Интернет и JSON

Работа с сетью – одна из базовых потребностей многих приложений. Для этого можно использовать как базовые средства, которые предлагает операционная система, так и сторонние библиотеки.

Для отправки и получения данных через сеть есть много разных способов, как классы, встроенные в сам Android, так и разнообразные сторонние библиотеки, которые облегчают работу разработчику приложения. Однако в конечном итоге все эти способы отправляют и получают данные через сетевые протоколы HTTP/HTTPS. При изучении этой темы весьма желательно понимание того как работают эти сетевые протоколы, да и вообще основ компьютерных сетей.

Обязательным условием для работы сетевого кода является добавление разрешения `INTERNET` в манифест приложения:

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

Это разрешение не требует явного запроса у пользователя: если оно задекларировано в манифесте, то доступ будет предоставлен автоматически.

Другой важный момент, который относится к работе с сетью: код, выполняющий сетевые запросы, должен быть вынесен в отдельный поток или корутину. Дело в том, что сетевые запросы по своей природе очень медленные по сравнению с обычной работой программ, и если они выполняются в основном потоке, то блокируют его: приложение «подвисает» и «тормозит», исчезает плавность работы интерфейса. Поэтому в Android запрещено выполнять сетевые запросы в основном потоке.

Android Studio пытается обнаружить такой код и предупредить разработчика о необходимости вынести его из основного потока. Если этого не сделать, то при выполнении приложение будет остановлено с ошибкой `NetworkOnMainThreadException`.

Все примеры, которые рассматриваются в этой теме, предполагают, что код запускается не в основном потоке, то есть обрамлён в конструкцию наподобие такой:

```
thread {
    // Тут запрос данных из сети
    // ...
    runOnUiThread {
        // Тут, если нужно, вывод полученных данных в интерфейс
        // ...
    }
}
```

HttpURLConnection

Android имеет в своём составе классы, которые позволяют приложению скачивать или отправлять данные без использования сторонних библиотек. Однако использование этих классов требует дополнительных действий со стороны разработчика.

Алгоритм работы с классом `HttpURLConnection` выглядит так:

1. Получить экземпляр `HttpURLConnection` с помощью вызова метода `URL.openConnection()`.
2. Подготовить запрос: обязательным элементом является URI, дополнительно можно добавить и другие данные, такие как cookies и заголовки протокола HTTP.
3. Выгрузить тело запроса, если оно требуется: нужно сообщить о наличии тела запроса с помощью метода `setDoOutput(true)`, а затем записать сами данные в поток, возвращаемый методом `URLConnection.getOutputStream()`.
4. Прочитать ответ, полученный от сервера: можно проанализировать заголовки ответа (например, длину тела ответа, дату последнего изменения контента, cookie и пр.), а также прочитать само тело ответа из потока, возвращаемого методом `URLConnection.getInputStream()`.
5. Завершить работу с помощью метода `disconnect()`.

В следующем примере загружается главная страница сайта ТПУ:

```
val url = URL("https://www.tpu.ru/")
val con = url.openConnection() as HttpsURLConnection
try {
    val stream = BufferedInputStream(con.inputStream)
    // Читаем данные из потока stream
} finally {
    con.disconnect()
}
```

Здесь в примере в переменной stream находится поток, из которого можно читать данные с помощью методов класса `BufferedInputStream`.

Если требуется отправка каких-то данных на сервер, то пример становится немного сложнее:

```
val url = URL("https://www.tpu.ru/scripts/upload/")
val con = url.openConnection() as HttpsURLConnection
try {
    // Указание на наличие тела запроса
    con.setDoOutput(true)
    con.setChunkedStreamingMode(0)

    // Получение потока для тела запроса и запись в него
    val outStream = BufferedOutputStream(con.outputStream)
    outStream.write(12345)

    // Получение потока для тела ответа и чтение из него
    val inStream = BufferedInputStream(con.inputStream)
    val b = inStream.read()
} finally {
    con.disconnect()
}
```

Здесь сначала вызывается метод `setDoOutput`, который сообщает о наличии тела запроса, а затем метод `setChunkedStreamingMode`, который говорит о том что длина запроса неизвестна и нужно отправлять данные по мере поступления. Затем получается поток для тела запроса,

и уже в него осуществляется запись данных для отправки на сервер, после чего читается ответ сервера.

Если для работы с сервером требуется установка cookie, то можно включить работу с ними на уровне всего приложения:

```
val cookieManager = CookieManager()
CookieHandler.setDefault(cookieManager)
```

При отправке запроса система автоматически получит cookie от сервера. Если требуется установить какие-то дополнительные cookie, то нужно создать новый объект, установить в нём требуемые свойства, и добавить его в полученный экземпляр класса `CookieManager`:

```
val cookie = HttpCookie("lang", "en")
cookie.domain = "www.tpu.ru"
cookie.path = "/"
cookie.version = 0
cookieManager.cookieStore.add(URI("https://www.tpu.ru/"), cookie)
```

Библиотека OkHttp

Сторонние библиотеки как правило предлагают более комфортные способы работы с сетью. Однако они не избавляют от необходимости декларировать разрешение в манифесте или запускать код не в основном потоке.

Популярная сетевая библиотека OkHttp обеспечивает более низкие задержки и повышенную эффективность сетевых запросов за счёт повторного использования соединений, очередей запросов, кэширования и других механизмов. Для использования этих преимуществ нужно создать объект клиента один раз, и впоследствии все действия производить через него:

```
private val client = OkHttpClient()
```

Каждый запрос формируется с помощью класса `Request.Builder`, который позволяет настраивать заголовки, использование cookie и других параметров. По окончании настройки вызывается метод `build()`, который возвращает сформированный объект запроса:

```
val request = Request.Builder()
    .url("https://www.tpu.ru/robots.txt")
    .addHeader("lang", "en")
    .header("User-Agent", "IndexBot/1.0")
    .build()
```

В этом примере создаётся запрос к указанному адресу, в запрос добавляется новый заголовок `lang` со значением `en`, и модифицируется стандартный заголовок `User-Agent`.

Для отправки данных на сервер нужно использовать метод `post()`. Например, так можно отправить текстовые данные:

```
val str = "Текст для отправки"
val request = Request.Builder()
    .url("https://www.mysite.ru/scripts/send/")
    .post(str.toRequestBody())
    .build()
```

Для более сложных объектов можно реализовать свой класс-наследник от класса `RequestBody`.

После того как запрос подготовлен, его нужно передать для выполнения. Для этого служит метод `newCall()`: он подготавливает запрос к выполнению и возвращает объект `Call`:

```
val call = client.newCall(request)
```

Теперь можно либо выполнить запрос синхронно, либо поставить его в очередь на асинхронное выполнение. Разница заключается в том, что синхронный способ блокирует текущий поток до получения данных из сети, и потому должен выполняться не в основном потоке, а асинхронный способ немедленно возвращает управление, а при поступлении данных будет вызвана функция обратного вызова.

Синхронное выполнение запроса выглядит так:

```
call.execute().use { response ->
    if (response.isSuccessful) {
        val data = response.body!!.string()
        // Обработка полученных данных
    }
}
```

Асинхронное выполнение выглядит чуть более громоздко, но зато не задерживает выполнение основного потока и позволяет отдельно обрабатывать ошибки:

```
call.enqueue(object : Callback {

    override fun onFailure(call: Call, e: IOException) {
        // Запрос завершился с ошибкой
    }

    override fun onResponse(call: Call, response: Response) {
        // Запрос завершился успешно
        response.use {
            if (response.isSuccessful) {
                val data = response.body!!.string()
                // Обработка полученных данных
            }
        }
    }
})
```

Поскольку асинхронные запросы выполняются не в основном потоке, их не нужно вручную выносить в корутину или оборачивать в конструкцию `thread`. Однако, если полученные данные требуется вывести на экран, то в нужном месте всё равно требуется использовать конструкцию `runOnUiThread`.

Формат JSON

Данные в интернете могут передаваться в различных форматах: в виде простого текста, в XML, в JSON, в двоичном виде, ничто не мешает изобрести и какой-то свой формат. Однако использование стандартных форматов вроде XML или JSON позволяет сэкономить время

на написании и отладке кода для формирования данных для передачи и их последующей расшифровке, ведь для этих целей уже написано много библиотек. Работа с форматом XML кратко разбиралась в теме про многопоточность (лабораторная работа «Курсы валют ЦБ РФ»), ниже будет показано как можно работать с форматом JSON.

JSON представляет собой текстовый файл, в котором данные хранятся в виде множества пар формата *ключ-значение*. Значения могут быть строками, числами, массивами, или даже вложенными множествами пар формата ключ-значение. Например, базовую информацию о писателе можно было бы хранить в следующем виде:

```
{
  "year_born": 1818,
  "year_died": 1883,
  "name": {
    "first": "Иван",
    "middle": "Сергеевич",
    "last": "Тургеньев"
  },
  "popular_works": [
    "Отцы и дети",
    "Дворянское гнездо",
    "Муму"
  ]
}
```

Годы рождения (`year_born`) и смерти (`year_died`) представляют собой числа, имя (`name`) является вложенным объектом с тремя полями строкового типа (`first`, `middle`, `last`), а список популярных произведений (`popular_works`) – это массив со строками.

В Android есть два способа работы с данными в формате JSON: низкоуровневое чтение с использованием класса `JSONObject`, или использование сторонних библиотек.

Класс `JSONObject` встроен в Android и может использоваться для «ручного» разбора JSON-данных. Если при создании объекта передать ему в качестве параметра JSON-строку, то она будет разобрана и помещена во вновь созданный объект:

```
val jo = JSONObject(jsonStr)
```

Здесь jsonStr – это строка с приведённым выше примером, а результат парсинга заносится в переменную jo. Далее можно получать значения по ключу с помощью методов `getInt()`, `getString()`, `getDouble()`, `getLong()` или `getBoolean()`:

```
val year_born = jo.getInt("year_born")
```

Если нужно обратиться к вложенному объекту, то его можно получить с помощью метода `getJSONObject()`, он вернёт вложенный объект в виде точно такого же экземпляра класса `JSONObject`:

```
val name = jo.getJSONObject("name")
val first = name.getString("first")
```

Обратиться к массиву можно с помощью метода `getJSONArray()`, он возвращает объект класса `JSONArray`, у которого есть методы `length()` для получения количества элементов массива, и `get()` для получения элемента массива с указанным номером:

```
val works = jo.getJSONArray("popular_works")
var i = 0
while (i < works.length()) {
    val item = works.get(i)
    i++
}
```

Для более удобной работы с форматом JSON можно использовать сторонние библиотеки, например, библиотеку Gson, разработанную компанией Google. Перед использованием её, как и все сторонние библиотеки, нужно подключить в проект: для этого в файле Gradle, который относится к модулю приложения, нужно добавить соответствующую строку в раздел `dependencies`:

```
implementation("com.google.code.gson:gson:2.10.1")
```

Для разбора данных используется метод `fromJson()`, на вход которому передаются данные в формате JSON, а также ссылка на класс с распознанными данными. Например, если требуется произвести разбор данных о писателе из примера выше, то класс может выглядеть следующим образом:

```
data class AuthorName(
    val first: String,
    val middle: String,
    val last: String
)

data class Author(
    val year_born: Int,
    val year_died: Int,
    val name: AuthorName,
    val popular_works: Array<String>
)
```

Следует обратить внимание, что названия полей должны в точности соответствовать ключам из JSON-файла, в противном случае соответствующие данные не будут сопоставлены и занесены в результирующий объект.

Разбор данных можно произвести следующим кодом:

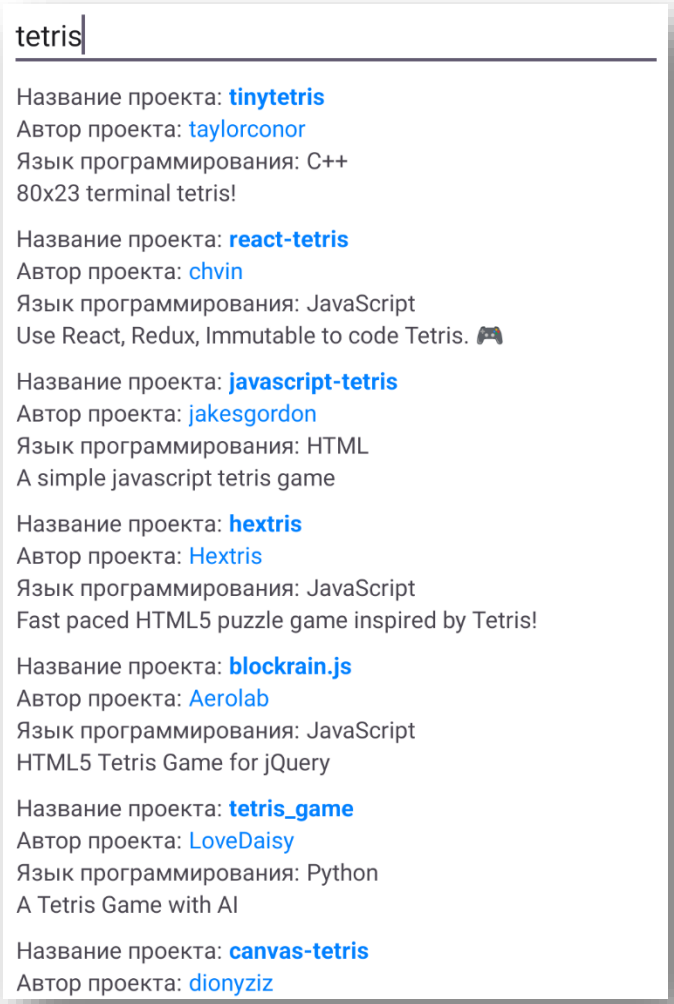
```
val t = Gson().fromJson(jsonStr, Author::class.java)
```

Здесь `jsonStr` – это строка, которая содержит JSON-данные из примера выше, `Author` – ссылка на класс с данными, а в переменную `t` будут занесены распознанные данные, если в формате не было ошибок.

Задание

Разработайте приложение для поиска репозиториев в сервисе GitHub. Пользователь вводит текст для поиска в поле ввода, приложение отправляет запрос на GitHub, и после получения

результатов выводит их в виде списка. Внешний вид программы может быть примерно таким:



Название репозитория и логин владельца репозитория должны быть выделены другим цветом, при нажатии на них в браузере по умолчанию должна открываться страница проекта или страница владельца репозитория, соответственно.

Поиск репозиториев осуществляется путём отправки запроса по следующему адресу:

```
https://api.github.com/search/repositories?q=<текст_для_поиска>
```

Например, чтобы найти все репозитории, в названии или описании которых содержится слово `tetris`, запрос следует отправить по такому адресу:

```
https://api.github.com/search/repositories?q=tetris
```

В ответ сервер присылает JSON-файл с результатами:

```
{
  "items": [
    {
      "name": "tinytetris",
      "owner": {
        "login": "taylorconor",
        "html_url": "https://github.com/taylorconor",
```

```
      },
      "html_url": "https://github.com/taylorconor/tinytetris",
      "description": "80x23 terminal tetris!",
      "language": "C++",
    },
    {
      "name": "react-tetris",
      "owner": {
        "login": "chvin",
        "html_url": "https://github.com/chvin",
      },
      "html_url": "https://github.com/chvin/react-tetris",
      "description": "Use React, Redux, Immutable to code Tetris. 🎮",
      "language": "JavaScript",
    },
    ...
  ]
}
```

Значений в возвращаемых данных очень много, в примере оставлены только те, которые существенны для выполнения задания.

Подробная информация о запросах к API GitHub может быть найдена в документации сервиса.

Запрос в сеть и разбор JSON-данных можно осуществлять любым из описанных в данной теме способов. По предварительному согласованию с преподавателем допускается использование других методов для выполнения запроса и разбора данных, например, использование библиотеки Retrofit.