Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский политехнический университет

Петра Великого»

Институт компьютерных наук и технологий

# Отчёт по лабораторной работе №6 Дисциплина: Проектирование мобильных приложений

Выполнил студент гр. 3530901/70202 \_\_\_\_\_\_\_\_\_\_\_ Павлов Д.В.

*(подпись)*

Преподаватель \_\_\_\_\_\_\_\_\_\_\_ Кузнецов А.Н.

*(подпись)*

Санкт-Петербург

2019

**1. Цели работы**

* Получить практические навыки разработки многопоточных приложений:
  + 1. Организация обработки длительных операций в background (worker) thread.
    2. Публикация данных из background (worker) thread в main (ui) thread.
* Освоить 3 основные группы API для разработки многопоточных приложений:
  + 1. Kotlin Coroutines
    2. AsyncTask
    3. Java Threads

**2. Выполнение работы**

# 2.1. Задание 1

Используйте приложение "не секундомер", получившееся в результате выполнениня Лабораторной работы №2. Разработайте несколько альтернативных приложений "не секундомер", отличающихся друг от друга организацией многопоточной работы.

# 2.1.1. Решение с помощью Java Threads

Был модернизирован код из 2 лабораторной работы. Осуществилось добавление метода onInitTask(), в котором переменной *backgroundTask* присваивается поток и производится его запуск. Также были переопределены методы *onPause()/onResume(),* которые теперь отвечают за корректное завершение потока и его порождение соответственно. Так как sleep(1000) не гарантирует остановку потока ровно на 1000 мс, было решено использовать *System.currentTimeMillis()* (текущее время) и сравнивать его с переменной *endTime = System.currentTimeMillis()+1000.* Далее мы производим сравнение до тех пора, пока текущее время не нагонит endTime, после чего икрементируем секундомер. Код представлен в Листинге 2.1.1.

Листинг 2.1.1.

class MainActivity : AppCompatActivity() {  
 private val KEY\_SECONDS\_ELAPSED = "key\_seconds\_elapsed"  
 private var isWorking = false  
 private var secondsElapsed = 0  
 private var backgroundTask: Thread? = null  
  
 private val TAG = "STATE"  
  
 private fun thread() = Thread **{** while (backgroundTask != null) {  
 try {  
 val endTime = System.currentTimeMillis() +1000  
 while (System.currentTimeMillis()<endTime){  
 *synchronized*(this)**{** try{  
 Thread.sleep(1000)  
 } catch (e: Exception) {  
 Log.e("error", "" + e.message)  
 e.printStackTrace()  
 }  
 **}** }  
 runOnUiThread **{** timeSecondsElapsed.*text* = "Seconds elapsed: " + secondsElapsed++  
 **}** } catch (e: InterruptedException) {return@Thread}  
 }  
 **}** override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 log("onCreate")  
 setContentView(R.layout.*activity\_main*)  
 onInitTask()  
 }  
  
 private fun onInitTask(){  
 backgroundTask = thread()  
 backgroundTask?.start()  
 isWorking = true  
 }  
  
 override fun onResume() {  
 super.onResume()  
 if(!isWorking) onInitTask()  
 log("onResume")  
 }  
  
 override fun onPause() {  
 backgroundTask?.interrupt()  
 backgroundTask = null  
 isWorking = false  
 super.onPause()  
 log("onPause")  
 }  
  
 private fun log(msg: String) {  
 Log.i(TAG, msg)  
 }  
  
 override fun onRestoreInstanceState(savedInstanceState: Bundle) {  
 super.onRestoreInstanceState(savedInstanceState)  
 secondsElapsed = savedInstanceState.getInt(KEY\_SECONDS\_ELAPSED)  
 log("onRestoreInstanceState")  
 }  
  
 override fun onSaveInstanceState(outState: Bundle) {  
 super.onSaveInstanceState(outState)  
 outState.putInt(KEY\_SECONDS\_ELAPSED, secondsElapsed)  
 log("onSaveInstanceState = $outState")  
 }  
}

Проверка правильности работы приложения при сворачивании, разворачивании и изменении ориентации экрана прошла успешно.

# 2.1.2. Решение с помощью AsyncTask

AsyncTask позволяет правильно и легко использовать поток пользовательского интерфейса. Этот класс позволяет выполнять фоновые операции и публиковать результаты в потоке пользовательского интерфейса без необходимости манипулирования потоками/обработчиками.

Соответствующим образом были модернизированы методы onInitTask(), onPause() и onResume(). Остановка на 1 секунду реализована таким же алгоритмом, как в прошлом пункте. Код представлен в Листинге 2.1.2.

Листинг 2.1.2.

class MainActivity : AppCompatActivity() {  
 private val KEY\_SECONDS\_ELAPSED = "key\_seconds\_elapsed"  
 private var secondsElapsed = 0  
 private lateinit var timer: TimerTask  
  
 private val TAG = "STATE"  
  
 @SuppressLint("StaticFieldLeak")  
 inner class TimerTask : AsyncTask<Int, Int, Void>(){  
 override fun doInBackground(vararg params: Int?): Void? {  
 try{  
 while(true){  
 if(*isCancelled*) return null  
 val endTime = System.currentTimeMillis() +1000  
 while (System.currentTimeMillis()<endTime) {  
 *synchronized*(this) **{** try {  
 *Random*(Int.MAX\_VALUE)  
 } catch (e: Exception) {  
 Log.e("error", "" + e.message)  
 e.printStackTrace()  
 }  
 **}** }  
 publishProgress(secondsElapsed)  
 }  
 } catch (e: InterruptedException){  
 Toast.makeText(*applicationContext*, e.message, Toast.*LENGTH\_SHORT*).show()  
 }  
 return null  
 }  
  
 @SuppressLint("SetTextI18n")  
 override fun onProgressUpdate(vararg values: Int?) {  
 super.onProgressUpdate(\*values)  
 timeSecondsElapsed.*text* = "Seconds elapsed: " + secondsElapsed++  
 }  
 }  
  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 log("onCreate")  
 setContentView(R.layout.*activity\_main*)  
 onInitTask()  
 }  
  
 private fun onInitTask(){  
 timer = TimerTask()  
 timer.execute(secondsElapsed)  
 }  
  
 override fun onResume() {  
 super.onResume()  
 if(timer.*status* != AsyncTask.Status.RUNNING) onInitTask()  
 log("onResume")  
 }  
  
 override fun onPause() {  
 super.onPause()  
 timer.cancel(true)  
 log("onPause")  
 }  
  
 private fun log(msg: String) {  
 Log.i(TAG, msg)  
 }  
  
 override fun onRestoreInstanceState(savedInstanceState: Bundle) {  
 super.onRestoreInstanceState(savedInstanceState)  
 secondsElapsed = savedInstanceState.getInt(KEY\_SECONDS\_ELAPSED)  
 log("onRestoreInstanceState")  
 }  
  
 override fun onSaveInstanceState(outState: Bundle) {  
 super.onSaveInstanceState(outState)  
 outState.putInt(KEY\_SECONDS\_ELAPSED, secondsElapsed)  
 log("onSaveInstanceState = $outState")  
 }  
}

Произошло добавление следующих методов:

* doInBackground() – основной метод. Выполняется в новом потоке и не имеет доступа к UI.
* onProgressUpdate() – обработчик для публикации промежуточных обновлений в пользовательский интерфейс. При вызове синхронизируется с потоком GUI (имеет доступ к UI).
* publishProgress() – можно вызвать в doInBackground() для показа промежуточных результатов в onProgressUpdate() (в данном случае производит увеличение переменной secondsElapsed).

Проверка правильности работы приложения при сворачивании, разворачивании и изменении ориентации экрана прошла успешно.

# 2.1.3. Решение с помощью Kotlin Coroutines

Kotlin Coroutines по своей сути являются «легковесными потоками». Они не требуют столько ресурсов для создания асинхронных вычислений, сколько требуют обычные потоки. В Kotlin все корутины работают внутри [CoroutineScope.](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.experimental/-coroutine-scope/)

В *gradle* были добавлены следующие зависимости:

implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.1"  
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.1.1"

В данном случае я отказался от предыдущего алгоритма подсчета секунд, так как из корутин мы можем вызвать suspend функцию receive(). Код представлен в Листинге 2.1.3.

Листинг 2.1.3.

@ExperimentalCoroutinesApi  
@ObsoleteCoroutinesApi  
class MainActivity : AppCompatActivity() {  
 private val KEY\_SECONDS\_ELAPSED = "key\_seconds\_elapsed"  
 private var secondsElapsed = 0  
 private lateinit var timer: CoroutineScope  
  
 private val TAG = "STATE"  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 log("onCreate")  
 setContentView(R.layout.*activity\_main*)  
 onInitTask()  
 }  
  
 private fun onInitTask(){  
 val ticker = *ticker*(delayMillis = 1\_000, initialDelayMillis = 1\_000)  
 timer = *CoroutineScope*(Dispatchers.Default + *SupervisorJob*())  
 timer.*launch*(Dispatchers.IO)**{** while(true){  
 ticker.receive()  
 timer.*launch*(Dispatchers.Main)**{** timeSecondsElapsed.*text* = "Seconds elapsed: " + secondsElapsed++**}** }  
 **}** }  
  
 override fun onResume() {  
 super.onResume()  
 if(!timer.*isActive*) onInitTask()  
 log("onResume")  
 }  
  
 override fun onPause() {  
 super.onPause()  
 timer.*cancel*()  
 log("onPause")  
 }  
  
 private fun log(msg: String) {  
 Log.i(TAG, msg)  
 }  
  
 override fun onRestoreInstanceState(savedInstanceState: Bundle) {  
 super.onRestoreInstanceState(savedInstanceState)  
 secondsElapsed = savedInstanceState.getInt(KEY\_SECONDS\_ELAPSED)  
 log("onRestoreInstanceState")  
 }  
  
 override fun onSaveInstanceState(outState: Bundle) {  
 super.onSaveInstanceState(outState)  
 outState.putInt(KEY\_SECONDS\_ELAPSED, secondsElapsed)  
 log("onSaveInstanceState = $outState")  
 }  
}

Проверка правильности работы приложения при сворачивании, разворачивании и изменении ориентации экрана прошла успешно.

# 2.2. Задание 2. Загрузка картинки в фоновом потоке с помощью AsyncTask

Было добавлено следующее разрешение в manifest:

<uses-permission android:name="android.permission.INTERNET" />

Код представлен в Листинге 2.2.1.

Листинг 2.2.1.

class MainActivity : AppCompatActivity() {  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
 DownloadTask(findViewById(R.id.*image\_view*)).execute("https://versiya.info/uploads/posts/2017-11/1509554686\_maxresdefault.jpg")  
 }  
}  
  
 @SuppressLint("StaticFieldLeak")  
 private class DownloadTask(private var bmImageView: ImageView) :  
 AsyncTask<String, Void, Bitmap>() {  
 override fun doInBackground(vararg params: String?): Bitmap? {  
 val url = params[0]  
 var dIcon: Bitmap? = null  
 try{  
 val `in` = java.net.URL(url).openStream()  
 dIcon = BitmapFactory.decodeStream(`in`)  
 } catch (e: Exception) {  
 Log.e("Warning",e.message)  
 e.printStackTrace()  
 }  
 return dIcon  
 }  
  
 override fun onPostExecute(result: Bitmap){  
 bmImageView.setImageBitmap(result)  
 }  
}

При запуске приложения первые несколько секунд пользователь видит только белый фон, затем в ImageView в фоновом потоке загружается изображение, ссылка на которое вставлена в коде.

Результат представлен на Рис.1.

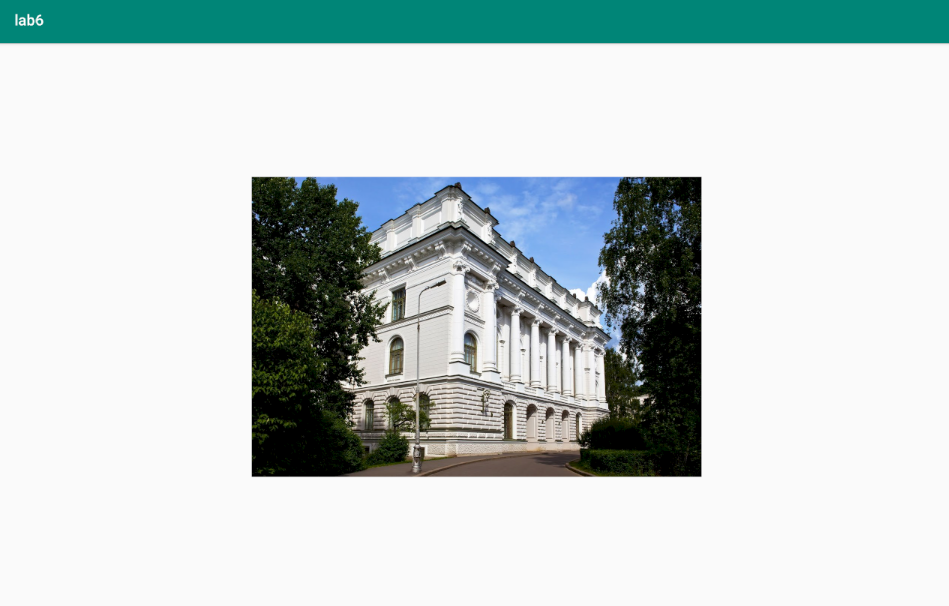


Рис.1.

# 2.3. Задание 3. Загрузка картинки с помощью Kotlin Coroutines

Программа из задания 2 была изменен с использованием Kotlin Coroutines. Код представлен в Листинге 2.3.1.

Листинг 2.3.1.

class MainActivity : AppCompatActivity() {  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
  
 val imageLoader = *CoroutineScope*(Dispatchers.Default + *SupervisorJob*())  
 imageLoader.*launch*(Dispatchers.IO) **{** var dImage: Bitmap? = null  
 try {  
 val inStream =  
 URL("https://versiya.info/uploads/posts/2017-11/1509554686\_maxresdefault.jpg").openStream()  
 dImage = BitmapFactory.decodeStream(inStream)  
 } catch (e: Exception) {  
 Log.e("Warning", e.message)  
 e.printStackTrace()  
 }  
 imageLoader.*launch*(Dispatchers.Main) **{** image\_view.setImageBitmap(dImage) **}  
 }** }  
}

В результата загрузка произошла несколько быстрее, чем в случае с AsyncTask().

# 2.4. Задание 4. Загрузка картинки с использованием сторонних библиотек

Для выполнения данной задачи использована библиотека Picasso. Код представлен в Листинге 2.4.1.

Листинг 2.4.1.

class MainActivity : AppCompatActivity() {  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
 val ivBasicImage : ImageView = findViewById(R.id.*image\_view*)  
 Picasso.with(*applicationContext*).load("https://versiya.info/uploads/posts/2017-11/1509554686\_maxresdefault.jpg").into(ivBasicImage)  
 }  
}

Легко заметить, что код сократился в несколько раз (в сравнении с пунктами 2 и 3).

# 3. Выводы

В ходе выполнения данной лабораторной работы получены практические навыки разработки многопоточных приложений. Были рассмотрены 3 основные группы API для разработки многопоточных приложений:

1)Java Threads – объект-потомок класса Thread. Этот класс инкапсулирует стандартные механизмы работы с потоками

2)AsyncTask – используется для перемещения ресурсозатратных операций в фоновой поток. Работа AsyncTask не зависит от жизненного цикла Activity. Это значит, что, при условном повороте экрана (когда вызывается метод onDestroy()), AsyncTask продолжит работать до своего завершения. Это является существенным недостатком, так как мы будем хранить ссылку на не существующее Activity, а это ведет к утечкам память.

3)Kotlin Coroutines – легковесные и простые аналоги потоков. Контролируются CoroutineScope. Он может выбирать поток для каждой из coroutine (в том числе главный), отменить все запущенные. При своей приостановке в главном потоке, сам главный поток не остановится.

Так как в главном потоке нельзя выполнять тяжелые операции (существенно тормозит работу приложения и в крайних случаях приводит к ошибке), в каждом варианте задач ожидание и скачивание выполнялось в отдельных потоках и корутинах. Для первой задачи лучшим оказался вариант с использованием Kotlin Coroutines, а для решения задачи с загрузкой изображения удобнее использовать стороннюю библиотеку (была выбрана библиотека Picasso).

Решение Задачи 1 не ограничивается 3 способами, вот некоторые известные для меня:

1)IntentService – должен являться выбором по умолчанию разработчика для загрузки\выгрузки больших файлов, что требует длительной обработки.

2)CountDownTimer – обратный отсчет (на каком-то этапе применял в своем курсовом проекте). Может работать из любого потока и не блокирует использование приложения пользователем, даже при инициализации из основного потока. Лучше не использовать для тяжелых операций.  
  
3)Android HandlerThread – используется для обработки сообщений фонового потока, запуска сервиса в фоновом потоке.

4)Java Timer – можно использовать для планирования обработки в фоновом потоке. Однако есть важный момент, этот API не знает о жизненном цикле android, поэтому любая грубая ссылка на Activity, Fragment или View может привести к утечке памяти.