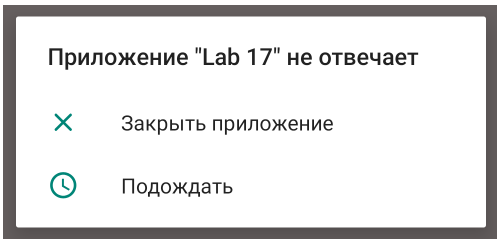
Многопоточность

Когда запускается какой-либо компонент приложения, Android запускает новый процесс с одним потоком выполнения. По умолчанию все компоненты приложения выполняются в одном и том же потоке, называемом *основным* потоком. Если при запуске компонента Android обнаруживает, что для этого приложения уже есть процесс, то компонент запускается внутри этого процесса и использует тот же самый поток выполнения.

Основной поток очень важен, он отвечает за отправку событий представлениям на экране. По этой причине основной поток иногда называют *потоком пользовательского интерфейса* (UI thread). Функции обратного вызова, которые вызываются системой при каких-то событиях (например, касании кнопки на экране), тоже выполняются в потоке пользовательского интерфейса.

Если приложение выполняет интенсивную работу в ответ на действия пользователя, это может привести к снижению производительности. Кроме того, выполнение длительных операций в основном потоке, таких как доступ к сети или запросы к базе данных, блокирует весь пользовательский интерфейс.

Когда поток заблокирован, никакие события не могут быть отправлены, включая события рисования. С точки зрения пользователя приложение зависает. Хуже того, если основной поток пользовательского интерфейса блокируется более чем на несколько секунд, пользователю отображается диалоговое окно «Приложение не отвечает». Пользователь может решить выйти из приложения или даже удалить его.



В однопоточной модели Android существуют два правила:

- Не блокируйте основной поток.
- Не обращайтесь к пользовательскому интерфейсу извне основного потока.

Если нужно выполнить длительные операции, следует выполнять их в отдельных *фоновых* или *рабочих* потоках. Только нужно помнить, что из таких потоков не получится обновлять пользовательский интерфейс.

Классические фоновые потоки

Традиционный путь для выполнения фрагмента кода в фоне – это создание дополнительного потока. В Kotlin это делается просто, с помощью функции `thread`:

```
fun myButton_Click(view: View) {
    thread {
        var n = 0
        while (true) {
            n++
            if (n % 10_000_000 == 0)
```

```
                // Что-то делаем...
        }
    }
}
```

В этом примере при нажатии кнопки запускается отдельный поток, в котором выполняется некая тяжёлая вычислительная работа – для её имитации запускается бесконечный цикл с увеличением переменной n.

Если попытаться вывести значение переменной n в поле `EditText`, то программа упадёт с ошибкой:

```
editText.setText(n.toString())
```

Чтобы корректно взаимодействовать с пользовательским интерфейсом, нужно соответствующий код запускать в основном потоке:

```
thread {
    var n = 0
    while (true) {
        n++
        if (n == 10_000_000) {
            runOnUiThread {
                et.setText(m.toString())
            }
        }

    }
}
```

Казалось бы, всё замечательно, проблема решена: код работает в фоновом потоке, не особо мешает основному потоку, и даже может взаимодействовать с пользовательским интерфейсом.

Есть, однако, существенное «но»: потоки – это довольно тяжёлый инструмент, переключение из основного кода в поток требует больше времени, памяти и других ресурсов. Поэтому если часто запускать потоки, то на производительности программы это может сказаться не лучшим образом.

Рекомендуется использовать потоки в тех сценариях, где требуются более тяжёлые фоновые вычисления и интенсивная обработка данных. А для сценариев, в которых основное время занимает ожидания чего-либо (например, окончания скачивания файла) рекомендуется использовать *корутины*.

Корутины

Корутина (англ. *coroutine*, на русский язык правильнее переводить как «сопрограммы», однако название «корутины» уже устоялось, поэтому далее используется именно оно) – это фрагмент программы, который выполняется поверх потока. Иногда корутины называют «легковесными потоками», потому что они во многом похожи на классические потоки, однако требуют гораздо меньше накладных расходов при использовании: потоки

запускаются и управляются операционной системой, а корутины реализуются в самом языке программирования.

Корутины по умолчанию не являются частью языка Kotlin, они реализованы в дополнительных библиотеках компании JetBrains, которая и создала язык Kotlin. Для подключения библиотек нужно добавить соответствующие зависимости в файл Gradle:

```
implementation("org.jetbrains.kotlin:kotlin-stdlib")
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")
```

Простой пример работы корутин (поскольку он работает в консоли, его нужно запускать в Kotlin Playground):

```
fun main() {
    runBlocking {
        launch {
            delay(1000L)
            println("Корутина выполнена")
        }
        println("Программа выполнена")
    }
}
```

Если запустить этот код, то в консоли появится такой текст:

```
Программа выполнена
Корутина выполнена
```

Почему получился такой порядок? Потому что сначала с помощью функции `launch` запускается корутина, но поток не блокируется и продолжает выполнять команды, появляется надпись «Программа выполнена». В это время корутина с помощью функции `delay` уходит в паузу на одну секунду, а затем печатает на экране надпись «Корутина выполнена». Оба эти кода выполняются параллельно.

Корутины всегда запускаются в каком-либо *контексте*. В примере выше контекстом служит объект `runBlocking` – он блокирует поток, запускает внутри себя корутины и дожидается их выполнения. Вот слегка модифицированный пример:

```
fun main() {
    runBlocking {
        launch {
            delay(100L)
            println("A")
        }
        launch {
            delay(90L)
            println("B")
        }
        launch {
            delay(80L)
            println("C")
        }
    }
    println("Программа выполнена")
}
```

В этом примере внутри `runBlocking` запускаются сразу три корутины, которые выполняются параллельно. Поскольку время задержки у них разное, то сначала завершится третья, затем вторая, и последней – первая. И только потом появится надпись «Программа выполнена», поскольку она стоит уже после программного блока `runBlocking`:

```
C
B
A
Программа выполнена
```

Можно вынести повторяющийся код в отдельную функцию – такие функции должны быть с модификатором `suspend`, что говорит о том, что они могут быть приостановлены без блокировки потока:

```
suspend fun wait(ms: Long, text: String) {
    delay(ms)
    println(text)
}

fun main() {
    runBlocking {
        launch { wait(100L, "A") }
        launch { wait(90L, "B") }
        launch { wait(80L, "C") }
    }
    println("Программа выполнена")
}
```

`runBlocking` может быть полезен если требуется, например, скачать из сети три файла и только потом продолжить выполнение программы. Но для приложения на Android это также будет означать блокирование основного потока, которое выглядит как зависание программы.

Поэтому в Android-приложениях вместо контекста `runBlocking` обычно используются другие контексты, их можно создать самим с помощью функции `CoroutineScope()`. В качестве параметра функция принимает один из классов диспетчера, определяющий в каком потоке (и с какой основной целью) будут запускаться корутины в этом контексте. Чаще всего используются два диспетчера, `IO` и `Main`:

- `Dispatchers.IO`: применяется для запуска корутин, выполняющих операции ввода-вывода – сетевые запросы, операции с файлами и т.д.
- `Dispatchers.Main`: применяется там, где необходимо взаимодействие с пользовательским интерфейсом.

Вот так можно создать корутину для скачивания файла:

```
val scope = CoroutineScope(Dispatchers.IO)
scope.launch {
    // Тут качается файл
    // ...
}
```

Здесь сначала создаётся свой контекст с диспетчером `IO`, внутри которого запускается скачивание файла.

Android строго следит за правильностью выбранного контекста: если попытаться выполнить сетевой запрос из контекста `Main`, программа упадёт – нельзя обращаться к сети из основного потока (а код хоть и выполняется в корутине, но поверх основного потока, система про эти ваши корутины ничего не знает и поэтому возмущается). Точно так же, если попытаться из контекста `IO` обновить текст на экране – программа тоже упадёт, с интерфейсом можно взаимодействовать только из основного потока, а контекст `IO` всегда выполняется в каком-то другом потоке.

Что же делать, если в корутине нужно сначала скачать файл, а затем вывести его на экран? Для этого можно временно изменить контекст, в котором выполняется корутина, с помощью функции `withContext`, а затем вернуться обратно – это примерный аналог `runOnUiThread`, который используется в потоках:

```
val scope = CoroutineScope(Dispatchers.IO)
scope.launch {
    // Тут качается файл
    // ...
    withContext(Dispatchers.Main) {
        // Тут что-то выводится на экран
        // ...
    }
    // Тут можно делать ещё что-то
    // ...
}
```

Важный момент: когда код выполняется внутри корутины, ключевое слово `this` ссылается уже не на объект активности, а на контекст корутины, в данном случае `MainScope`. Обращаться к полям и методам класса это никак не помешает, а вот если какой-либо функции требуется контекст, то вместо обычного `this` можно использовать `applicationContext` или другие аналогичные функции:

```
...
withContext(Dispatchers.Main) {
    // Вместо привычного this используется applicationContext:
    Toast.makeText(applicationContext, "Это корутина", Toast.LENGTH_SHORT).show()
}
...
```

Функция `launch` не просто запускает корутину, но ещё и возвращает объект класса `Job`, с помощью которого этой корутиной можно управлять:

```
val job = scope.launch{
    ...
}
```

У такого объекта есть целый ряд полезных функций и свойств:

- `join()` – ждать завершения корутины (работает только внутри другой корутины или suspend-функции)
- `cancel()` – прервать выполнение корутины
- `isActive` – выполняется ли корутина в данный момент?
- `isCompleted` – завершилась ли корутина естественным образом?
- `isCancelled` – была ли корутина прервана?

Например, если активность запустила корутину для скачивания файла, а пользователь в это время передумал и нажал кнопку отмены, то с помощью функции `cancel()` можно прервать корутину, скачивающую файл.

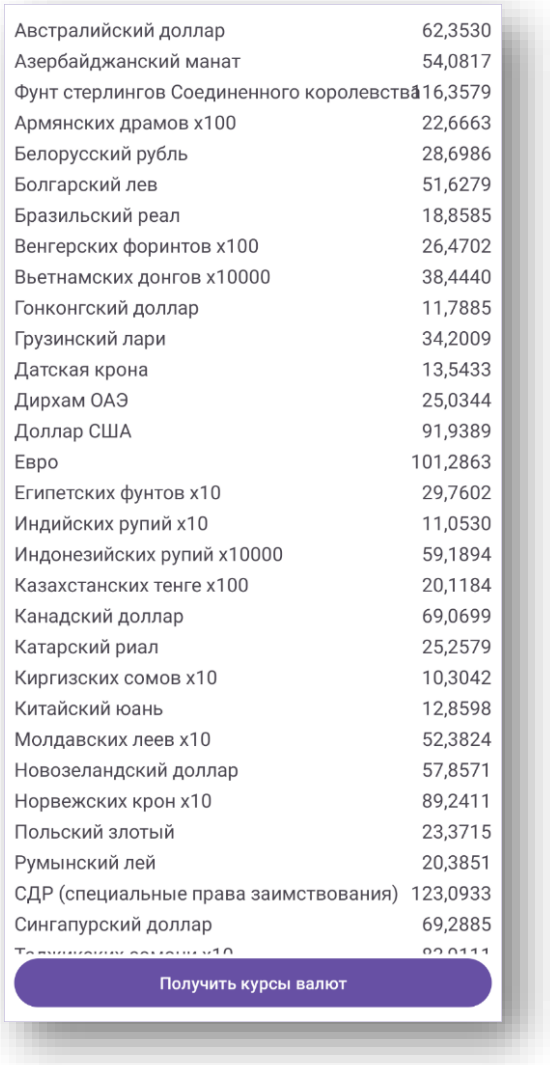
Дополнительная информация

Приведённая информация об использовании корутин касается лишь базовых аспектов использования корутин, необходимых для фоновых задач в Android-приложениях. Но тема корутин гораздо шире, содержит немало непростых моментов и нюансов: как передавать данные между двумя корутинами, как синхронизировать их работу и т.д. Если есть желание продолжить знакомство с корутинами, в интернете можно найти различные руководства и курсы, например:

- Учебник по языку Kotlin на сайте Metanit, главы 8 и 9: https://metanit.com/kotlin/tutorial/8.1.php
- Статья «Работа с асинхронными операциями с помощью Kotlin Coroutines»: https://habr.com/ru/articles/747858/

Задание

Напишите приложение, которое скачивает с сайта ЦБ РФ и отображает в списке актуальные курсы валют. Приложение должно выглядеть примерно следующим образом:



Изначально экран пустой, только в нижней части присутствует кнопка «Получить курсы валют». При нажатии этой кнопки в центре экрана появляется крутящийся элемент `ProgressBar`, и в фоне (в потоке или корутине, по желанию) начинается загрузка файла по адресу https://www.cbr.ru/scripts/XML_daily.asp – это XML-файл следующего вида:

```
<ValCurs Date="23.12.2023" name="Foreign Currency Market">
    <Valute ID="R01010">
        <NumCode>036</NumCode>
        <CharCode>AUD</CharCode>
        <Nominal>1</Nominal>
        <Name>Австралийский доллар</Name>
        <Value>62,3530</Value>
        <VunitRate>62,353</VunitRate>
    </Valute>
    <Valute ID="R01020A">
        <NumCode>944</NumCode>
        <CharCode>AZN</CharCode>
        <Nominal>1</Nominal>
        <Name>Азербайджанский манат</Name>
        <Value>54,0817</Value>
        <VunitRate>54,0817</VunitRate>
    </Valute>
    ...
</ValCurs>
```

Работа с сетью будет изучаться в последующих темах курса, пока же для загрузки данных просто используется следующий код:

```
val xml = URL("https://www.cbr.ru/scripts/XML_daily.asp")
    .readText(Charset.forName("Windows-1251"))
```

По историческим причинам для сохранения обратной совместимости с внешними приложениями файл находится в кодировке Windows-1251, а в Android / Kotlin все строки кодируются в UTF-8, поэтому для правильной перекодировки текста при загрузке сразу указывается требуемая кодировка.

Доступ к сети требует наличия соответствующего разрешения (о них также будет отдельная тема), поэтому следует внести его в манифест:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>

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

    <application ...>
        ...
    </application>

</manifest>
```

Для работы с XML в Android есть лишь самые базовые средства, одно из них – класс `XmlPullParser`. Для начала нужно получить экземпляр класса:

```
val parser = XmlPullParserFactory.newInstance().newPullParser()
```

Затем передать ему строку, в которой находится скачанный XML-файл:

```
parser.setInput(StringReader(xml))
```

Парсер будет проходить элемент за элементом и сообщать нам о каждом из них: начало документа, открывающий тег, текст, закрывающий тег, следующий открывающий тег, следующий текст, …, конец документа. Пусть, например, парсеру передан вот такой XML-код:

```
<book>
    <author>John R. R. Tolkien</author>
    <title>The Lord of the Rings</title>
</book>
```

В этом примере парсер будет передавать в программу следующие данные:

eventType	name	text
START_DOCUMENT		
START_TAG	book	
START_TAG	author	
TEXT		John R. R. Tolkien
END_TAG	author	
START_TAG	title	
TEXT		The Lord of the Rings
END_TAG	title	
END_TAG	book	
END_DOCUMENT		

Пример кода, который использует парсер:

```
while (parser.eventType != XmlPullParser.END_DOCUMENT) {
    if (parser.eventType == XmlPullParser.START_TAG)
        // Начало тега, можно сохранить его где-то...
    else if (parser.eventType == XmlPullParser.TEXT)
        // Текст внутри тега, что-то с ним можно сделать, если тег полезный
    else if (parser.eventType == XmlPullParser.END_TAG && parser.name == "...")
        // Закрытие тега...
    parser.next()
}
```