**Практическая работа 11. AndroidDev на Kotlin. Activity Lifecycle. Интерфейс LifecycleObserver**

Как вы уже знаете из прошлых уроков, Активити — это, по сути, экран нашего приложения. Каждый пользователь андроид устройства запускает, сворачивает экраны приложения, запускает другие приложения, вертит устройство в руках. Что происходит при этом с активити? Как изменяются его состояния? И главный вопрос, как разработчик приложения должен это учитывать? Сейчас мы это выясним.

**Создаем приложение в Android Studio**

Давайте создадим приложение и на его примере будем изучать поведение активити.

Каждый раз при создании нового приложения по умолчанию создается класс главного экрана приложения — MainActivity.

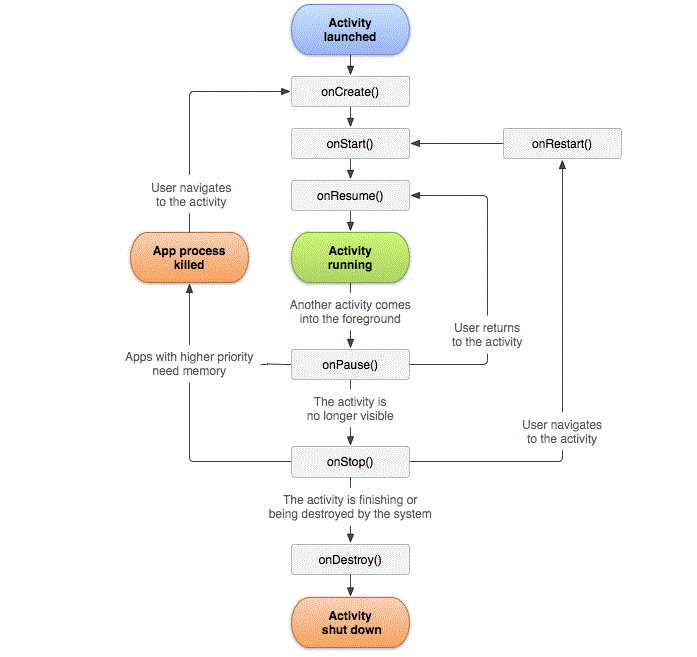
Когда пользователь нажимает ярлык приложения, система запускает активити, которое указано как стартовое в манифесте приложения. По умолчанию это MainActivity. После запуска мы видим активити на экране. Мы можем свернуть его или открыть другое приложение, экран настроек и т.д. При этом наше активити уходит на второй план и перестает быть видимым. Однако оно не уничтожается, а продолжает работать в фоне, и мы можем в любой момент к нему вернуться. Наконец, мы можем совсем закрыть приложение, или его может закрыть система, если ей будет не хватать оперативной памяти. Тогда активити прекратит свою работу и будет уничтожено.

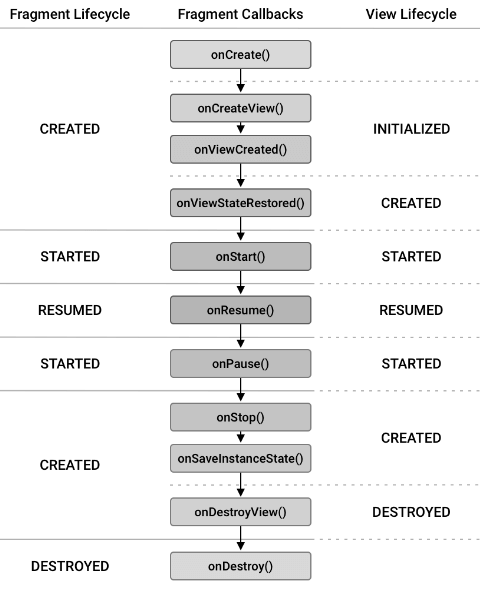
Кроме того, активити может пересоздаваться — уничтожаться и создаваться заново. Это происходит при событиях изменения конфигурации, таких как поворот устройства из портретной ориентации в альбомную (и обратно), смена размера экрана или языка интерфейса. О смене конфигурации мы более подробно поговорим на следующем уроке.

**Схема Lifecycle**

А сейчас посмотрите на схему состояний, через которое проходит активити в процессе работы, и событий, которые при этом происходят.

Когда активити запускается системой, оно инициализируется (INITIALIZED) и происходит событие ON\_CREATE. При этом активити переходит к состоянию “создано” (CREATED). В этот момент должен инициализироваться пользовательский интерфейс, поскольку активити готовится отобразиться пользователю. Далее происходит событие ON\_START и активити переходит к состоянию “запущено” (STARTED). Следующее событие ON\_RESUME. Активити переходит в состояние — RESUMED (возобновлено) — выходит на передний план, получает фокус и отображается пользователю. Если активити в процессе работы теряет фокус и частично перекрывается, например, диалоговым окном или другим активити, то переходит обратно в состояние STARTED. При этом происходит событие ON\_PAUSE. В этом состоянии активити приостанавливается, но может быть все еще видимым на экране, например, в многооконном режиме. Если же активити полностью перекрыто, то система его останавливает и переводит в состояние CREATED. Выполняется событие ON\_STOP. Активити пока не уничтожается системой и может в любой момент возобновить работу. Но поскольку оно не видно пользователю, то в этом состоянии целесообразно отключать некоторые функции, например, воспроизведение анимации. Если пользователь закрыл активити или система испытывает недостаток памяти, или изменилась конфигурация устройства (произошел поворот), активити уничтожается системой. При этом происходит событие ON\_DESTROY. В этот момент необходимо освобождать ресурсы, используемые активити.



****

**Функции обратного вызова жизненного цикла активити**

На рассмотренной нами схеме указаны события жизненного цикла, всего их шесть: ON\_CREATE, ON\_START, ON\_RESUME, ON\_PAUSE, ON\_STOP и ON\_DESTROY..

Для обработки событий класс [Activity](https://developer.android.com/reference/android/app/Activity" \t "_blank)имеет колбеки — функции обратного вызова, соответствующие каждому событию жизненного цикла. Например, это уже известная вам функция onCreate(savedInstanceState: Bundle?) {…} в которой происходит инициализация пользовательского интерфейса и вызовы некоторых других функций.

Но кроме одноименных функций жизненного цикла, в активити есть несколько дополнительных, таких как onPostResume() и onRestart(). Мы их будем использовать в качестве вспомогательных для большей наглядности.

Во всех функциях обратного вызова нужно переопределять одноименные методы суперкласса во избежание ошибок.

Функции обратного вызова полезны для изучения жизненного цикла. До недавнего времени они массово использовались разработчиками для управления поведением приложения. Однако сейчас Гугл рекомендует другой, более правильный подход, который мы рассмотрим далее в этом уроке. А пока переопределим эти функции в активити.

**onCreate()**

Первая функция, которая переопределена по умолчанию в каждом активити, отвечает событию ON\_CREATE. Выполняется перед стартом активити, когда активити переходит из состояния INITIALIZED к состоянию CREATED.

В теле функции onCreate должна выполняться базовая логика приложения, например связывание данных со списками, инициализация переменных и т.д.



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onCreate(savedInstanceState: Bundle?) {          super.onCreate(savedInstanceState)          setContentView(R.layout.activity\_main)  } |

В качестве параметра функция onCreate получает объект типа Bundle, который представляет собой набор пар “ключ — значение” и может быть использован для сохранения предыдущего состояния активити. Если активити запускается впервые, объект имеет значение null.

Переопределим остальные функции для каждого события жизненного цикла.

Но для начала создадим функцию toastMeState, которая будет отображать тост с информацией о том, какое событие произошло и какое состояние активити получило.



|  |  |
| --- | --- |
| 1  2  3 | private fun toastMeState(message: String) {          Toast.makeText(this, "${lifecycle.currentState}, $message", Toast.LENGTH\_LONG).show()      } |

Текущее состояние активити можно получить с помощью свойства объекта *lifecycle*.*currentState*, мы еще вернемся к нему и рассмотрим подробнее.

Длительность отображения тостов будет одинаковой и не сможет отражать реальную длительность событий и состояний активити. Тем не менее мы получим наглядную информацию об их очередности.

**onStart()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onStart() {          super.onStart()          toastMeState("ON\_START")      } |

Этот колбек вызывается, когда активити стартует и готовится выйти на передний план, чтобы стать видимым пользователю. Событие ON\_START  проходит очень быстро, и активити не задерживается долго в состоянии STARTED, а сразу переходит к состоянию RESUMED после вызова функции onResume().

Функцию toastMeState() мы будем вызывать в каждом колбеке, передавая туда имя функции обратного вызова.

**onResume()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onResume() {          super.onResume()          toastMeState("ON\_RESUME")      } |

Вызывается при событии ON\_RESUME, когда активити становится видимым пользователю и переходит в состояние RESUMED. Здесь можно стартовать анимацию, получать доступ к камере и отображать превью и т.п. Имейте в виду, что состояние RESUMED не гарантирует на 100%, что ваша активность видна пользователю — системное окно, например клавиатура, может перекрывать экран. Используйте метод onWindowFocusChanged (boolean), чтобы точно знать, что ваша активность видна пользователю.

**onPostResume()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onPostResume() {          super.onPostResume()          toastMeState("onPostResume")      } |

Вызывается после onResume() когда процесс возобновления активити завершился. Обычно эта функция не переопределяется в приложениях, а используется системой для запуска кода, зависящего от отображения активити пользователю. Мы будем использовать этот колбек для демонстрации текущего состояния активити — RESUMED.

**onPause()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onPause() {          super.onPause()          toastMeState("ON\_PAUSE")      } |

Этот колбек вызывается при событии ON\_PAUSE когда активити уходит с переднего плана и переходит из состояния RESUMED в состояние STARTED. При этом активити может оставаться видимым, например, в многооконном режиме. При этом событии можно прекращать операции, которые должны выполняться только когда активити на переднем плане, например, отображение превью с камеры. Однако не стоит использовать событие ON\_PAUSE для операций сохранения данных по причине его скоротечности.

**onStop()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onStop() {          super.onStop()          toastMeState("ON\_STOP")      } |

Событие ON\_STOP происходит, когда активити становится невидимым пользователю и возвращается в состояние CREATED. Это может произойти когда его полностью перекрывает вновь открытое активити. При этом событии можно освобождать те ресурсы, которые не требуются, пока активити остается невидимым пользователю, а также выполнять ресурсоемкие операции сохранения данных в БД и т.п.

**onRestart()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onRestart() {          super.onRestart()          toastMeState("onRestart")      } |

Этот колбек вызывается после события ON\_STOP  если активити собирается вернуться на передний план и будет отображено пользователю.  После вызова этого колбэка выполняются события ON\_START и ON\_RESUME. Если возврата активити не происходит, то вместо вызова onRestart() может произойти следующее событие — ON\_DESTROY.

**onDestroy()**



|  |  |
| --- | --- |
| 1  2  3  4 | override fun onDestroy() {          super.onDestroy()          toastMeState("ON\_DESTROY")      } |

Вызывается непосредственно перед уничтожением активити системой в связи с нехваткой памяти, нажатием кнопки “Назад”, сменой конфигурации устройства (поворот, смена размера экрана, локали и т.п), или вызова функции finish(). Не используйте это событие для выполнения длительных операций сохранения данных. Здесь можно отписаться от подписок или освободить ресурсы, которые не были освобождены в более ранних событиях жизненного цикла.

## Наглядное отображение событий жизненного цикла

Запустите приложение в эмуляторе или на смартфоне и посмотрите, как будут появляться тосты с информацией о состояниях активити и событиях.

Когда активити запускается системой, оно инициализируется (INITIALIZED) и происходит событие ON\_CREATE. При этом активити переходит к состоянию “создано” (CREATED). Далее происходит событие ON\_START и активити переходит к состоянию “запущено” (STARTED). Следующее событие ON\_RESUME. Активити переходит в состояние — RESUMED (возобновлено) — выходит на передний план, получает фокус и отображается пользователю. Если активити в процессе работы теряет фокус и частично перекрывается, например, диалоговым окном или другим активити, то переходит обратно в состояние STARTED. При этом происходит событие ON\_PAUSE. В этом состоянии активити приостанавливается, но может быть все еще видимым на экране, например, в многооконном режиме. Если же активити полностью перекрыто, то система его останавливает и переводит в состояние CREATED. Выполняется событие ON\_STOP. Активити пока не уничтожается системой и может в любой момент возобновить работу. При возврате к нему выполняется колбэк onRestart() и затем происходят события ON\_START и ON\_RESUME с возобновлением соответствующих состояний.

Если пользователь закрыл активити или система испытывает недостаток памяти, или изменилась конфигурация устройства (произошел поворот), активити уничтожается системой. При этом происходит событие ON\_DESTROY.

Также активити уничтожается и создается заново при повороте устройства, или переключении в многооконный режим. При этом активити обычно проходит поочередно через все состояния и события. Пересоздание активити также происходит при смене локали на устройстве. Пересоздание активити связано с необходимостью изменения конфигурации и загрузки новых ресурсов. На следующем уроке мы разберемся, как сохранять состояние активити при смене конфигурации.

Имейте в виду, что могут быть ситуации, когда активити может быть уничтожено без вызова некоторых колбеков. Например, если в теле функции onCreate() прописать вызов функции finish(), то после onCreate() сразу будет вызван onDestroy(). Однако и onDestroy() может не быть вызван в некоторых случаях при уничтожении активити системой.

## Почему так не нужно делать?

Почему же Гугл предостерегает от злоупотребления использованием функций обратного вызова жизненного цикла активити?

Если мы посмотрим на код MainActivity, то увидим, что количество строк в нем уже достигает сотни. А ведь мы еще ничего особо не писали в колбэках. В реальных приложениях эти колбэки могут стать местом нагромождения кода, что приведет к ухудшению читабельности, поддерживаемости и тестируемости кода. В маленьких приложениях это не кажется таким уж страшным, но в реальных больших проектах это становится серьезной проблемой.

## Интерфейс LifecycleObserver

## !!! используется только на версии под 26API мы его не делаем

Не так давно Гугл предоставил сообществу разработчиков так называемый [Android Jetpack](https://developer.android.com/jetpack/" \t "_blank). Это набор современных инструментов и рекомендаций для построения эффективных приложений с учетом актуальных требований индустрии, таких как чистая архитектура, модульность,  тестируемый код и т.п.

Плохим примером является приложение, весь функционал которого сосредоточен в активити. Чистая архитектура, которая становится стандартом мобильной разработки, предполагает модульное приложение,  разделенное на компоненты, каждый из которых отвечает за определенный функционал. В то же время компоненты, работающие с активити, должны реагировать на события жизненного цикла и обрабатывать их.

Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют интерфейс [LifecycleOwner](https://developer.android.com/reference/android/arch/lifecycle/LifecycleOwner" \t "_blank). Именно этот интерфейс и добавляет им метод getLifecycle, к которому мы обращались через вызов lifecycle.currentState

Метод getLifecycle() возвращает объект Lifecycle. На этот объект можно подписать слушателей, которые будут получать уведомления при смене lifecycle-состояния Activity.

Для этой цели служит интерфейс [LifecycleObserver](https://developer.android.com/reference/android/arch/lifecycle/LifecycleObserver" \t "_blank), который мы реализуем в новом классе MyObserver. Интерфейс LifecycleObserver не имеет методов. Мы можем создать любую функцию и привязать ее к жизненному циклу активити. Для привязки служит аннотация @OnLifecycleEvent, которая принимает объект Lifecycle с перечислением Event, содержащим константы для каждого события жизненного цикла.

Функция, помеченная такой аннотацией, будет вызываться при соответствующем событии жизненного цикла активити.

Также есть константа ON\_ANY для реагирования на все события жизненного цикла.

Такая функция будет выводить в лог все события жизненного цикла активити:



|  |  |
| --- | --- |
| 1  2  3  4 | @OnLifecycleEvent(Lifecycle.Event.ON\_ANY)      fun onAny(source: LifecycleOwner, event: Lifecycle.Event) {          Log.d("TAG", event.toString())      } |

Чтобы подписать этот класс на события жизненного цикла, в MainActivity объявляем переменную класса MyObserver с поздней инициализацией:



|  |  |
| --- | --- |
| 1 | lateinit var myObserver: MyObserver |

В теле функции onCreate() инициализируем переменную и подписываем на объект lifecycle:



|  |  |
| --- | --- |
| 1  2 | myObserver = MyObserver()  lifecycle.addObserver(myObserver) |

Теперь запускаем приложение и убедимся, что в логах у нас отображаются все события жизненного цикла активити при запуске, работе и остановке приложения.