

# Многопоточность и сетевые запросы

Павел Галанин

Осень 2023



# Вы находитесь здесь



Немного теории



Многопоточность в  
Java и Kotlin



Сетевые запросы

# Процесс и поток

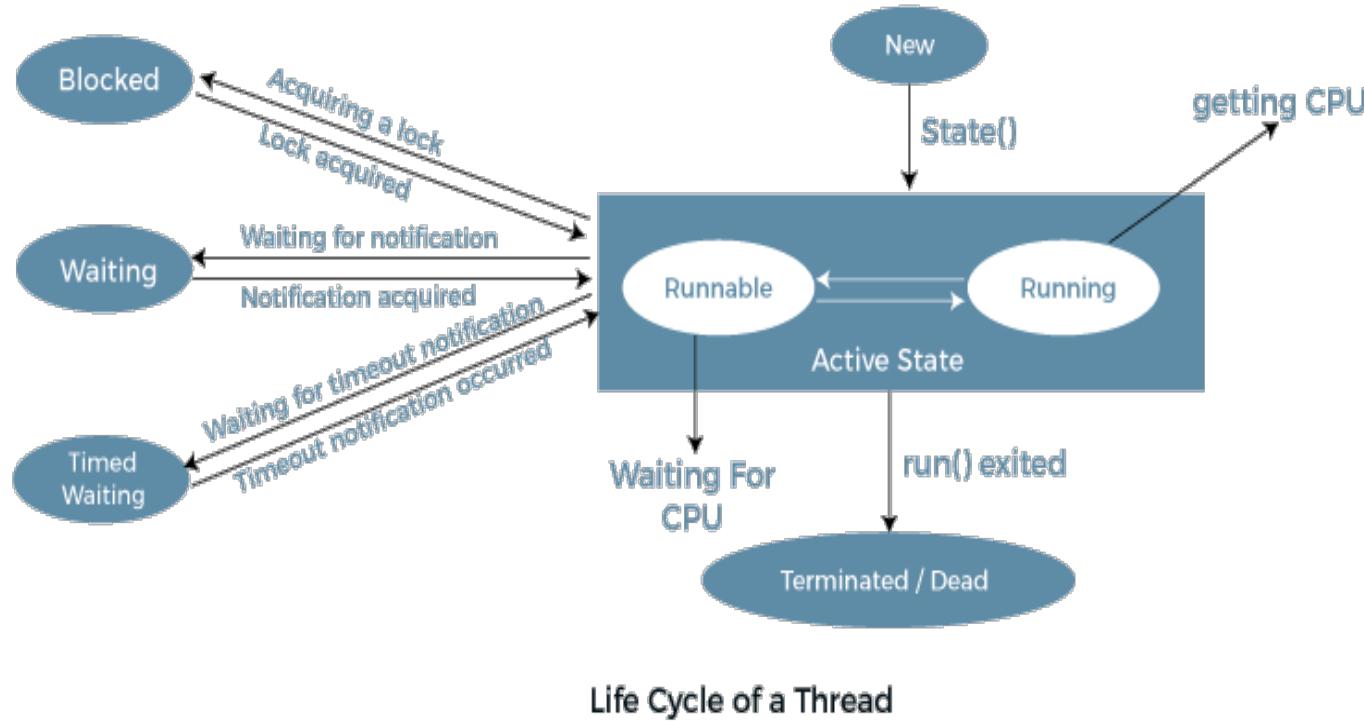
## Процесс (Process)

- ① «экземпляр программы во время выполнения»
- ① ОС выделяет ресурсы на процесс: память, доступ к файлам, устройствам
- ① имеет изолированное адресное пространство – для общения между процессами нужны специальные механизмы межпроцессного взаимодействия
- ① процесс содержит как минимум один поток

## Поток (Thread)

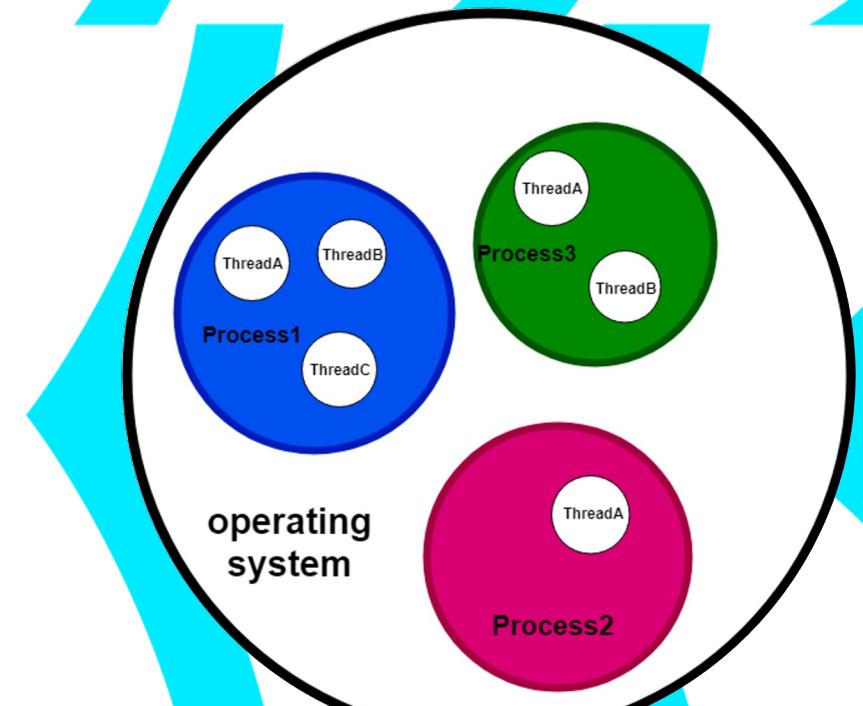
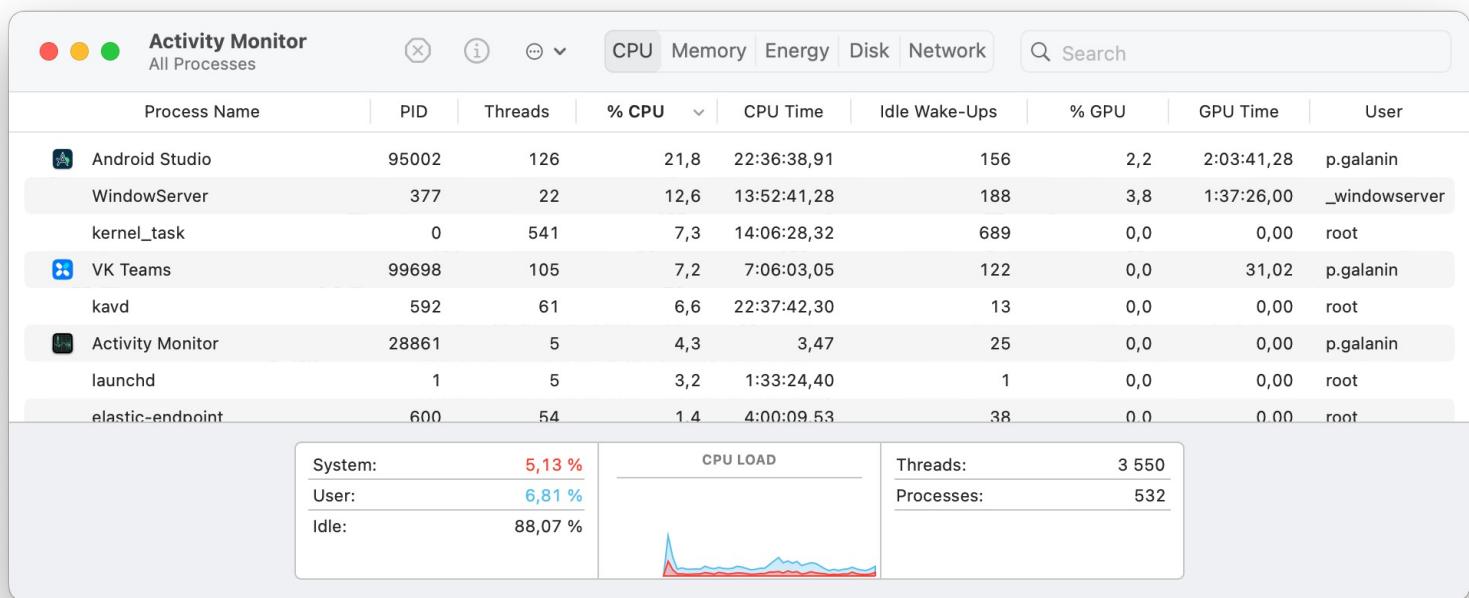
- ① «поток исполнения инструкций», единица планирования исполнения
- ① на поток выделяется ресурс – процессорное время (CPU)
- ① ресурсы, выделяемые на процесс (например адресное пространство), – общие на все потоки процесса
- ① поток имеет небольшую собственную область памяти – стек потока

# Жизненный цикл потока



Enum values	
<b>Thread.State</b>	<b>BLOCKED</b> Thread state for a thread blocked waiting for a monitor lock.
<b>Thread.State</b>	<b>NEW</b> Thread state for a thread which has not yet started.
<b>Thread.State</b>	<b>RUNNABLE</b> Thread state for a runnable thread.
<b>Thread.State</b>	<b>TERMINATED</b> Thread state for a terminated thread.
<b>Thread.State</b>	<b>TIMED_WAITING</b> Thread state for a waiting thread with a specified waiting time.
<b>Thread.State</b>	<b>WAITING</b> Thread state for a waiting thread.

# Процесс и поток



# Процессы и потоки в Android

- при запуске приложения Android создаёт новый Linux-процесс с одним потоком – **главным потоком (main thread)**
- по умолчанию все компоненты приложения (активити, сервисы и т.д.) запускаются в одном и том же процессе на главном потоке
- в манифесте для отдельного компонента можно указать собственный процесс

```
<activity  
    android:name=".viewer.ui.ViewerActivity"  
    android:process="viewer_process" />
```

# Main thread

В Android с визуальными компонентами можно работать только из главного потока – поэтому его ещё называют **UI-поток**.

Из любого потока можно вызвать действие на главном потоке, например с помощью:

- `View.post(Runnable)`
- `Activity.runOnUiThread(Runnable)`
- `View.postDelayed(Runnable, long)`

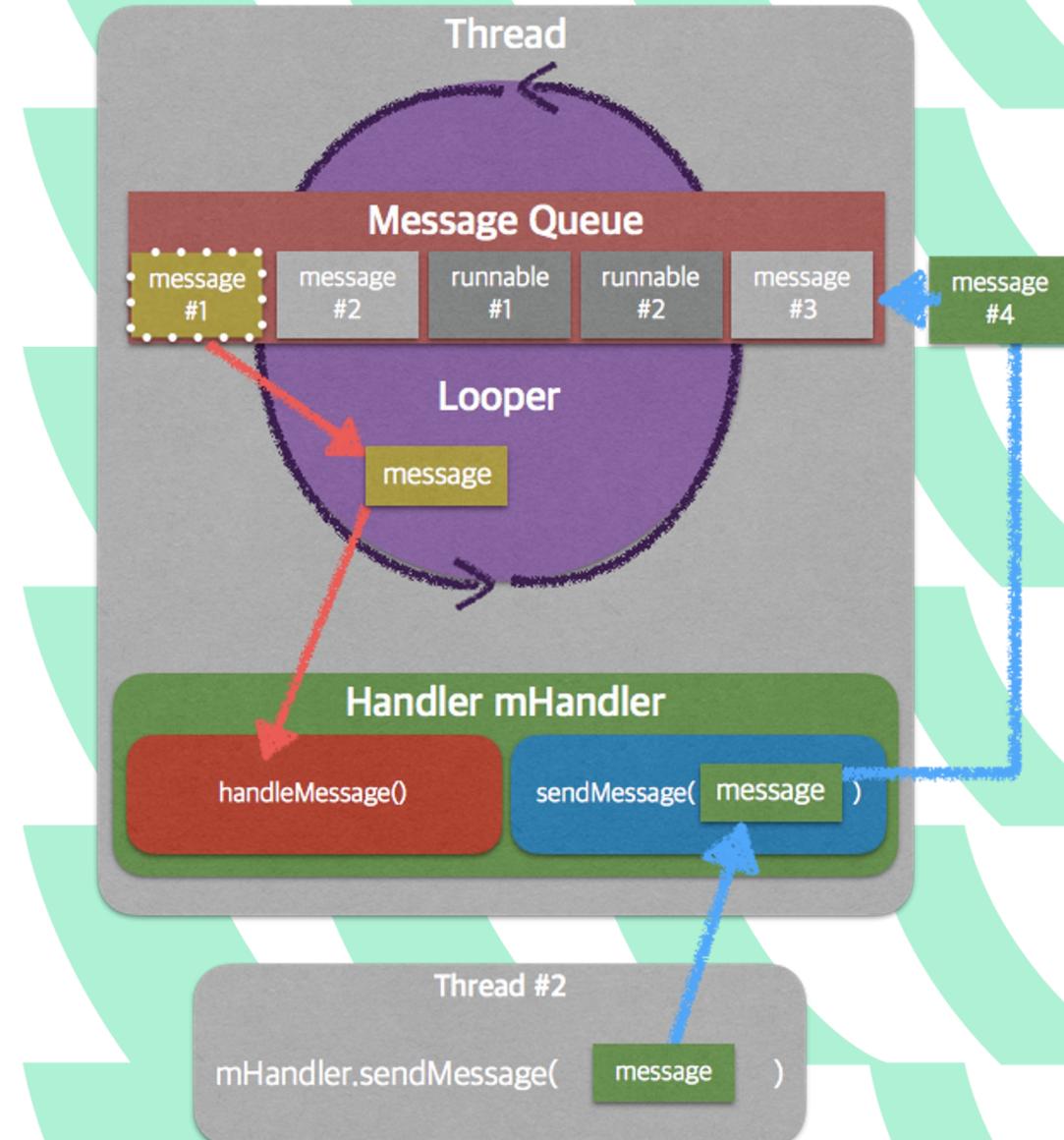
```
fun onClick(v: View) {
    Thread(Runnable {
        // A potentially time consuming task.
        val bitmap = processBitMap(s: "image.png")
        imageView.post {
            imageView.setImageBitmap(bitmap)
        }
    }).start()
}
```

# Main thread

На главном потоке работает **Looper** – бесконечный цикл по выполнению действий. Действия попадают в очередь на выполнение в виде сообщений (**Message**).

Из любого потока можно создать **Handler** и связать его с лупером главного потока (`Looper.getMainLooper()`). Далее через этот handler можно отправлять сообщения – блоки кода – на главный поток:

```
val handler = Handler(Looper.getMainLooper())
handler.post {
    textView.text = "New text"
}
```



# ANR

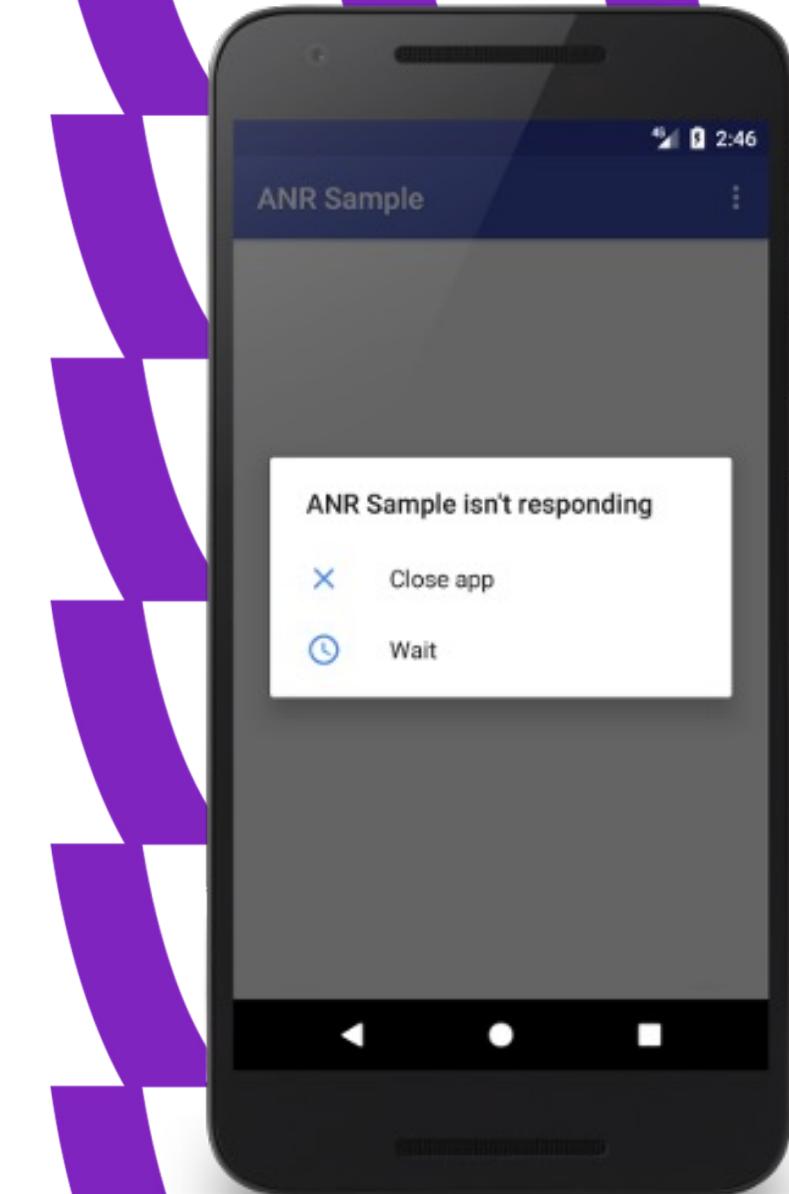
[Application Not Responding](#) – появляется, когда главный поток заблокирован на существенное время и не выполняет UI-операции из очереди.

ANR возникнет, если событие ввода (например касание экрана) не будет обработано за 5 секунд.

Причина ANR – выполнение долгих операций на главном потоке:

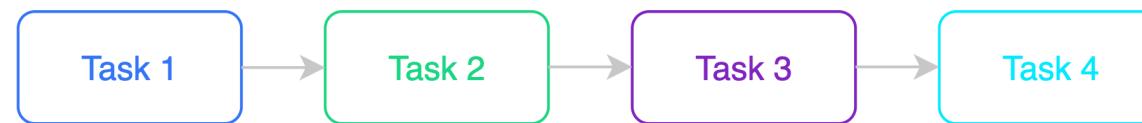
- поход в сеть, в базу данных, в файловое хранилище
- большая вычислительная операция

Для профилактики ANR можно включить [StrictMode](#) (`StrictMode.setThreadPolicy`). Для него можно настроить, чтобы при походе в сеть или файловую систему из главного потока появлялась бы запись в логе или вообще крешилось приложение.

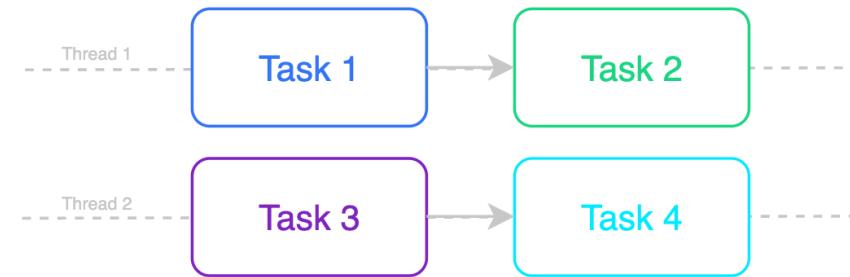


# Последовательное и параллельное программирование

Последовательное

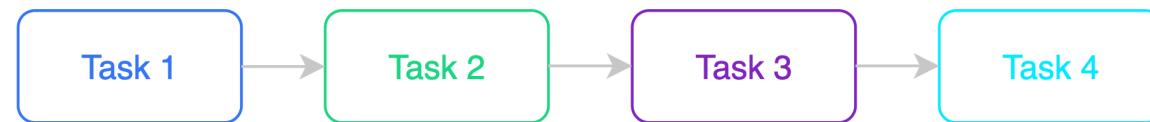


Параллельное

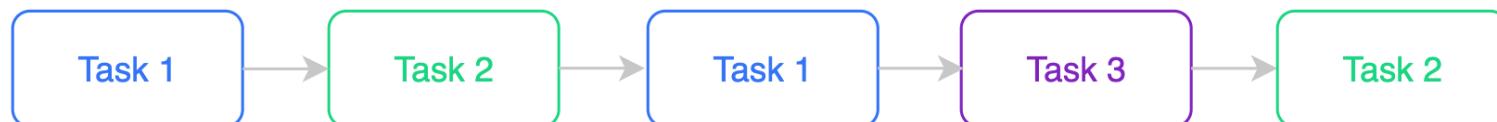


# Синхронное и асинхронное программирование

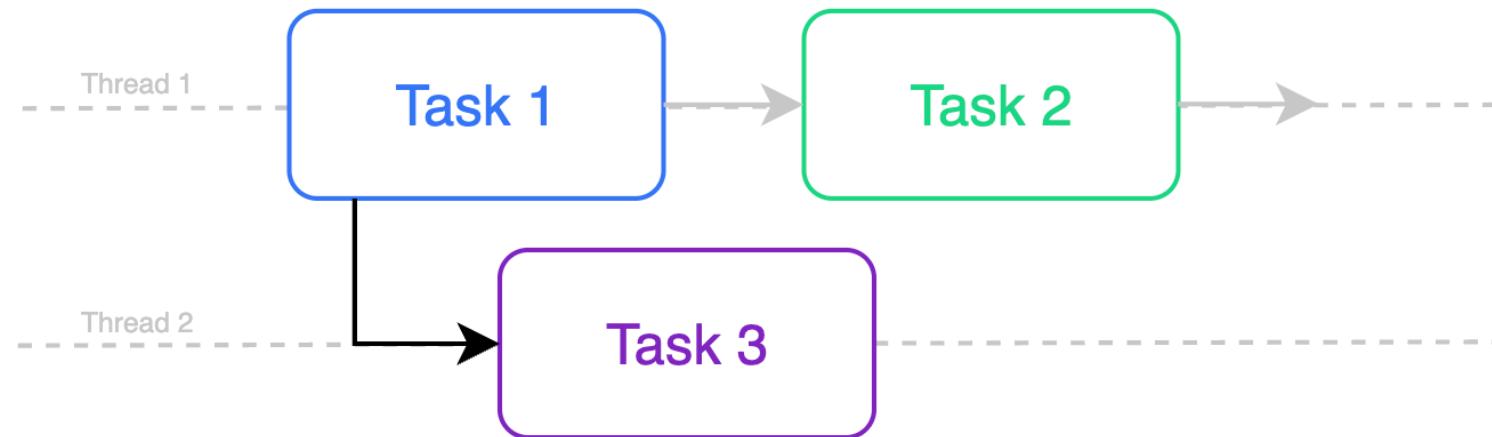
Синхронное



Асинхронное



# Параллельное асинхронное программирование

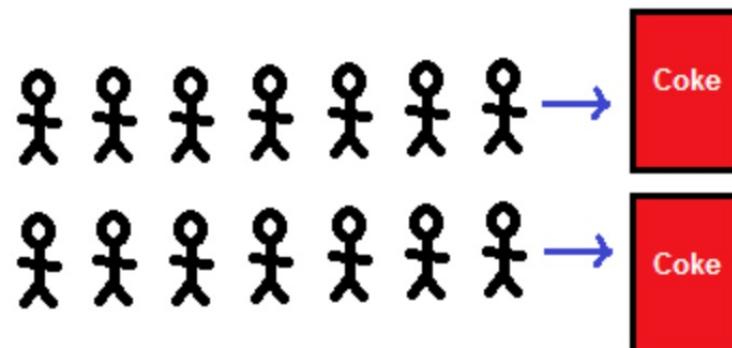


# Конкурентность (Concurrency)

Неточное определение: одновременное выполнение задач на ограниченном общем наборе ресурсов



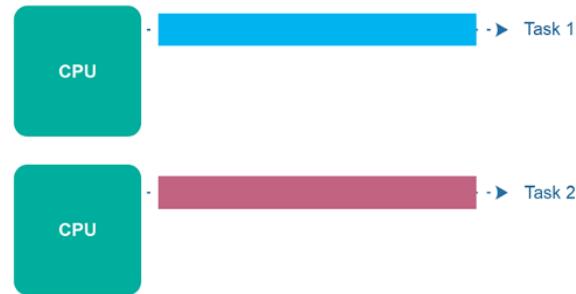
Concurrent: 2 queues, 1 vending machine



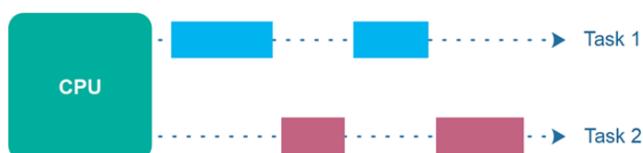
Parallel: 2 queues, 2 vending machines

# Конкурентность (Concurrency)

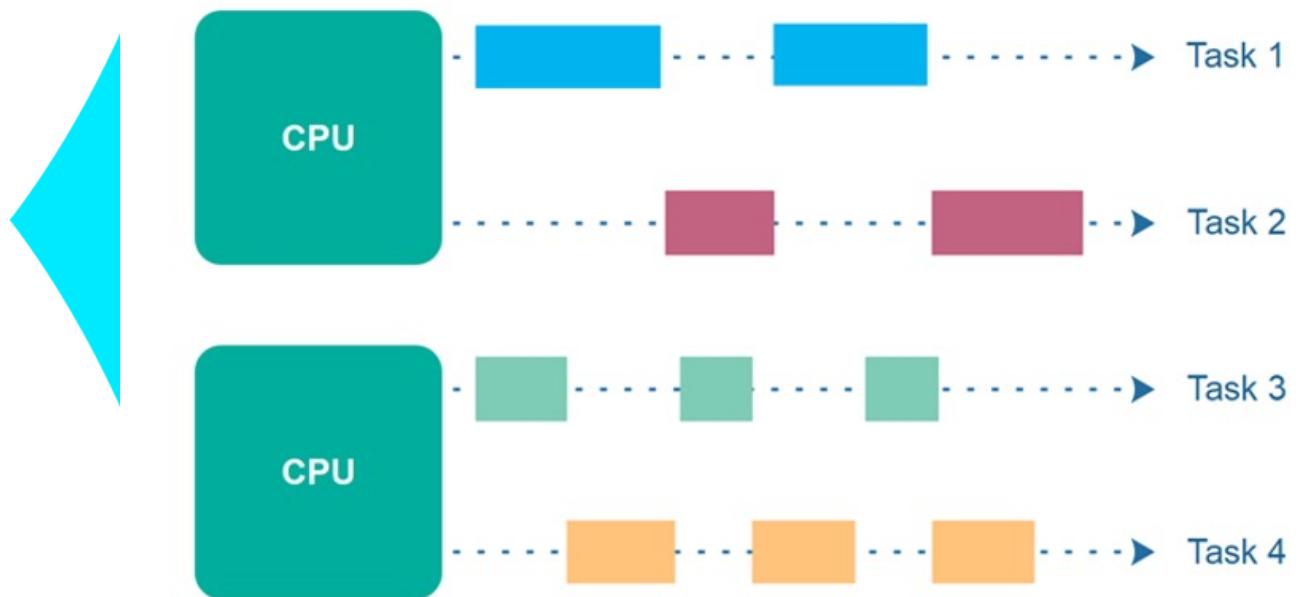
## Parallel Execution



## Concurrency



## Parallel Concurrent Execution



# Вы находитесь здесь



Немного теории



Многопоточность в  
Java и Kotlin



Сетевые запросы

# java.lang.Thread

Самый базовый способ запуска исполнения кода в другом потоке.

```
val thread = Thread(object : Runnable {  
    override fun run() {  
        // some work in another thread  
    }  
})
```

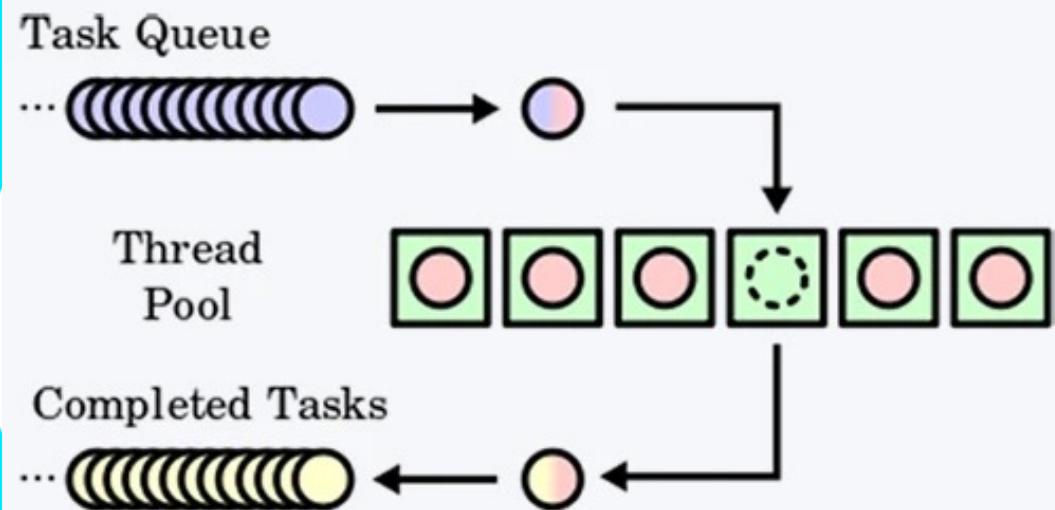
# Пул потоков

Создание потока – тяжёлая операция + для нового потока нужно выделить ресурсы (память под стек).

Если нужно запланировать много задач и под каждую из них создавать поток, можно ухудшить производительность и достичь лимита по ресурсам.

Но можно создать фиксированное число потоков и переиспользовать их. Эти потоки образуют [пул потоков](#) (thread pool).

Задачи для пула добавляются в очередь. По мере освобождения потоков в пуле достаются задачи и отправляются на выполнение.



# Executor, ExecutorService

**Executor** – интерфейс для запуска потоков.  
Позволяет скрыть детали создания потока и передачи задач на исполнение.

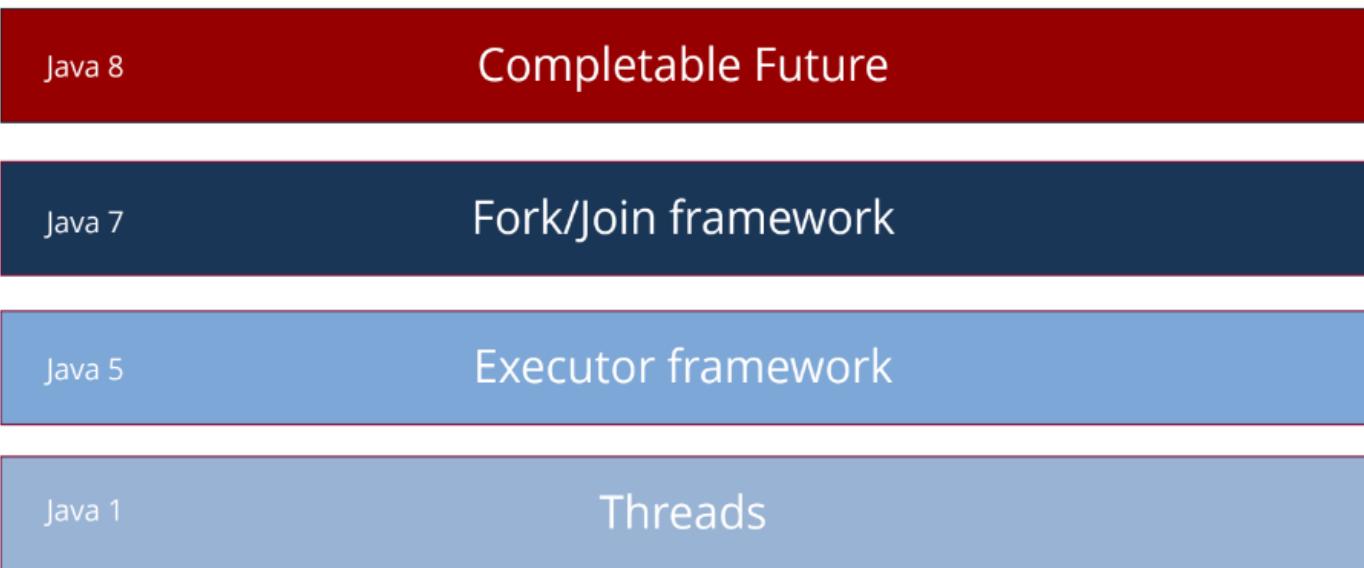
**ExecutorService** расширяет возможности Executor по управлению исполнением. Он определяет очередь исполнения и пул потоков, на котором задачи будут выполняться.

В классе `java.util.concurrent.Executors` находятся методы для создания `ExecutorService`, например `newFixedThreadPool`.

```
public interface ExecutorService extends Executor {  
    no usages  
    void shutdown();  
  
    no usages  
    List<Runnable> shutdownNow();  
  
    no usages  
    boolean isShutdown();  
  
    no usages  
    boolean isTerminated();  
  
    no usages  
    boolean awaitTermination(long var1, TimeUnit var3);  
  
    no usages  
    <T> Future<T> submit(Callable<T> var1);  
  
    no usages  
    <T> Future<T> submit(Runnable var1, T var2);  
  
    no usages  
    Future<?> submit(Runnable var1);  
  
    no usages  
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> var1);  
  
    no usages  
    <T> T invokeAny(Collection<? extends Callable<T>> var1);  
}
```

# CompletableFuture

В Java 8 появился [CompletableFuture](#) – дальнейшее развитие `ExecutorService`. Предоставляет методы для комбинирования нескольких задач, создания цепочек обработки.



# Реактивное программирование



Реактивное  
программирование —  
асинхронность,  
соединенная с  
потоковой  
обработкой данных.

## Зачем?

Частота одного ядра практически не растёт

- ↪ в компьютерах всё больше ядер
- ↪ нужно оптимально использовать (утилизировать) ядра
- ↪ больше потоков богу потоков
- ↪ сложнее координировать работу потоков
- ↪ нужен подход, чтобы было как можно меньше ожиданий/блокировок

В реактивном программировании выстраиваются цепочки обработки событий. Такая цепочка активируется при наступлении некоторого события (т.е. цепочка является **реакцией** на событие). Событием может быть, например, действие пользователя, получение ответа из сети или от базы данных.

# Паттерн Observer

Реактивное программирование основано на паттерне [наблюдатель](#) (observer).

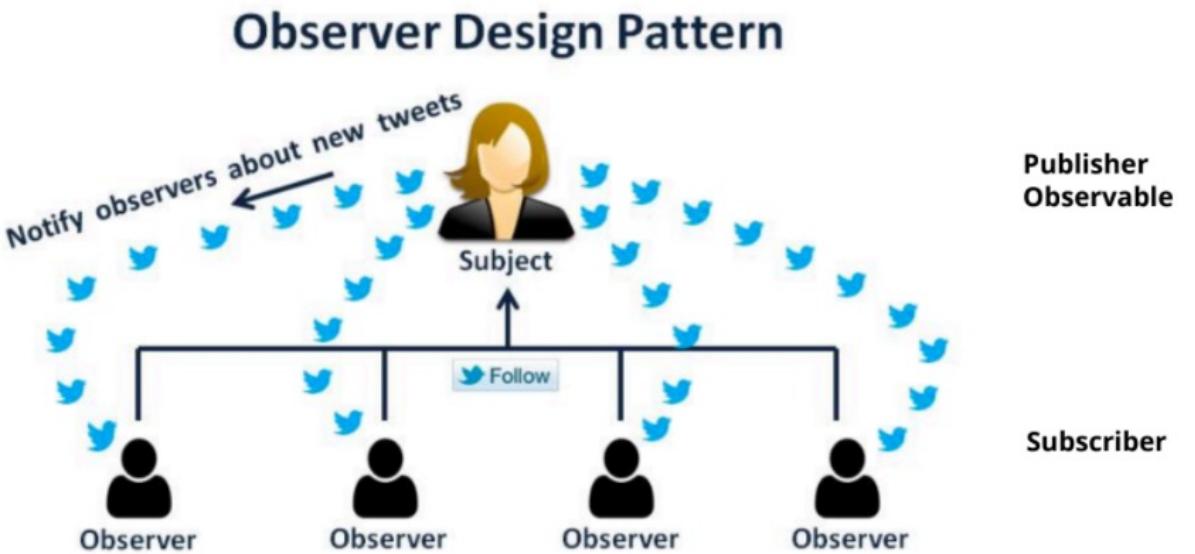
[Publisher \(Observable\)](#) – публикует события

[Subscriber \(Observer\)](#) – подписывается на Publisher

[Subscription](#) – «факт» подписки Subscriber на Publisher

Publisher публикует событие, Subscriber получает уведомление.

Пока нет новых сообщений, Subscriber не блокируется, выполняет другие задачи. При появлении события Subscriber активирует код обработки.



# RxJava

## io.reactivex.rxjava3

Действующие лица:

- **Observable** – генерирует события
- **Observer** – подписывается на события:
  - Next* - очередная порция данных
  - Error* - произошла ошибка
  - Completed* - поток завершен
- **Subscription** – связь между Observable и Observer, позволяет управлять подпиской (например прекращать)
- **Subject** – наследуется от Observable и одновременно реализует Observer – позволяет обработать и ретранслировать события другим подписчикам

```
public interface Observer<@NotNull T> {
```

Provides the **Observer** with the means of cancelling (disposing) the connection (channel) with the **Observable** in both synchronous (from within `onNext(Object)`) and asynchronous manner.

Params: `d` – the **Disposable** instance whose `Disposable.dispose()` can be called anytime to cancel the connection

Since: 2.0

```
void onSubscribe(@NotNull Disposable d);
```

Provides the **Observer** with a new item to observe.

The **Observable** may call this method 0 or more times.

The **Observable** will not call this method again after it calls either `onComplete` or `onError`.

Params: `t` – the item emitted by the **Observable**

```
void onNext(@NotNull T t);
```

Notifies the **Observer** that the **Observable** has experienced an error condition.

If the **Observable** calls this method, it will not thereafter call `onNext` or `onComplete`.

Params: `e` – the exception encountered by the **Observable**

```
void onError(@NotNull Throwable e);
```

Notifies the **Observer** that the **Observable** has finished sending push-based notifications.

The **Observable** will not call this method if it calls `onError`.

```
void onComplete();
```

```
}
```

# Многопоточность в RxJava

`subscribeOn` – задаёт thread, на котором будет выполняться генерация событий

`observeOn` – задаёт thread, на котором будет выполнена обработка событий  
(`onNext/onError/onCompleted`)

`Scheduler` определяет, на каком потоке будет выполнена генерация и обработка:

- `immediate` - выполнит синхронно, в том же потоке
- `trampoline` - в том же потоке, но после завершения обработки текущей задачи
- `newThread` - новый поток для выполнения задачи
- `computation` - для сложных вычислений на CPU
- `io` - для операций ввода/вывода
- `test` - удобен для тестирования и отладки

```
val observable = Observable.create { o ->
    o.onNext( value: 1)
    o.onNext( value: 2)
    o.onComplete()
}.subscribeOn(Schedulers.newThread())
```

```
observable
    .observeOn(Schedulers.newThread())
    .subscribe { i ->
        println(i)
    }
```



# RxJava

- + гибкий, много возможностей под разные кейсы
- + большая распространённость – много документации, разобранных кейсов

- достаточно сложный синтаксис, высокий порог входа
- завязывает на себя API приложения – есть шанс очень сильно завязать приложение на этот фреймворк
- не очень kotlin-friendly

# Пульс-опрос

Пройдите, пожалуйста



# Road to coroutines...

Колбеки – громоздкие (**callback hell**).

Future, RxJava – сложные, практически отдельный язык для написания асинхронного кода.

## Callback hell

```
connectToDatabase()
  .then((database) => {
    return getUsers(database)
      .then((users) => {
        return getUserSettings(database)
          .then((settings) => {
            return enableAccess(user, settings);
          })
        })
      })
  })
```

## RxJava complexity

```
// rx java
fun openTask() {
  mTaskDetailView.setLoadingIndicator(true)
  mCompositeDisposable.add(mTasksRepository
    .getTask(mTaskId)
    .filter(Optional::isPresent)
    .map(Optional::get)
    .subscribeOn(mSchedulerProvider.computation())
    .observeOn(mSchedulerProvider.ui())
    .subscribe(
      // onNext
      this::showTask,
      // onError
      throwable -> {
      },
      // onCompleted
      () -> mTaskDetailView.setLoadingIndicator(false)))
}
```

# Корутины

Корутины позволяют

писать асинхронный код в  
том же стиле, в каком мы  
пишем синхронный.

Можно использовать  
обычные операторы:  
циклы, условия, try-catch\*,  
коллекции и т.д.

```
// rx java
fun openTask() {
    mTaskDetailView.setLoadingIndicator(true)
    mCompositeDisposable.add(mTasksRepository
        .getTask(mTaskId)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .subscribeOn(mSchedulerProvider.computation())
        .observeOn(mSchedulerProvider.ui())
        .subscribe(
            // onNext
            this::showTask,
            // onError
            throwable -> {
                },
            // onCompleted
            () -> mTaskDetailView.setLoadingIndicator(false)))
}

// coroutines
fun openTask() = launch(UI, parent = job) {
    mTaskDetailView.setLoadingIndicator(true)

    val task = withContext(CommonPool) { mTasksRepository.getTask(mTaskId) }
    if(task != null) showTask(task)

    mTaskDetailView.setLoadingIndicator(false)
}
```

\* с try-catch есть нюансы

# Корутины. Suspension

В основе идея **приостановки** (suspension):

Suspend-метод может приостановить своё выполнение (например на время ожидания ответа от сервера) и затем, позже, продолжить своё выполнение с этого же места.

Такие методы должны выполняться внутри **корутин**. Корутины можно воспринимать как очень легковесные потоки.

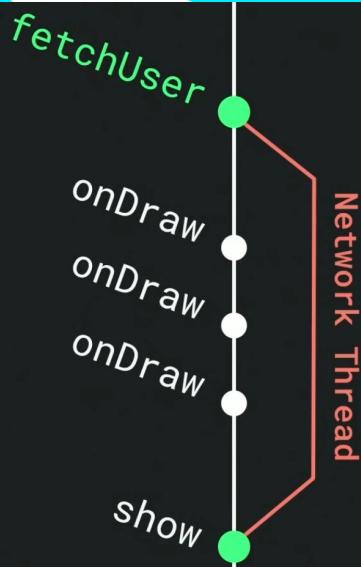
blocking.kt

```
fun loadUser() {  
    val user = api.fetchUser()  
    show(user)  
}
```



coroutines.kt

```
suspend fun loadUser() {  
    val user = api.fetchUser()  
    show(user)  
}
```

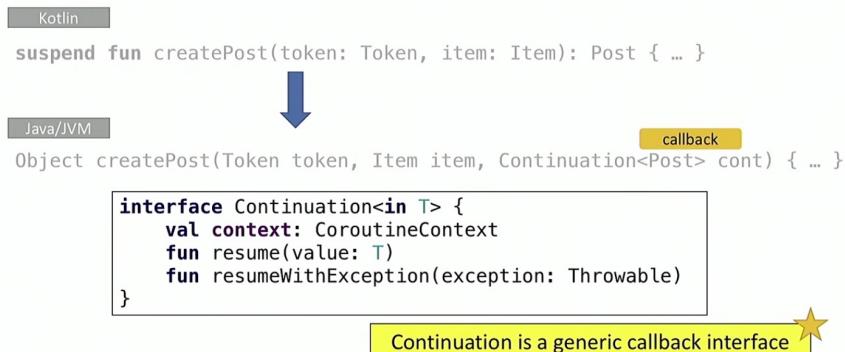


# Корутины под капотом

В suspend-методы при компиляции добавляется аргумент [Continuation](#).

По сути это тот же колбек, но:

- генерирует компилятор
- используется один мета-колбек на всю цепочку вызовов



Внутри Continuation реализована стейт-машина.

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Kotlin

```
↳ val token = requestToken()
↳ val post = createPost(token, item)
processPost(post)
```

Java/JVM

```
switch (cont.label) {
    case 0:
        cont.label = 1;
        requestToken(cont);
        break;
    case 1:
        Token token = (Token) prevResult;
        cont.label = 2;
        createPost(token, item, cont);
        break;
    case 2:
        Post post = (Post) prevResult;
        processPost(post);
        break;
}
```

Compiles to state machine  
(simplified code shown)

# Создание корутин

## Билдеры корутин:

- `launch` – «запустил и забыл», возвращает объект `Job`, с помощью которого можно дождаться или завершить корутину
- `async` – как `launch`, но можно получить значение, которое вернула корутина; `async` возвращает объект `Deferred<T>`
- `runBlocking` -- запустит корутина и дождётся окончания выполнения

```
fun example() = runBlocking { this: CoroutineScope
    val job = launch { this: CoroutineScope
        delay( timeMillis: 1000L)
    }

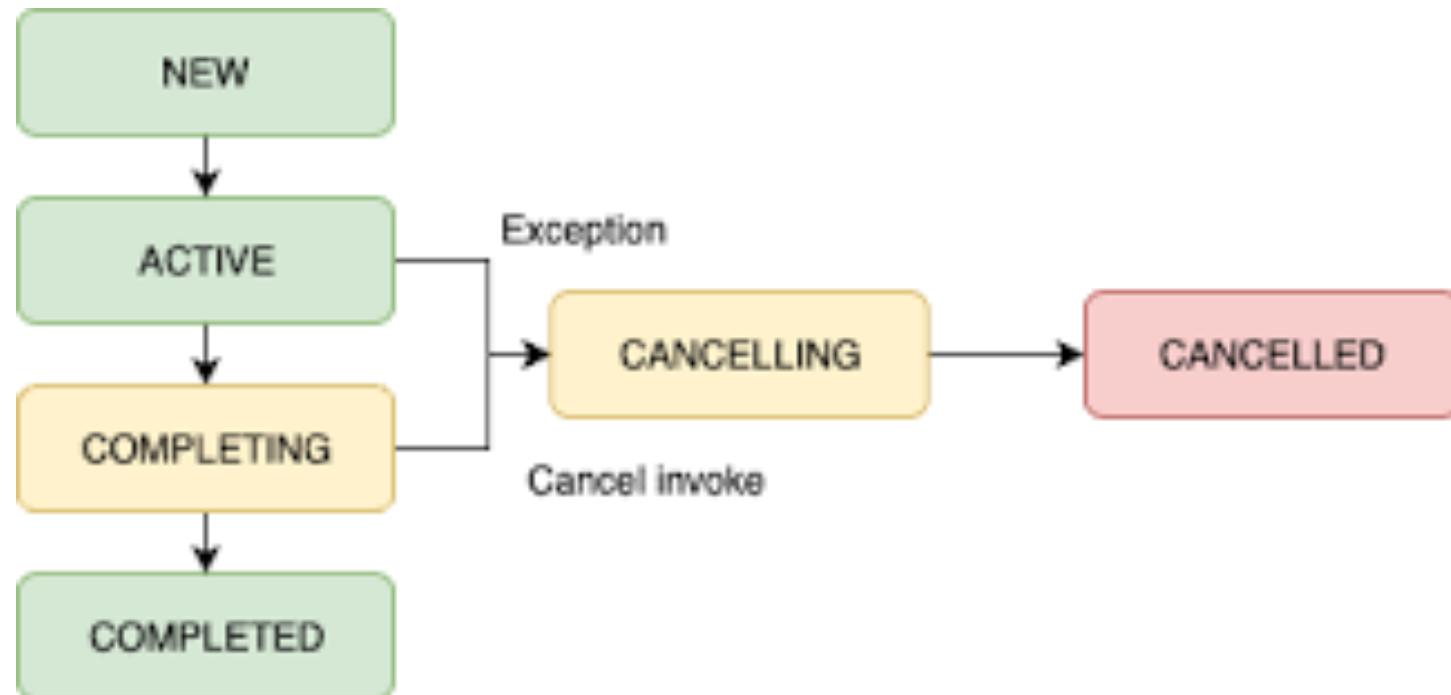
    val result: Deferred<String> = async { this: CoroutineScope
        delay( timeMillis: 1500L)
        "Result" ^async
    }

    job.join()
    println(result.await())

    println("Done")
}
```



# Жизненный цикл корутины



# Скоуп корутины

Для запуска корутины нужен **скоуп** (scope), который определяет жизненный цикл всех относящихся к нему корутин.

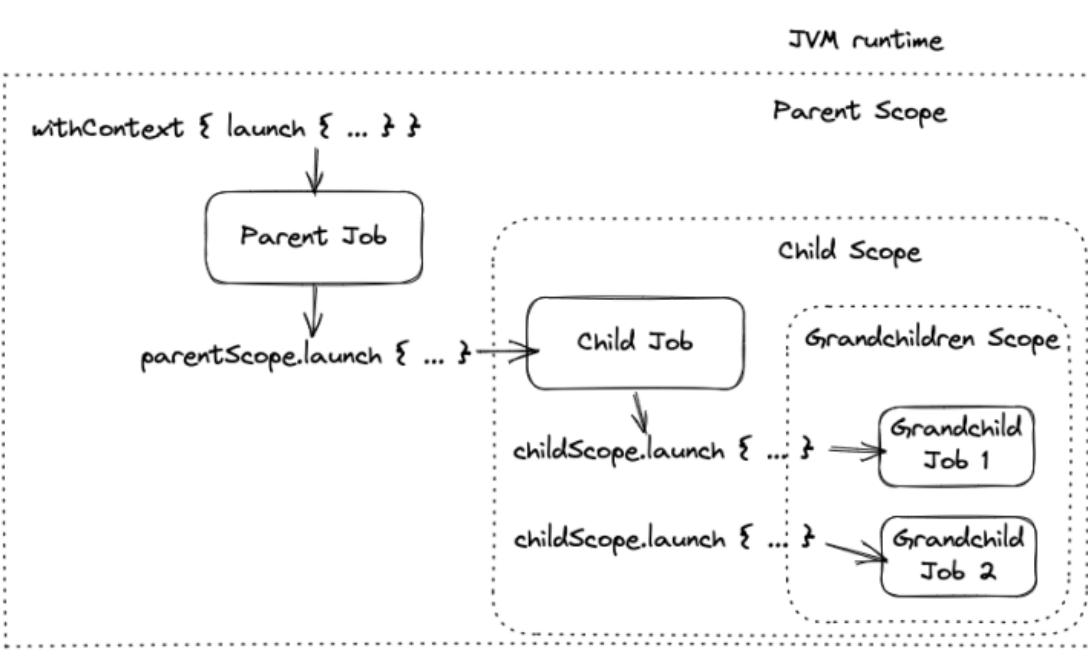
Билдеры `launch` и `async` – функции-расширения `CoroutineScope`.

Функция-расширение `cancel` позволяет завершить все корутины в скоупе.

В библиотеке корутин из коробки есть `GlobalScope` – единый скоуп на весь процесс.

В библиотеках Android предопределены:

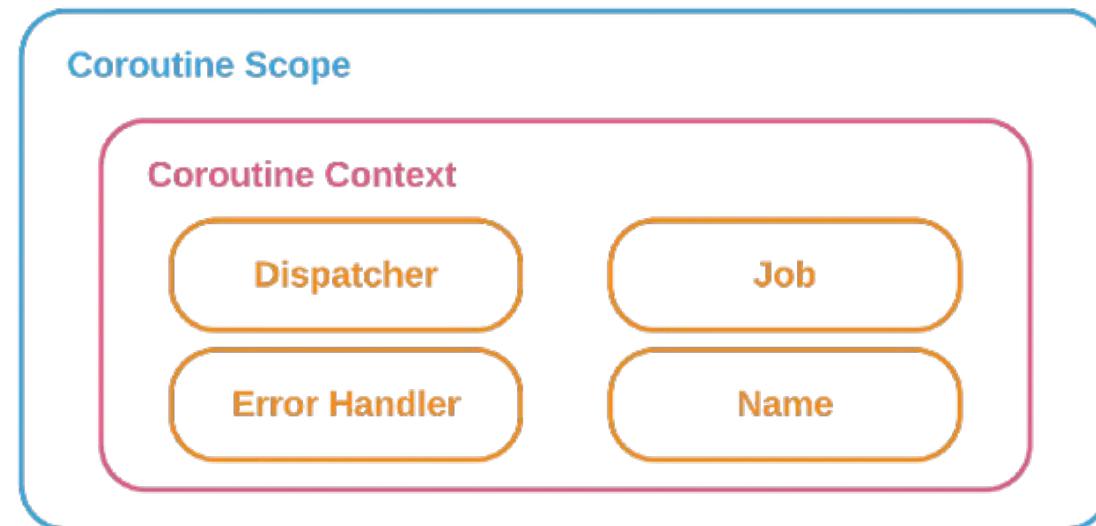
- `viewModelScope`
- `lifecycleScope`



# Контекст корутины

Контекст корутины (CoroutineContext) – хранилище настроек и данных, необходимых для её работы:

- **Job** – объект для управления корутиной (ожидание – `join`, завершение – `cancel`)
- **Dispatcher** – определяет потоки (пул), на которых будет выполняться корутина
- **Name** – просто имя корутины
- **Error Handler** – обработчик исключений



# Dispatchers

`Dispatchers.Unconfined` – корутина запустится на том же потоке

`Dispatchers.Default` – пул для интенсивных CPU-вычислений (число потоков совпадает с числом ядер)

`Dispatchers.IO` – пул для операций ввода/вывода (число потоков превышает число ядер)

`Dispatchers.Main` – в Android соответствует главному потоку



# Обработка исключений в корутинах

Если корутина создавалась через `launch`, то исключение, брошенное внутри, будет проброшено до обработчика исключений **корневой** корутины.

Оборачивание `launch` в `try-catch` не даст эффекта – при возникновении исключения код **не** попадёт в блок `catch`.

Если корутина создавалась через `async`, то выброшенное в корутине исключение будет «отложено» до выполнения действия над полученным `Deferred` – например, до вызова `await()`.

Вызов `await()` **можно** обернуть в `try-catch` и поймать исключение.

```
▶ fun main() = runBlocking { this: CoroutineScope
    // root coroutine with launch
    val job = GlobalScope.launch { this: CoroutineScope
        println("Throwing exception from launch")
        // Will be printed to the console by Thread.defaultUncaughtExceptionHandler
        throw IndexOutOfBoundsException()
    }
    job.join()
    println("Joined failed job")
    // root coroutine with async
    val deferred = GlobalScope.async { this: CoroutineScope
        println("Throwing exception from async")
        // Nothing is printed, relying on user to call await
        throw ArithmeticException()
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
```



# Scope-методы создания корутин

Scope-методы позволяют создавать корутины – как и билдеры, но есть отличия:

Билдеры	Scope-функции
launch, async	coroutineScope, withContext, withTimeout
функции-расширения интерфейса CoroutineScope	suspend-функции
контекст корутины берут из объекта, для которого вызывается функция-расширение	контекст корутины получают из Continuation
исключения пробрасываются в родительскую Job	исключения бросаются как обычно, можно использовать try-catch
запускают асинхронную корутину	корутина запускается сразу же, родительская корутина ждёт окончания новой корутины



# Flow

Flow в общем случае генерирует поток значений, обработка которых выполняется в корутинах. Обработкой значений занимается наблюдатель(и).

**Холодные потоки** – потоки без состояния – создаются по требованию каждый раз, когда наблюдатель подписывается на поток. Каждый наблюдатель получает собственную последовательность значений.

**Горячие потоки** – генерируют значения независимо от наличия наблюдателей. Все наблюдатели получают одни и те же значения.

```
val flow = flow<String> { this: FlowCollector<String>
    → emit(value: "1")
    → delay(timeMillis: 1000L)
    → emit(value: "2")
    → delay(timeMillis: 1000L)
    → emit(value: "3")
}
println("Flow created")
→ delay(timeMillis: 1000L)
println("Subscribe to flow")

runBlocking { this: CoroutineScope
    → flow.collect { it: String
        | println("Value received: $it")
    }
}
```



# StateFlow и SharedFlow

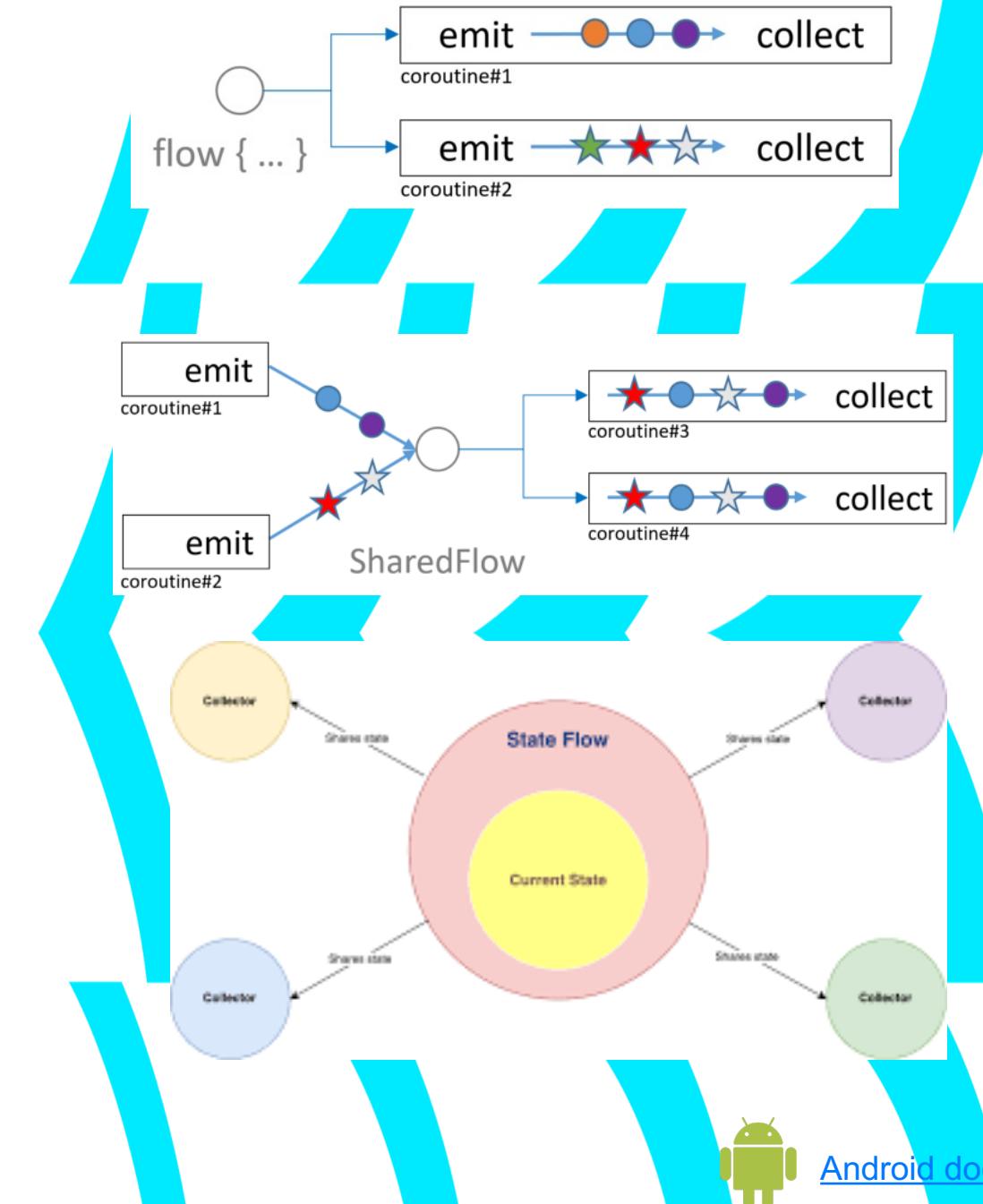
## StateFlow

- горячий поток
- всегда имеет состояние (изначально – дефолтное)
- при подписке слушатель получает текущее состояние
- при изменении состояния все слушатели получают один и тот же его экземпляр

## SharedFlow

- горячий поток
- может хранить и отдавать слушателям n последних значений (параметр `replay`)
- не требует дефолтного значения

SharedFlow – обобщение StateFlow, который можно воспринимать как SharedFlow с `replay=1`.



# Вы находитесь здесь



Немного теории



Многопоточность в  
Java и Kotlin



Сетевые запросы

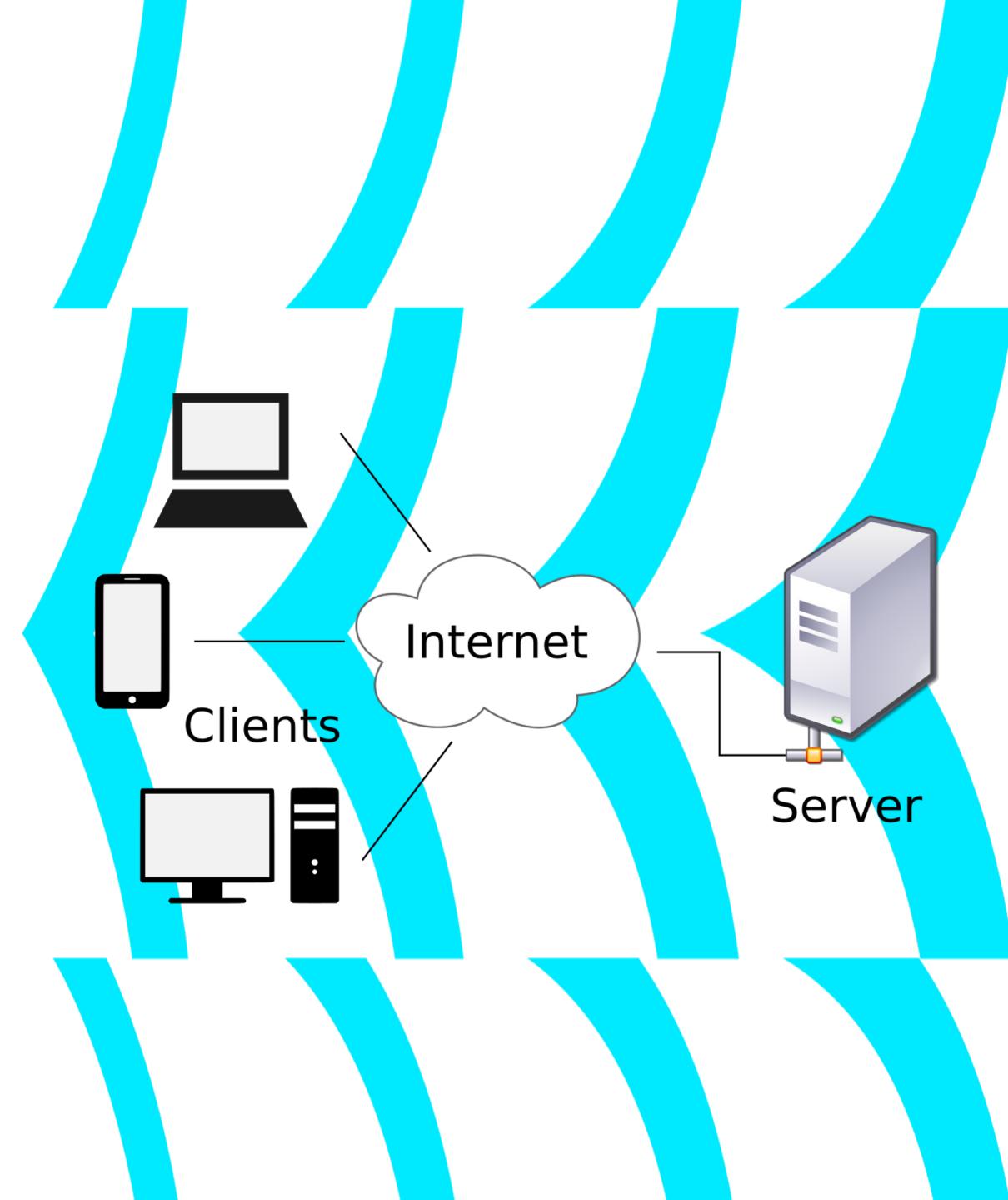
# Клиент - сервер

Подавляющее большинство современных приложений использует интернет (авторизация, синхронизация данных, настроек, получение рекламы).

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

**API (Application Programming Interface)** – интерфейс для взаимодействия; в контексте сети – взаимодействия с сервером.

Самой распространённой архитектурой сетевого API является **REST**, построенный на основе **HTTP**.



# HTTP

HTTP (Hypertext Transfer Protocol) позволяет передавать через web информацию: текст, медиа, файлы и т.д.

Клиент отправляет к серверу **запрос** (request), сервер возвращает **ответ** (response). Запрос и ответ состоят из **заголовка** (header) и **тела** (body).

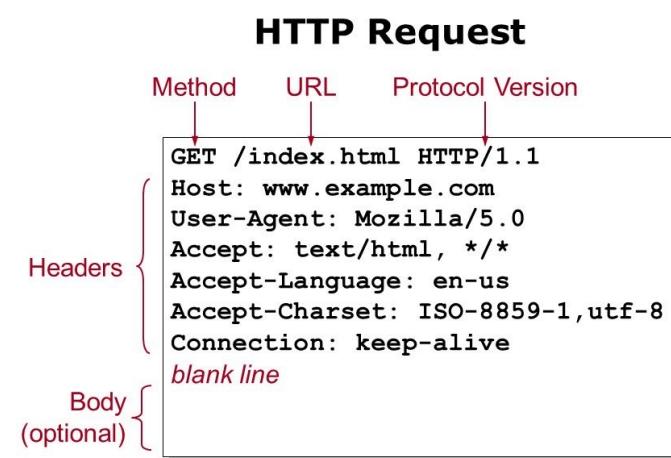
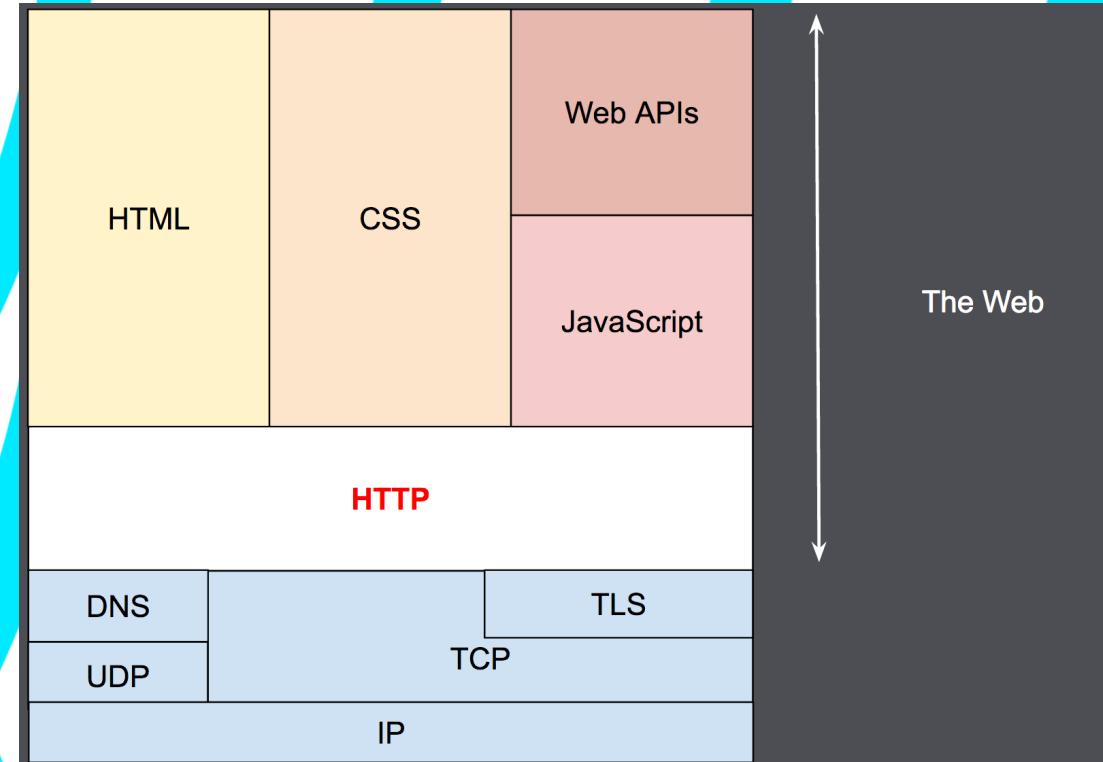
Основные поля заголовка запроса:

- адрес сервера в интернете (host)
- URL требуемого ресурса
- **метод**: GET, POST, PUT, DELETE

В ответе есть поле **status code**. Самые распространённые коды:

- 200 OK
- 300 Moved Permanently
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 500 Internal Server Error

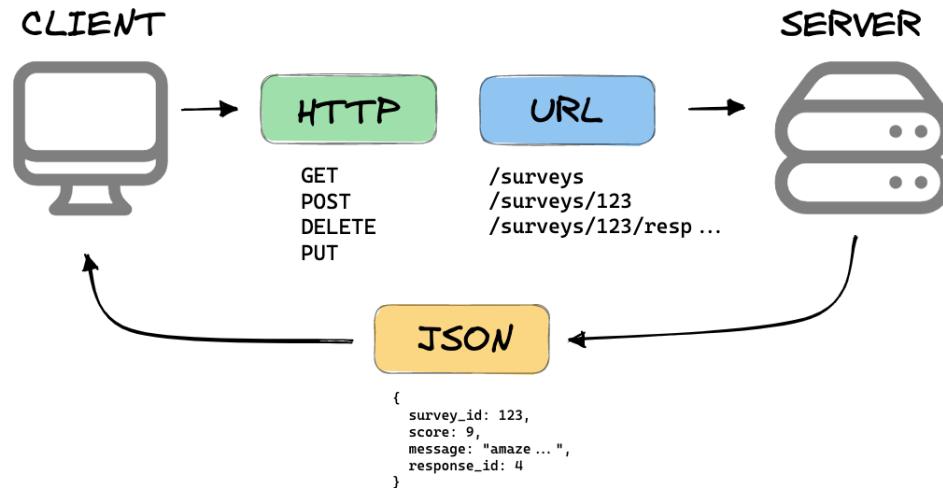
**HTTPS**: HTTP + SSL/TLS – тело запроса и ответа шифруются + с помощью сертификатов проверяется, что клиент и сервер – «те, за кого себя выдают».



# REST

REST API (REpresentational State Transfer API) построен на передаче «репрезентативного состояния» ресурса через протокол HTTP. Понятия REST:

- Ресурс – данные/объект, к которым можно обратиться через этот API
- URI ресурса – уникальный идентификатор ресурса (может иметь вид, например, /users/{id})
- Действие над ресурсом: в общем случае ресурс можно получить (GET), создать (POST/PUT), обновить (PUT/POST), удалить (DELETE)
- Результат действия – можно увидеть в теле (например, JSON с запрошенным ресурсом) и/или в заголовках (например, код 200 в ответ на удаление ресурса)



# OkHttp

com.squareup.okhttp3:okhttp

OkHttp – библиотека для выполнения HTTP-запросов.



Гибкость, широкая функциональность



Для типовых действий приходится писать бойлерплейт-код.

```
val request = Request.Builder() Request.Builder
    .get() Request.Builder!
    .url("https://publicobject.com/helloworld.txt")
    .addHeader(name: "Authorization", value: "Bearer 12345")
    .addHeader(name: "Content-Type", value: "application/json")
    .build()

try {
    client.newCall(request).execute().use { response ->
        if (!response.isSuccessful) {
            throw IOException("Запрос к серверу не был успешен:" +
                " ${response.code()} ${response.message()}")
        }
        // пример получения конкретного заголовка ответа
        println("Server: ${response.header(name: "Server")}")
        // вывод тела ответа
        println(response.body()!!.string())
    }
} catch (e: IOException) {
    println("Ошибка подключения: $e");
}
```

# Retrofit

com.squareup.retrofit2:retrofit

По сути является обёрткой над OkHttp, упрощающей некоторые типовые задачи.

В частности:

- упрощается создание запроса – часть кода генерируется автоматически при компиляции
- просто настраивается парсер ответа (GSON подключается в одну строчку)

Создан и поддерживается той же компанией, что и OkHttp – Square.

```
interface Api {  
    @GET("users/{user}/repos")  
    suspend fun listRepos(  
        @Path("user") user: String  
    ): Response<List<Repo>>  
  
    @Headers("User-Agent: Retrofit-Sample-App")  
    @GET("users/{username}")  
    fun getUser(  
        @Path("username") username: String  
    ): Response<User>  
  
    @POST("users/new")  
    fun createUser(@Body user: User): Response<User>  
}
```

# GSON

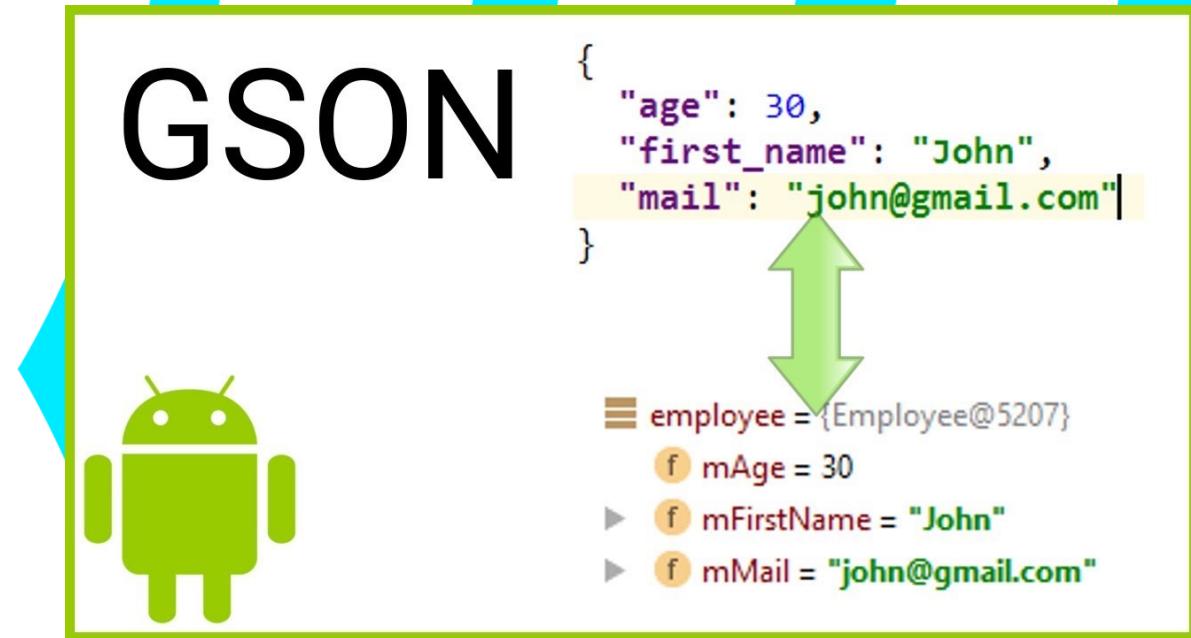
com.google.code.gson:gson

GSON – библиотека для **сериализации** и **десериализации**: конвертирует из JSON-строки в объект и обратно.

Retrofit + GSON – одна из самых популярных связок библиотек для работы с сетью в Android-приложениях.

Retrofit предоставляет расширение для быстрого подключения GSON:

```
implementation 'com.squareup.retrofit2:converter-gson'
```



# Библиотеки для загрузки изображений

Библиотеки для загрузки изображений берут на себя загрузку из сети, кеширование и отображение изображений.

## Picasso

- + быстро и легко настроить базовую загрузку
- не так много продвинутых фичей
- не особо активное добавление новых возможностей

## Glide

- + богатая функциональность, из коробки есть фичи для преобразования изображений (обрезка, вращение и т.д.)
- + поддержка Compose
- достаточно сложная настройка
- большой объём либы

## Coil

- + самая свежая – все самые модные фичи
- + Kotlin-first, корутины
- + поддержка Compose
- пока мало историй использования

Спасибо за  
внимание!

