ТЕХНОТРЕК

Занятие №5

Современная разработка под Android

Юрий Береза
Кирилл Филимонов

Напоминание

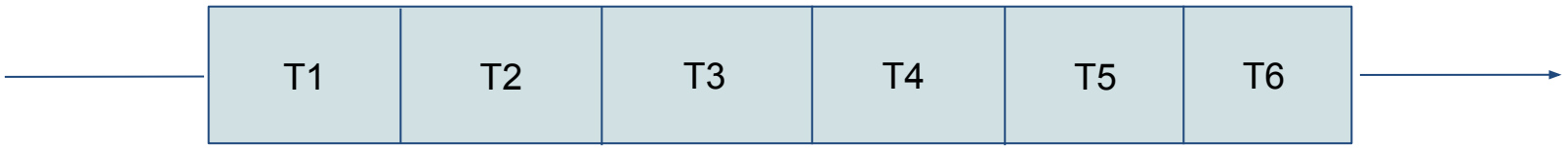
А ты отметился о присутствии на занятии?

Agenda

1. Многопоточность и асинхронность
2. Процессы и потоки
3. Примитивы синхронизации
4. Ошибки конкурентного доступа
5. Java и java.util.concurrent
6. Инструменты Kotlin
7. Инструменты Android
8. RxJava
9. Coroutines

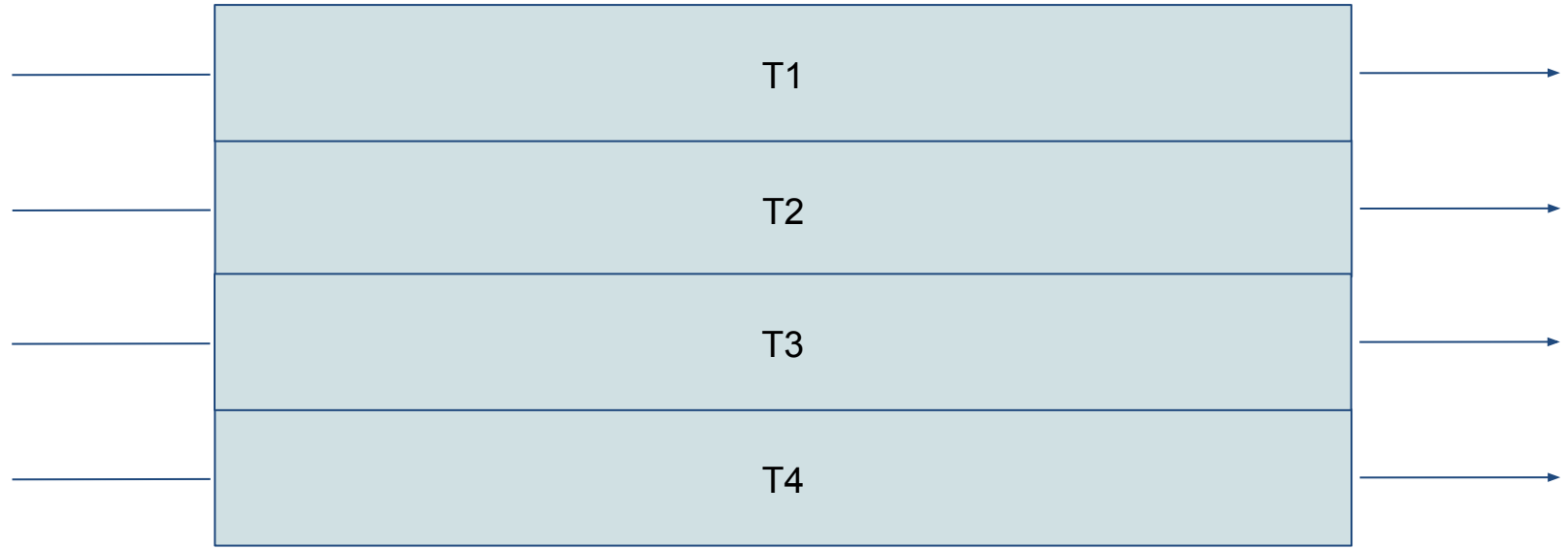
Многопоточность и асинхронность

Синхронное однопоточное выполнение



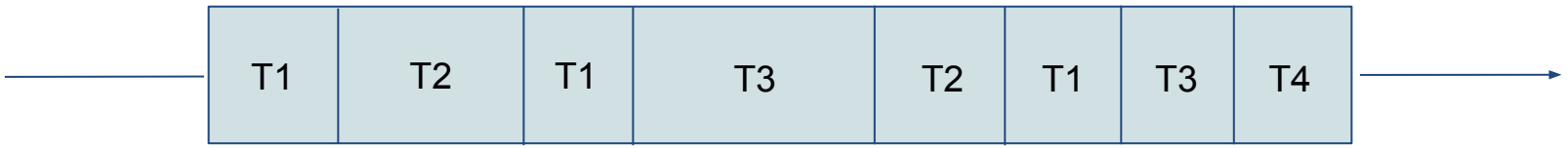
Многопоточность и асинхронность

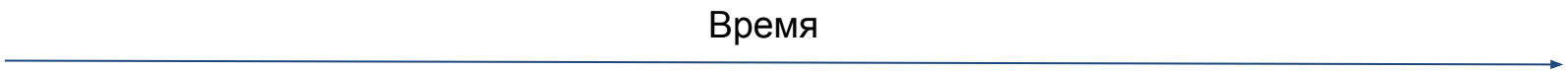
Синхронное многопоточное выполнение



Многопоточность и асинхронность

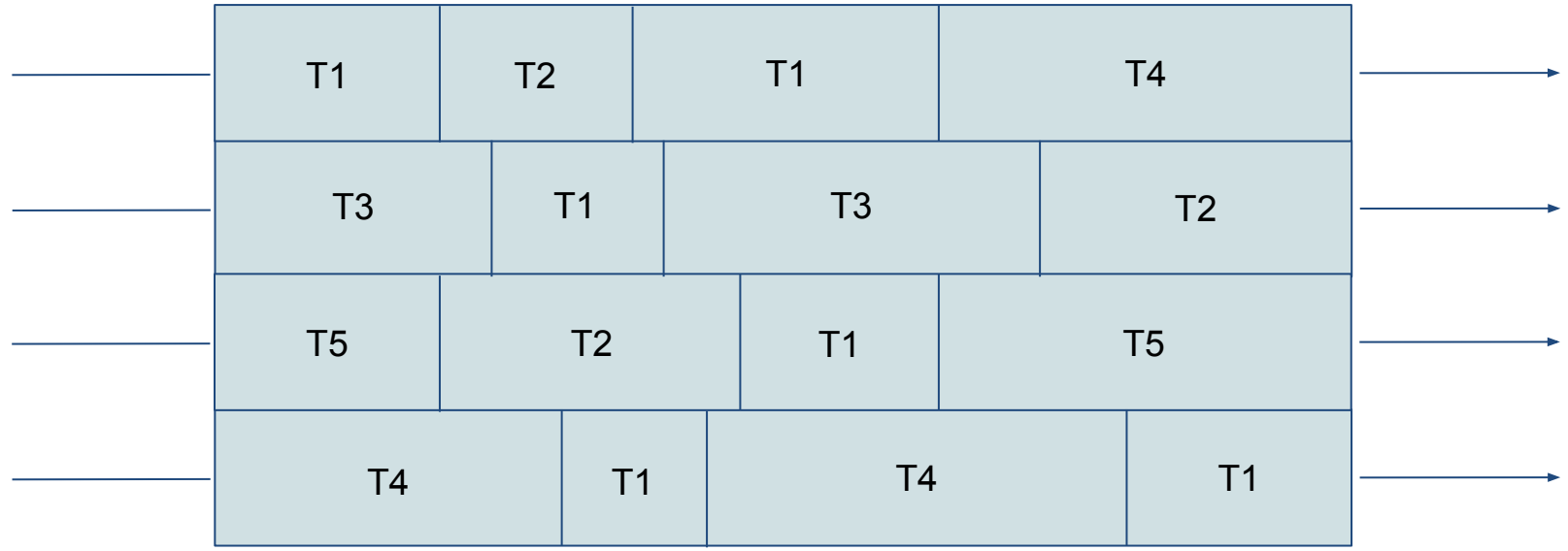
Асинхронное однопоточное выполнение





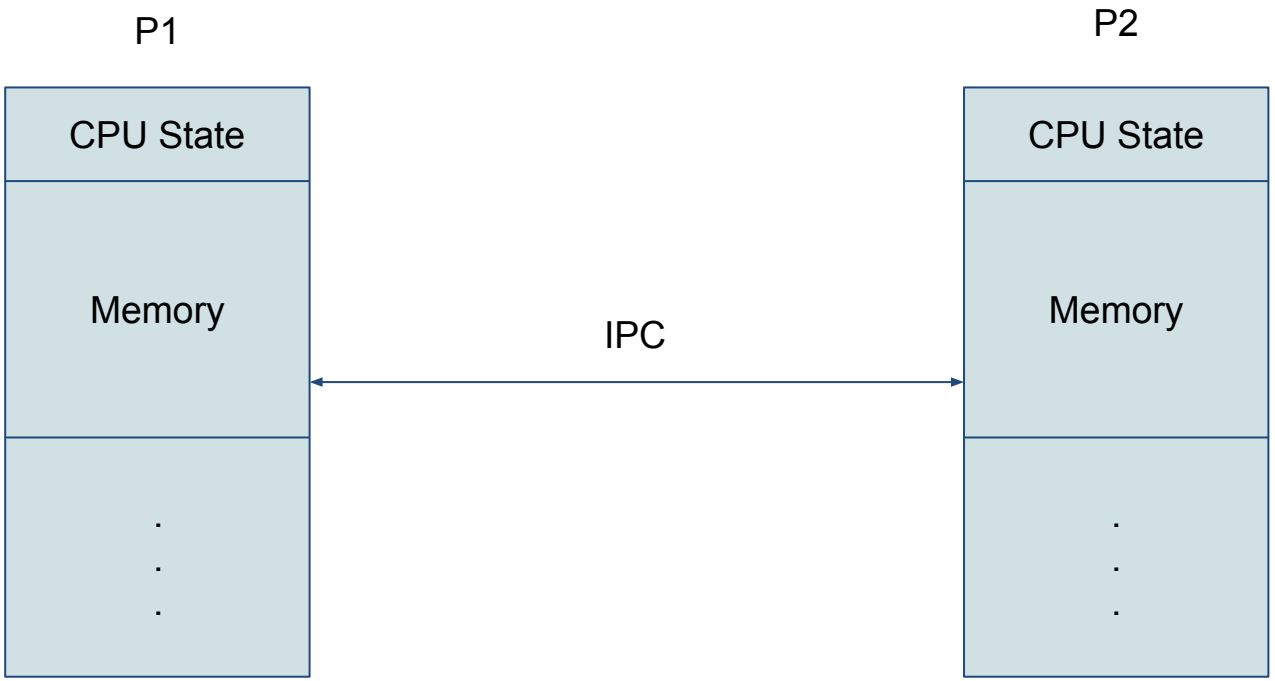
Многопоточность и асинхронность

Синхронное многопоточное выполнение

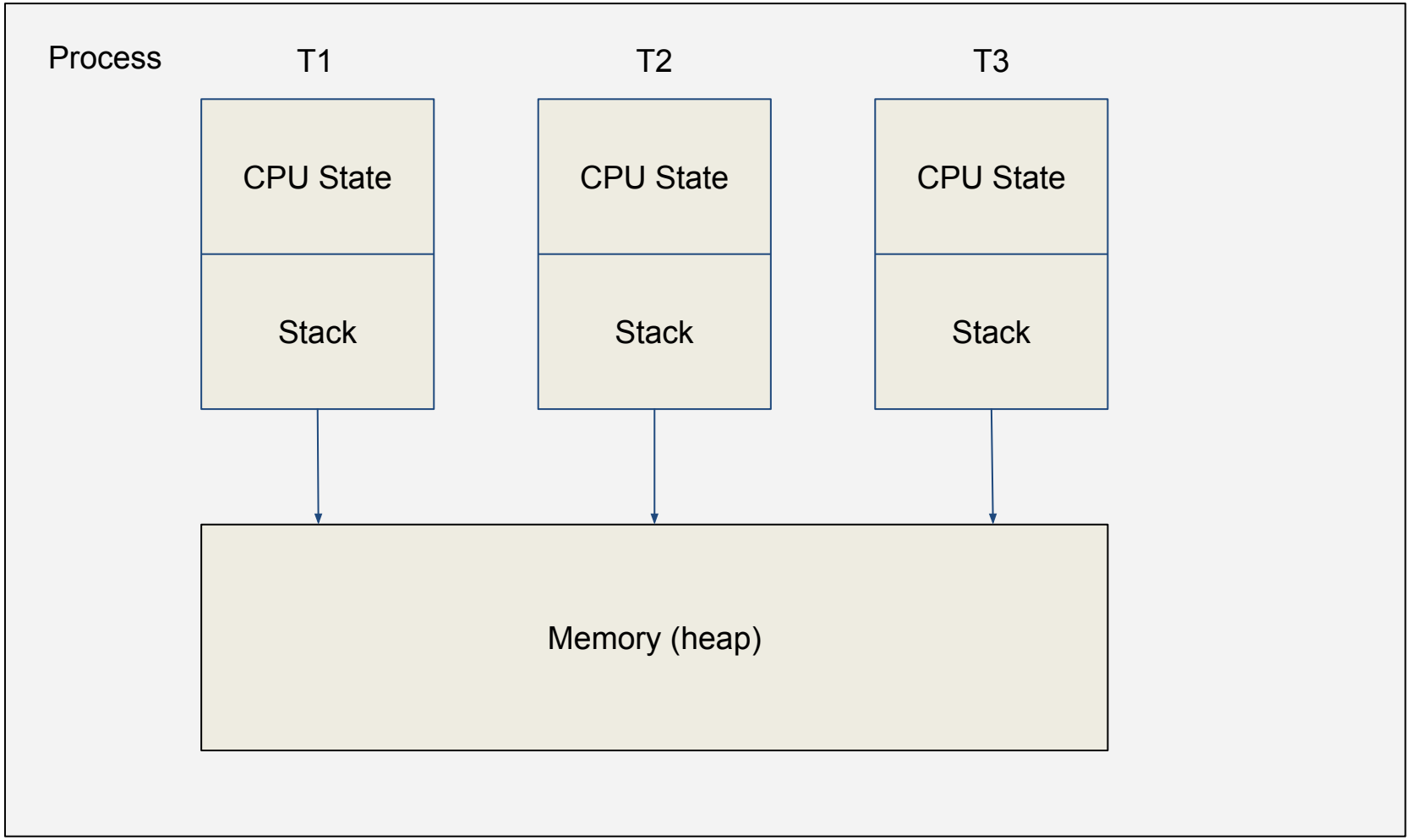


Процессы и потоки

Процесс - экземпляр выполняемой программы, включая счетчик команд, адресное пространство, регистры процессора, стек, файловые дескрипторы.



Процессы и потоки



Процессы и потоки

Потокобезопасность (Thread safety) - сохранение **корректного** поведения кода/алгоритма при доступе из нескольких потоков **независимо** от операций переключения потоков и работы планировщика потоков, не требующее **дополнительной** синхронизации вызывающего кода

Обеспечение потокобезопасноти:

- Не использовать общее состояние
- Использовать неизменяемое состояние
- Синхронизировать доступ к общему состоянию

Процессы и потоки

- Атомарность

  Для **консистентности** состояния, изменение этого состояния должно быть одной **атомарной** операцией. **Атомарная операция** - это единое и неделимое действие.

- Видимость

  Данные могут **кэшироваться** в регистрах CPU, JVM может менять **порядок** выполнения операций

- Неизменяемость (immutability)

  Неизменяемые объекты потокобезопасны

Примитивы синхронизации

- Semaphore
- Mutex
- Conditional Lock
- Barriers

Проблема спящего парикмахера

Проблема спящего парикмахера

```
fun barberProblem() {
    val extraChairs = 5
    val occupiedChairs = 0
    val barbers = 0

    fun customers() {
        while (true) {
            down(extraChairs)
            up(occupiedChairs)
            down(barbers)
            cutMe()
        }
    }
    fun barber() {
        while (true) {
            down(occupiedChairs)
            up(extraChairs)
            up(barbers)
            cutHair()
        }
    }
}
```

Semaphore

Предложены Дийкстрой еще в 1965г

Это объекты имеющие счётчик и всего два метода

- Метод up()
- Метод down()

При вызове метода down(), если счетчик достигает значения 0 - поток засыпает до тех пор, пока счетчик не станет больше 0.

Вызов каждого из этих методов - **атомарный**, при этом метод up() не приводит к блокировке потока

Mutex

- Упрощенная версия семафора
- Не способен считать, а лишь управляет доступом к ресурсу

Реализация очень проста. Фактически одна переменная и два метода

- Метод lock()
- Метод unlock()

Если мьютекс уже захвачен, повторный вызов метода lock() приведет к **вызову метода thread_yeld()**

Conditional Lock

- Более высокоуровневый примитив, чем семафоры и мьютексы
- Работает в паре с мьютексом
- Позволяет не держать захваченным мьютекс

Обычно в реализации используется три метода

- Метод wait()
- Метод signal()
- Метод broadcast()

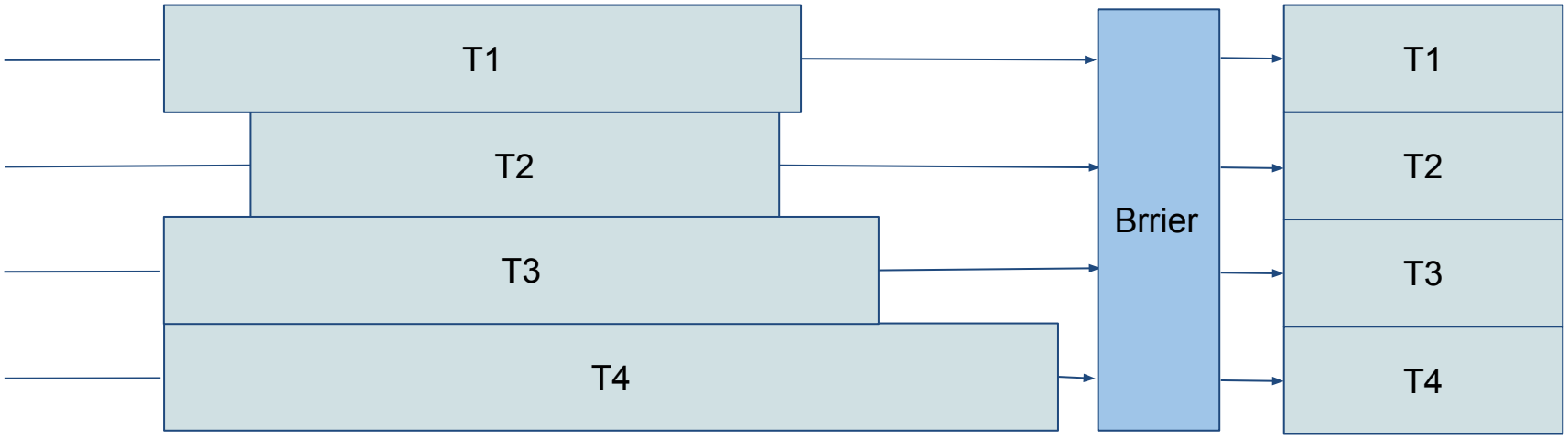
Conditional Lock - Пример

```
fun barberProblem() {
    val mutex = Mutex()
    val cond = Condition()
    val maxsize = 5
    val buffer = ArrayList<Int>()

    fun put() {
        mutex.lock()

        while(buffer.size == maxsize) {
            cond.wait(mutex)
        }
        buffer.add(0)
        cond.signal(mutex)
        mutex.unlock()
    }

    fun get() {
        mutex.lock()

        while(buffer.size == 0) {
            cond.wait(mutex)
        }
        val item = buffer.remove(0)
        cond.signal(mutex)
        mutex.unlock()
    }
}
```

Barriers

Механизм синхронизации, предназначенный для группы потоков или процессов.

Эффективен при параллельной обработке данных группой “производителей”, результата которой дожидается один “потребитель”



Concurrency в Java - Потоки

- За работу с потоками отвечает класс Thread

Для выполнения кода в отдельном потоке, есть два способа

- Наследуемся от класса Thread
- Реализуем интерфейс Runnable

Concurrency в Java - Потоки

```
fun runBackgroundRunnable() {
    val thread = Thread(Runnable {
        println("Hello, from background thread")
    })
    thread.start()
    thread.join()
    println("Welcome back to main thread")
}

fun runBackgroundThread() {
    val thread = object : Thread() {
        override fun run() {
            println("Hello, from background thread")
        }
    }
    thread.start()
    thread.join()
    println("Welcome back to main thread")
}
```

Concurrency в Java - Монитор

Монитор - объект сочетающий в себе Mutex и Conditional Lock

Для работы с монитором предусмотрены три метода:

- wait
- notify
- notifyAll

Обращаться к этим методам можно **только в синхронизированном** контексте (synchronized)

Concurrency в Java - Монитор

```
public synchronized void put(Object o) {
    while (buf.size()==MAX_SIZE) {
        wait();
    }
    buf.add(o);
    notify();
}

public synchronized Object get() {
    while (buf.size()==0) {
        wait();
    }
    Object o = buf.remove(0);
    notify();
    return o;
}
```

Concurrency в Java

- synchronized - Встроенный механизм для обеспечения **атомарности операции**

- volatile - ключевое слово **предотвращающее оптимизацию** переменной и требующее обновления ее состояния в памяти (memory barrier)

Concurrency в Java - synchronized

```
public class Synchronized {
    private final Object locker = new Object();

    private synchronized void synchronizedMethod() {
        //do some thread safe work
    }

    private void syncThis() {
        //call some methods
        synchronized (this) {
            //do some thread safe work
        }
        //do thread unsafe work
    }

    private void doSomeWork() {
        //call some methods
        synchronized (locker) {
            //do some thread safe work
        }
        //do thread unsafe work
    }
}
```

Concurrency в Java - Atomic

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicIntegerFieldUpdater<T>
- AtomicLong
- AtomicLongArray
- AtomicLongFieldUpdater<T>
- AtomicReference<V>
- AtomicReferenceArray<E>

Многие из этих классов реализованы с помощью **CAS** (Compare-and-swap) операций

Concurrency в Java - Atomic

```
public class VisitorsCounter
{
    private volatile int count = 0;
    public void upateVisitors()
    {
        ++count;
    }
}

public class AtomicVisitorsCounter
{
    private AtomicInteger count = new AtomicInteger(0);
    public void upateVisitors()
    {
        count.incrementAndGet();
    }
}
```

java.util.concurrent

Интерфейс Lock

- ReentrantLock
- ReentrantReadWriteLock.ReadLock
- ReentrantReadWriteLock.WriteLock

Java.util.concurrent - Lock

```
interface Lock {
    fun lock()

    fun lockInterruptibly()

    fun tryLock() : Boolean

    fun tryLock(var1 : Long, var3 : TimeUnit) : Boolean

    fun unlock()

    fun newCondition() : Condition
}
```

Java.util.concurrent - ReentrantLock

```
class BoundedBuffer {
    val lock: Lock = ReentrantLock()
    val notFull = lock.newCondition()
    val notEmpty = lock.newCondition()

    fun put(x: Any) {
        lock.lock()
        try {
            while (buf.size == MAX_SIZE)
                notFull.await()
            buf.add(x)
            notEmpty.signal()
        } finally {
            lock.unlock()
        }
    }

    fun take(): Any {
        lock.lock()
        try {
            while (buf.isEmpty)
                notEmpty.await()
            val x = buf.remove(0)
            notFull.signal()
            return x
        } finally {
            lock.unlock()
        }
    }
}
```

java.util.concurrent

- CountDownLatch
- Semaphore
- Phaser

Java.util.concurrent - CountDownLatch

```
fun main() {
    val doneSignal = CountDownLatch(10)
    val e = Executors.newCachedThreadPool()

    for (i in 0 until 10) {
        e.execute(WorkerRunnable(doneSignal, i))
    }
    doneSignal.await()
}

class WorkerRunnable(private val doneSignal : CountDownLatch,
                     private val i : Int) : Runnable {

    override fun run() {
        try {
            doWork(i)
            doneSignal.countDown()
        } catch (e : InterruptedException) {}
    }

    fun doWork(i : Int) {  }
}
```

Java.util.concurrent - Phaser

```
class LongRunningAction implements Runnable {
    private String threadName;
    private Phaser ph;

    LongRunningAction(String threadName, Phaser ph) {
        this.threadName = threadName;
        this.ph = ph;
        ph.register();
    }

    @Override
    public void run() {
        ph.arriveAndAwaitAdvance();
        System.out.println(threadName + " is ready!");
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        ph.arriveAndAwaitAdvance();
        System.out.println(threadName + " phase two!");
        ph.arriveAndDeregister();
    }
}
```

```
class Main {
  public static void main(String[] args) {
    ExecutorService executorService = Executors.newCachedThreadPool();
    Phaser ph = new Phaser(1);

    executorService.submit(new LongRunningAction("thread-1", ph));
    executorService.submit(new LongRunningAction("thread-2", ph));
    executorService.submit(new LongRunningAction("thread-3", ph));

    ph.arriveAndAwaitAdvance();
    System.out.println("We are ready!");
    ph.arriveAndAwaitAdvance();
    System.out.println("Phase two!");
    ph.arriveAndDeregister();
  }
}
```

java.util.concurrent

Класс Executors является фабрикой для классов ExecutorService

- SingleThreadExecutor
- FixedThreadExecutor
- CachedThreadExecutor

Java.util.concurrent - Executor

```
class SerialExecutor(val executor: Executor) : Executor {
    val tasks: Queue<Runnable> = ArrayDeque<Runnable>()
    var active: Runnable? = null

    @Synchronized
    override fun execute(r: Runnable) {
        tasks.offer(Runnable {
            try {
                r.run()
            } finally {
                scheduleNext()
            }
        })
        if (active == null) {
            scheduleNext()
        }
    }

    @Synchronized
    protected fun scheduleNext() {
        active = tasks.poll()
        active.let {
            executor.execute(it)
        }
    }
}
```

```
class DirectExecutor : Executor {
    override fun execute(r: Runnable) {
        r.run()
    }
}

class ThreadPerTaskExecutor : Executor {
    override fun execute(r: Runnable) {
        Thread(r).start()
    }
}
```

Java.util.concurrent - ExecutorService

```
interface ExecutorService : Executor {
    val isShutdown: Boolean

    val isTerminated: Boolean
    fun shutdown()

    fun shutdownNow(): List<Runnable>

    fun awaitTermination(timeout: Long, timeUnits: TimeUnit): Boolean

    fun <T> submit(callable: Callable<T>): Future<T>

    fun <T> submit(runnable: Runnable, result: T): Future<T>

    fun submit(var1: Runnable): Future<*>

    fun <T> invokeAll(var1: Collection<Callable<T>>): List<Future<T>>

    fun <T> invokeAll(var1: Collection<Callable<T>>, timeout: Long, units: TimeUnit): List<Future<T>>

    fun <T> invokeAny(var1: Collection<Callable<T>>): T

    fun <T> invokeAny(var1: Collection<Callable<T>>, timeout: Long, units: TimeUnit): T
}
```

java.util.concurrent

Concurrent collections

- CopyOnWrite ArrayList
- BlockingQueue
- ConcurrentMap
- ConcurrentNavigableMap

Concurrency в Kotlin

- нет ключевого слова synchronized
- нет ключевого слова volatile
- объект типа Any (аналог Object в Java) не содержит методов wait(), notify(), notifyAll()

abreslav JetBrains Team Apr '13

Kotlin deliberately has no constructs for concurrency built into the language. We believe this should be handled by libraries. Use Java's locks and the synchronized(x) {} function.

Concurrency в Kotlin

- `@Synchronized`
- `@Volatile`
- **public inline fun** <R> synchronized(lock: Any, block: () -> R): R
- инструменты j.u.concurrent
- методы java.lang.Object
- **public inline fun** <T> Lock.withLock(action: () -> T): T
- инструменты kotlin.concurrent

Concurrency в Kotlin- synchronized

```kotlin
class SynchronizedKotlin {
    private val locker = java.lang.Object()

    @Synchronized
    private fun synchronizedMethod() {
        //do some thread safe work
    }

    private fun syncThis() {
        //call some methods
        synchronized(this) {
            //do some thread safe work
        }
        //do thread unsafe work
    }

    private fun doSomeWork() {
        //call some methods
        synchronized(locker) {
            //do some thread safe work
        }
        //do thread unsafe work
    }
}
```

Concurrency в Kotlin - volatile

```
class BackgroundWorker {
    @Volatile var sync = true
    val tasks : CopyOnWriteArrayList<Runnable> = CopyOnWriteArrayList<Runnable>()

    val worker = object : Thread() {
        override fun run() {
            while (sync) {
                if (tasks.isNotEmpty()) {
                    val task = tasks.removeAt(0)
                    task.run()
                }
            }
        }
    }

    fun addTask(task : Runnable) {
        tasks.add(task)
    }

    fun start() = if (sync) worker.start() else throw IllegalStateException("Background is dead!")

    fun shutdown() {
        sync = false
        worker.join()
    }
}
```

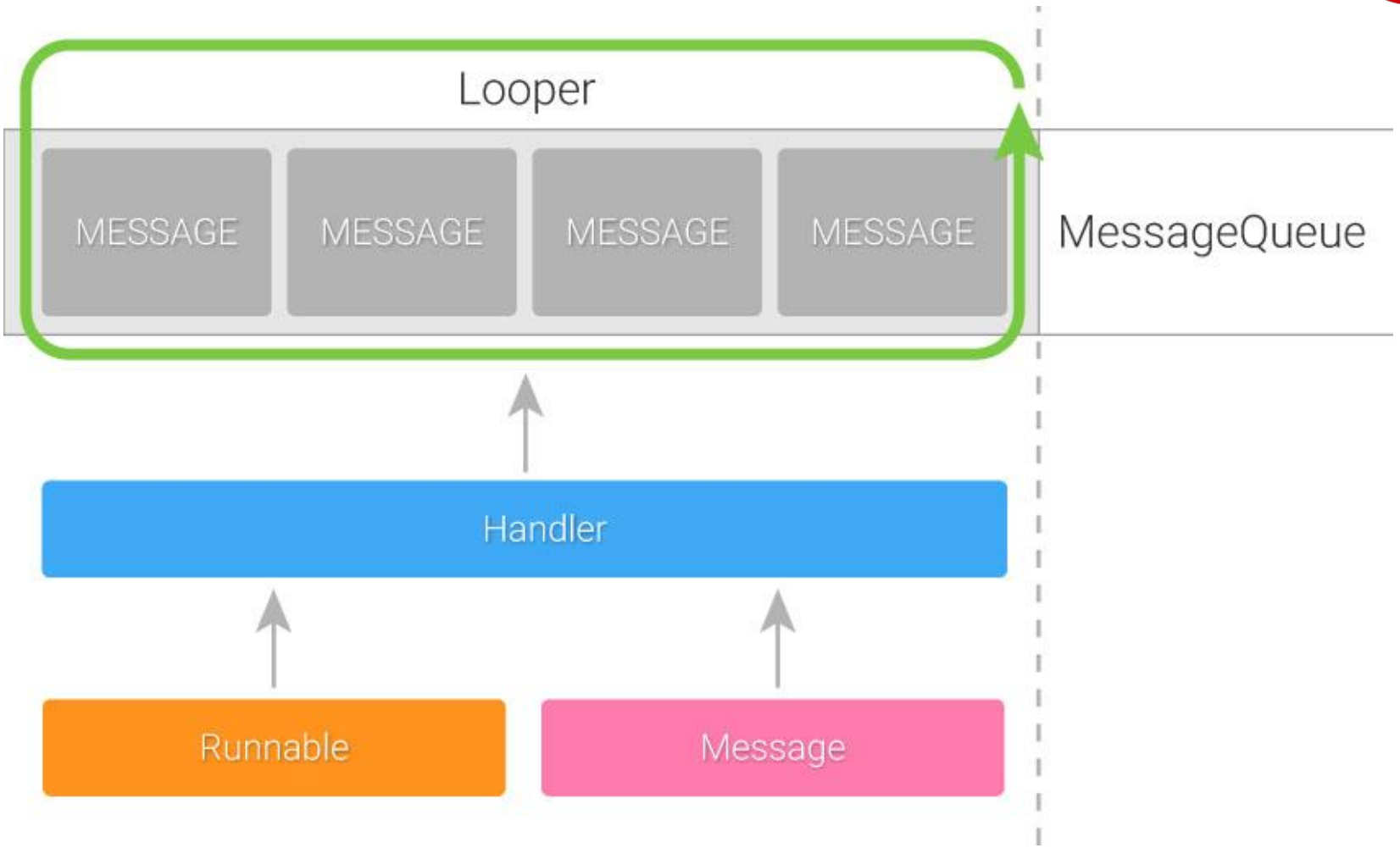
Concurrency в Kotlin - withLock

```
class Guarded {
    var lock : Lock = ReentrantLock()

    fun doSomething() {
        lock.withLock {
            //thread safe call
        }
    }
}
```

Concurrency в Android

- HaMeR Framework (Handler, Message, Runnable)
- HandlerThread
- IntentService/JobIntentService
- JobSchedule/WorkManager
- *AsyncTask, Loader*

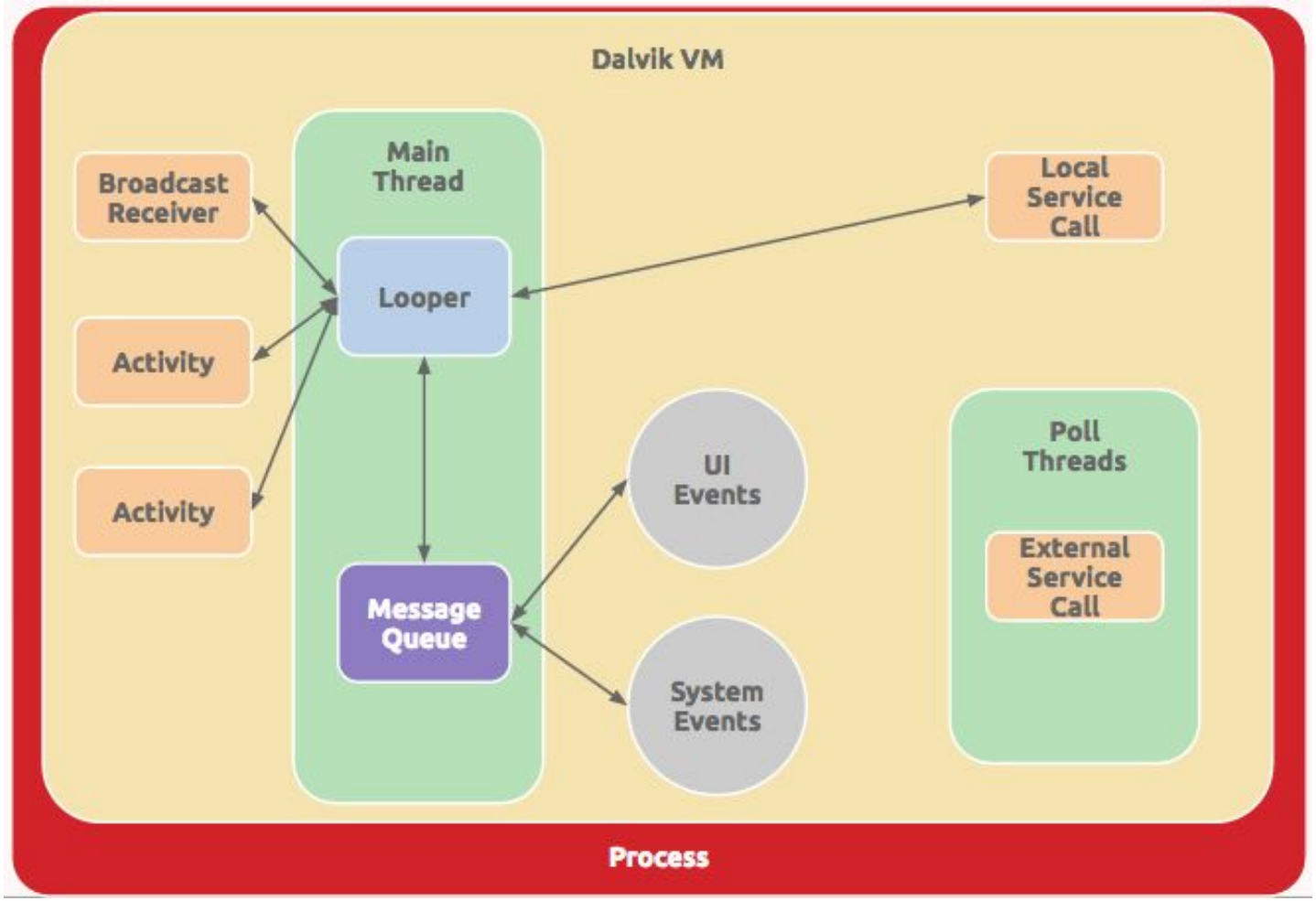
HaMeR Framework



HaMeR Framework

- Looper
- Handler
- Message
- MessageQueue
- Runnable

Android UI


Dalvik VM
Main Thread
Broadcast Receiver
Looper
Activity
Activity
Local Service Call
UI Events
Message Queue
System Events
Poll Threads
External Service Call
Process

Android UI

Взаимодействие с UI потоком

- view.post(..)
- activity.runOnUiThread(...)
- handler.post(...) на MainLooper

HandlerThread

- поток, который содержит Looper, MessageQueue
- Обеспечивает последовательное выполнение задач
- Имеет жизненный цикл

IntentService/JobIntentService

IntentService

- Service с проинициализированным HandlerThread
- Позволяет последовательно выполнять задачи
- Возврат результата нетривиальная задача
- Завершается после выполнения всех задач
- Попадает под ограничения использования фоновых задач

JobIntentService

- Предпочтителен для использования с API 28
- Использует JobScheduler

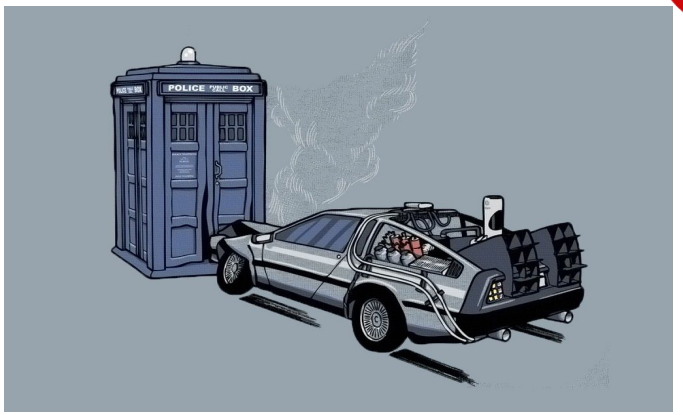
JobScheduler/WorkManager

- JobScheduler
- WorkManager

Callback approach

```
fun handlePerson() {
    requestPerson({response ->
        handleResponse(response, {person ->
            persistPerson(person, {
                // save to db
            })
        }, {
            // handle error
        })
    }, {
        // handle error
    })
}

fun requestPerson(success: (Response) -> Unit, failure: (Exception) -> Unit) {
    TODO("request from network")
}

fun handleResponse(response: Response, success: (Person) -> Unit, failure: (Exception) -> Unit ) {
    TODO("map response to Person")
}

fun persistPerson(person: Person, success: () -> Unit) {
    TODO("persist Person to DB")
}
```

Callback Hell



```
(tail =>
    (is_nil =>
        (isEmpty =>
            ($0 =>
                ($1 =>
                    ($2 =>
                        ($3 =>
                            (is_$0 =>
                                ($$ =>
                                    (is_$$ =>
                                        (succ =>
                                            (add =>
                                                (pair_succ =>
                                                    (prev =>
                                                        (sub =>
                                                            (len =>
                                                                (sum =>
                                                                    (rcon =>
                                                                        (rev =>
                                                                            (reverse =>
                                                                                (elems =>
                                                                                    (list =>
                                                                                        (map =>
                                                                                            map(list(elems($1)($2)($3)))(elem => sub(elem)($1))
                                                                                        )(y(map => l => f => when(isEmpty(l))(_ => nil)(_ => co
                                                                                    )(es => reverse(es($$)))
                                                                                )(rcon(nil))
                                                                            )(l => rev(nil)(l))
                                                                        )(y(rev => r => l => when(isEmpty(l))(_ => r)(_ => rev(con(head(l))(r))
                                                                    )(y(rcon => t => h => when(is_$$(h))(_ => t)(_ => rcon(con(h)(t)))))
                                                                )(y(sum => l => when(isEmpty(l))(_ => $0)(_ => add(head(l))(sum(tail(l))))))
                                                            )(y(len => l => when(isEmpty(l))(_ => $0)(_ => add($1)(len(tail(l))))))
                                                        )(m => n => n(prev)(m))
                                                    )(n => left(n(pair_succ)(pair(no_use)($0))))
                                                )(p => pair(right(p))(succ(right(p))))
                                            )(m => n => n(succ)(m))
                                        )(n => f => x => f(n(f)(x)))
                                    )(n => n(_ => no)(no))
                                )(_ => _ => yes)
                            )(n => n(_ => no)(yes))
                        )(f => x => f(f(f(x))))
                    )(f => x => f(f(x)))
                )(f => x => f(x))
            )(_ => x => x)
        )(is_nil)
    )(l => l(_ => _ => no))
)(right)
```

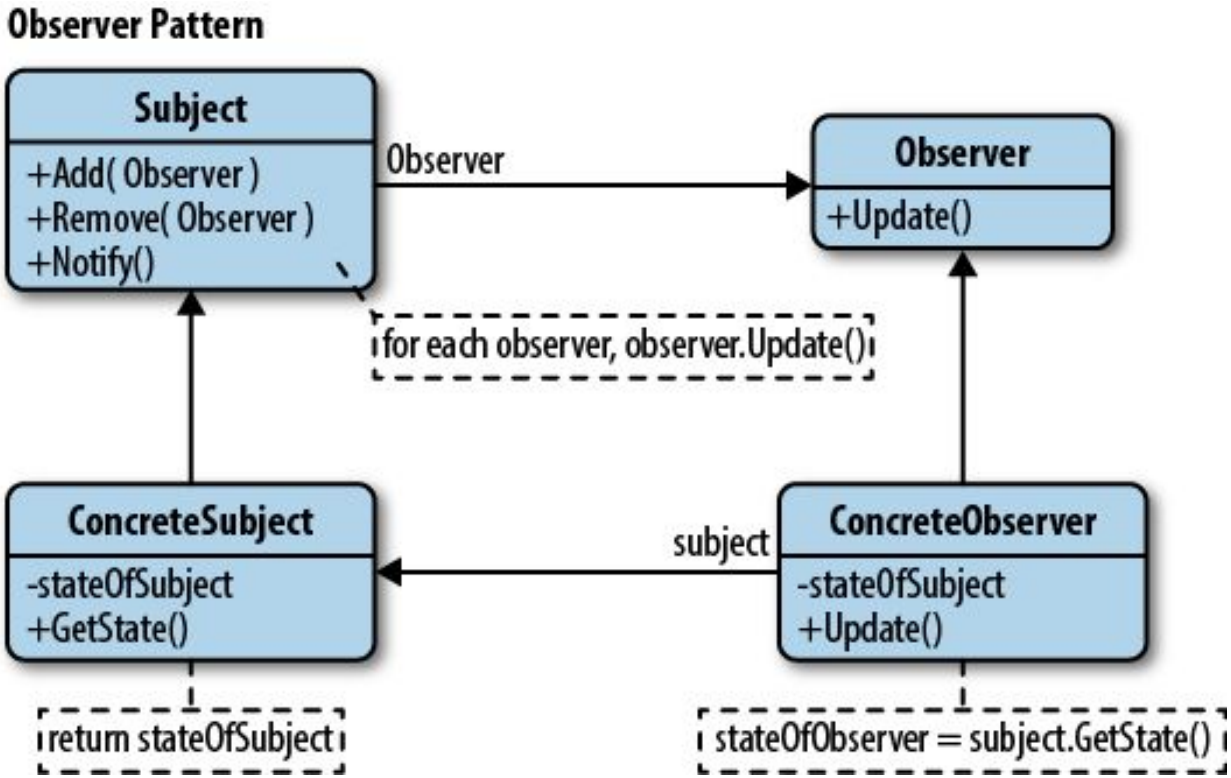
RxJava

- Библиотека (большая) для работы с асинхронными и событийными программа с помощью наблюдаемых потоков
- Основана на шаблоне Наблюдатель
- Используются функциональные преобразования потоков
- Легкое переключение между потоками

RxJava

Reactive programming mantra

RxJava


Observer Pattern
Subject
+Add(Observer)
+Remove(Observer)
+Notify()
Observer
Observer
+Update()
for each observer, observer.Update()
ConcreteSubject
-stateOfSubject
+GetState()
subject
ConcreteObserver
-stateOfSubject
+Update()
return stateOfSubject
stateOfObserver = subject.GetState()

RxJava

Сущности:

- Observable - генерирует события
- Observer - получает события

События:

- Next
- Error
- Complete

RxJava

```
fun parsePhoneNumber(phoneNumber: String): Observable<PhoneNumber> {
        return Observable.fromCallable(object : Callable<PhoneNumber>() {
            override fun call(): PhoneNumber {
                val parsedPhone = phoneNumberUtil.parse(phoneNumber, Locale.getDefault().getCountry())
                return PhoneNumber.create(parsedPhone.getCountryCode(), parsedPhone.getNationalNumber())
            }
        })
}

fun getPhone(phone: String) {
    parsePhoneNumber.subscribe(object : Observer<String> {
        override fun onNext(phone: String) {
            // handle phone
        }

        override fun onError(exception: Exception) {
          // handle exception
        }

        override fun onComplete() {
            // handle completion
        }
    })
}
```

RxJava

Создание Observable

- create
- defer
- from: fromIterable, fromArray, fromCallable, fromRunnable, fromFuture
- just
- error

RxJava

Операторы:

- Operators (Alphabetical List)
  - Async
  - Blocking
  - Combining
  - Conditional & Boolean
  - Connectable
  - Creation
  - Error management
  - Filtering
  - Mathematical and Aggregate
  - Parallel flows
  - String
  - Transformation
  - Utility
  - Notable 3rd party Operators (Alphabetical List)

RxJava

- Позволяет легко управлять потоками
- subscribeOn
- observeOn

RxJava

```
import io.reactivex.Completable
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers

class RxExample {

    fun handlePerson() {
        requestPerson()
                .flatMap { response -> handleResponse(response) }
                .flatMapCompletable { person -> persistPerson(person) }
                .subscribeOn(Schedulers.io())
                .observeOn(Schedulers.single())
                .subscribe { print("request completed") }
    }

    fun requestPerson(): Observable<Response> {
        TODO("request from network")
    }

    fun handleResponse(response: Response): Observable<Person> {
        TODO("map response to Person")
    }

    fun persistPerson(person: Person): Completable {
        TODO("persist Person to DB")
    }
}
```

Coroutines

- “легковесные” потоки
- возможно выбора исполнителя кода
- последовательная семантика асинхронного кода
- близкий родственник async/await
- structured concurrency
- shared communication

Coroutines

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("world!")
    }

    println("Hello,")
    Thread.sleep(2000)
}
```

Coroutines

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    GlobalScope.launch { helloWorldPrinter() }
    println("Hello,")
    Thread.sleep(2000)
}

suspend fun helloWorldPrinter() {
    delay(1000L)
    println("world!")
}
```

Coroutines

```
import kotlinx.coroutines.*

fun main() {
        GlobalScope.launch {
            val response = loadFromNetwork()
            val person =  handleResponse(response)
            persistPerson(person)
        }
}

suspend fun loadFromNetwork(): Response {
    return doRequest().await()
}

suspend fun handleResponse(response: Response): Person {
    return parseRepsonse(response).await()
}

fun persistPerson(person: Person) {
    TODO("save to database")
}

fun doRequest(): Deferred<Response> {
    TODO("make netowk request")
}

fun parseRepsonse(response: Response): Deferred<Person> {
    TODO("parse response")
}
```

Coroutines

Конструкторы корутин:

- launch

  Запускает корутину, не возвращает результат
- async

  Запускает корутину, возвращает Deferred
- runBlocking

  Запускает корутину, ждет ее завершения

Coroutines

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    handlePerson()
}

suspend fun handlePerson() = coroutineScope {
    val response = async { loadFromNetwork() }
    val person = async { handleResponse(response.await()) }
    persistPerson(person.await())
}

suspend fun loadFromNetwork(): Response {
    return doRequest().await()
}

suspend fun handleResponse(response: Response): Person {
    return parseRepsonse(response).await()
}

fun persistPerson(person: Person) {
    TODO("save to database")
}

fun doRequest(): Deferred<Response> {
    TODO("make netowk request")
}

fun parseRepsonse(response: Response): Deferred<Person> {
    TODO("parse response")
}
```

Coroutines

- CoroutineContext - контекст выполнения корутины, содержит Job и CoroutineDispatcher

- CoroutineDispatcher - компонент, который определяет поток или пул потоков для выполнения корутины

- CoroutineBuilder принимает CoroutineDispatcher в качестве параметра

Coroutines

```
launch {
    // runBlocking
}

launch(Dispatchers.Default) {
    //  CoroutineDispatcher по-умолчанию
}

launch(newSingleThreadContext("MyOwnThread")) {
    // Job будет выполняться в новом потоке
}
```

Coroutines

- Structured concurrency - корутины запускаются в своей **области видимости**, позволяет избежать утечек памяти, контролировать обработку исключений
- Корутины **отменяемые**
- Позволяют писать асинхронный код в **последовательной** семантике
- Каналы - позволяют перейти от **разделяемого состояние** к **разделяемому взаимодействию**, избегая параллельного изменения состояния и необходимости синхронизации

Ваш проект

1. На данном этапе в рамках вашего проекта необходимо реализовать сплеш-скрин (заставку) и основные окна вашего приложения, со списками элементов. В приложении должны использоваться Фрагменты и Списки.
2. Ваше приложение должно уже работать с данными, загружаемыми как по сети так и из файлов. Фактически это уже простое, но полноценно работающее приложение, которое может ходить в сеть, а в случае отсутствия сети брать локальные данные
3. После ознакомления с современным дизайном приложений, ваше приложение должно содержать основные компоненты современного Android приложения. Toolbar, Floating Navigation Button, Navigation Drawer или NavigationView. Все должно корректно отображаться на смартфонах с разными экранами
4. У вас должно быть полностью рабочее приложение, на которое не страшно посмотреть. Интерфейс работает плавно, используются Анимации, Custom View и нотификации. Старые ошибки должны быть исправлены. Приложение не должно падать или нестабильно работать.

Ваш проект

- Защита №1 - 13.11
- Защита №2 - 4.12
- Защита №3 - 25.12
- Защита №4 - 15.01

Ваш проект - источник вдохновения

https://github.com/toddmotto/public-apis

ТЕХНОТРЕК

Спасибо за внимание!

Юрий Береза – ybereza@gmail.com

Кирилл Филимонов - kirill.filimonov@gmail.com