Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Отчёт по лабораторной работе № 6

Дисциплина: Проектирование мобильных приложений

Тема: Многопоточные Android приложения.

Выполнил студент гр. 3530901/90201 _____З.А. Фрид
(подпись)

Принял преподаватель _____ А.Н. Кузнецов
(подпись)

“____” _____ 2021 г.

Санкт-Петербург

2021

Оглавление

Ссылка на проекты

https://github.com/zina-frid/AndroidLabs/tree/main/projects/lab6

Цели

- Получить практические навыки разработки многопоточных приложений:
  1. Организация обработки длительных операций в background (worker) thread:
     - Запуск фоновой операции (Coroutine/ExecutionService/Thread)
     - Остановка фоновой операции (Coroutine/ExecutionService/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. ExecutionService
  3. Java Threads

Задачи

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.

Указания

К моменту выполнения работы Вам должно быть известно, как минимум, 4 принципиально разных подхода к решению задачи. В отчете должны появиться, как минимум, 3 решения:

1. С помощью Java Threads (фактически, это оригинальный код, однако на этот раз необходимо убедиться, что потоки запускаются и останавливаются в определенные моменты времени. Опишите эти моменты времени в

отчете. Обратите внимание, что потоки не должны существовать, когда приложение не отображается на экране).

2. С помощью ExecutionService. Это та же программа, что и в предыдущем пункте, только в этой версии программы потоки не будут создаваться вручную. Потоки могут существовать, когда приложение находится в background, однако нужно убедиться, что этот поток ничего не делает.
3. С помощью Kotlin Coroutines. В качестве знакомства с корутинами рекомендуется выполнить шаги 1-6 (включительно) из codelab: https://codelabs.developers.google.com/codelabs/kotlin-coroutines

Во всех случаях необходимо обращать внимание на то, когда запускаются потоки/задачи и когда они останавливаются: приложение не должно тратить впустую ресурсы ОС, когда оно не отображается на экране.

Как и в любом другом приложении, необходимо уделить особое внимание поведению приложения в ситуации перезапуска Activity (например, в результате поворота экрана). Не должно появляться более одного активного потока/задачи для подсчета времени.

Задача 2. Загрузка картинки в фоновом потоке

Создайте приложение, которое скачивает картинку из интернета и размещает ее в ImaveView в Activity. Используйте ExecutorService для решения этой задачи.

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

Перепишите предыдущее приложение с использованием Kotlin Coroutines.

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.

Задача 1

Использовалась реализация "не секундомер" с SharedPreferences, получившееся в результате выполнения Лабораторной работы №2.

Задача 1.1. Threads

Как сказано в указаниях, фактически, получился оригинальный код, однако теперь мы запускаем и останавливаем потоки в определенные моменты времени.

Листинг 1.1 MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    var secondsElapsed: Int = 0
    lateinit var textSecondsElapsed: TextView
    private lateinit var sharedPref: SharedPreferences
    private lateinit var backgroundThread: Thread

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        textSecondsElapsed = findViewById(R.id.timer)
        sharedPref = getSharedPreferences(SECONDS, Context.MODE_PRIVATE)
    }

    override fun onStart() {
        secondsElapsed = sharedPref.getInt(SECONDS, 0)
        Log.d("MainActivity", "OnStart: seconds = $secondsElapsed")
        backgroundThread = Thread {
            try {
                while (!Thread.currentThread().isInterrupted) {
                    Log.d("MainActivity", "${Thread.currentThread()} is
iterating")
                    Thread.sleep(1000)
                    textSecondsElapsed.post {
                        textSecondsElapsed.text = "${secondsElapsed++}"
                    }
                }
            } catch (e: InterruptedException) {
                Thread.currentThread().interrupt()
            }
        }
        backgroundThread.start()
        super.onStart()
    }

    override fun onStop() {
        val editor = sharedPref.edit()
        editor.putInt(SECONDS, secondsElapsed)
        editor.apply()
        Log.d("MainActivity", "OnStop: seconds = $secondsElapsed")
        backgroundThread.interrupt()
        super.onStop()
    }

    companion object {
        const val SECONDS = "Seconds"
```

```
    }
}
```

Поскольку приложение не должно тратить впустую ресурсы ОС, когда оно не отображается на экране, используется пара callback методов onStart()/onStop() для работы с потоками. Во-первых, была удалена переменная, выполнявшая роль флага, находится ли приложение на экране или нет. Затем добавлена переменная backgroundThread для потока.

В методе onStart() инициализируется и запускается поток, а в методе onStop() он прерывается. Строка while(true) была изменена на while (!Thread.currentThread().isInterrupted), чтобы приложение каждый раз не создавало новый поток при попадании в состояние Stopped. Добавлен блок try…catch, так как при установленном флаге interrupted метод Thread.sleep(1000) кинет исключение InterruptedException, которое мы должны обработать.

Теперь посмотрим логи:

```
2021-11-21 12:50:18.614 3779-3779/com.zinafrid.threads D/MainActivity: OnStart: seconds = 0
2021-11-21 12:50:18.618 3779-3805/com.zinafrid.threads D/MainActivity: Thread[Thread-2,5,main] is iterating
2021-11-21 12:50:19.619 3779-3805/com.zinafrid.threads D/MainActivity: Thread[Thread-2,5,main] is iterating
2021-11-21 12:50:20.621 3779-3805/com.zinafrid.threads D/MainActivity: Thread[Thread-2,5,main] is iterating
2021-11-21 12:50:21.623 3779-3805/com.zinafrid.threads D/MainActivity: Thread[Thread-2,5,main] is iterating
2021-11-21 12:50:22.624 3779-3805/com.zinafrid.threads D/MainActivity: Thread[Thread-2,5,main] is iterating
2021-11-21 12:50:23.468 3779-3779/com.zinafrid.threads D/MainActivity: OnStop: seconds = 4
2021-11-21 12:50:26.426 3779-3779/com.zinafrid.threads D/MainActivity: OnStart: seconds = 4
2021-11-21 12:50:26.455 3779-3833/com.zinafrid.threads D/MainActivity: Thread[Thread-3,5,main] is iterating
2021-11-21 12:50:27.456 3779-3833/com.zinafrid.threads D/MainActivity: Thread[Thread-3,5,main] is iterating
2021-11-21 12:50:28.458 3779-3833/com.zinafrid.threads D/MainActivity: Thread[Thread-3,5,main] is iterating
2021-11-21 12:50:28.874 3779-3779/com.zinafrid.threads D/MainActivity: OnStop: seconds = 6
2021-11-21 12:50:31.488 3779-3779/com.zinafrid.threads D/MainActivity: OnStart: seconds = 6
2021-11-21 12:50:31.521 3779-3849/com.zinafrid.threads D/MainActivity: Thread[Thread-4,5,main] is iterating
2021-11-21 12:50:32.522 3779-3849/com.zinafrid.threads D/MainActivity: Thread[Thread-4,5,main] is iterating
2021-11-21 12:50:33.523 3779-3849/com.zinafrid.threads D/MainActivity: Thread[Thread-4,5,main] is iterating
2021-11-21 12:50:34.525 3779-3849/com.zinafrid.threads D/MainActivity: Thread[Thread-4,5,main] is iterating
2021-11-21 12:50:35.395 3779-3779/com.zinafrid.threads D/MainActivity: OnStop: seconds = 9
```

Рис. 1-1

Мы видим, что при вызове onStop() старый поток завершается, а при вызове onStart() запускается новый, то есть работает только один поток.

Задача 1.2. ExecutorService

Это та же программа, что и в предыдущем пункте, однако для решения используется интерфейс ExecutorService.

В документации говорится, что «создание потоков обходится дорого, поэтому вам следует создавать пул потоков только один раз при инициализации вашего приложения. Обязательно сохраните экземпляр ExecutorService в своем Application классе или в контейнере для внедрения зависимостей».

Создадим пул потоков на 1 поток в классе Application, который мы можем использовать для выполнения фоновых задач.

Листинг 1.2.1 MainApplication.kt

```
class MainApplication: Application() {
    val threadPool: ExecutorService = Executors.newFixedThreadPool(1)
}
```

В классе MainActivity в методе onStart() запускаем поток, в котором будем исполнять наш процесс, с помощью метода submit(). Метод submit() запускает поток и возвращает экземпляр класса Future, который, согласно документации, представляет результат асинхронного вычисления.

В методе onStop() у объекта Future вызывается метод cancel(boolean mayInterruptIfRunning), то есть попытка отменить выполнение этой задачи. Эта попытка не удастся, если задача уже завершена, уже отменена или не может быть отменена по какой-либо другой причине. В случае успеха, и эта задача не была запущена при cancel вызове, эта задача никогда не должна запускаться. Если задача уже запущена, то mayInterruptIfRunning параметр определяет, следует ли прервать поток, выполняющий эту задачу, при попытке остановить задачу. Я вызываю метод cancel с аргументом true, и если всё хорошо, то данный поток должен завершиться.

Для того, чтобы получить доступ к нашему пулу тредов используем Application Context.

Листинг 1.2.2 MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    var secondsElapsed: Int = 0
    lateinit var textSecondsElapsed: TextView
    private lateinit var sharedPref: SharedPreferences
    private lateinit var background: Future<*>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        textSecondsElapsed = findViewById(R.id.timer)
        sharedPref = getSharedPreferences(SECONDS, Context.MODE_PRIVATE)
    }

    override fun onStart() {
        secondsElapsed = sharedPref.getInt(SECONDS, 0)
        Log.d("MainActivity", "OnStart: seconds = $secondsElapsed")
        val executor = (applicationContext as MainApplication).threadPool
        background = executor.submit {
            while (!executor.isShutdown) {
                Log.d("MainActivity", "${Thread.currentThread()} is
iterating")
                Thread.sleep(1000)
                textSecondsElapsed.post {
                    textSecondsElapsed.text = "${secondsElapsed++}"
                }
            }
        }
        super.onStart()
    }

    override fun onStop() {
        val editor = sharedPref.edit()
        editor.putInt(SECONDS, secondsElapsed)
        editor.apply()
        Log.d("MainActivity", "OnStop: seconds = $secondsElapsed")
        background.cancel(true)
        super.onStop()
    }

    companion object {
        const val SECONDS = "Seconds"
    }
}
```

Теперь посмотрим логи, из них видно, что потоки выполняются в одном пуле, а пул не создается каждый раз при вызове onStart заново, как это было в мой предыдущей реализации:

```
2021-12-17 14:33:21.540 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStart: seconds = 0
2021-12-17 14:33:21.548 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:22.550 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:23.552 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:24.554 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:25.555 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:44.583 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:45.586 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:46.251 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStop: seconds = 24
2021-12-17 14:33:49.198 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStart: seconds = 24
2021-12-17 14:33:49.199 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:50.238 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:51.240 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:52.241 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:53.243 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:53.660 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStop: seconds = 28
2021-12-17 14:33:56.411 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStart: seconds = 28
2021-12-17 14:33:56.411 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:57.462 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:58.255 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStop: seconds = 29
2021-12-17 14:33:58.354 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStart: seconds = 29
2021-12-17 14:33:58.355 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:33:59.356 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:34:00.358 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:34:01.358 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
2021-12-17 14:34:01.402 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStop: seconds = 32
2021-12-17 14:34:01.514 6161-6161/com.zinafrid.executionservice D/MainActivity: OnStart: seconds = 32
2021-12-17 14:34:01.515 6161-6187/com.zinafrid.executionservice D/MainActivity: Thread[pool-2-thread-1,5,main] is iterating
```

Рис. 1-2

Задача 1.3. Coroutines

Корутина запускается с помощью lifecycleScope и его метода launchWhenStarted(), то есть только в состоянии started каждую секунду увеличивается значение счетчика. В цикле while используем isActive. Аналогом метода sleep() у корутин является delay().

При разрушении activity (метод onDestroy) корутина завершается.

Листинг 1.3 MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    var secondsElapsed: Int = 0
    lateinit var textSecondsElapsed: TextView
    private lateinit var sharedPref: SharedPreferences

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        textSecondsElapsed = findViewById(R.id.timer)
        sharedPref = getSharedPreferences(SECONDS, Context.MODE_PRIVATE)

        val job = lifecycleScope.launchWhenStarted {
            Log.d("MainActivity", "Coroutine is launched")
            while (isActive) {
                Log.d("MainActivity", "Coroutine is working")
                delay(1000)
                textSecondsElapsed.text = "${secondsElapsed++}"
```

```
            }
        }

        job.invokeOnCompletion {
            Log.d("MainActivity", "Coroutine is completed")
        }
    }

    override fun onStart() {
        secondsElapsed = sharedPref.getInt(SECONDS, 0)
        Log.d("MainActivity", "OnStart: seconds = $secondsElapsed")
        super.onStart()
    }

    override fun onStop() {
        val editor = sharedPref.edit()
        editor.putInt(SECONDS, secondsElapsed)
        editor.apply()
        Log.d("MainActivity", "OnStop: seconds = $secondsElapsed")
        super.onStop()
    }

    companion object {
        const val SECONDS = "Seconds"
    }
}
```

Для демонстрации в логах работы корутины используется интерфейс Job, хотя в самой работе и при заданных условиях задачи его наличие необязательно.

Теперь посмотрим на логи:

```
2021-12-17 14:49:28.474 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:29.478 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:30.461 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStop: seconds = 36
2021-12-17 14:49:33.948 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStart: seconds = 36
2021-12-17 14:49:33.949 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:34.952 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:35.955 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:36.536 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStop: seconds = 39
2021-12-17 14:49:43.016 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStart: seconds = 39
2021-12-17 14:49:43.016 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:44.058 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:44.381 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStop: seconds = 41
2021-12-17 14:49:44.385 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is completed
2021-12-17 14:49:44.461 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStart: seconds = 41
2021-12-17 14:49:44.461 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is launched
2021-12-17 14:49:44.462 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:45.464 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:46.468 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:47.472 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
2021-12-17 14:49:48.361 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStop: seconds = 44
2021-12-17 14:49:48.364 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is completed
2021-12-17 14:49:48.435 6615-6615/com.zinafrid.coroutines D/MainActivity: OnStart: seconds = 44
2021-12-17 14:49:48.438 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is launched
2021-12-17 14:49:48.438 6615-6615/com.zinafrid.coroutines D/MainActivity: Coroutine is working
```

Рис. 1-3

Из логов видно, что coroutine запускается, работает, затем, когда activity выходит из состояния started, счет прекращается. Лог в методе onStop() возвращает сохраняемое количество секунд. Потом при возвращении activity в состояние started, работа coroutine продолжается, а лог в методе onStart() возвращает количество сохраненных секунд. После разрушения activity, lifecycleScope завершает coroutine, и мы видим соответствующее сообщение в логах “Coroutine is completed”.

Задача 2

В данной задаче необходимо создать приложение, которое скачивает картинку из интернета и размещает ее в ImaveView в Activity. Сначала я реализовала приложение, которое в onCreate() проверяет, не пустой ли imageView, и если пустой, то скачивает картинку. Таким образом, при запуске приложения почти сразу видна картинка, поэтому я подумала, что для того, чтобы показать, что сначала imageView пустой, а потом заполняется изображением, стоит добавить, например, кнопку, при нажатии на которую происходило бы скачивание и размещение картинки.

Как и при решении задачи с таймером с помощью ExecutorService, создаю пул потоков на 1 поток в классе Application, который мы можем использовать для выполнения фоновых задач.

Листинг 2.1 MainApplication.kt

```
class MainApplication: Application() {
    val threadPool: ExecutorService = Executors.newFixedThreadPool(1)
}
```

MainViewModel

Для того, чтобы загружать картинку используем ViewModel, которая и будет запускать поток для наших целей. ViewModel — это класс, который отвечает за подготовку и управление данными для файла Activity или Fragment.

В классе MainViewModel есть функция downloadImage(), которая качает картинку. Bitmap – это класс, предназначенный для работы с растровыми изображениями. BitmapFactory – класс, который позволяет создать объект Bitmap из файла, потока или байтового массива, с его помощью входной поток преобразуется в картинку. Метод postValue перенаправляет вызов в UI поток.

Листинг 2.2 MainViewModel.kt

```
class MainViewModel(application: Application) :
AndroidViewModel(application) {

    val mutableLiveData = MutableLiveData<Bitmap>()
    private val executor: ExecutorService =
getApplication<MainApplication>().threadPool

    fun downloadImage(url: String) {
        executor.execute {
            val stream = URL(url).openConnection().getInputStream()
            val bitmap = BitmapFactory.decodeStream(stream)
            mutableLiveData.postValue(bitmap)
        }
    }
}
```

Поскольку нам нужно использовать контекст внутри вашей ViewModel, я использую AndroidViewModel, потому что она содержит контекст приложения. Чтобы получить контекстный вызов, то есть получить экземпляр MainApplication, используем метод getApplication () в классе mainActivity.kt.

MainActivity

При нажатии на кнопку происходит вызов функции downloadImage() из viewModel, то есть картинка скачивается.

С помощью метода observe() наша activity подписывается на mutableLiveData (объект типа MutableLiveData<Bitmap>()). После изменения данных во viewModel поле imageView заполняется картинкой.

Листинг 2.3 MainActivity.kt

```
class MainActivity : AppCompatActivity() {

    private lateinit var imageView: ImageView
    lateinit var btn : Button
    private val viewModel: MainViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        setContentView(R.layout.activity_main)
        btn = findViewById(R.id.downloader)
        btn.setOnClickListener {
            viewModel.downloadImage(MY_URL)
        }
        imageView = findViewById(R.id.imageView)

        viewModel.mutableLiveData.observe(this) {
             imageView.setImageBitmap(it)
        }
    }

    companion object {
        private const val MY_URL = "https://www.meme-
arsenal.com/memes/ecdaf55fffca12b0feaca5b3431acdff.jpg"
    }
}
```

Листинг 2.4 main_activity.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/downloader"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="@string/download"
        android:layout_marginTop="100dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="350dp"
        app:layout_constraintTop_toBottomOf="@id/downloader"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        android:layout_gravity="center"
        android:contentDescription="@string/image"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Стоит отметить, что при попытке скачать картинку из интернета появляется сообщение об отказе в доступе, поэтому в манифесте необходимо прописать следующую строку:

Листинг 2.4

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

Задача 3

Необходимо переписать предыдущее приложение с использованием Kotlin Coroutines. Вместо метода postValue(), воспользуемся Dispatchers.Main, с помощью которого корутина переключится на главный поток.

MainViewModel

Листинг 3.1 MainViewModel.kt

```
class MainViewModel : ViewModel(){

    val mutableLiveData = MutableLiveData<Bitmap>()

    fun downloadImage(url: String) {
        viewModelScope.launch(Dispatchers.IO){
            val stream = URL(url).openConnection().getInputStream()
            val bitmap = BitmapFactory.decodeStream(stream)
            withContext(Dispatchers.Main) {
                mutableLiveData.value = bitmap
            }
        }
    }
}
```

Задача 4

Используем библиотеку Picasso для загрузки изображения.

Листинг 3.1 MainActivity.kt

```
class MainActivity : AppCompatActivity() {

    private lateinit var imageView: ImageView
    lateinit var btn : Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        imageView = findViewById(R.id.imageView)
        btn = findViewById(R.id.downloader)
        btn.setOnClickListener {
            Picasso.get().load(MY_URL).into(imageView)
        }
    }

    companion object {
        private const val MY_URL = "https://www.meme-
arsenal.com/memes/ecdaf55fffca12b0feaca5b3431acdff.jpg"
    }
}
```

Вывод

В ходе выполнения лабораторной работы мы познакомились с многопоточными Android приложениями. Получены практические навыки разработки многопоточных приложений: запуск фоновой операции, остановка фоновой операции, публикация данных из background (worker) thread в main (ui) thread. Освоены три основные группы API для разработки многопоточных приложений: Java Threads, ExecutionService и Kotlin Coroutines.

В первом пункте данной работы мы научились эффективно использовать ресурсы операционной системы. Попробовали исправить некорректно написанную программу «не секундомер» тремя способами: с помощью Java Threads, ExecutionService и Kotline Coroutine. Мне кажется, что для решения данной задачи больше подходят корутины, потому что есть специальный удобный конструктор, который знает про lifecycles.

Во второй части работы было написано приложение, которое качает картинку из интернета и размещает в поле ImageView. Использовалось три разных способа: с использованием ExecutorService, Kotlin Coroutines, а также стандартной библиотеки Picasso.

Соберем полученные знания об этих API и заполним таблицу с информацией о том, как достигается каждая из задач в разных группах API. Таблица приведена ниже.

	Запуск фоновой задачи	Остановка фоновой задачи	Передача данных из фонового потока в UI поток
Java Threads	Создаётся объект Thread, в конструкторе у которого указывается Runnable. После этого можно запускать новый поток с помощью метода start().	У потока есть метод stop(), но использовать его не рекомендуется, поскольку он оставляет приложение в неопределённом состоянии. Обычно используют подход с методом interrupt(). Данный метод выставляет "internal flag called interrupt status". То есть у каждого потока есть внутренний флаг, недоступный напрямую. Но у нас есть методы для взаимодействия с этим флагом. Про флаг isInterrupted стоит отметить, что если мы поймали InterruptedException, флаг isInterrupted сбрасывается, и тогда isInterrupted будет возвращать false.	Android предлагает несколько способов доступа к UI потоку из других потоков: • Activity.runOnUiThread(Runnable) • View.post(Runnable) • View.postDelayed(Runnable, long)
Execution Service	У ExecutionSrvice есть метод execute(). Метод Executor.execute() принимает Runnable и выполняет заданную команду когда-нибудь в будущем.	ExecutorService может быть отключен (shut down), что приведет к отклонению новых задач. Предусмотрены два разных метода завершения работы ExecutorService. Метод shutdown() позволит выполнить ранее отправленные задачи до	Такие же, как и у Java Threads: • Activity.runOnUiThread (Runnable) • View.post(Runnable) • View.postDelayed(Runnable, long)

	Ещё есть метод submit(). Метод submit расширяет базовый метод Executor.execute (java.lang.Runnable), создавая и возвращая Future, который можно использовать для отмены выполнения и/или ожидания завершения.	завершения, в то время как метод shutdownNow() предотвращает запуск ожидающих задач и пытается остановить выполнение текущих задач. У экземпляров класса Future есть метод cancel(Boolean), который отменяет выполняемую задачу, поэтому если при запуске использовался метод submit(), то можно вызвать cancel(true) у возвращаемого Future.	
Kotlin Coroutines	Для начала стоит определить, так называемую, область действия для новых корутин или scope. А затем используется метод launch, который асинхронно запускает корутину. В задаче с секундомером был использован lifecycleScope, который определяется для	При запуске launch возвращает экземпляр Job, у него есть метод cancel(), с помощью которого можно отменить выполнение. Также, как было написано, при использовании некоторых scope корутина завершится, если жизненный цикл объекта, в котором она запущена, завершён.	Если используется lifecycleScope.launch{}, то блок выполняется в main (UI) потоке. Так же с помощью Dispathchers можно переключать контексты (потоки). В задаче со скачиванием использовалось withContext(Dispatchers.Main) для передачи данных из фонового потока в UI поток. В целом, можно воспользоваться и теми же методами, что и у Java Threads.

	каждого Lifecycle объекта. Любая корутина, запущенная в этой области видимости, отменяется при Lifecycle уничтожении. В задаче со скачиванием использовался ViewModelScope, который определяется для каждого ViewModel в приложении. Любая корутина, запущенная в этой области видимости, автоматически отменяется, если ViewModel очищается.		

Список источников

1. https://github.com/andrei-kuznetsov/android-lectures
2. https://developer.android.com/guide/background/threading#creating-multiple-threads
3. https://developer.android.com/reference/java/util/concurrent/Future
4. https://developer.android.com/guide/background/threading
5. https://developer.android.com/kotlin/coroutines
6. https://developer.android.com/topic/libraries/architecture/coroutines
7. https://dzone.com/articles/how-to-handle-the-interruptedexception
8. https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/525-urok-2-livedata.html
9. https://developer.android.com/reference/androidx/lifecycle/ViewModel
10. https://developer.android.com/reference/android/arch/lifecycle/ViewModel
11. https://developer.android.com/reference/android/arch/lifecycle/AndroidViewModel
12. https://stackoverflow.com/questions/44148966/androidviewmodel-vs-viewmodel
13. https://developer.android.com/reference/android/content/Context#public-constructors
14. https://square.github.io/picasso/