Проблеми при ползването на AsyncTask в Android

Фрагментация, изтичане на памет (memory leaks), проблеми с жизнения цикъл на Activity

Божидар Николаев Захов
фак. № 121217024, група 37
e-mail: bzahov1998@gmail.com
LinkedIn: https://www.linkedin.com/in/bzahov98/

***Резюме: В статията ще погледнем* AsyncTask, от всички гледни точки, ще обясним какви са му положителните и какви са отрицателните му страни, възможните му подводни камъни и за какво трябва да внимаваме при употребата му. Накратко ще обясним и процесите, от които зависи правилната му работа, така че да минимизираме неяснотите, с които ще се сблъскате в бъдеще.**

Ключови думи: AsyncTask, AndroidOS, Parallel Programming, Memory leak, Android Fragmentation, Problems with Android Lifecycle

I. Въведение

В тази статия ще разгледаме основните причини, защо AsyncTask API (*Application Programming Interface*) е била и ще бъде, що се отнася до по-старите версии на Андроид, едно от най-използваните решения на много важния проблем, с паралелната обработка и изчисления в Андроид приложенията.

Също ще разгледаме плюсовете и минусите и, защо вече не се препоръчва използването и в най-новите версии на Андроид според официалния сайт за разработка[1] и с какво може да я заменим. Ще обясним и как работи и как да я използваме, както и някои специфики на операционната система Андроид(Android OS/OC), така че читателя да добие, най-добра представа, за поставения проблем.

II. Кога, къде и как да го използваме

A. Какъв проблем разрешава AsyncTask?

Когато потребителят стартира приложение, операционната система на Android създава изпълнителна нишка (UI thread – нишка на потребителския интерфейс), наречена главна (main thread). Тази нишка е много важна, тъй като отговаря за управлението на събитията, които потребителят заснема в съответните компоненти, а също така включва събитията, които рисуват екрана.

Защо ни трябва изобщо паралелна обработка, а не използваме вече наличната? Отговорът на този въпрос се крие в самият него, а именно, че главната UI нишка се грижи за всичко, което потребителят вижда, задачите по които работи и т.н. Тоест, ако ние трябва да изпълним много обемна и времеемка задача последователно, то това ще блокира потребителя и няма да му позволи да взаимодейства, по никакъв начин с интерфейса(UI) и ще изглежда така, сякаш приложението е забило. Тези операции могат да бъдат заявки към интернет ресурси, SQL(*Structured Query Language*) заявки към база от данни DB (Data base), XML / HTML / JSON разбор, както и сложна графична обработка[2]. Дори индикаторът за зареждане няма да се движи/изпълнява и приложението ще замръзне след 5 секунди, от което операционната система ще попита потребителя дали да убие процеса. Мотото на Андроид що се отнася до паралелното програмиране е следното:

"*Андроидима една [главна] нишка отговорна за взаимодействието с потребителския интерфейс 60FPS(Frame per second/ 60 кадъра в секунда) — стремим се да правим, колкото е възможно по-малко операции нея"[4]*

Затова програмистът може да създаде/извика множество нишки, във всеки един момент, докато приложението се изпълнява. Андроид предлага множество различни начина за създаване и управление на нишки, обаче с толкова много различни подхода е много важно да изберем правилния, а това може да не е лесна задача. Най-примитивният и на най-ниско ниво метод за създаване на нишки, е както при всяка JVM(Java Virtual Machine) и Java код, с например наследяване и имплементиране на класа Thread, което може да види на фрагмента от код .1 :

```
//наследяване и задаване на бъдещата задача, чрез метода run()
class MyThread extends Thread{
        public void run() {// изпълним код}
}
// създаване и извикване на новата нишка
        MyThread thread1 = new MyThread("background task", 1000);
        thread1.start();
```

Блок от код 1.

Главния проблем при този подход е, че тези нишки нямат право да задават команди към потребителския интерфейс и неговите контроли (UI), такова право както споменахме по-горе, има само главната нишка. Или по друг начин казано, тези нишки се създават, изпълняват своята задача, но нямат вградена възможност да променят нищо, пряко свързано с потребителския интерфейс и да доставят резултата от задачата им. Това може да се заобиколи чрез Прихващач(Handler).

За да разрешим горепоставения проблем едното възможно решение, е чрез AsyncTask API, за което ще поговорим след малко.

B. Как да използваме AsyncTask?

В тази глава ще обясним накратко как работи AsyncTask API. За да използваме класа е необходимо първо да наследим AsyncTask по начина показан на блока от код 2:

```
private class MyAsyncTask extends AsyncTask< Params, Progress, Result >
```

Блок от код 2. Шаблон за наследяване на AsyncTask [1]

В основата му стоят трите параметъра представени чрез шаблонни типове (Generics).

- `Params` или типа на параметрите,, изпратени към задачата по време на изпълнението на задачата в doInBackground(Params…params)
- `Progress` или типа на параметрите, получени по време на междинните моменти от частичното изпълнение на задачата в onProgressUpdate(`Progress`… progress).
- `Result` или типа на параметрите,получени при крайния резултат от изпълнението на задачата в onPostExecute(`Result`…result). [1]

Може да извикаме задачата за изпълнение чрез блока от код 3 по-долу

```
new MyAsyncTask().execute(url1, url2, url3);
```

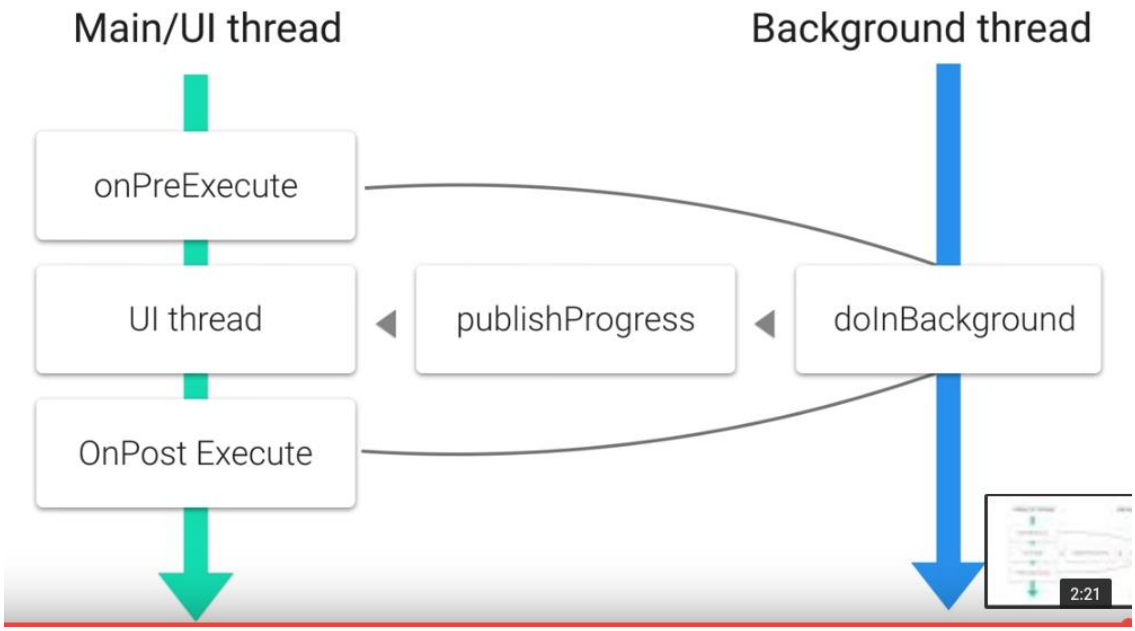
Блок от код 3. Извикване на задачата за изпълнение

Според официалната документация[1] , след като бъде извикана задачата, за изпълнение, жизнения цикъл на AsyncTask, следва следните стъпки, :

- onPreExecute() – Извиква се на UI нишката преди да бъде изпълнена задачата. Тук обикновено се подготвя всичко необходимо за правилното изпълнение на задачата в последствие.
- **`doInBackground`**`(Params...)` – Извиква се на паралелната нишка веднага след **`onPreExecute`**`()`.

- Извършва сложните и тромави сметки необходими, за изпълняването на задачата, която му е зададена да изпълни
- Параметрите(Params) на асинхронната задача се прехвърлят тук
- Резултата от сметките трябва да бъде върнат от тук и ще бъде прехвърлен обратно до предишната стъпка.
- Тази стъпка може да използва **publishProgress**(Progress...) да публикува един или няколко подстъпки за прогрес, публикувани на UI нишката, чрез **onProgressUpdate**(Progress...) стъпка.

- **onProgressUpdate**(Progress...) – Извиква се на UI нишката, когато има нова публикувана информация чрез **publishProgress**(Progress...)
  - По време на тази стъпка, успоредно с нея продължава изпълнението на задачата на паралелната нишка в **doInBackground**()
- **`onPostExecute`**`(Result)` – Извиква се на UI нишката, когато задачата е изпълнена.
  - Получава резултата от изчисленията изпълнени в **doInBackground**().[1]



Фиг. 1. Жизнения цикъл на AsyncTask и комуникацията между основната UI нишка и паралелната нишка[4]

C. Прекъсване на Задача

Задачите поставени на AsuncTask могат да бъдат прекъснати чрез метода **`cancel`**`(`**`boolean`**`)`. Извикването и ще направи така ,че във всеки един момент на изпълнението на задачата да може да се провери дали тя не е била прекъсната преждевременно, чрез метода **`isCancelled`**`()`.

Това е много важно да се проверява периодично в **doInBackground**(), защото ако това не бъде направено, първо ще се натоварва безсмислено процесора и второ може да доведе до изтичане (задържане) на памет (Memory leak).[1]

D. Какви са недостатъците на AsyncTask

1) Проблеми с фрагментацията

Фрагментацията на Android е проблем, който възникна в резултат на твърде честото обновяване на платформата и многото производители на устройства, използващи различен хардуер и работещи под нейно управление. С всяка нова версия на операционната система се вкарва нова функционалност, изглаждат се стари неизправности, подобрява се бързодействието и и т.н, но се налага да се приспособяват и към старите версии или: [6]

"Фрагментацията на Android се отнася до тревога за тревожния брой различни версии на операционната система (OS) на пазара. Основният проблем е потенциалната намалена оперативна съвместимост между устройствата с приложения, кодирани с помощта на комплекта за разработка на софтуер за Android (Android SDK)".[10]

Тя е най-често изтъквания недостатък на платформата и въпреки всичките усилия на производителите да премахнат проблемите, те все пак са факт, и дори стават все по-големи с течение на времето, тъй като все повече потребители настояват да притежават най-новата и най-добрата версия на платформата на телефоните си. [7]

AsyncTask е стара концепция още от началото на Андроид ("Added in API level 3" [1]), а с новите версии на платформата се развила и в резултат на това, поведението и се различава при различните версии и устройства. Това е част от по – големия проблем с фрагментацията на операционната система.

И тъй като поведението и може да варира, както казахме по-горе, вследствие на това и поведението на нашето приложение няма да е такова, каквото очакваме.

а) Какво може да направим, за да сведем вероятността от неочаквано поведение на програмата ни, до минимум?

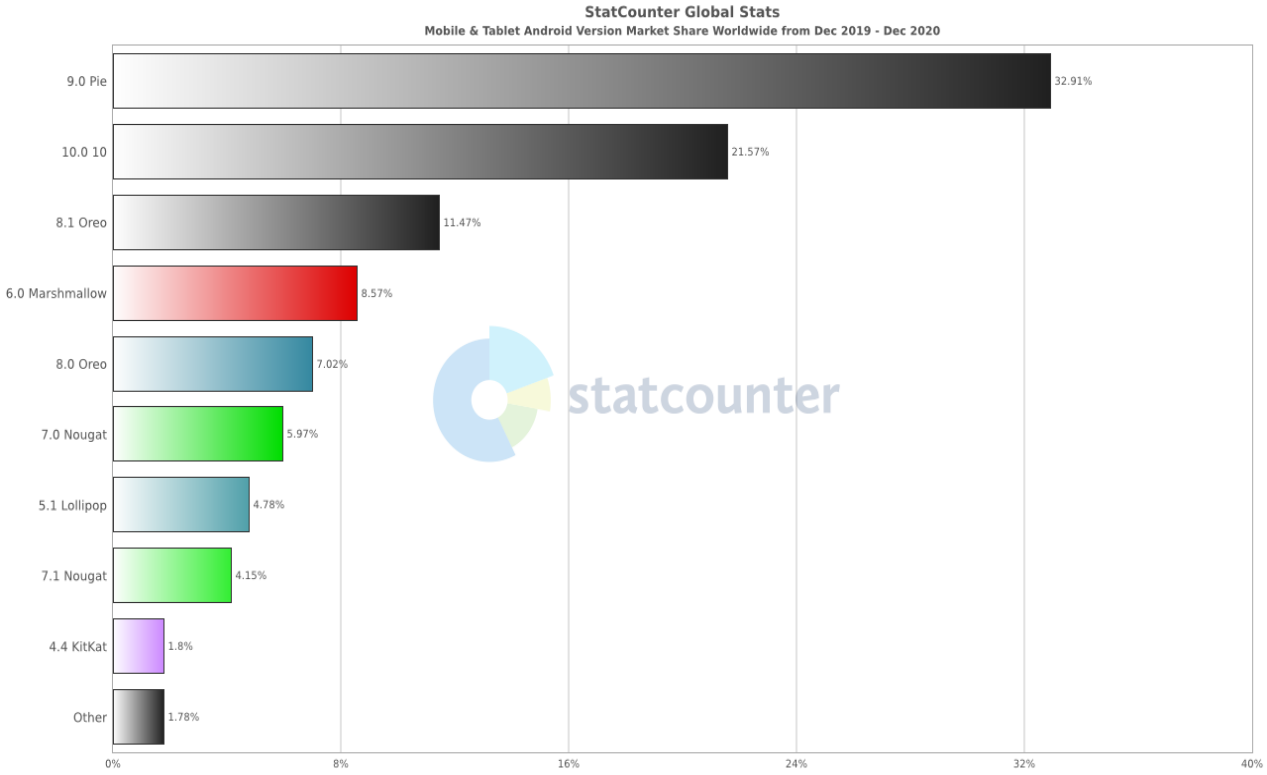
Най-очевидното решение е да използваме набор от поддържани версии, при които всичките ни функции биха се държали по един и същи начин. Например за AsyncTask след Honeycomb 3.0, когато се променя поведението и и преди Android 11, когато AsyncTask спира официално да се поддържа и трябва да се избягва(Deprecated). Това означава, че нашите потребители извън тези граници, няма да могат да използват нашата програма, ако ние не направим отделна функционалност за тях, и на практика по този начин ще изгубим потребители.

*"This class was deprecated in API level 30."*според официалната документация на Андроид[1]

Трябва да отбележим, че към 2013 това е бил доста сериозен проблем, защото когато AsyncTask се е ползвала масово, огромен брой потребители е бил под версията Honeycomb 3.0 и е било неподходящо да се ограничава толкова много пазарния дял на приложението. [7][8]

Вторият вариант е да проектираме нашия код внимателно и да тестваме изчерпателно, върху редица устройства - разбира се това винаги е добра практика, но както видяхме, паралелното програмиране е достатъчно трудно, а като трябва да вземем предвид и допълнителната сложност на фрагментацията и неизменната истина, че винаги ще се възпроизведат и допълнителни грешки(bugs). [7]

Третото решение, което е предложено от Android общността (Android development community), е да се имплементира наново AsyncTask в рамките на вашия собствен проект. Така ще сте независими от ограниченията на средата(SDK-Software Development Kit). Но тогава си задаваме въпроса дали не е по-добре да се насочим към други налични решения, които са на разположение от Android. [7]



Фиг. 2. Разпространението на различните Андроид версии през 2020 година[11].

2) *Проблеми с изтичане (задържане) на памет(Memory leak) и с жизнения цикъл на екрана (lifecycle activity problems)*

а) Какво е изтичане (задържане) на памет(Memory leak)

Средата за изпълнение на Java (Java Runtime Environment - JRE) използва техниката на боклуко-събирач (garbage - collector) за премахване на ненужните обекти. Боклуко-събирача се включва в определен момент от работата на програмата, като той работи в ниско-приоритетна нишка (low-level thread) и наблюдава обектите. Когато всички псевдоними (references) на даден обект спрат да се използват, обектът престава да е достъпен, бива маркиран за изчистване. По някое време, никой не може да гарантира кога ще е това, боклуко-събирача разрушава обекта и изпраща паметта му в зоната на свободното пространство, след което тази памет може да бъде преизползвана.

В повечето случаи боклуко-събирача си свършва работата, както е предвидено, без проблеми, но има някои сценарии, при които може да се случи неочаквано изтичане на памет. Това са случаите, когато даден обект не се използва никъде в програмата, но по някаква причина, той не може да бъде изчистен от боклуко-събирача. [9]

Някои от основните ситуации, при които се случва това са:

- Статично контекстно поле (static context object)
  Жизненият цикъл (lifecycle) на статичен обект започва, когато му се присвои стойности завършва, само тогава, когато тази стойност бъде заличена (виж блок от код 5. и 6.) [9]:

```
public class MyActivity extends AppCompatActivity { // нестатичен клас
// Статично контекстно поле в нестатичен клас
        private static Context sContext;
;
```

Блок от код 4. Статично контекстно поле в нестатичен клас

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        (…)
        sContext = this;
```

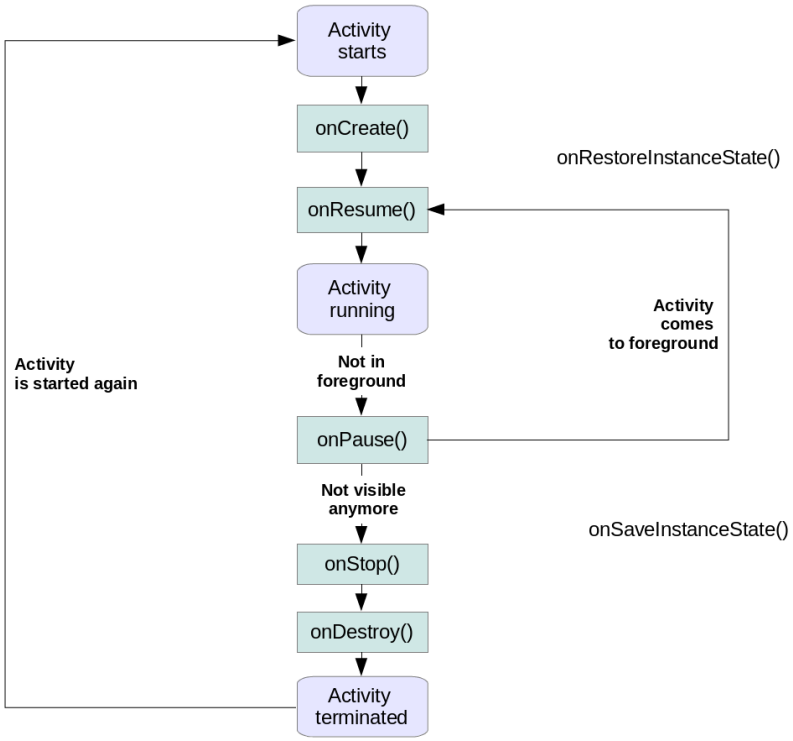
Блок от код 5. Започване на жизнения цикъл на статичен обект, чрез задаване на препратка към екрана

```
@Override
protected void onDestroy() {
        sContextReference = null;
```

Блок от код 6. Предразполагане за заличаване на жизнен цикъл на статичен обект, чрез премахване на препратката

За да разрешим проблемът със статичния контекст, ние бихме могли да подходим по няколко начина, като най-простия е като просто премахнем препратката на контекста преди да се разруши екрана в onDestroy(), виж блок от код 4. [9]

- изтичане (задържане) на памет (Memory leak), в жизнения цикъл на екран



Фиг. 1. Жизнен цикъл на екран [12]

Винаги е много важно да премахваме всякакви статични препратки, активни слушатели на събития (callbacks listeners) и всякакви обекти свързани с потребителския интерфейс, в съответния за това метод от жизнения път на даден екран. Например onPause(), onStop(), onDestroy().

Реципрочно на това,когато е необхидимо, трябва да ги активираме, инициализираме и т.н - в съответните за това методи като например onCreate(), onResume(), OnStart(). Ако не направим това, рискуваме да се получи изтичане (задържане) на памет (Memory leak), тъй като ще има заделена памет за даден екран, а поради това този обект няма да може да бъде боклуко-събран и заличен.

В контекстът на AsyncTask, трябва да имаме в предвид, че ако асинхронната задача, която се изпълнява във метода **doInBackground**(), възможно да надживее живота на екрана, при което все още няма да сме получили резултата в метода **onPostExecute**(). При това положение AsyncTask обектът все още не е приключил операцията си и не подлежи на боклуко-събиране, тоест няма как нито той, нито екрана да бъде изчистен от боклуко-събирача(Garbage-Collector). Така това причинява изтичане (задържане) на памет (Memory leak) и трябва да бъде избегнато.

Не трябва да забравяме, че даден екран се унищожава по-често от колкото си мислим. Например най-честия случай за спонтанно унищожаване е, когато потребителя завърти екрана. Затова силно се препоръчва да няма никакви AsyncTask свързани директно с активитито. За да се случи това се

използват различни Архитектури, като Model View Controler(MVC), Model View ViewModel (MVVM) и Model View Presenter(MVP). Ще поговорим за AsyncTask от гледна точка на MVC, и чрез тази архитектура ще го обясним:

- Нека допуснем, че активитито е контролера в нашия пример и че контролера не трябва да започва операции, които надживяват View-то. Това е накратко,
- Модел - AsyncTask трябва да се използва от Модела, или по друг начин казано от клас, който не е свързан с жизнения път на екрана, все пак той се унищожава при завъртане.
- Тоест е грешно да използваме производните на AsyncTask в методите на Екраните. Изводът от това е, че ако не трябва да ги ползваме така, то това да използваме AsyncTask губи привлекателност

- Static view object

Когато препратка на визуален обект(View) се съдържа в екран(Activity), боклуко-ссъбирачът не може да изчисти обекта, или по друг начин казано - наличието на статичен визуален обект е еквивалентно на това да имаме Статично поле в нестатичен клас, а защо това не е добре, обяснихме по-горе: [9]

"A view object holds a reference to the activity that it is housed in, so having a static view object is equivalent to having a static context object."[9]

- При нестатичен вграден анонимен клас (Non-static anonymous inner class)

Леко особения проблем в случая, е че инстанцията на нестатичен вграден анонимен клас има имплицитна препратка към инстанцията на основния клас [9]. Този случай е много често срещан при употребата на AsyncTask,защото по този начин не е необходимо да имаме нов клас наследяващ го, както и фактът, че вградените класове имат достъп до всички скрити полета на външния клас.

```
public class MyActivity extends AppCompatActivity {

 private TextView resultTextView;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
     resultTextView = (TextView)
                          findViewById(R.id.txt_result);

     // Start an asyncTask
     new AsyncTask<Void, Void, String>() {
             @Override
             protected String doInBackground( final Void ... params )
{
                 // a methods takes very long time
                 String result = aMethodTakesVeryLongTime();
                 return result;
             }

             @Override
             protected void onPostExecute( final String result ) {
                 // continue what you are doing...
                 resultTextView.setText(result);
             }
         }.execute();

 }
```

Блок от код 7. Инстанцията на нестатичен вграден анонимен клас с имплицитна препратка към инстанцията на основния клас (по-горе) [9]

Затова ще го разгледаме малко по-подробно. Нека видим блока от код 5, който показва вграждането на AsyncTask като анонимен клас в активитито.

Както обяснихме по-рано, ако докато се изпълнява асинхронната задача, преждевременно прекратим екрана, ще настъпи изтичане на памет(виж изтичане на памет в жизнения цикъл на екран),

Тук ще покажем как да разрешим този проблем с помощта на класа WeakReference[14] и нов клас наследяващ AsyncTask – виж блок от код 6. WeakReference не позволява на боклуко-събирача да бъдат финализирани, а в последствие унищожени, освен при подходящи обстоятелства и в правилния момент, като така се избягва изтичане на памет.

```
public class MyActivity extends AppCompatActivity {

 private TextView resultTextView;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
     resultTextView = (TextView)
                          findViewById(R.id.txt_result);

     // Start an asyncTask
     new MyTask(this).execute();

}

private static class MyTask extends AsyncTask<Void, Void, String> {

  private final WeakReference<MyActivity> mActivityRef;

  public MyTask(MyActivity activity){
       mActivityRef = new WeakReference<>(activity);

}

  @Override
  protected String doInBackground(final Void ... params) {
        // a methods takes very long time
        String result = aMethodTakesVeryLongTime();
        return result;
  }

  @Override
  protected void onPostExecute( final String result ) {
        // continue what you are doing...
```

Блок от код 8. Подобрена версия на блок от код 7. Имаме отделен клас с WeakReference(по-долу) [9]

3) *AsyncTask е остарял (Deprecated)*

Според официалната документация на Андроид , AsyncTask API вече се смята за остаряла (Deprecated) технология от Android 11 (**API level 30).** Това означава, че всички разработчици на всяка цена, трябва да избягват употребата за приложения използващи версии след Android 11+:

“This class was deprecated in API level 30.
Use the standard java.util.concurrent or Kotlin concurrency utilities instead.”[1]

Официалната страница предлага използването на доста по-модерните алтернативи, като Kotlin функциите за паралелност (Kotlin concurrency utilities) които си струва да проучите, или стандартните функции за паралелност на Java (`java.util.concurrent`).[1]

III. Заключение

Андроид предлага много различни решения за справяне с предизвикателствата на многонишковото програмиране, но никое от тях не е перфектно за абсолютно всички ситуации, всяко едно решение има добри и лоши черти, с които разработчика трябва да се съобразява. Затова програмистът предварително трябва да проучи характеристиките на тези решения и да прецени кое да използва.

В тази статия ние обяснихме често срещаните проблеми на AsyncTask и как, и дали може да ги преодолеем . AsyncTask е сравнително лесно за разбиране, дори ползвателят и да не е много запознат , с характеристиките на паралелното програмиране, но разработчика задължително трябва да проучи предварително, кое е най-подходящото решение, за да не открие в движение множеството “подводни камъни”, които безспорно всяко едно от тях притежава.

IV. Литература

[1] https://developer.android.com/reference/android/os/AsyncTask - Официална документация на андроид за AsyncTask

[2] https://bg.admininfo.info/programar-android-procesos - admininfo.info - Употреба на Асинхронни операции

~~[3] **https**://abhiandroid.com/programming/asynctask - Syntax of AsyncTask In Android:~~

[4] https://x-wei.github.io/andev_p1e2_internet.html/ - Фигура 1 – AsyncTask Lifecycle

[5] https://stackoverflow.com/questions/22932862/when-to-use-asynctask-and-when-to-use-thread-in-android - Кога да използваме AsyncTask

[6] https://stackoverflow.com/questions/3357477/is-asynctask-really-conceptually-flawed-or-am-i-just-missing-something?rq=1 – Концепционални проблеми при AsyncTask

[7] https://hub.packtpub.com/common-asynctask-issues/ - Честосрещани асинхронни проблеми

[8] https://en.wikipedia.org/wiki/Android_version_history - Wikipedia - Android история на версиите(Version History)

[9] https://medium.com/@zhangqichuan/memory-leak-in-android-4a6a7e8d7780 - Zhang QiChuan - Memory Leaks

[10] https://bg.begin-it.com/3486-android-fragmentation-problem - Geoffrey Carr, Techopedia, Определяне за фрагментацията на Android

[11] https://gs.statcounter.com/android-version-market-share/mobile-tablet/bulgaria#monthly-201912-202012-bar

[12] https://www.vogella.com/tutorials/AndroidLifeCycle/article.html - Lars Vogel - Жизнен цикъл на активити

[13] https://planerny-ndv.ru/bg/the-tax-system/processy-i-potoki-processy-i-potoki-processy-i-potoki.html

[14] https://developer.android.com/reference/java/lang/ref/WeakReference - Android Docs - WeakReference