ТЕХНОПОЛИС

Лекция 4

Асинхронная работа

Грицук Александр

Введение

Процессы в Android

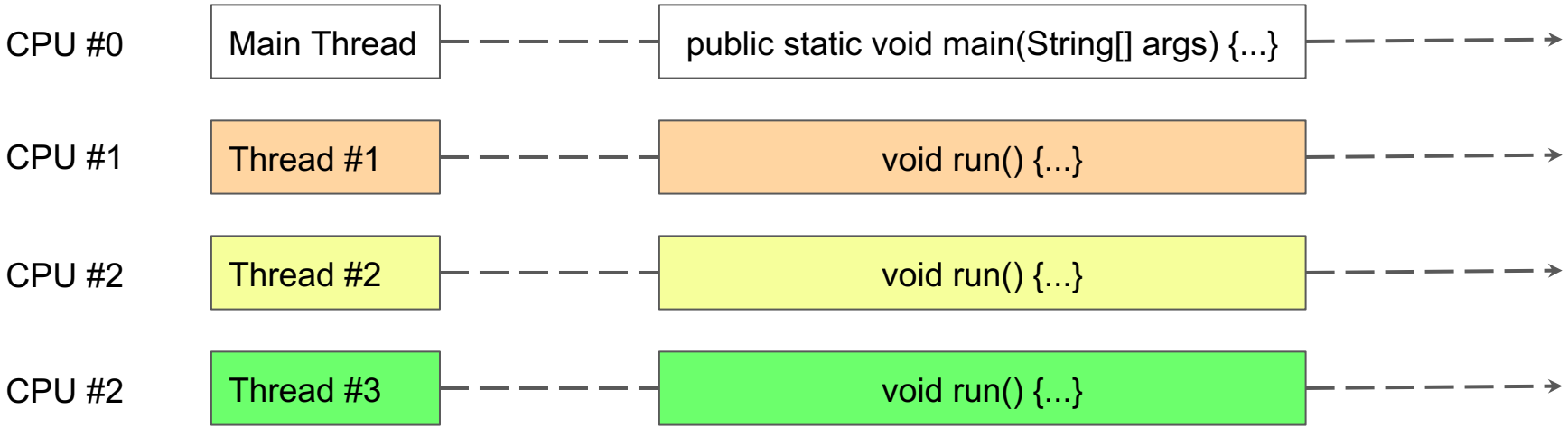
Multi-threading

Concurrency

https://developer.android.com/guide/components/processes-and-threads.html

- **Приложения** в Android являются набором данных и исполняемого кода, который упакован в файл
- Обычно одному **приложению** соответствует один **процесс**
- Внутри **процесса** исполняется, как минимум, один **главный** (**main**) **поток** (**Thread**)
- Каждому **процессу** выделяется **общая** (**для всех потоков процесса**) память
- Один процесс не может напрямую обращаться к памяти другого процесса
- **Поток** - последовательно исполняемый набор команд

- **Потоки** могут выполняться **параллельно**
- **Потоки** могут выполняться на разных или на одном **CPU**
- Количество потоков не ограничено (в пределах доступной памяти)



Потоки в Java

- Каждому потоку соответствует объект **Thread**
- Текущий поток: **Thread.currentThread()**
- Запуск нового потока: **new Thread.start()**
- Интерфейс **Runnable** для определения кода, который будет выполняться потоком:

```
public interface Runnable {
    // Этот метод будет выполнен в отдельном потоке
    public void run();
}
```

- Стартуем поток:

```
public final class ThreadTask implements Runnable {
    @Override
    public void run() {
        // … do something
    }
}

final ThreadTask task = new ThreadTask()

new Thread(task).start();
```

- Стартуем поток, используя анонимный **Runnable**:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        // do something
    }
}).start();
```

- Стартуем поток, **Java 8** (**Android 7.0**):

```
new Thread(() -> { /* do something */ }).start();
new Thread(SomeClass::doSomething).start();
```

Потоки в Android

Основной (UI, main) поток

- При создании **процесса** приложения, создается **основной** (**UI**, **main**) поток
- Главные задачи, возлагаемые на **основной** поток:
  - отображение графического интерфейса приложения - **UI**
  - обработка пользовательского ввода (клавиатура и т.д.)
- Жизненный цикл Android-компонент приложения ‘привязан’ к основному потоку, например: **Activity.onCreate/onDestroy**

- Основной поток содержит бесконечный цикл (**Looper**) выборки сообщений (**MessageQueue**)
- Для работы с сообщениями используется **Handler**:

```
public final class MyHandler extends Handler {
    public MyHandler(final Looper looper) {
        super(looper);
    }

    @Override
    public void handleMessage(final Message msg) {
        // …
    }
}
```

- Отправляем сообщение (**Message**), которое будет обработано в основном потоке:

```
final static int MY_MESSAGE = 0;

final MyHandler handler = new MyHandler(Looper.getMainLooper());
handler.obtainMessage(MY_MESSAGE).sendToTarget();
```

- ‘Отправляем’ **Runnable**, который выполнится в основном потоке (**Java 8**):

```
handler.post(() -> { /* do something */ });
```

- **MessageQueue** содержит сообщения - объекты типа **Message**
- Каждое сообщение содержит **payload** - данные или объект типа **Runnable**
- **Handler** добавляет сообщения в **MessageQueue** через **Looper**
- **Looper**, посредством метода **Looper.loop()**, ‘заставляет’ поток выполняться бесконечно долго. Кроме того, **Looper** отвечает за доставку сообщений **Handler**-у или выполнение **Runnable**
- Вызов метода **Looper.quit()** приводит к выходу из цикла выборки сообщений и завершению потока

- Все методы в основном потоке **должны** работать быстро (не более **~16 мс**)
- Если какой-то из методов (‘на’ **UI**-ом потоке) будет выполняться дольше **16 мс**, скорее всего будут видны ‘артефакты’ в отрисовке графического интерфейса приложения или задержки при обработке пользовательского ввода - приложение будет “тормозить”

- Что нельзя делать в основном потоке:
  - выполнять операции с файлами
  - выполнять сетевые запросы
  - выполнять операции с базами данных (основа - файлы)
  - выполнять любые длительные операции, например: сложные математические расчёты, декодирование графических изображений
  - выполнять вызовы ‘ожидания’, например: **Thread.sleep()**, **Object.wait()** и т.д.

- В силу изложенного выше видно, что основного (**UI**) потока, в общем случае, не достаточно для решения задач

Потоки в Android

Фоновые потоки

Способы выполнения задач в фоновых потоках на Android:

- **Executor** – как в обычной Java
- **AsyncTask** – работает на Executor, удобные методы передачи результата в main поток
- **Loader** – решает проблему жизненного цикла, основное средство загрузки данных в фоне
- **AsyncTask** и **Loader** используют ‘под капотом’ **Handler, Looper, HandlerThread, Message**.

Переиспользование потоков

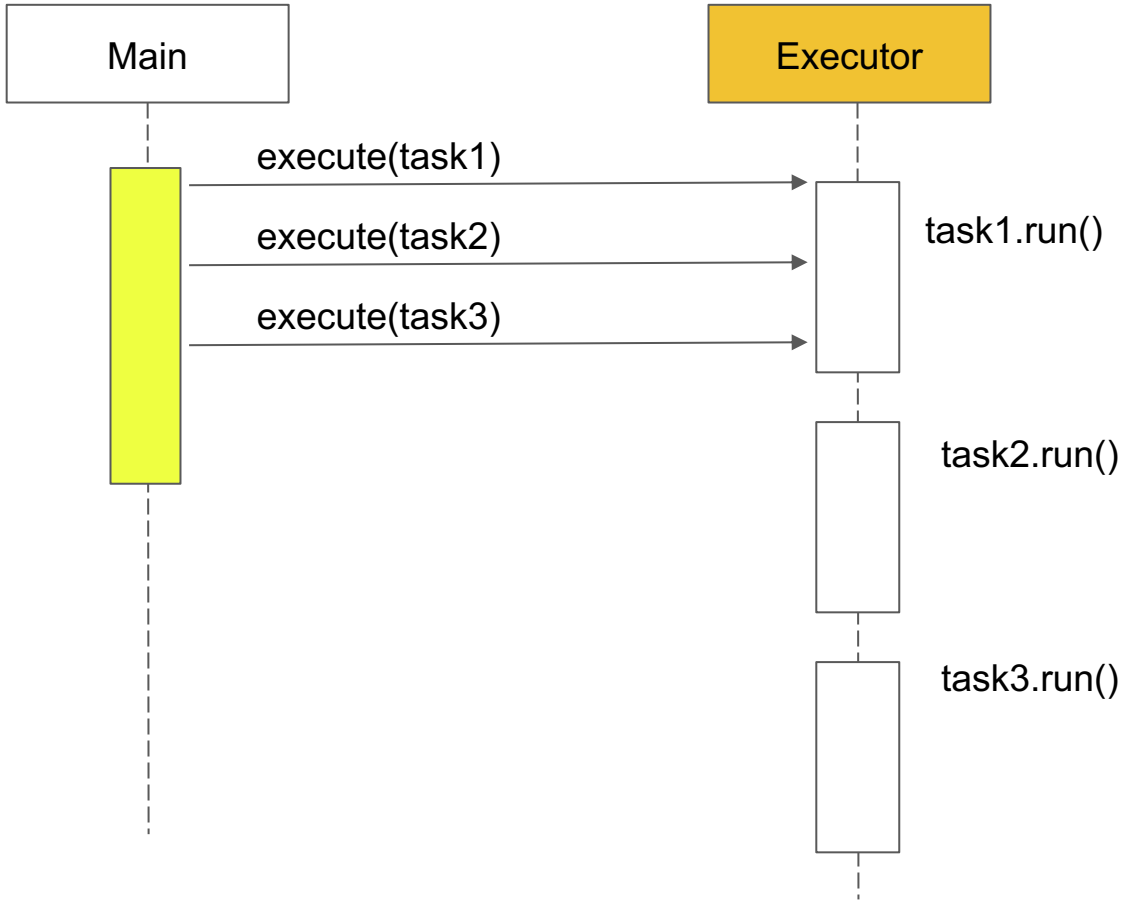
- Создание нового потока – дорогая операция (На Android выделяется 1Мб памяти для стэка)
- Вместо создания нового потока для каждой новой задачи надо использовать Executor

```
~~Thread~~(task).~~start~~();
// Получаем и сохраняем где-нибудь объект executor-а
Executor executor = ...
// Используем его для запуска всех задач
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2()); ...
```

Single Thread Executor

```
Executor executor = Executors.newSingleThreadExecutor();
```

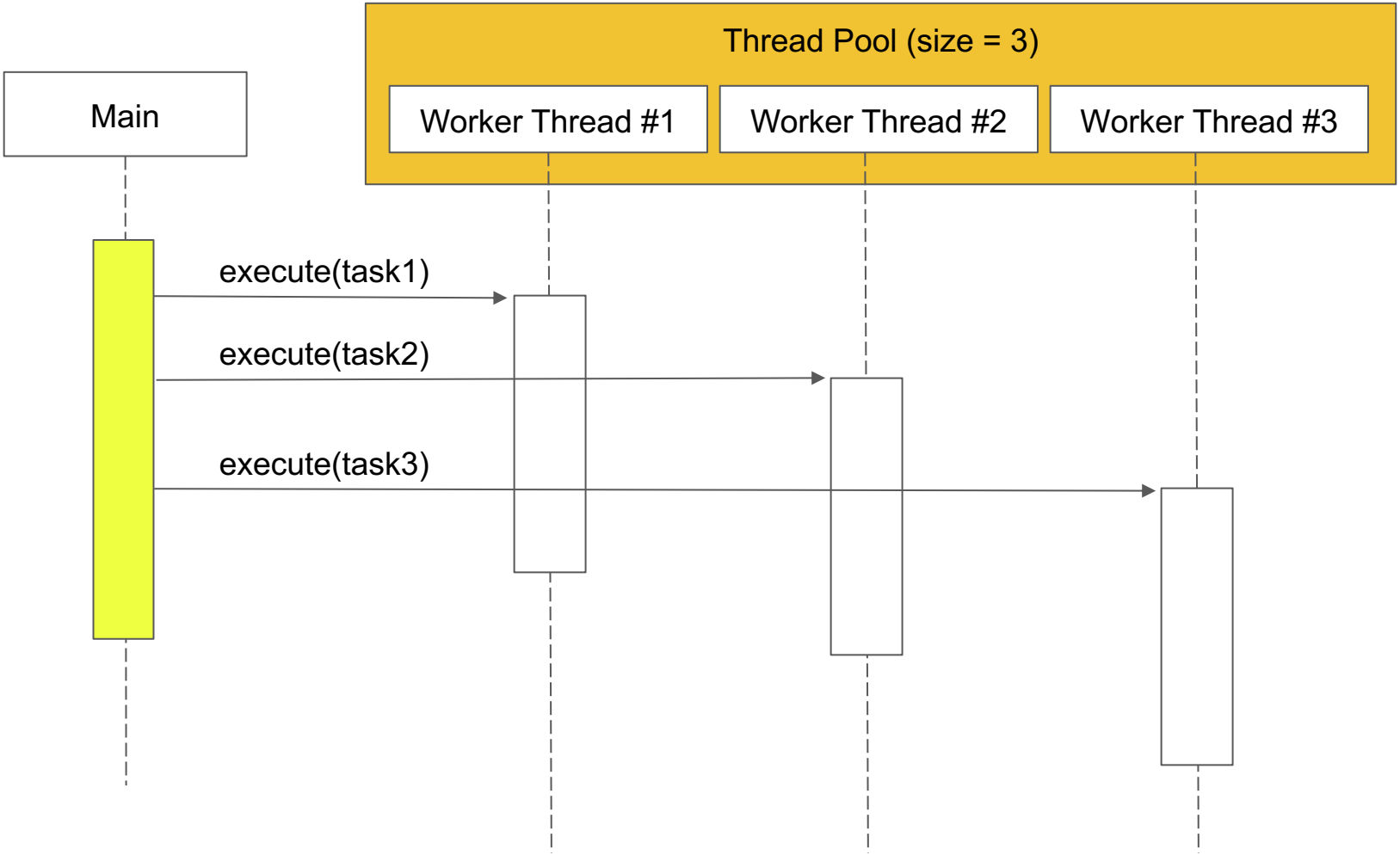
- Выполняет задачи по очереди в одном потоке, одна за другой.
- Одновременно выполняется только одна задача.
- Задачи выполняются в порядке поступления


Main
Executor
execute(task1)
execute(task2)
execute(task3)
task1.run()
task2.run()
task3.run()

Thread Pool

```
Executor executor = Executors.newFixedThreadPool(N);
```

- Использует не более N потоков
- Выполняет до N задач одновременно
- Порядок выполнения не гарантирован


Thread Pool (size = 3)
Main
Worker Thread #1
Worker Thread #2
Worker Thread #3
execute(task1)
execute(task2)
execute(task3)

Что нужно знать ещё

- Конкурентный доступ к памяти
- Атомарные операции
- Синхронизация
- Deadlock
- Потокобезопасность
- Классика: **Java Concurrency In Practice**, *Brian Goetz*
- Java Concurrency: https://docs.oracle.com/javase/tutorial/essential/ concurrency/index.html

Сервис (IntentService)

- **Service** это компонент приложения, который выполняется в фоне и, непосредственно, не связан с пользовательским интерфейсом приложения (**UI**) и его жизненным циклом
- **Service** предназначен для выполнения потенциально длительных операций
- **Service** имеет более высокий приоритет, чем ‘не активный или не видимый’ пользовательский интерфейс (**Activity**)

- Вы можете указать системе, что ваш сервис очень важен для пользователя и его нельзя завершать при нехватке памяти:

```
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

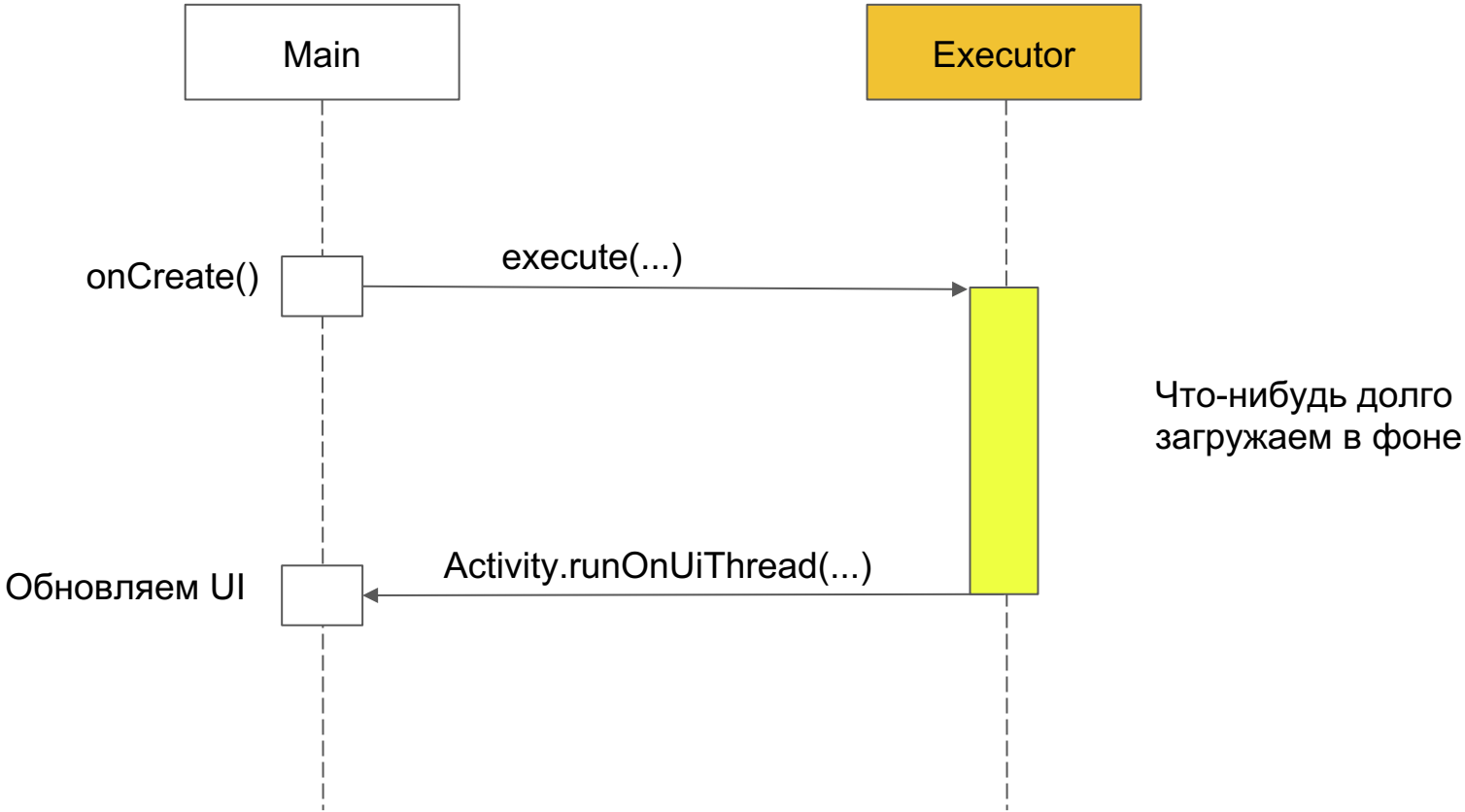
- **IntentService** - это сервис, который обрабатывает ‘входящие’ запросы последовательно, ‘на одном’ и том же потоке (**HandlerThread**)
- **IntentService** завершается, после выполнения всех запросов (**HandlerThread.quit()**)

https://developer.android.com/reference/android/app/IntentService

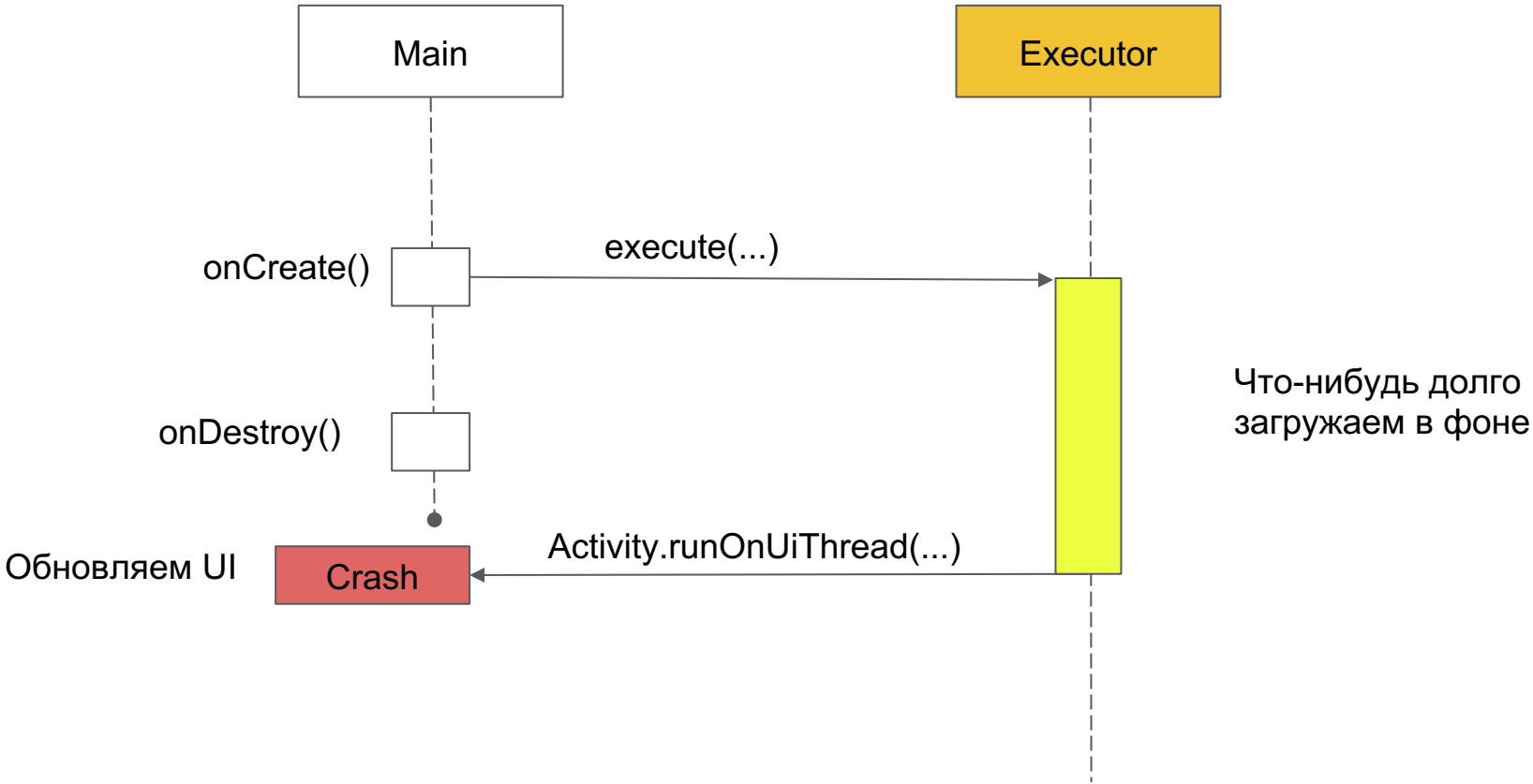
Executor

Можно просто кинуть задачу на выполнение в **Executor** и забыть... Но что если хотим получить результат обратно в main потоке?

```
final Activity activity = this;
final ImageView imageView = findViewById(R.id.image_view);
executor.execute(() -> {
    Bitmap bitmap = ... // Загружаем картинку из сети
        activity.runOnUiThread(() -> {
            imageView.setImageBitmap(bitmap);
        });
    });
```


Main
Executor
onCreate()
execute(...)
Что-нибудь долго
загружаем в фоне
Обновляем UI
Activity.runOnUiThread(...)

Проблема жизненного цикла



Проблема жизненного цикла

В callback методах, которые «приходят» из фоновых потоков, всегда проверять, жив ли еще UI?

```
executor.execute(() -> {
    Bitmap bitmap = ... // Загружаем картинку из сети
        activity.runOnUiThread(() -> {
            if (...) { // Проверяем, что UI ещё жив
                imageView.setImageBitmap(bitmap);
            }
        });
});
```

Проверяем, что UI ещё «жив»:

- !Activity.isFinishing()
- View.isAttachedToWindow()
- Fragment.getActivity() != null

Потоки в Android

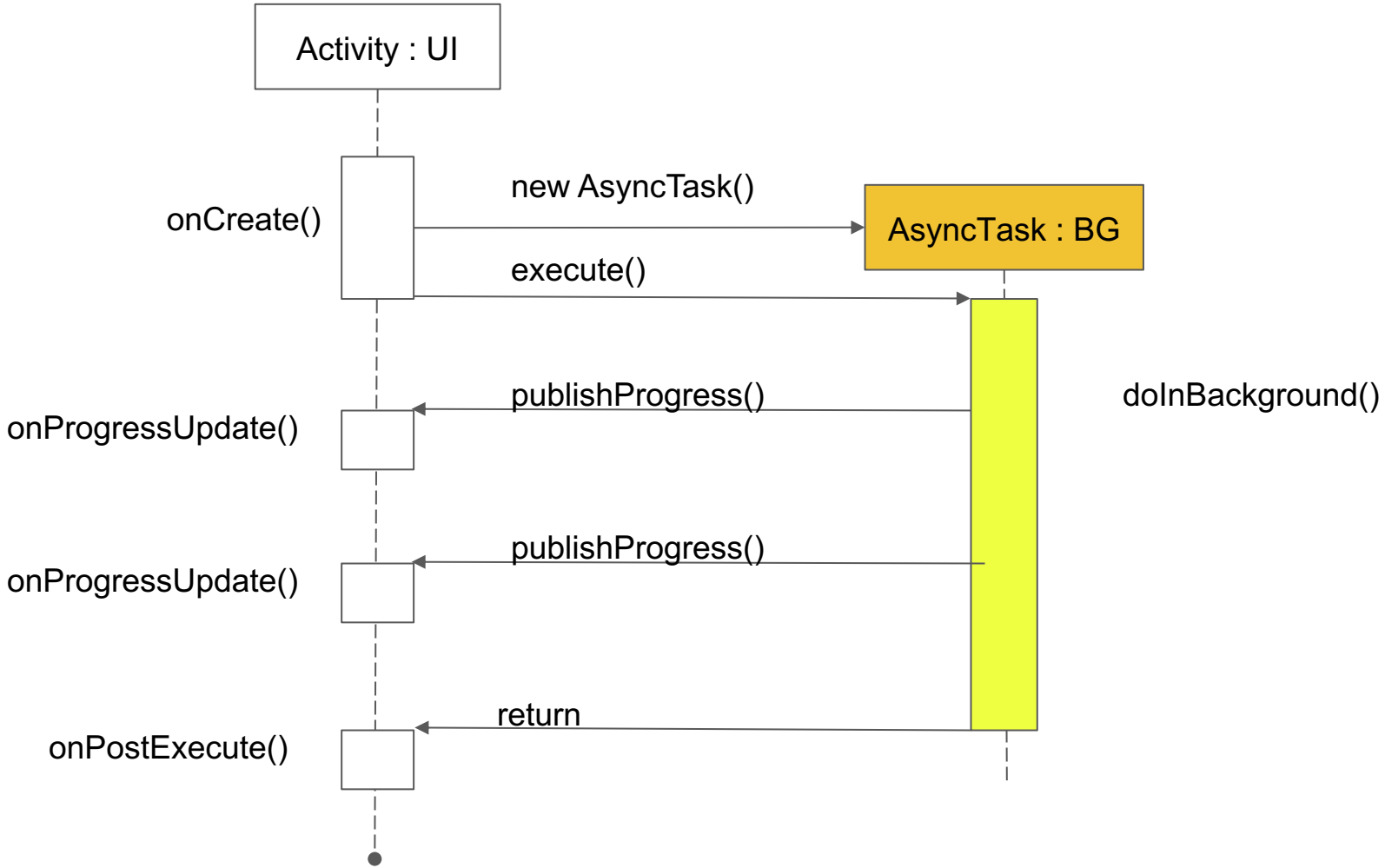
AsyncTask – выполнение задачи в фоновом потоке и передача результата в UI поток

Сценарий: скачивание файла

- При старте приложение начинает скачивание изображения из сети
- Пока идет скачивание, показывает индикатор прогресса
- После завершения скачивания приложение показывает изображение на экране
- Скачивание выполняется в фоновом потоке при помощи **android.os.AsyncTask**

AsyncTask

- doInBackground(Param... params)
  выполняется в фоновом потоке
- execute(Params... Params) запускает задачу из UI потока
- onPostExecute(Result result)
  выполняется в UI потоке после завершения
- publishProgress(Progress progress)
  вызывается из кода doInBackground
- onProgressUpdate(Progress... values)
  выполняется в UI потоке


Activity : UI
onCreate()
new AsyncTask()
AsyncTask : BG
execute()
publishProgress()
doInBackground()
onProgressUpdate()
publishProgress()
onProgressUpdate()
return
onPostExecute()

```
class GetImageTask extends AsyncTask<Void, Void, Bitmap> {
    @Override
    protected Bitmap doInBackground(Void... ignore) {
        // Этот метод выполняется в фоновом потоке
        try {
            return downloadImage(downloadUrl);
        } catch (Exception e) {
            Log.e(TAG, "Error downloading file: " + e, e);
            return null;
        }
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {
    }
}
```

```
public class LoadImageActivity extends Activity {
    private ProgressBar progressBarView;
    private ImageView imageView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_load_image);

        progressBarView.setVisibility(View.VISIBLE);new GetImageTask().execute();

        new GetImageTask().execute();
    }
}
```

AsyncTask: отображение прогресса

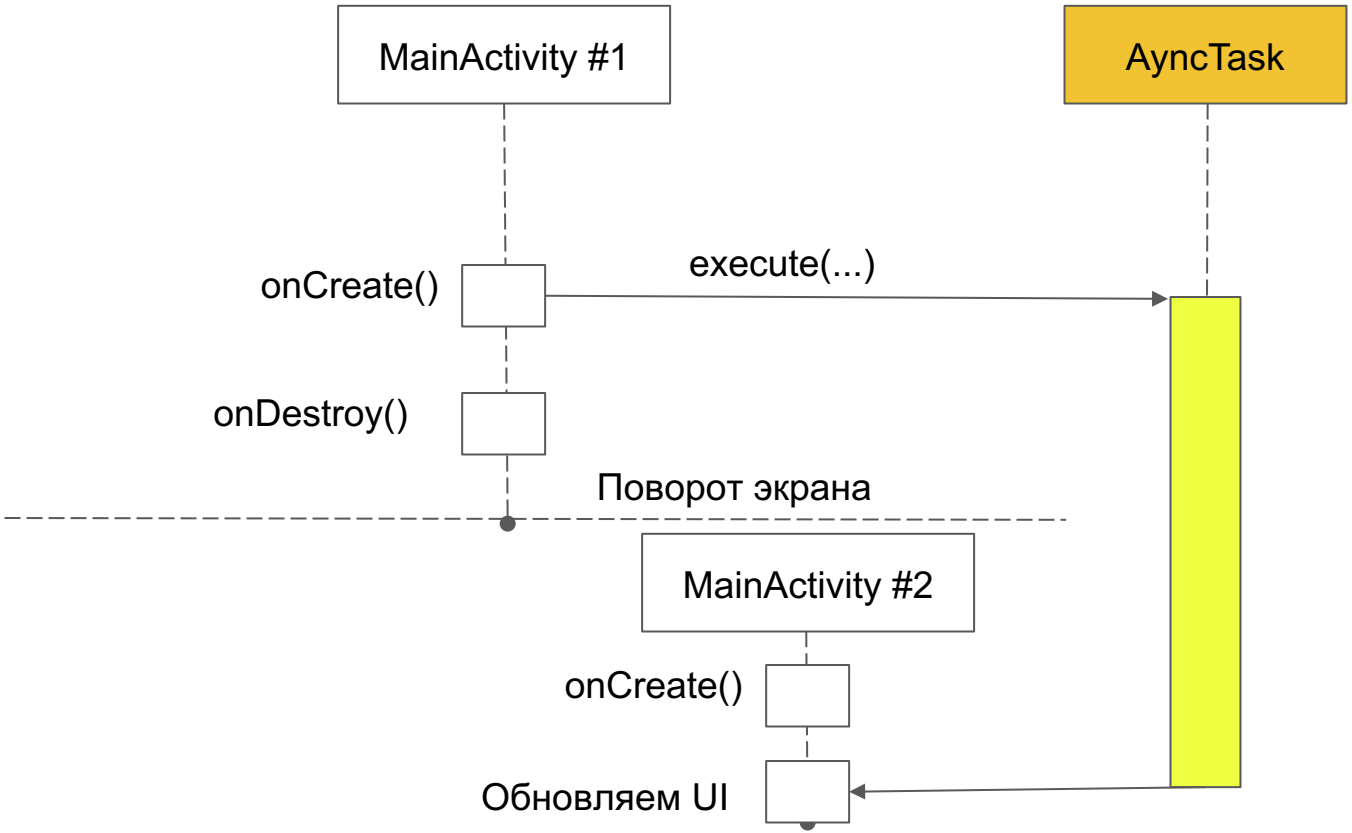
```
/**
 * Callback интерфейс для получения уведомления о прогрессе.*/
public interface ProgressCallback {
    /**
     * Вызывается при изменении значения прогресса.
     * @param progress новое значение прогресса от 0 до 100.*/
    void onProgressChanged(int progress);
}
```

```
class GetImageTask extends AsyncTask<Void, Integer, Bitmap>
    implements ProgressCallback {
    protected Bitmap doInBackground(Void... ignore) {
        return downloadImage(url, this /*progressCallback*/);
    }

    // Метод ProgressCallback, вызывается в фоновом потоке
    public void onProgressChanged(int progress) {
        publishProgress(progress);
    }

    // Метод AsyncTask, вызывается в UI потоке
    protected void onProgressUpdate(Integer... values) {
        int progress = values[values.length -1];
        progressBarView.setProgress(progress);
    }
}
```

Смена конфигурации



Смена конфигурации

- При смене конфигурации создается новый объект Activity
- Запущенный AsyncTask продолжаетработать!
- AsyncTask должен получить ссылку на новый объект Activity для отображения прогресса
- Новый объект Activity не должен запускать новый AsyncTask,а должен «связаться» со старым.

```
public static class GetImageTask extends AsyncTask {
    // Текущий объект Activity, храним для обновления отображения
   private LoadImageActivity activity;

    GetImageActivity(LoadImageActivity activity) {
        this.activity = activity;
    }
    void attachActivity(LoadImageActivity activity) {
        this.activity = activity;
        updateView();
    }
    void updateView() {
        if (activity != null && !activity.isFinishing()) {
            activity.imageView.setImageBitmap(...);
            activity.progressBarView.setProgress(...);
        }
}
```

```
public class GetImageActivity extends Activity {
    // Выполняющийся таск загрузки изображения

   private GetImageTask getImageTask;

   @Override
    public Object onRetainNonConfigurationInstance() {
        // Этот метод вызывается при смене конфигурации,
        // когда текущий объект Activity уничтожается. Объект,
        // который мы вернем, не будет уничтожен, и его можно
        // будет использовать в новом объекте Activity
        return getImageTask;
    }
}
```

```
public class LoadImageActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        if (savedInstanceState != null) {
            // Пытаемся получить ранее запущенный таск
            getImageTask = (GetImageTask) getLastNonConfigurationInstance();
        }
        if (getImageTask == null) {
            // Создаем новый таск, только если не было// ранее запущенного таска
            getImageTask = new GetImageTask(this);
            getImageTask.execute();
        } else {
            // Передаем в ранее запущенный таск текущий
            // объект Activity
            getImageTask.attachActivity(this);
        }
    }
```

Loader

- Гибкий фреймворк для асинхронной загрузки чего-нибудь
- Решает проблему жизненного цикла
- Проще,чем **AsyncTask**
- Базовыйкласс **AsyncTaskLoader** выполняет задачу на пуле потоков (в отличие от **AsyncTask**),то есть может выполняться несколько задач параллельно

https://developer.android.com/guide/components/loaders.html

RxJava

- Гибкий (open-source) фреймворк для асинхронной загрузки чего-нибудь
- Позволяет указать callback (**Observer**), который будет вызван из фреймворка, когда данные готовы
- Позволяет указать, в каком потоке будет выполняться ‘загрузка’ данных и в каком потоке будет вызван callback
- Удобен в использовании

https://github.com/ReactiveX/RxJava