

Многопоточность и сетевые запросы

что, почему, как?

Клещин Никита

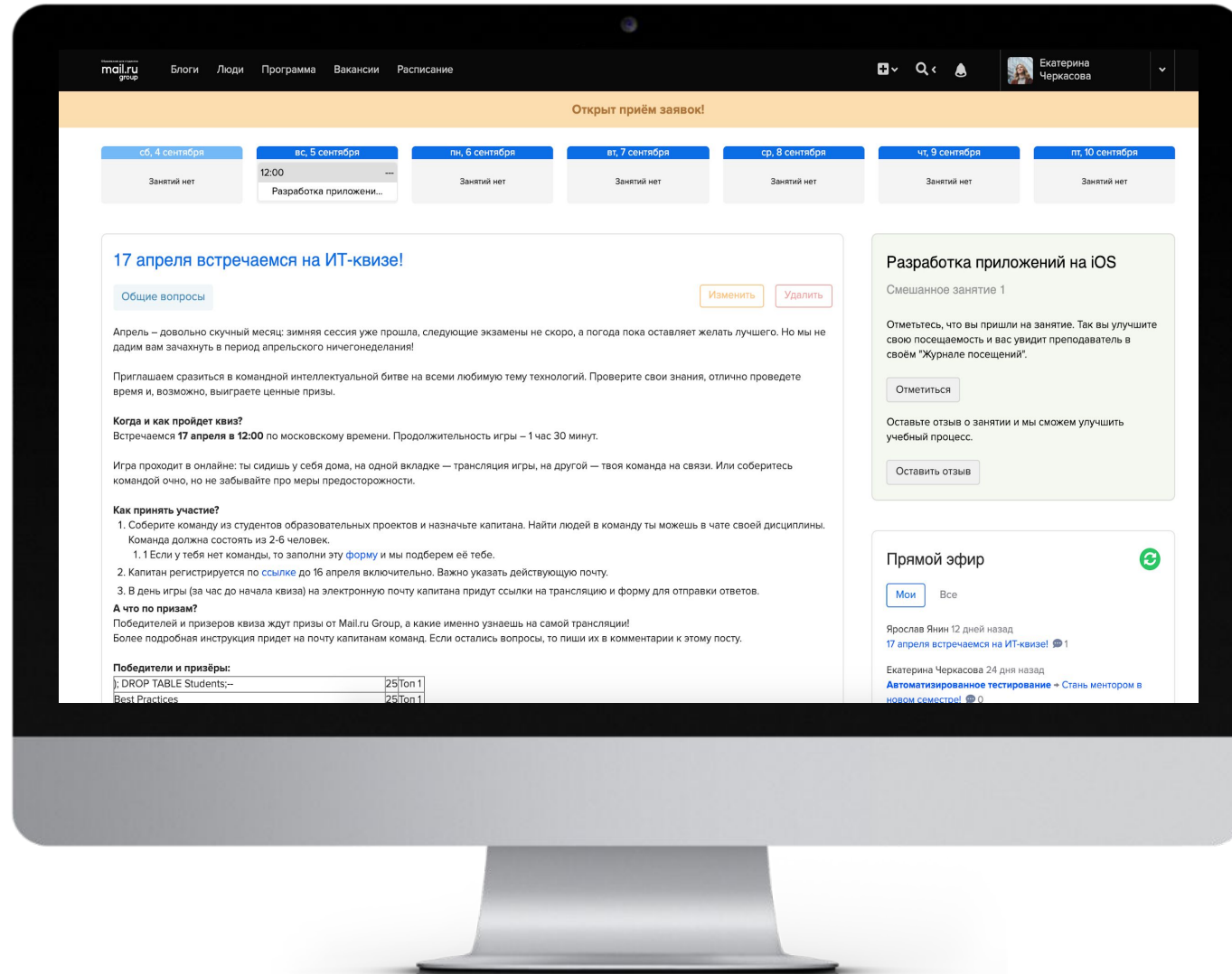




Содержание занятия

1. Процесс vs Поток
2. Многопоточность
3. Инструменты
4. Запросы
5. Приложение
6. Домашнее задание №2

Не забудьте отметиться!



Давайте сначала
еще раз поговорим
про менторов и
процесс обучения

Процесс vs Поток

Что?

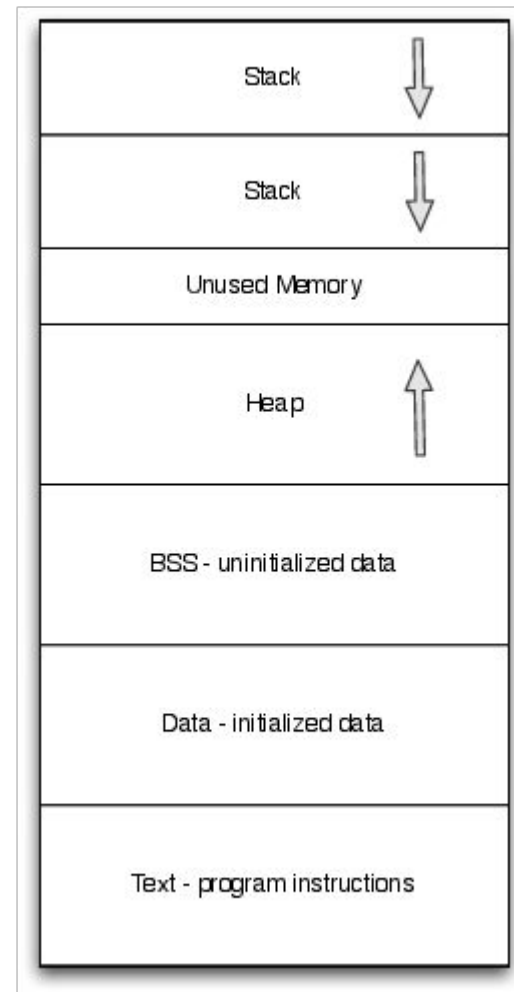


Процесс

Простыми словами - выделенное виртуальное пространство со своими ресурсами.

Между процессами нет общей памяти. Отсюда вытекают проблемы передачи данных между процессами.

Для Android есть такое правило - в каждом процессе есть один экземпляр Application. По дефолту, приложение работает в одном процессе.



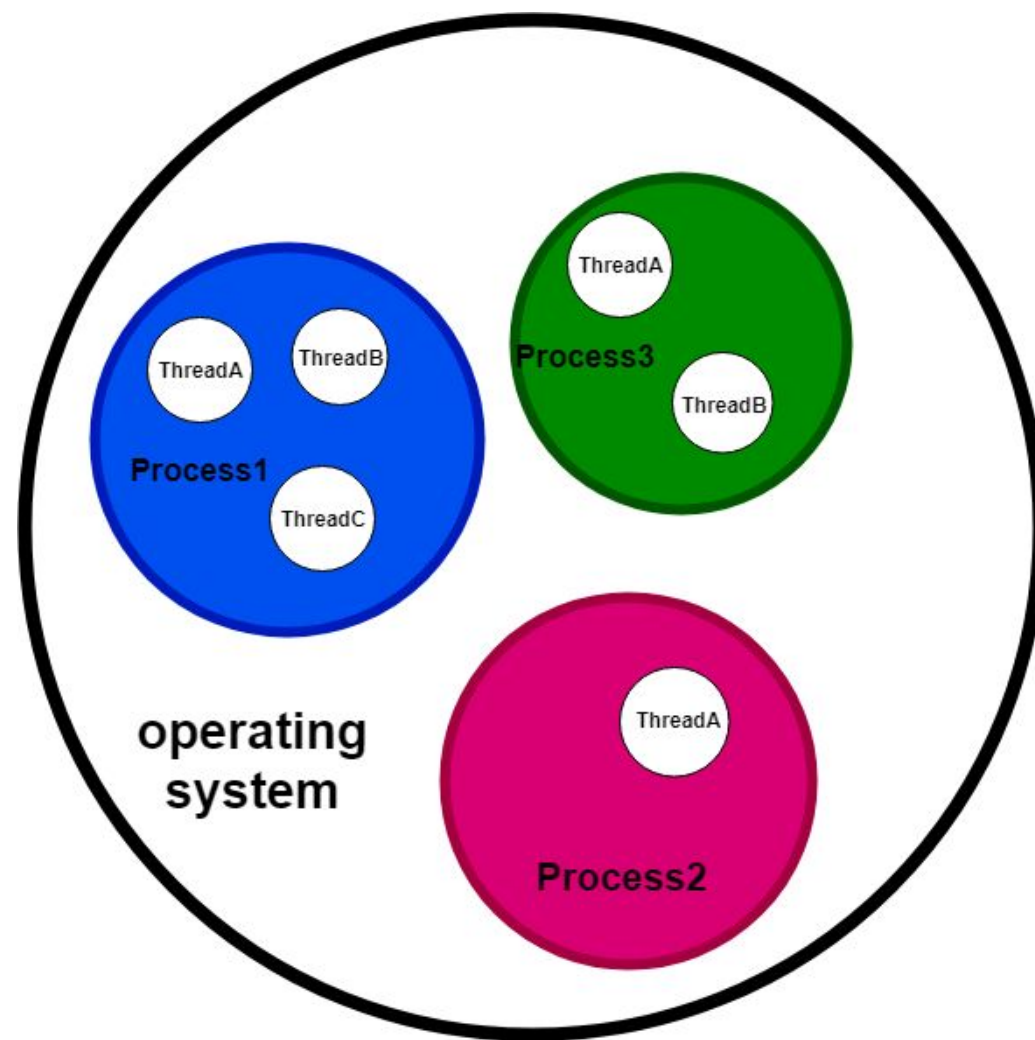
Поток

Использует ресурсы, выделенные для процесса, для исполнения инструкций.

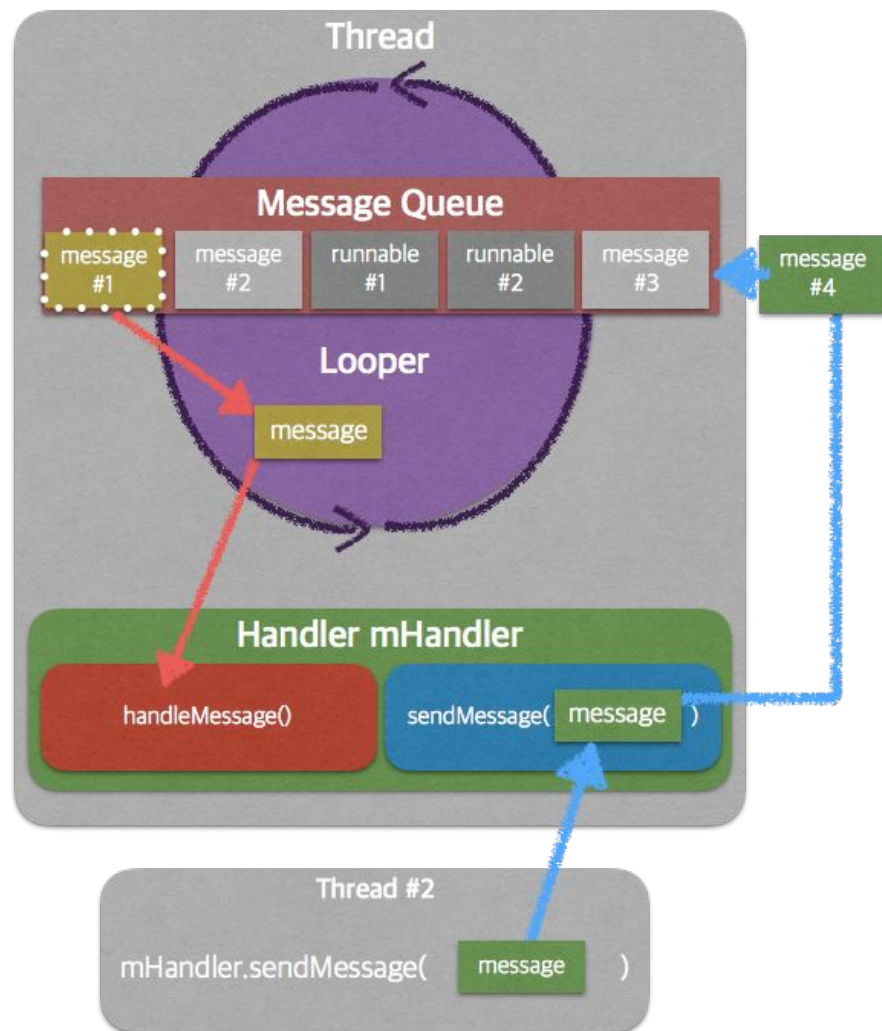
Память между потоками уже общая, но есть нюансы в виде кэша самого потока.

В Android для отрисовки интерфейса есть main поток. И если он перегружается тяжелыми задачами, то интерфейс начинает лагать.

Android также может замечать, что если приложение начинает делать операции для похода в сеть, и рубить весь процесс (StrictMode).



(Android) Looper, Handler... Main Thread



Просто способ что-то исполнить в главном потоке

```
val MAIN_HANDLER = Handler(Looper.getMainLooper())  
MAIN_HANDLER.post {  
    ...  
}
```

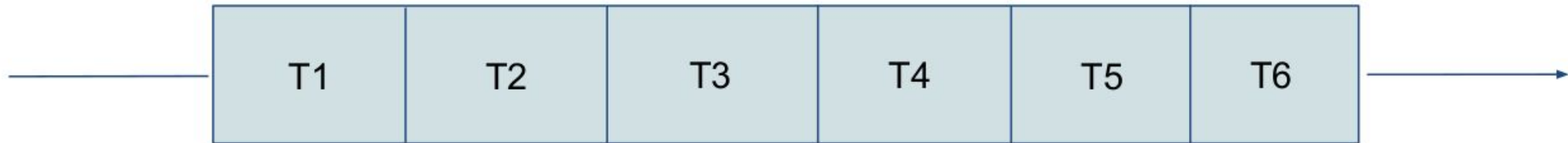

Многопоточность

Почему?

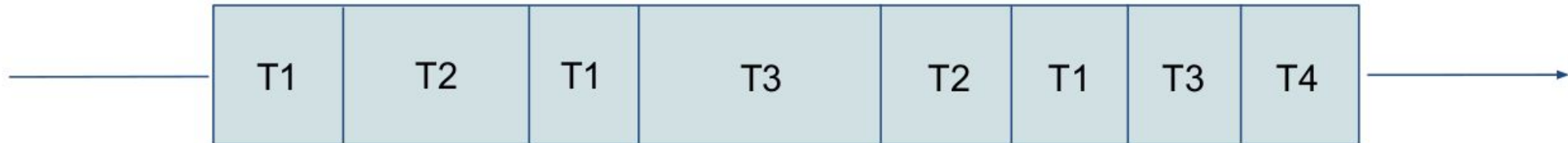


Поточное выполнение

Синхронное однопоточное выполнение



Асинхронное однопоточное выполнение



Время

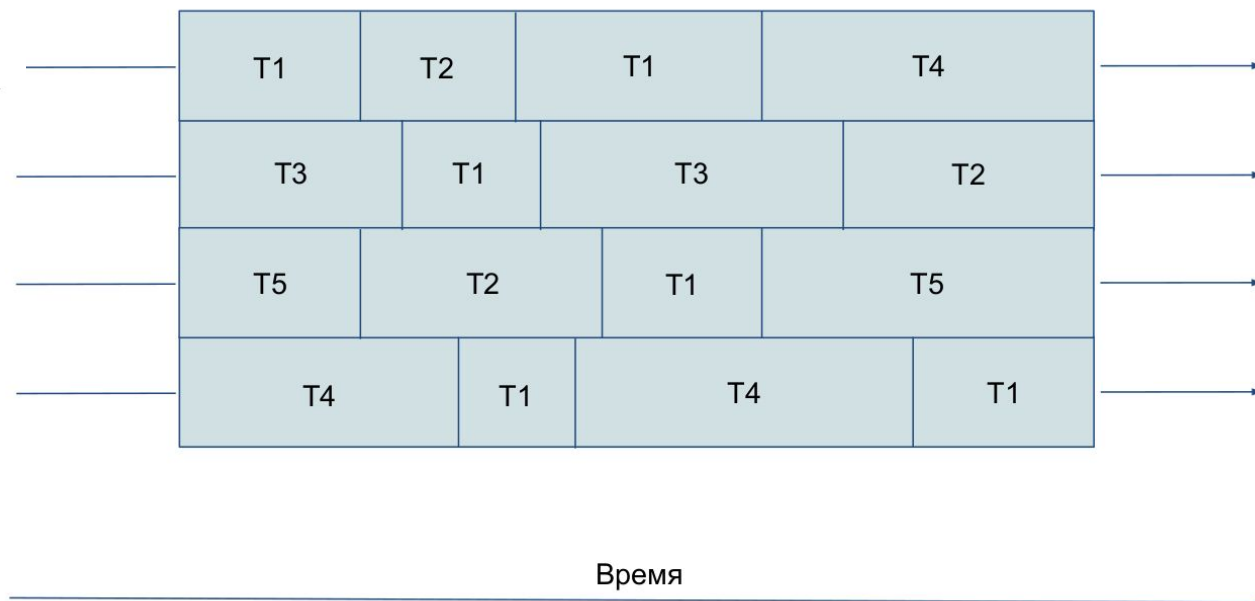


Многопоточное выполнение

Синхронное многопоточное выполнение



Асинхронное многопоточное выполнение

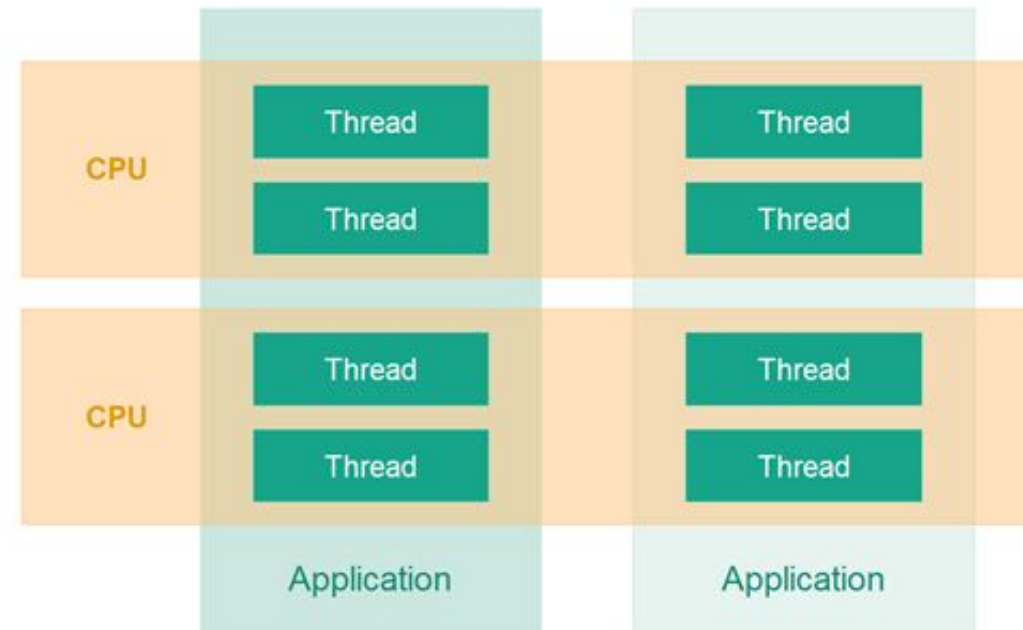


Concurrency

Потоки поочередно используют ресурсы процесса, для выполнения своих задач.

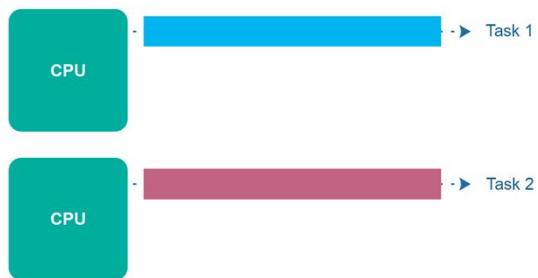
Если кажется что задачи выполняются одинаково (на одном ядре), это означает что что они выполняют задачи конкурентно.

Чем больше ядер у устройства, тем менее конкурентны будут выполнения.

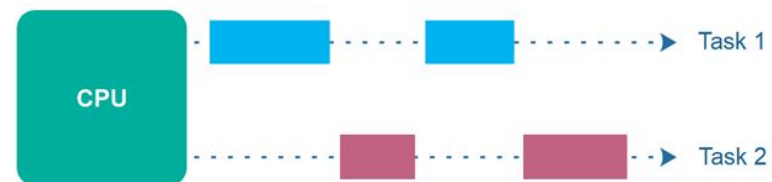


Многопоточность в картинках

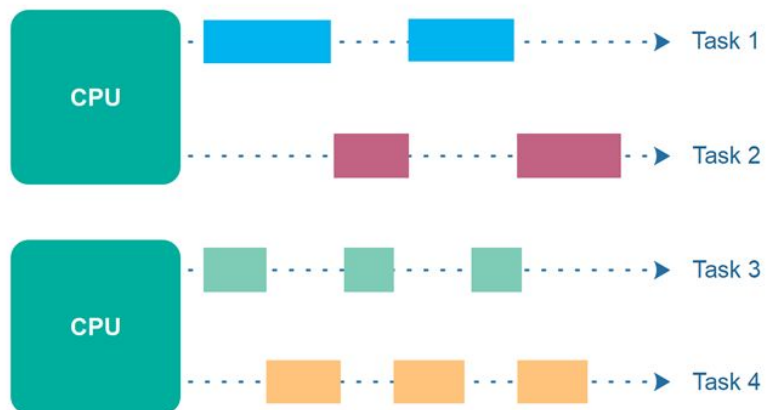
Parallel Execution



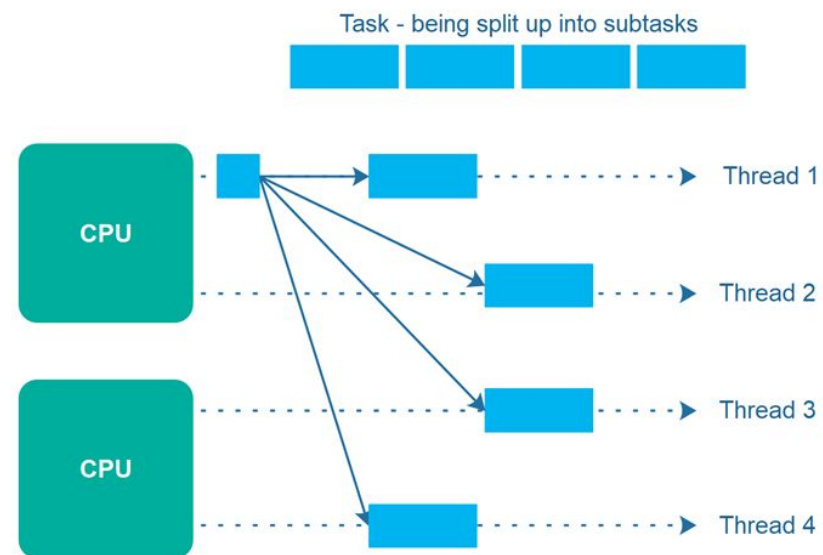
Concurrency



Parallel Concurrent Execution



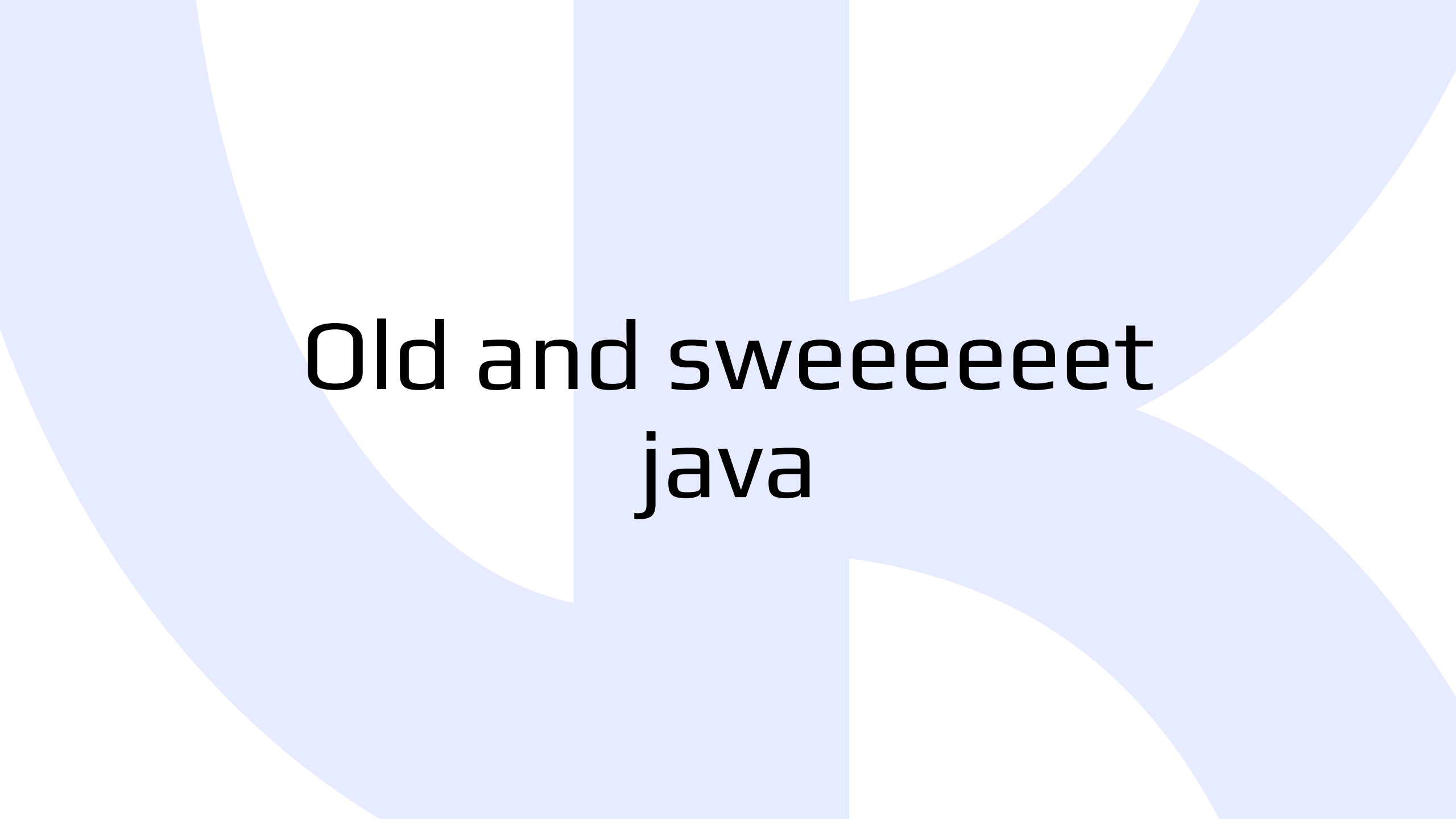
Parallelism



Инструменты работы с многопоточностью

Как?





Old and sweeeeeet
java

Thread

Плюсы

- Это самая базовая механика
- Можно сильно оптимизировать исполнения скорость выполнения

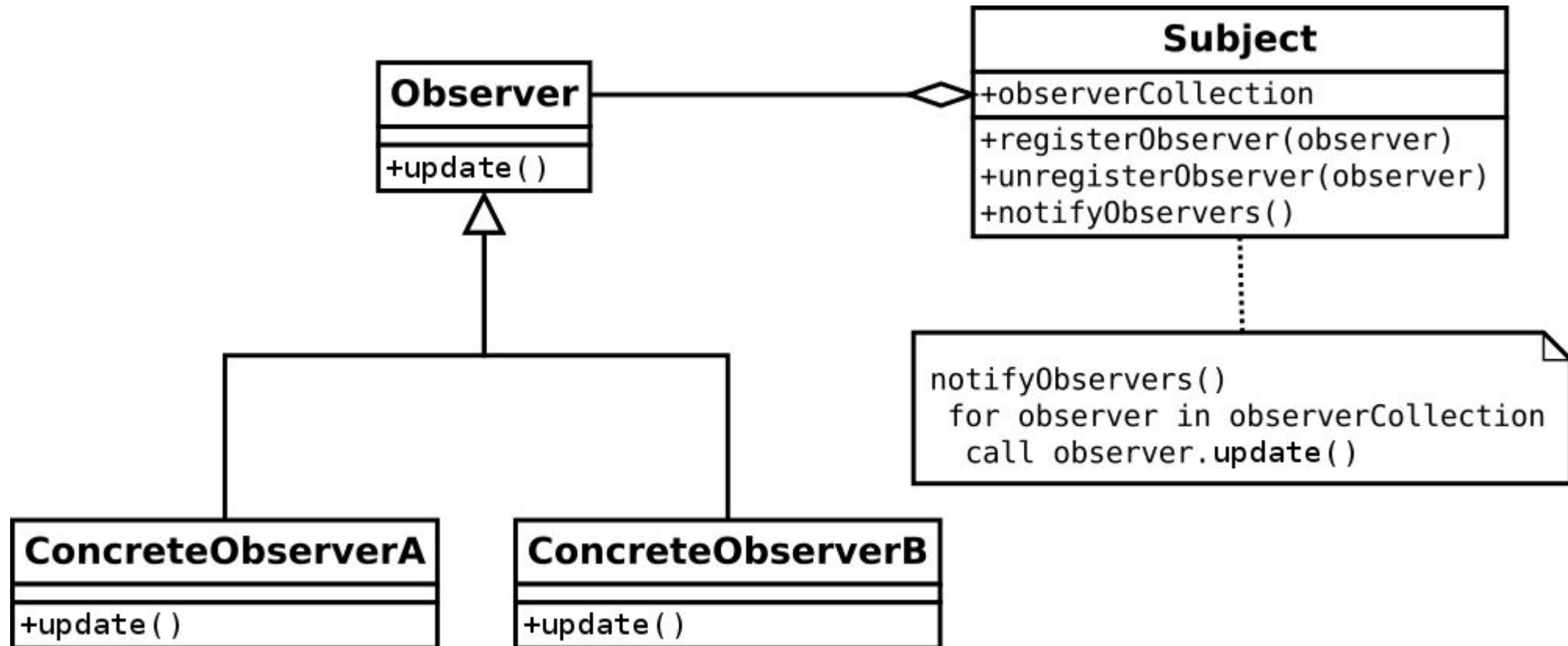
Минусы

- Это самая базовая механика
- Лучше написать свой фреймворк на основе пула тредов чтобы этим было удобно пользоваться во всем проекте
- Есть много информации, но ее местами не так просто усвоить

Этот подход можно использовать когда есть очень глубокое понимание всей механики работы

```
thread {  
    val result = accessor.items()  
    MAIN_HANDLER.post { callback(result) }  
}
```


Observer Pattern



RxJava

Плюсы

- Реактивно
- Все сложные кейсы обернуты в простые вызовы
- Достаточно популярная вещь, есть много информации

Минусы

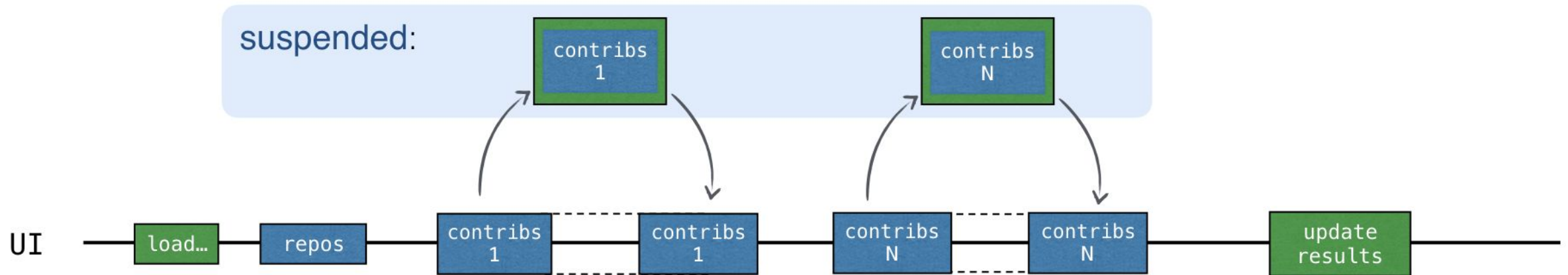
- Не является частью языка
- Есть шанс очень сильно завязать приложение на этот фреймворк
- Не совсем kotlin-friendly

До прихода kotlin+coroutine это был самый удобный фреймворк для работы с многопоточностью. Многие проекты до сих пор с ним

```
Observable.fromSingle<List<Item>> { accessor.items() }  
    .subscribeOn(Schedulers.newThread())  
    .observeOn(AndroidSchedulers.mainThread())  
    .subscribe(  
        { callback(it) },  
        { it.printStackTrace() }  
    )
```

```
val observable = Observable.create {  
    val result = accessor.items()  
    val response = mutableListOf<Item>()  
  
    result.forEach { item ->  
        sleep(500)  
        response.add(item)  
        it.onNext(response.toList())  
    }  
  
    it.onComplete()  
}.subscribeOn(Schedulers.newThread())  
  
observable.observeOn(AndroidSchedulers.mainThread())  
    .subscribe(  
        {callback(it)},  
        {it.printStackTrace()}  
    )
```

Suspending



Coroutine

Плюсы

- Почти часть языка (kotlin-friendly)
- Все сложные кейсы обернуты в простые вызовы
- Достаточно популярная вещь, есть много информации
- Позволяет писать код императивно
- Продвигается гуглом и интегрируется в Android

Минусы

- По началу можно написать что-то не оптимальное

Многие проекты их уже давно успешно используют.

```
scope.launch {  
    try {  
        val result = withContext(Dispatchers.IO) { accessor.items() }  
        callback(result)  
    } catch (error: Throwable) {  
        error.printStackTrace()  
    }  
}
```

Flow

Это как расширение корутин

Плюсы

- Почти часть языка (kotlin-friendly)
- Все сложные кейсы обернуты в простые вызовы
- Дает корутинам стать реактивными
- Продвигается гуглом

Минусы

- По началу можно написать что-то не оптимальное
- Зачастую код выглядит многоэтажно

Некоторые компании активно используют это расширение.

```
val flow = flow {  
    val result = accessor.items()  
    val response = mutableListOf<Item>()  
  
    result.forEach {  
        delay(500)  
        response.add(it)  
        emit(response.toList())  
    }  
}.flowOn(Dispatchers.IO)  
  
scope.launch {  
    flow.collectLatest {  
        callback(it)  
    }  
}
```

Кто использовал RxJava и
Coroutines
Что лучше?:)

Запросы

boy



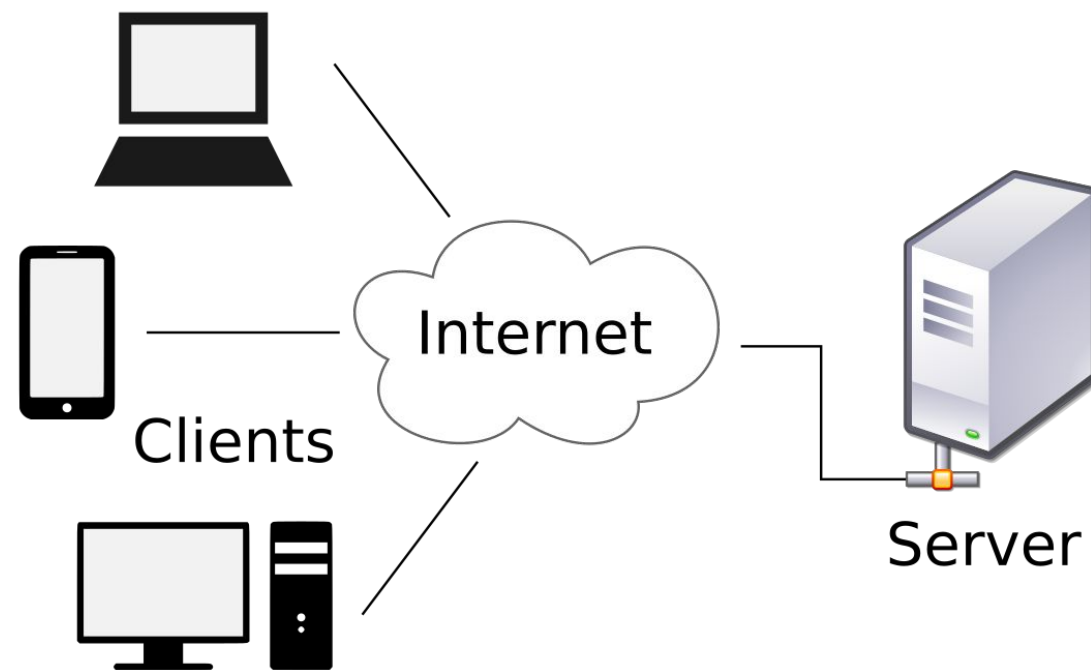
Приложение Клиент - Сервер

Сейчас уже сложно найти приложение, которое не требует интернет для корректной работы.

Выход в сеть будет требоваться минимум для аналитики и рекламы:)

Также - уже сложно для каталожного типа весь каталог записать в приложение. А приложения, в которых надо будет совершать оплату, доступ в интернет будет нужен минимум исходя из валидации платежа.

Когда говорят про клиент-сервер, подразумевают, что приложение будет взаимодействовать с API сервера.



Кейс: Вы открываете браузер, вводите там `vk.com`, жмете Enter.

Что происходит в этот момент?

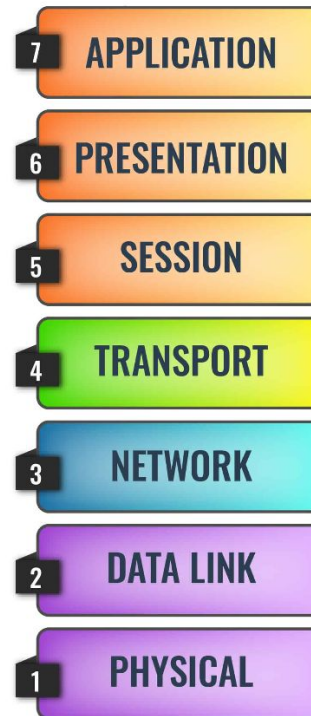
HyperText Transfer Protocol + Representational state transfer

HTTP (HTTPS - Secure) на основе **TCP/IP**. Самый распространенный формат взаимодействия клиента и сервера. Он нам диктует определенные ограничения для создания запросов, про которые так же знает и сервер (заголовки, тело, запрос).

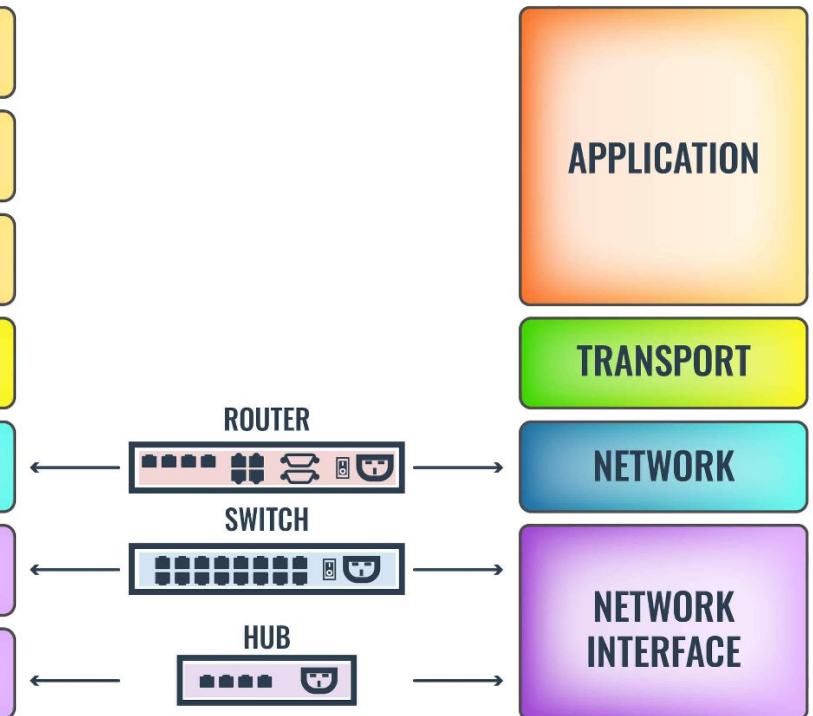
REST - архитектурный стиль. Тоже определенные ограничения по составлению запросов. Он помогает ограничить разнообразие методов до необходимого минимума, чтобы мы могли выполнить необходимые операции. Также подсказывает логику формирования необходимых запросов

Поскольку это общепринятые стандарты, то за нас уже написали необходимые фреймворки, чтобы не надо было сильно вникать в механику работы.

OSI REFERENCE MODEL



TCP/IP CONCEPTUAL LAYERS



Немного древностей

В Android есть все необходимые библиотеки для написания запросов. Но с ними примерно такая же история, как и с голыми потоками - на основе них надо написать свой фреймворк, который упростит работу с запросами.

Данные вещи необходимо писать разве что в ситуации, когда вы пишете библиотеку, которой должны будут пользоваться другие приложения, и вы точно не знаете на основе какого фреймворка у них устроена запросная часть.

Сообщество отдает предпочтение уже написанным фреймворкам по типу OkHttpClient, Retrofit, Ktor и прочее.

```
val connection = URL(url).openConnection() as HttpURLConnection

connection.apply {
    connectTimeout = TIMEOUT_CONNECT
    readTimeout = TIMEOUT_READ
    instanceFollowRedirects = followRedirects
}

var connection: HttpURLConnection? = null
try {
    connection = createConnection(url)
    connection.connect()

    val responseCode = connection.responseCode
    if (responseCode == HttpURLConnection.HTTP_OK) {
        return downloadBody(connection.inputStream)
    } else {
        val errorResponse = downloadBody(connection.errorStream)
        throw IllegalStateException(errorResponse)
    }
} finally {
    connection?.disconnect()
}
```

И это еще не полный код....

Спасибо комьюнити

Как упоминал выше - любой стандарт можно превратить в фреймворк.

И благодаря кодогенерации (Annotation Processor), можно бойлерплейт сократить.

Самый часто встречаемый фреймворк для упрощения работы с запросами - **Retrofit**.

Для парсинга данных - **GSON**, **Moshi**. Так же набирает популярность **Kotlin Serializable**.

В моих проектах получается такая связка:

Coroutines + Retrofit с OkHttpClient + Gson

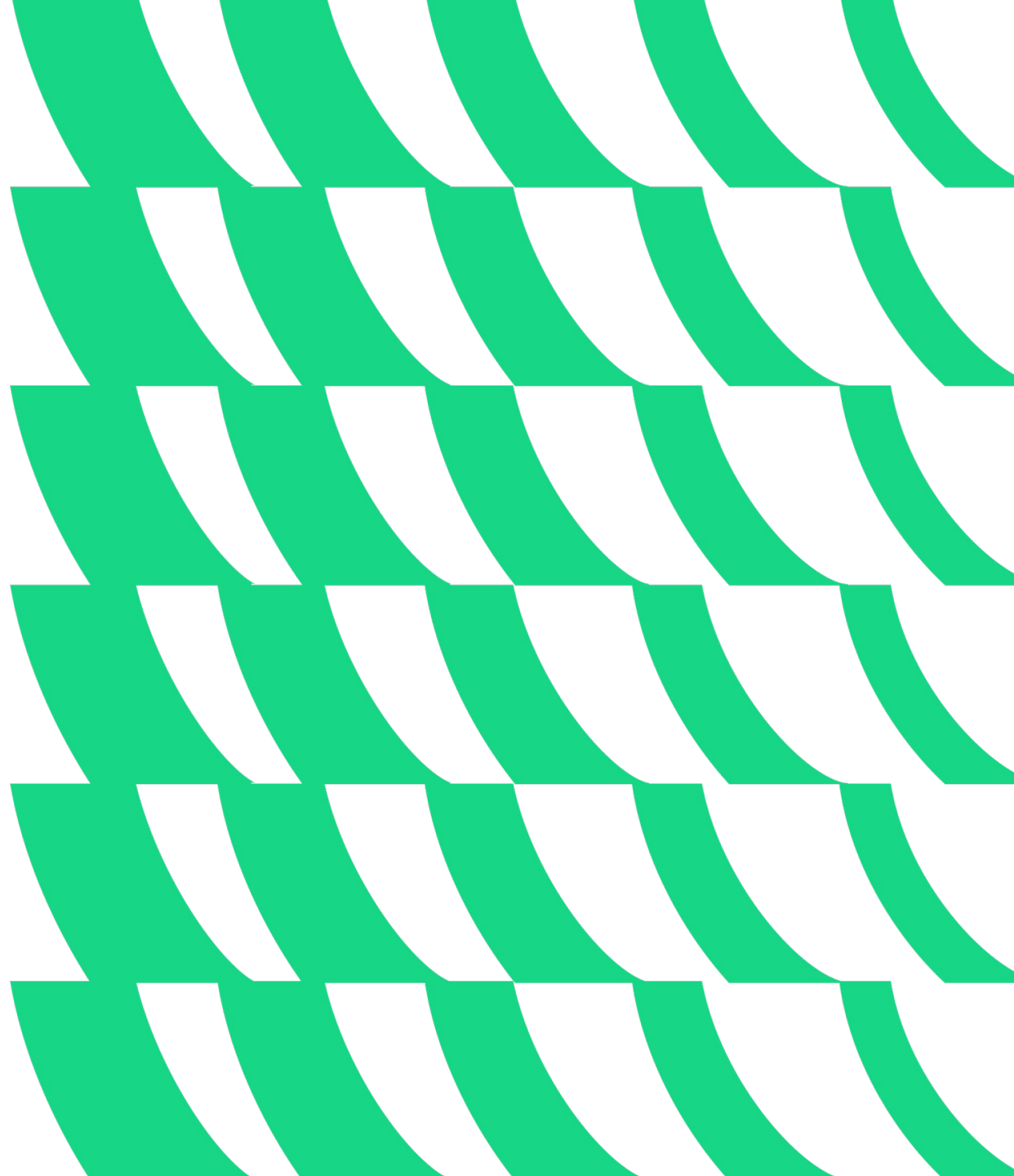
```
internal interface IItemAccessor2 {
    @GET("/api/cats?skip=0&limit=100")
    suspend fun items(): List<Item>
}

val retrofit = Retrofit.Builder().apply {
    addConverterFactory(GsonConverterFactory.create(gson))
    client(okHttpClient)
    baseUrl(baseUrl)
}.build()

val accessor = retrofit.create(IItemAccessor2::class.java)
```

Приложение

Как быстро начать



Android команда Google ранее

“Once we have gotten in to this entry-point to your UI, we really don't care how you organize the flow inside.”

- Dianne Hackborn, Android Framework team

Android команда Google вносит свои коррективы

Проект **Jetpack**:

- MVVM
- Navigation
- Paging
- Compose :)
- Интеграции с Hilt
- и много чего еще



Android команда также помогает обеспечивать обратную совместимость фич внутри androidx компонентов.

Когда-то давно об обратной совместимостью приходилось заниматься самим разработчикам. И это было порой очень “дорого”, после презентации нового дизайна.

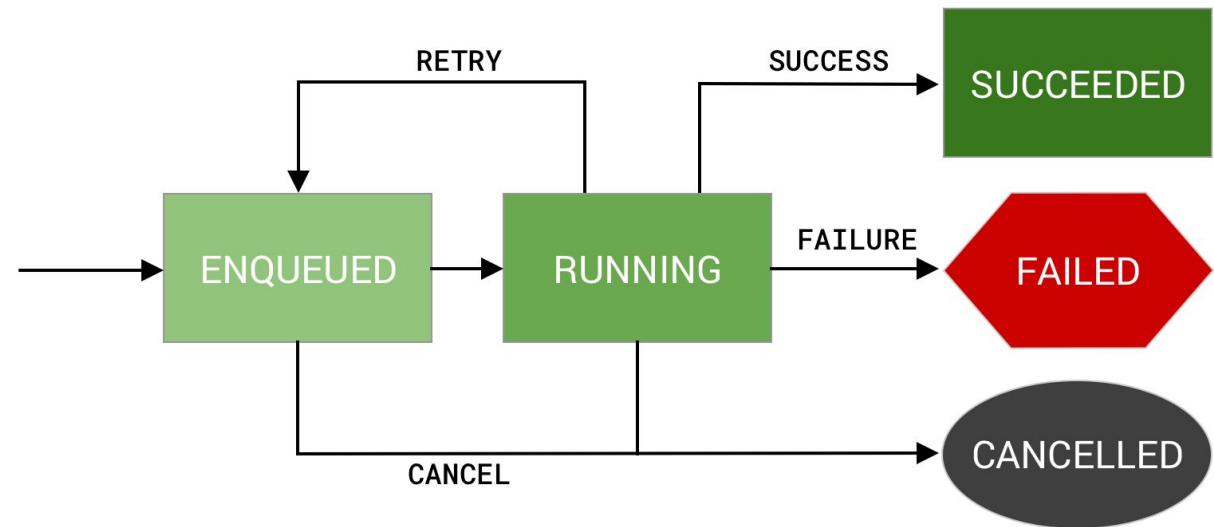
Постепенно многие команды отказываются от сторонних и самописных фреймворков и переходят на гугловые фреймворки.

Обработка ошибок и ожиданий

Неопытные разработчики не учитывают такие моменты, и приложение пишется сначала по пути success-case, к которому прикручивают сбоку обработку ошибок, затягивая время разработки.

Не надо так!:)

Клиент-сервер предполагает, что для получения данных нужно время, а данные могут к тому же не прийти по разным причинам.



Загрузка изображений

Кажется что это очень простая фича.

Но на самом деле - если писать ее с нуля, пробуя учесть все лучшие практики и требования к интерфейсам... то создание такого фреймворка потребует очень много времени.

Настоятельно рекомендуется сразу использовать необходимые фреймворки. Т.к. они учитывают:

- Загрузку
- Кэширование
- Оптимизацию памяти
- Состояния
- Размер конечного ImageView
- Смерть Activity или View
- Переиспользование View

Самый простой фреймворк с минимум фичей:

- **Picasso**

Самый гибкий фреймворк:

- **Glide**

Самый оптимизированный и сложный фреймворк:

- Fresco (Но он требует свои View для отображения)

Самый новый фреймворк:

- Coil

Немного
полайвкодим

Домашнее Задание #2

Опять?



Постановка задачи

Требуется разработать мини приложение, которое будет брать данные по API и обрабатывать ошибки.

Полный текст по [ссылки](#)

Когда сдавать: на РК2 (29 Ноября)

Как защищать: Аналогично ДЗ1. Проверка кода, кейсов и вопросы по коду.

Что будет на РК2: Аналогично РК1. Будут задачи на модификацию кода ДЗ с учетом всех лекций до РК.

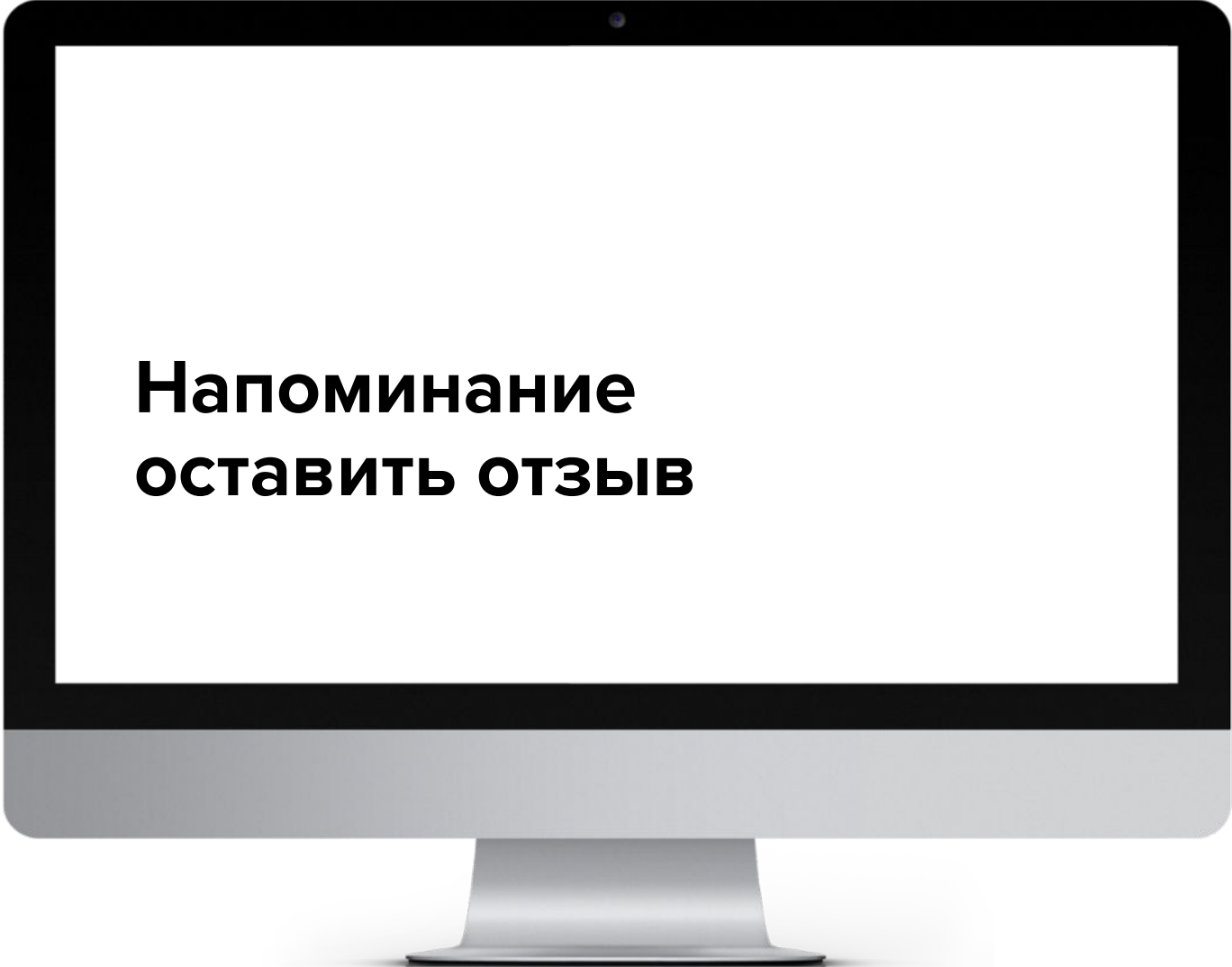
Как будет проходить контроль: Аналогично РК1. Офлайн и онлайн сессии с преподавателями.

Как записаться: Аналогично РК1. Запись через Google-таблицу, с указанием ссылки на профиль и репозиторий.

Пожалуйста, не надо слать код на портале:)

О всех проблемой постарайтесь сообщить заранее. И активнее пользуйтесь помощью менторов в процессе выполнения.





**Напоминание
оставить отзыв**

Спасибо за
внимание!

