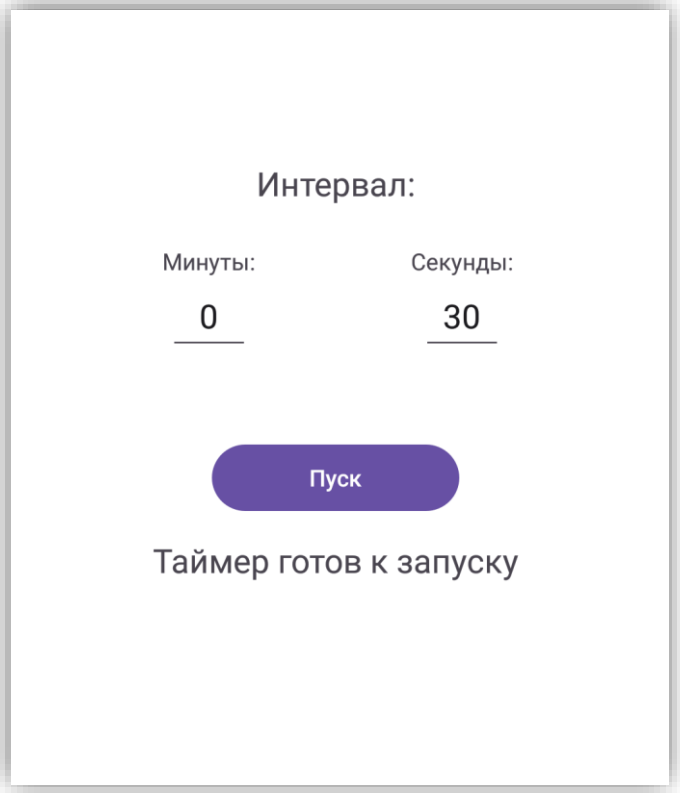
Когда сервис становится не нужен, от него можно отвязаться:

```
unbindService(connection)
```

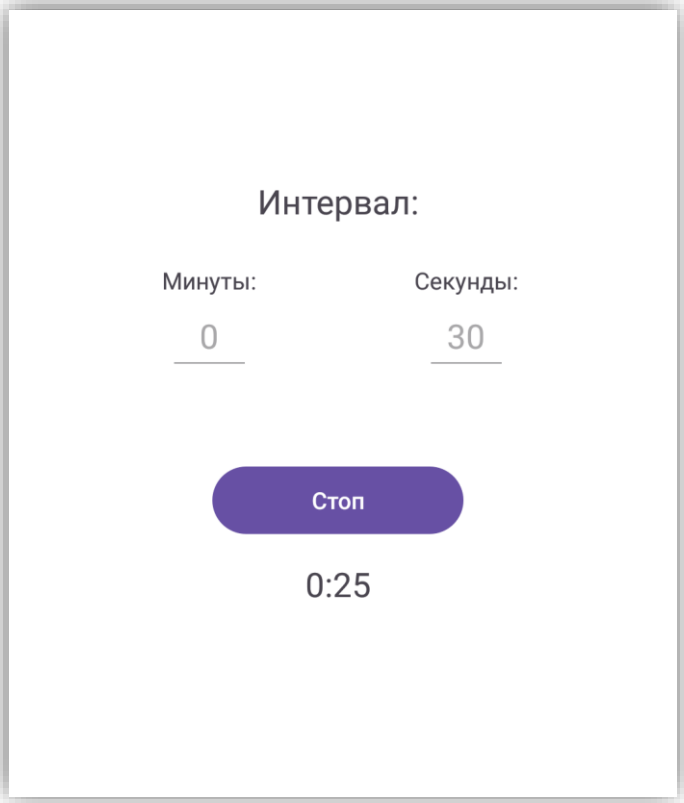
Привязываемый сервис существует только для обслуживания привязанных к нему компонентов приложения, а когда к сервису не привязаны никакие компоненты – система уничтожает его. Не требуется останавливать привязываемый сервис с помощью `stopSelf()` или `stopService()`.

Задание

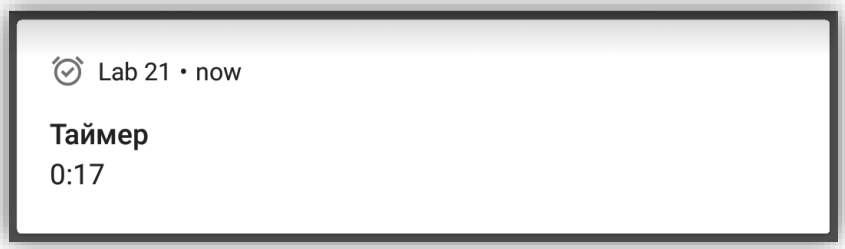
Напишите программу-таймер. Интерфейс программы должен выглядеть примерно так:



Пользователь может ввести числовые значения в поля минут и секунд. При нажатии кнопки «Пуск» начинается обратный отсчёт с тех значений, которые введены в поля минут и секунд, до нуля. На время отсчёта поля ввода блокируются, кнопка «Пуск» становится кнопкой «Стоп», а вместо надписи «Таймер готов к запуску» отображается оставшееся время:



Одновременно с этим отображается уведомление, в котором также идёт обратный отсчёт:



По окончании отсчёта программа возвращается в исходное состояние, готовая к новому запуску.

Если в процессе отсчёта пользователь закрывает программу, отсчёт должен продолжаться, уведомление должно оставаться активным (вместе с продолжающимся отсчётом), и при повторном запуске программы – отсчёт в окне программы также должен быть продолжен.

При реализации программы необходимо использовать сервис переднего плана, реализация отсчёта и уведомлений должны быть сделаны в сервисе.