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

Отчёт по лабораторной работе № 5

Дисциплина: Проектирование мобильных приложений

Тема: Мультипоточные приложения

Выполнил студент гр. 3530901/90201 _____ Д.Е. Бакин
(подпись)

Принял преподаватель _____ А.Н. Кузнецов
(подпись)

“____” _____ 2021 г.

Санкт-Петербург

2021

Репозиторий GitHub:

https://github.com/donebd/Android_spbstu2021/tree/main/labs/lab5

1. 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

2. Задачи

- Альтернативные решения задачи "не секундомер" из Лаб. 2. Реализация на: Java Threads, ExecutionService, Kotlin Corutines.
- Загрузка картинки в фоновом потоке. Реализация на: ExecutionService, Kotlin Corutines, сторонняя библиотека.

3. Ход работы

Задача 1. Java Threads

Преобразим наше приложение из второй лабораторной работы. Возьмем реализацию с сохранением данных в onSave/RestoreInstantState. Нам необходимо останавливать background поток, когда мы не считаем наши секунды, возьмем парные колбэки onStart/onStop. Для этого будем использовать метод потока interrupt(). Он выставляет флаг прерывания у потока, и метод потока isInterrupted позволяет нам получить значение флага, после чего он сбрасывается в false. Но если есть блокирующие вызовы внутри потока, например sleep(), и наш поток был прерван во время сна, то наш поток кинет исключение InterruptedExeption, которое нам нужно отрабатывать, для этого обернем наш sleep блоком try catch.
Будем прерывать наш поток в методе onStop, после чего он будет останавливаться, а в методе onStart будем вызывать новый поток, а сборщик мусора соберет наши завершенные потоки.

Также в предыдущей работе мы не учитывали особенность погрешности нашей реализации подсчета времени. По сути, мы считаем количество методов sleep(1000), но мы можем сделать так, что например:
Войдем в приложение подождем 900ms, свернем, снова откроем, подождем 900ms, и так можем делать до бесконечности, и наш таймер все время будет на отметке 0 секунд. Это, не говоря о том, что сам по себе метод sleep(n), не всегда отсчитывает ровно n ms.
Предложенное решение: давайте оставим наш метод sleep(1000), но будем обновлять время путем подсчета реального времени, используя метод System.currentTimeMillis(). Мы в каждом методе updateTime() будем добавлять разницу между замерами времени, которые мы будем делать (первый внутри onStart), а после в самой функции updateTime(). А также будем перед прерыванием потока в методе onStop, будем вызывать updateTime(), чтобы не потерять наше время, которое прошло с момента sleep предыдущего цикла. Таким образом мы решим нашу проблему с накапливающейся ошибкой до какого-то порядка.

Листинг реализации:

Листинг 1.1 ThreadRealization

```
class ThreadRealization : AppCompatActivity() {

    private val MY_LOG = this.javaClass.name
    private val SAVE_KEY = "Key"

    private var secondsElapsed: Double = 0.0
    private lateinit var textSecondsElapsed: TextView
    private lateinit var backgroundThread: Thread
    private var startTime: Long = 0
    private var currentTime: Long = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        textSecondsElapsed = findViewById(R.id.textSecondsElapsed)
    }

    override fun onStart() {
        super.onStart()
        backgroundThread = getThread()
        backgroundThread.start()
        Log.i(MY_LOG, "$backgroundThread is started")
        startTime = System.currentTimeMillis()
    }

    override fun onStop() {
        super.onStop()
        updateTime()
        backgroundThread.interrupt()
    }

    private fun getThread(): Thread = Thread {
        while (!Thread.currentThread().isInterrupted) {
            try {
                Thread.sleep(1000)
            } catch (e: InterruptedException) {
                Log.i(MY_LOG, "${Thread.currentThread()} is stopped")
                return@Thread
            }
            Log.i(MY_LOG, "${Thread.currentThread()} iterating")
            updateTime()
        }
    }

    private fun updateTime() {
        currentTime = System.currentTimeMillis()
        secondsElapsed += (currentTime - startTime) / 1000.0
        startTime = currentTime
        Log.i(MY_LOG, "$secondsElapsed")

        textSecondsElapsed.post {
            textSecondsElapsed.text = resources.getQuantityString(
                R.plurals.secCounter,
                secondsElapsed.toInt(),
                secondsElapsed.toInt()
            )
        }
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
```

```
        outState.run {
            putDouble(SAVE_KEY, secondsElapsed)
        }
    }

    override fun onRestoreInstanceState(savedInstanceState: Bundle) {
        super.onRestoreInstanceState(savedInstanceState)
        savedInstanceState.run {
            secondsElapsed = getDouble(SAVE_KEY)
        }
    }
}
```

Посмотрим на логи работы нашего приложения:

```
ab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] is started
pplication/ru.spbstu.icc.kspt.lab6.continuewatch.ThreadRealization: +1s252ms
ab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] iterating
ab6.continuewatch.ThreadRealization: 1.006
ab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] iterating
ab6.continuewatch.ThreadRealization: 2.0069999999999997
ab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] iterating
ab6.continuewatch.ThreadRealization: 3.0079999999999996
ab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] iterating
ab6.continuewatch.ThreadRealization: 4.0089999999999995
ab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] iterating
ab6.continuewatch.ThreadRealization: 5.01
```

Рис. 1 Логи приложения на Thread.

Можем видеть, как постепенно накапливается наша дробная часть, которую раньше мы просто откидывали.

```
.lab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] iterating
.lab6.continuewatch.ThreadRealization: 15.019999999999996
.lab6.continuewatch.ThreadRealization: 15.775999999999996
.lab6.continuewatch.ThreadRealization: Thread[Thread-2,5,main] is stopped
.lab6.continuewatch.ThreadRealization: Thread[Thread-3,5,main] is started
.lab6.continuewatch.ThreadRealization: Thread[Thread-3,5,main] iterating
.lab6.continuewatch.ThreadRealization: 16.816999999999997
.lab6.continuewatch.ThreadRealization: Thread[Thread-3,5,main] iterating
.lab6.continuewatch.ThreadRealization: 17.817999999999998
.lab6.continuewatch.ThreadRealization: Thread[Thread-3,5,main] iterating
.lab6.continuewatch.ThreadRealization: 18.819
.lab6.continuewatch.ThreadRealization: Thread[Thread-3,5,main] iterating
.lab6.continuewatch.ThreadRealization: 19.82
.lab6.continuewatch.ThreadRealization: 19.939
.lab6.continuewatch.ThreadRealization: Thread[Thread-3,5,main] is stopped
```

Рис. 2 Логи приложения на Thread.

На этом рисунке показана, как мы прерываем Thread, а перед этим запоминаем нашу уже отработанную часть (2 строчка). После этого Thread снова запускается в onStart, и мы продолжаем счет со всем учтенным временем.

Дальнейшие реализации будут разобраны менее подробно, т. к. по сути там только заменяется Thread, на что-то другое.

Задача 1. Execution Service

Здесь мы оперируем более удобной штукой как ExecutionService. Правда здесь нам не нужно делать еще один updateTime в методе onStop, т. к. сервис не прерывается, а просто завершает очередной цикл Thred'a.

Листинг 1.2 ExecutionServiceRealization

```
class ExecutorServiceRealization : AppCompatActivity() {

    private val MY_LOG = this.javaClass.name
    private val SAVE_KEY = "Key"

    private var secondsElapsed: Double = 0.0
    private lateinit var textSecondsElapsed: TextView
    private lateinit var timeExecutor: Future<*>
    private val executorService by lazy { (application as ExecutorServiceApplication).executor }
    private var startTime: Long = 0
    private var currentTime: Long = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        textSecondsElapsed = findViewById(R.id.textSecondsElapsed)
    }

    override fun onStart() {
        super.onStart()
        timeExecutor = executorService.submit(getThread())
        startTime = System.currentTimeMillis()
    }

    override fun onStop() {
        updateTime()
        timeExecutor.cancel(true)
        super.onStop()
    }

    private fun getThread(): Thread = Thread {
        while (!Thread.currentThread().isInterrupted) {
            try {
                Thread.sleep(1000)
            } catch (e: InterruptedException) {
                Log.i(MY_LOG, "${Thread.currentThread()} is stopped")
                return@Thread
```

```
            }
            Log.i(MY_LOG, "${Thread.currentThread()} iterating")
            updateTime()
        }
    }

    private fun updateTime() {
        currentTime = System.currentTimeMillis()
        secondsElapsed += (currentTime - startTime) / 1000.0
        startTime = currentTime
        Log.i(MY_LOG, "$secondsElapsed")

        textSecondsElapsed.post {
            textSecondsElapsed.text = resources.getQuantityString(
                R.plurals.secCounter,
                secondsElapsed.toInt(),
                secondsElapsed.toInt()
            )
        }
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.run {
            putDouble(SAVE_KEY, secondsElapsed)
        }
    }

    override fun onRestoreInstanceState(savedInstanceState: Bundle) {
        super.onRestoreInstanceState(savedInstanceState)
        savedInstanceState.run {
            secondsElapsed = getDouble(SAVE_KEY)
        }
    }
}

class ExecutorServiceApplication : Application() {
    var executor: ExecutorService = Executors.newSingleThreadExecutor()
}
```

Посмотрим логи:

```
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] iterating
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: 1.001
W/System: A resource failed to call close.
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] iterating
    2.003
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] iterating
    3.004
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] iterating
    4.005
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: 4.068
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] is stopped
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] iterating
    5.074999999999999
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: 5.728999999999999
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] is stopped
I/ru.spbstu.icc.kspt.lab6.continuewatch.ExecutorServiceRealization: Thread[pool-2-thread-1,5,main] iterating
    6.732999999999999
```

Рис. 3 Логи приложения на ExecutionService.

Задача 1. Kotlin Coroutines

Здесь ситуация еще проще, т. к. мы работаем с кортуинами, как не странно. Корутина может просто остановиться в точке останова, нам не надо ее прерывать. А также мы можем вообще даже не контролировать ее состояния, а просто указать scope работы activityLifecycle. Таким образом вся наша работа с потоками сконцентрирована в одном методе:

Листинг 1.3 CoroutineRealization

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    textSecondsElapsed = findViewById(R.id.textSecondsElapsed)
    lifecycleScope.launchWhenResumed {
        Log.i(MY_LOG, "$this is started")
        while (isActive) {
            Log.i(MY_LOG, "Coroutine works")
            delay(1000)
            updateTime()
        }
    }
}
```

Надо также упомянуть, что мы не можем обновлять наше UI из background Thread'a. Для решения этой проблемы мы используем удобный метод View post. Во всех реализациях мы используем этот метод.

Логи:

```
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 1.016
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 2.018
    Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 2.0199999999999996
    Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 3.0609999999999995
    Coroutine works
I/e.myapplicatio: Waiting for a blocking GC ProfileSaver
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 3.1139999999999994
    Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 3.1149999999999993
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 4.156999999999999
    Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 4.614999999999999
    Coroutine works
I/ru.spbstu.icc.kspt.lab6.continuewatch.CoroutineRealization: 5.616
```

Рис. 4 Логи приложения на корутинах.

Наблюдаем аналогичное поведение.

Задача 2. Execution Service

Напишем приложение загрузки картинки при помощи Execution Service. Также мы будем использовать для скачки ViewModel, для того чтобы избежать повторного скачивания картинки при Configuration Change event'ах. MainActivity будет выглядеть одинаково для всех трех реализаций следующим образом:

Листинг 2.0 MainActivity

```
class MainActivity : AppCompatActivity() {

    private val executorService by lazy { (application as ExecutorServiceApplication).executor }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

//        //for Executor Realization
//        val factory = MyViewModelFactory(executorService)
//        val mainViewModel = ViewModelProvider(this, factory)
//            .get(ExecutionViewModel::class.java)

        //for other realizations
        val mainViewModel: CorutineViewModel by viewModels()

        binding.downloadButton.setOnClickListener {
            mainViewModel.downloadImage()
        }
        binding.resetButton.setOnClickListener {
            mainViewModel.clearImage()
        }
        mainViewModel.bitmap.observe(this) {
            binding.imageView.setImageBitmap(it)
        }

    }

}

class ExecutorServiceApplication : Application() {
    var executor: ExecutorService = Executors.newSingleThreadExecutor()
}

class MyViewModelFactory(private val executor: ExecutorService) : ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return modelClass.getConstructor(ExecutorService::class.java).newInstance(executor)
    }

}
```

В ней будет меняться только реализация ViewModel.

Реализация ViewModel на Execution Service:

Листинг 2.1 ExecutionViewModel

```
class ExecutionViewModel(private val executorService: ExecutorService) : ViewModel() {

    private val TAG = "Download"
    private val MOOD_URL = URL(
        "Здесь очень большая ссылка на мем"
    )

    val bitmap: MutableLiveData<Bitmap> = MutableLiveData()

    fun downloadImage() {
        executorService.execute {
            Log.i(TAG, "Connecting to DB...")
            Thread.sleep(1000)
            Log.i(TAG, "Get request...")
            Thread.sleep(1000)
            Log.i(TAG, "Downloading in ${Thread.currentThread()}")
            bitmap.postValue(BitmapFactory.decodeStream(MOOD_URL.openConnection().getInputStream()))
        }
    }

    fun clearImage() {
        Log.i(TAG, "Clearing image")
        bitmap.value = null
    }

    override fun onCleared() {
        Log.i(TAG, "Clear ViewModel")
        executorService.shutdown()
        super.onCleared()
    }
}
```

Помимо скачивания картинки, я сделал имитацию каких-то длительных запросов, чтобы я успел посмотреть, как хорошо, что я использую ViewModel и могу, например поворачивать экран во время загрузки.
Посмотрим на логи, и на само приложение:

```
9841-9878/com.example.downloadimage I/Download: Connecting to DB...
9841-9878/com.example.downloadimage I/Download: Get request...
9841-9878/com.example.downloadimage I/Download: Downloading in Thread[pool-2-thread-1,5,main]
9841-9841/com.example.downloadimage I/Download: Clearing image
9841-9878/com.example.downloadimage I/Download: Connecting to DB...
9841-9878/com.example.downloadimage I/Download: Get request...
9841-9878/com.example.downloadimage I/Download: Downloading in Thread[pool-2-thread-1,5,main]
```

Рис. 5 Логи приложения скачивания картинки на Сервисе.

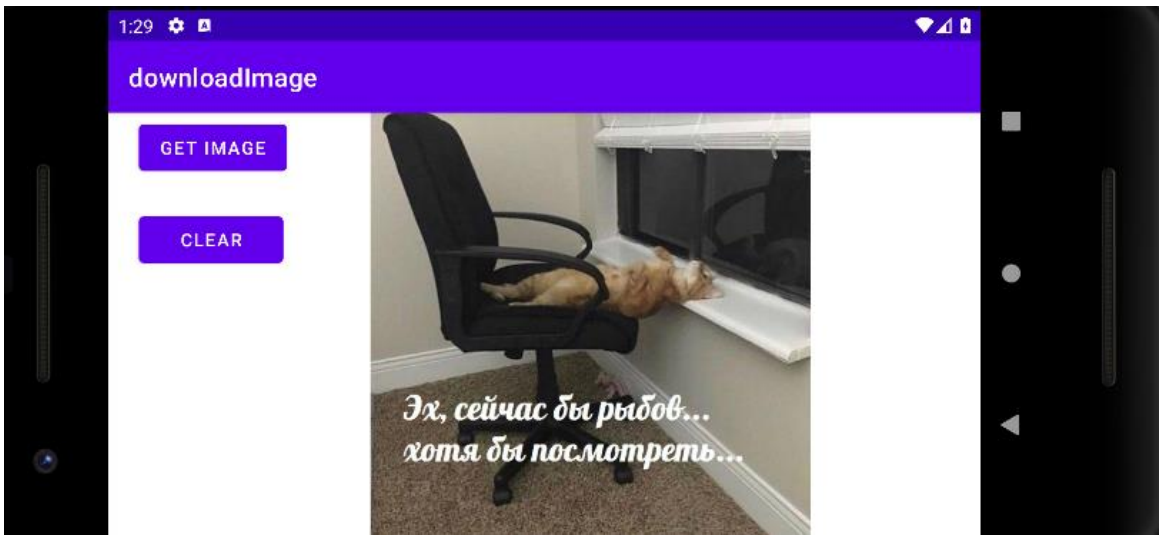

Рис. 6 Само приложение.

Все работает отлично, даже во время поворачивания экрана при загрузке.

Задача 2. Kotlin Coroutines

Перепишем нашу ViewModel на корутины.
По сути, мы меняем только метод downloadImage.

Листинг 2.2 CorutineViewModel

```
class CorutineViewModel : ViewModel() {

    private val TAG = "Download"
    private val MOOD_URL = URL(
        "https://sun9-
7.userapi.com/impg/7I7Iepv1qs91WHMCc5a16pR9c0tl2uyJVIulnw/dnqlc38WYsI.jpg?size=640x642&quality=96&si
gn=fa29f3ea252838887b29f958d674cbb1&type=album"
    )
    val bitmap: MutableLiveData<Bitmap> = MutableLiveData()
    fun downloadImage() {
        viewModelScope.launch(Dispatchers.IO) {
            Log.i(TAG, "Connecting to DB...")
            delay(1000)
            Log.i(TAG, "Get request...")
            delay(1000)
            Log.i(TAG, "Downloading in ${Thread.currentThread()}")
            val pic = BitmapFactory.decodeStream(
                    MOOD_URL.openConnection().getInputStream()
            )
            MainScope().launch {
                    bitmap.value = pic
            }
        }
    }

    fun clearImage() {
        Log.i(TAG, "Clearing image")
        bitmap.value = null
    }
}
```

Логи как-то семантически не изменятся.

Задача 2. Библиотека Picasso

Листинг 2.3 PicassoViewModel

```
class PicassoViewModel : ViewModel() {

    private val TAG = "Download"
    private val MOOD_URL = URL(
        "https://sun9-
7.userapi.com/impg/7I7Iepv1qs91WHMCc5a16pR9c0tl2uyJVIulnw/dnqlc38WYsI.jpg?size=640x642&quality=96&si
gn=fa29f3ea252838887b29f958d674cbb1&type=album"
    )

    val bitmap: MutableLiveData<Bitmap> = MutableLiveData()


    fun downloadImage() {
        Picasso.get().load(MOOD_URL.toString()).into(
            object : Target {
                override fun onBitmapLoaded(bitmap: Bitmap?, from: Picasso.LoadedFrom?) {
                    Log.i(TAG, "Loaded image")
                    this@PicassoViewModel.bitmap.postValue(bitmap)
                }

                override fun onBitmapFailed(e: Exception?, errorDrawable: Drawable?) {
                    Log.i(TAG, "Load image failed")
                }

                override fun onPrepareLoad(placeHolderDrawable: Drawable?) {
                    Log.i(TAG, "Prepare load image")
                }
            }
        )
    }

    fun clearImage() {
        Log.i(TAG, "Clearing image")
        bitmap.value = null
    }

    override fun onCleared() {
        Log.i(TAG, "Clear ViewModel")
        super.onCleared()
    }

}
```

Ну тут мы просто используем одну из множества библиотек, для решения определенной проблемы. That’s how we do it.
Также надо упомянуть что для работы с сетью, в манифесте приложения нужно прописать соответствующие запросы на использование сети.

4. Вывод

В работе я очень многое узнал о Thred’ах и многопоточных приложениях в целом, в частности о многопоточных приложениях под Android. Были изучены JavaThread, ExecutionService, KotlinCoroutine и методы работы с ними.

В работе также применл навыки разработки многопоточного приложения для реализации простой задачи – загрузки картинки из сети в фоновом потоке.

JavaThread:

1) Создание и запуск. Создаем поток через стандартный конструктор, передавая Runnable. Метод start() экземлпяра потока, запустит поток. Метод run() же просто запустит Runnable потока в текущем потоке.
2) Передача информации в UI Thread. Activity.runOnUiThread(Runnable) или View.post(Runnable) // View.postDelayed(Runnable, long)
3) Остановка потока. По факту нам нужно осуществить return из Runnable Thread’a. Для этого есть стандартный метод выставления флага interrupted, методом Thread’a Thread.interrupt(), а в самом Runnable проверять наличие флага Thread.isInterrupted. Причем нужно знать, что флаг сбрасывается, после получения. Также, надо знать, что блокирубщие методы типа Thread.sleep(), могут кинуть InterruptedException, которые нужно обрабатывать.
   Есть еще специфичный метод Thread.stop(), который вызывает исключение ThreadDeath, и если у нас есть какая, то синхронизация потоков, то у нас скорее всего появятся проеблемы, из-за того, что мы моментально освобождаем все мониторы.

ExecutorService:

1) Создание и запуск. Создаем наш экзекутор обычно через фабрику Executors, где есть обширный набор часто используемых экзекуторов. Ну или через конструкторов кастомных экзекуторов, если такие необходимы.

В экзекутор можно запланировать задачу методом ExecutorService.execute(Runnable). Или можно использовать метод ExecutorService.submit(Runnable), который возвращает экземпляр Future<?>, который помогает нам контроллировать и следить за данной задачей.

2) Передача информации в UI Thread. Аналогичные пункту из JavaThreadActivity.runOnUiThread(Runnable) или View.post(Runnable) // View.postDelayed(Runnable, long)
3) Остановка задачи/задач. При использовании Future, можно останваливать задачу методом cancel(), при этом есть атрибут mayInterruptIfRunning, которым можно регулировать поведения отмены задачи. У самого executor'а есть методы прекращения/досрочной остановки всех задач shutdown() shutdownNow().

Coroutine:

1) Создание и запуск. Создание при помощи Coroutine builders: launch, async, runBlocking. Также можно испольозовать стандартные скоупы, например в андроиде – lifecycleScope.
   Запуск методом launch(//диспетчер). Launch вернет экземпляр Job – семантический аналог Future, для управления корутиной.
2) Передача информации в UI Thread. Можно пользоваться методоми Thread'ов, или же приемами корутин- переключение контекста: withContext(otherDispatcher) {}, OtherScope.launch {}.
3) Остановка корутины. Для точной остановки корутины нам нужно у объекта job вызвать два последовательных метода job.cancel() и job.join(), или же просто job.cancelAndJoin(). Также нам нужно понимать особенности остановки корутины (точки останова), что она может не остановиться сразу, если у нас в Runnable корутины есть длительные операции нам там нужно проверять (по аналогии с флагом interrupted), жизнеспособность коруитны методом isActive.