Санкт-Петербургский политехнический университет Петра Великого

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №6**

**Дисциплина**: Проектирование мобильных приложений

**Тема**: Многопоточные Android приложения

Выполнил студент гр. 3530901/70203 И.Д. Иванов

(подпись)

Преподаватель И.С. Егорова

(подпись)

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург

2019

**1. Цели работы**

Получить практические навыки разработки многопоточных приложений:

* Организация обработки длительных операций в background (worker) thread.
* Публикация данных из background (worker) thread в main (ui) thread.

Освоить 3 основные группы API для разработки многопоточных приложений:

* Kotlin Coroutines
* AsyncTask
* Java Threads

**2. Программа работы**

**2.1. Задача 1.** Альтернативные решения задачи "не секундомер" из Лаб. 2

Используйте приложение "не секундомер", получившееся в результате выполнения Лабораторной работы №2. Разработайте несколько альтернативных приложений "не секундомер", отличающихся друг от друга организацией многопоточной работы. Опишите все известные Вам решения.

Исходный код решения с помощью Thread и Handler:

class MainActivity : AppCompatActivity() {  
 companion object{  
 var secondsElapsed: Int = 0  
 }  
 class MyHandler(private val act: AppCompatActivity) : Handler() {  
 override fun handleMessage(msg: Message) {  
 act.findViewById<TextView>(R.id.*textSecondsElapsed*).*text* =  
 "Seconds elapsed: " + ++secondsElapsed  
 }  
 }  
 private lateinit var sharedPref:SharedPreferences  
 private lateinit var backgroundThread: Thread  
  
 var handler: Handler = MyHandler(this)  
  
 private val timerRunnable: Runnable = *Runnable* **{** while (!Thread.interrupted()) {  
 try {  
 Thread.sleep(1000)  
 handler.sendEmptyMessage(0)  
 } catch (e: InterruptedException) {  
 Thread.currentThread().interrupt()  
 }  
 }  
 **}**

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
 }  
  
 override fun onResume() {  
 super.onResume()  
 backgroundThread = Thread(timerRunnable)  
 backgroundThread.start()  
 sharedPref = *applicationContext*.getSharedPreferences("myPrefs", *MODE\_PRIVATE*)  
 secondsElapsed=sharedPref.getInt("secondsElapsed",0)  
  
 }  
  
 override fun onPause() {  
 super.onPause()  
 backgroundThread.interrupt()  
 val editor = sharedPref.edit()  
 editor.putInt("secondsElapsed",secondsElapsed)  
 editor.apply()  
 }  
  
  
}

В данном случае мы создаём объект типа Handler, который при получении сообщения увеличивает счётчик секунд и обновляет textView. Кроме того, мы создаём объект типа Runnable, который отсчитывает секунды и отправляет сообщения Handler-у. Для того, чтобы поток останавливался, когда приложения не отображается на экране, используется метод Thread.interrupt() в onPause(). В методе onResume() из Runnable объекта создаётся и запускается новый поток.

Исходный код решения с помощью AsyncTasc:

class MainActivity : AppCompatActivity() {  
 companion object {  
 var secondsElapsed: Int = 0  
 }  
  
 private lateinit var sharedPref: SharedPreferences  
 private lateinit var timerAsyncTask: TimerAsyncTask  
  
 private class TimerAsyncTask internal constructor(private val act: AppCompatActivity) :  
 AsyncTask<Void, Void, Void>() {  
  
 override fun doInBackground(vararg params: Void?): Void? {  
 while (!*isCancelled*) {  
 TimeUnit.SECONDS.sleep(1)  
 publishProgress()  
 }  
 return null  
 }

override fun onProgressUpdate(vararg values: Void?) {  
 super.onProgressUpdate(\*values)  
 act.findViewById<TextView>(R.id.*textSecondsElapsed*).*text* =  
 "Seconds elapsed: " + ++secondsElapsed  
  
 }  
 }  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
 }  
  
  
 override fun onResume() {  
 super.onResume()  
 timerAsyncTask = TimerAsyncTask(this)  
 timerAsyncTask.execute()  
 sharedPref = *applicationContext*.getSharedPreferences("myPrefs", *MODE\_PRIVATE*)  
 secondsElapsed = sharedPref.getInt("secondsElapsed", 0)  
 }  
  
 override fun onPause() {  
 super.onPause()  
 timerAsyncTask.cancel(true)  
 val editor = sharedPref.edit()  
 editor.putInt("secondsElapsed", secondsElapsed)  
 editor.apply()  
 }  
}

В onResume() создаётся AsyncTask, в котором переопределены два метода doInBackground() и onProgressUpdate(). В первом методе определён цикл, который работает, пока AsyncTask не остановлен и раз в секунду вызывает метод publishProgress(). Вызов publishProgress() обрабатывается в методе onProgressUpdate(), который изменяет значение textView. В методе onPause() AsyncTask останавливается.

Исходный код решения с Kotlin Coroutines:

class MainActivity : AppCompatActivity() {  
 companion object {  
 var secondsElapsed: Int = 0  
 }  
 private val viewScope = *CoroutineScope*(Dispatchers.Main)  
 private lateinit var sharedPref: SharedPreferences  
 private lateinit var timer: Job

private fun startCoroutineTimer(repeatMillis: Long = 0, action: () -> Unit) =  
 viewScope.*launch* **{** while (true) {  
 *delay*(repeatMillis)  
 action()  
 }  
 **}** private fun getContinueTimer() = startCoroutineTimer(1000) **{** this.findViewById<TextView>(R.id.*textSecondsElapsed*).*text* = "Seconds elapsed: " + ++secondsElapsed  
 **}** override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
 }  
  
  
 override fun onResume() {  
 super.onResume()  
 timer = getContinueTimer()  
 sharedPref = *applicationContext*.getSharedPreferences("myPrefs", *MODE\_PRIVATE*)  
 secondsElapsed = sharedPref.getInt("secondsElapsed", 0)  
 }  
  
 override fun onPause() {  
 super.onPause()  
 viewScope.*cancel*()  
 val editor = sharedPref.edit()  
 editor.putInt("secondsElapsed", secondsElapsed)  
 editor.apply()  
 }  
}

В onResume() вызывается метод getContinueTimer(), в котором вызывается метод startCoroutineTimer() и создаётся корутина. В корутине выполняется бесконечный цикл, который раз в секунду обновляет textView. В методе onPause() действие корутины останавливается.

**2.2. Задача 2.** Загрузка картинки в фоновом потоке (AsyncTask)

Создайте приложение, которое скачивает картинку из интернета и размещает ее в ImaveView в Activity. За основу возьмите код со StackOverflow.

Листинг:

MainFragment.kt:

class MainFragment : Fragment() {  
  
 companion object {  
 fun newInstance() = MainFragment()  
 }  
  
 private var imgUrl: String = "https://namobilu.com/u/img/ib/003/195/195003-1.jpg"  
  
 private lateinit var viewModel: MainViewModel  
  
 override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
 ): View {  
 return inflater.inflate(R.layout.*fragment\_main*, container, false)  
 }  
  
 override fun onActivityCreated(savedInstanceState: Bundle?) {  
 super.onActivityCreated(savedInstanceState)  
 viewModel = ViewModelProviders.of(this).get(MainViewModel::class.*java*)  
 img.setOnClickListener **{** viewModel.download(imgUrl)  
 **}** viewModel.imgBitmap.observe(*viewLifecycleOwner*, *Observer* **{  
 it**?.*let* **{** img.setImageBitmap(**it**)  
 **}  
 }**)  
  
 }  
}

MainViewModel.kt:

class MainViewModel : ViewModel() {  
 private lateinit var asyncTask: DownloadAsyncTask  
  
 val imgBitmap = MutableLiveData<Bitmap>()  
  
 inner class DownloadAsyncTask :  
 AsyncTask<String, Void, Bitmap>() {  
  
 override fun doInBackground(vararg urls: String): Bitmap {  
 val url = urls[0]  
 var bitmap: Bitmap? = null  
 try {  
 val inputStream = URL(url).openStream()  
 bitmap = BitmapFactory.decodeStream(inputStream)  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 return bitmap ?: Bitmap.createBitmap(1, 1, Bitmap.Config.ARGB\_8888)  
 }  
  
 override fun onPostExecute(result: Bitmap) {  
 super.onPostExecute(result)  
 imgBitmap.*value* = result  
 }  
 }  
  
 fun download(url: String) {  
 asyncTask = DownloadAsyncTask()  
 asyncTask.execute(url)  
 }

override fun onCleared() {  
 if (::asyncTask.*isInitialized*)  
 asyncTask.cancel(true)  
 super.onCleared()  
 }  
  
}

Загрузка изображения с заданным URL осуществляется с помощью URL.openStream() и BitmapFactory.decodeStream() в фоновом потоке, созданном с помощью AsyncTask. AsyncTask создаётся при нажатии на ImageView. UI изменяется после того, как AsyncTask заканчивает свою работу.

Внешний вид приложения до нажатия на картинку:

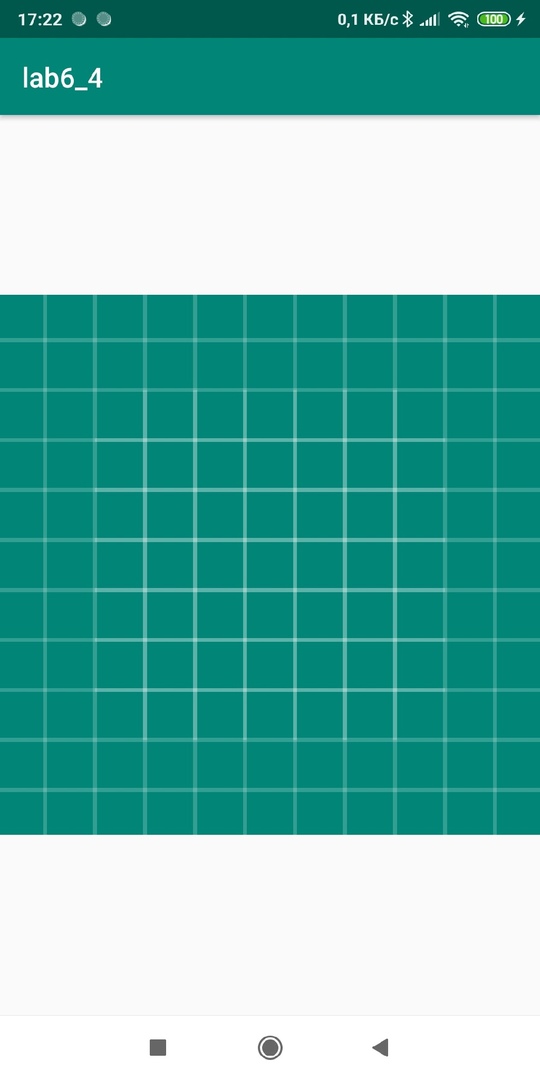


Рис.1. Внешний вид приложения до нажатия на картинку

Внешний вид приложения после нажатия на картинку:

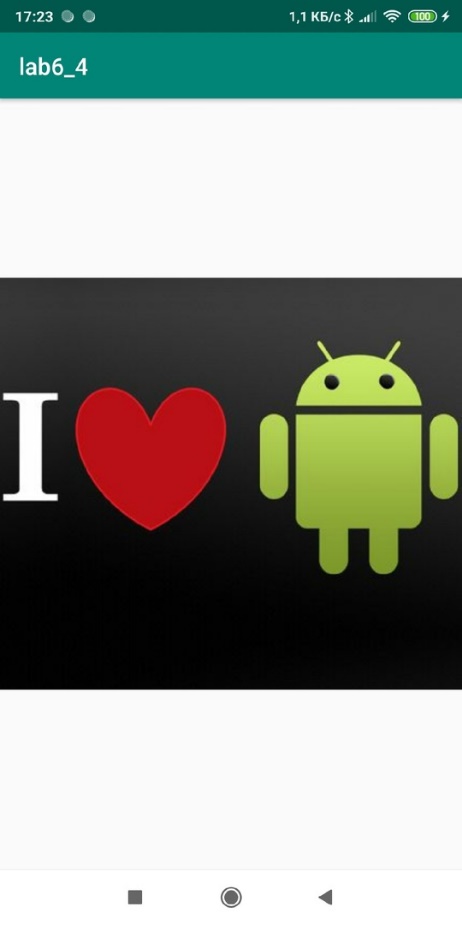


Рис.2. Внешний вид приложения после нажатия на картинку

**2.3. Задача 3.** Загрузка картинки в фоновом потоке (Kotlin Coroutines)

Перепишите предыдущее приложение с использованием Kotlin Coroutines.

Листинг:

MainFragment.kt:

class MainFragment : Fragment() {  
  
 companion object {  
 fun newInstance() = MainFragment()  
 }  
  
 private var imgUrl: String = "https://namobilu.com/u/img/ib/003/195/195003-1.jpg"  
  
 private lateinit var viewModel: MainViewModel  
  
 override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
 ): View {  
 return inflater.inflate(R.layout.*fragment\_main*, container, false)  
 }  
  
 override fun onActivityCreated(savedInstanceState: Bundle?) {  
 super.onActivityCreated(savedInstanceState)  
 viewModel = ViewModelProviders.of(this).get(MainViewModel::class.*java*)  
 img.setOnClickListener **{** viewModel.download(imgUrl)  
 **}** viewModel.imageBitmap.observe(*viewLifecycleOwner*, *Observer* **{  
 it**?.*let* **{** img.setImageBitmap(**it**)  
 **}  
 }**)  
  
 }  
}

MainViewModel.kt:

class MainViewModel : ViewModel() {  
 val imageBitmap = MutableLiveData<Bitmap>()  
  
 fun download(url: String) {  
 GlobalScope.*launch*(Dispatchers.IO) **{** val img = *async*(Dispatchers.IO) **{** java.net.URL(url).openStream().*use* **{** return@async BitmapFactory.decodeStream(**it**)  
 **}  
 }**.await()  
  
 *withContext*(Dispatchers.Main) **{** imageBitmap.*value* = img  
 **}  
 }** }  
  
}

Загрузка изображения с заданным URL осуществляется так же, как и в предыдущем задании - с помощью URL.openStream() и BitmapFactory.decodeStream(), но на этот раз используются Kotlin Coroutines. Async и await позволяют дождаться загрузки изображения, после чего изменяется UI.

Внешний вид приложения до нажатия на картинку:

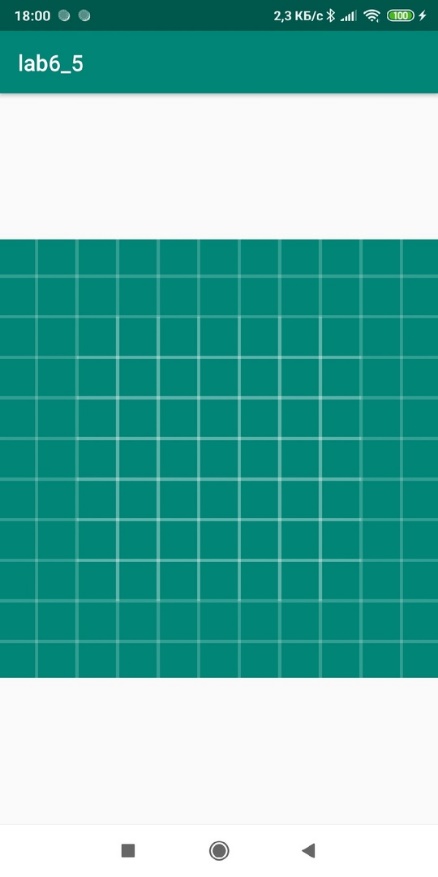


Рис.3. Внешний вид приложения до нажатия на картинку

Внешний вид приложения после нажатия на картинку:

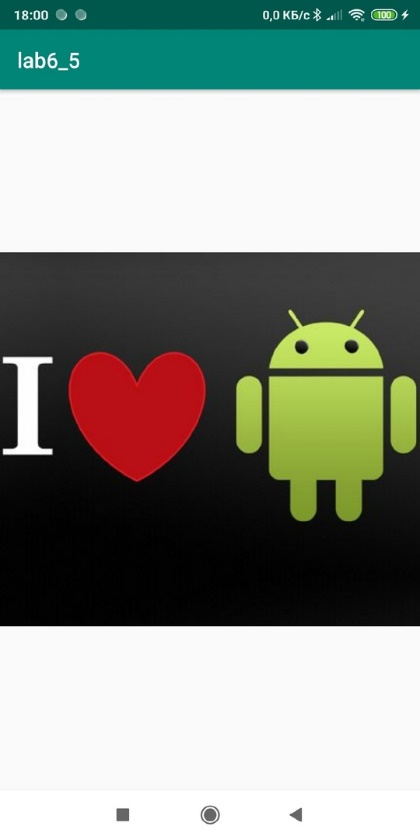


Рис.4. Внешний вид приложения после нажатия на картинку

**2.4. Задача 4.** Использование сторонних библиотек

Многие "стандартные" задачи имеют "стандартные" решения. Задача скачивания изображения в фоне возникает настолько часто, что уже сравнительно давно решение этой задачи занимает всего лишь несколько строчек. Убедитесь в этом на примере одной (любой) библиотеки Glide, picasso или fresco.

Листинг:

MainFragment.kt:

class MainFragment : Fragment() {  
  
 companion object {  
 fun newInstance() = MainFragment()  
 }  
  
 private var imgUrl: String = " https://namobilu.com/u/img/ib/003/195/195003-1.jpg"  
  
 override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
 ): View? {  
 return inflater.inflate(R.layout.*fragment\_main*, container, false)  
 }  
  
 override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
 super.onViewCreated(view, savedInstanceState)  
 image.setOnClickListener **{** Picasso.get().load(imgUrl).into(image)  
 **}** }  
  
 override fun onStop() {  
 super.onStop()  
 Picasso.get().cancelRequest(image)  
 }  
   
  
}

Использована библиотека Picasso. Изображение загружается в ImageView при помощи Picasso.get().load(imgUrl).into(image). В методе onStop() загрузка отменяется.

Внешний вид приложения до нажатия на картинку:

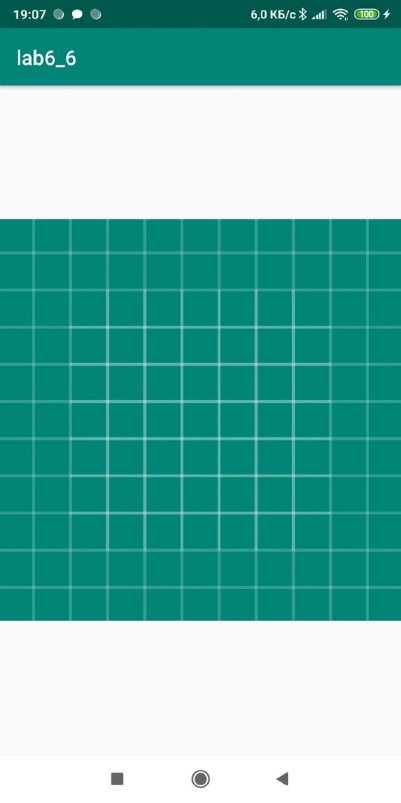


Рис.5. Внешний вид приложения до нажатия на картинку

Внешний вид приложения после нажатия на картинку:

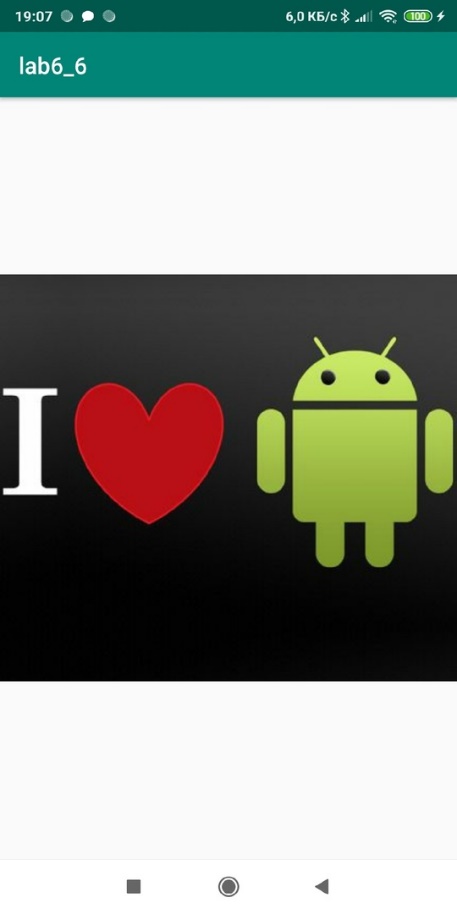


Рис.6. Внешний вид приложения после нажатия на картинку

**3. Вывод**

В ходе выполнения данной лабораторной работы были получены практические навыки разработки многопоточных приложений, а также освоены 3 основные группы API для разработки многопоточных приложений: Kotlin Coroutines, AsyncTask и Java Threads.