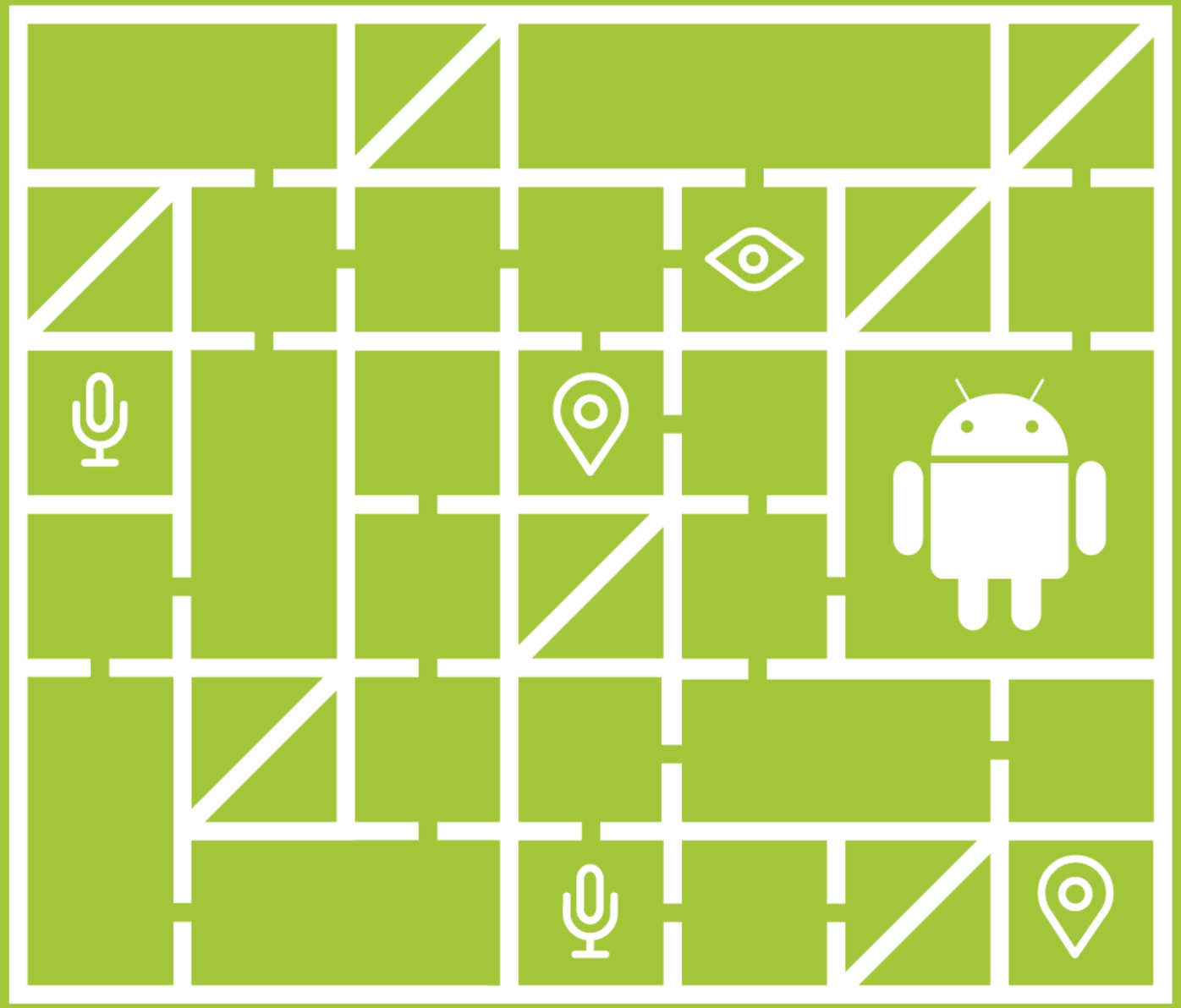


Яндекс Деньги

ANDROID PARANOID

Встреча для разработчиков

28/03

Evolution of tools of background work in Android

Disclaimer

Everything told here is a product of personal professional experience and not anything more.

About me

About me

— Vladimir Ivanov - Lead Software Engineer in EPAM

About me

— Vladimir Ivanov - Lead Software Engineer in EPAM

— Android apps: > 15 published, > 7 years

About me

— Vladimir Ivanov - Lead Software Engineer in EPAM

— Android apps: > 15 published, > 7 years

— Wide expertise in Mobile

Agenda

Agenda

— AsyncTasks

Agenda

— AsyncTasks

— Loaders

Agenda

— AsyncTasks

— Loaders

— Executors&Event bus

Agenda

— AsyncTasks

— Loaders

— Executors&Event bus

— RxJava

Agenda

— AsyncTasks
— Loaders
— Executors&Event bus
— RxJava
— Coroutines

Out-of-scope

Out-of-scope

— Services

Out-of-scope

— Services

— Sync Adapters

What it is your first Android OS you wrote code for?

Why do we need background?

Why do we need background?

— 60 FPS

Why do we need background?

— 60 FPS

— UI updates only on the main thread

Android 1.6: Async tasks

```
public class LoadWeatherForecastTask
    extends AsyncTask<String, Integer, Forecast> {

    public Forecast doInBackground(String... params) {
        HttpClient client = new HttpClient();
        HttpGet request = new HttpRequest(params[0]);
        HttpResponse response = client.execute(request);
        return parse(response);
    }
}
```

Which thread the doInBackground() is called on?

Which thread the doInBackground() is called on?

Fixed-sized threadpool executor of size 1

```
public static final Executor SERIAL_EXECUTOR
                                = new SerialExecutor();
```

```
public static final Executor SERIAL_EXECUTOR
                                = new SerialExecutor();
...
private static volatile Executor sDefaultExecutor
                                = SERIAL_EXECUTOR;
```

The result should be posted on the UI Thread

```
public class LoadWeatherForecastTask
    extends AsyncTask<String, Integer, Forecast> {

    public void onPostExecute(Forecast forecast) {

            mTemperatureView.setText(forecast.getTemp());
    }

}
```

Just Handler

Minuses

Minuses

— Boilerplate

Minuses

— Boilerplate

— Lifecycle not-aware

Minuses

— Boilerplate
— Lifecycle not-aware
— No result reusage across configuration changes

Minuses

— Boilerplate
— Lifecycle not-aware
— No result reusage across configuration changes
— Easy way to introduce memory leaks

Android 3.0: Loaders

```
inner class WeatherForecastLoaderCallbacks :
    LoaderManager.LoaderCallbacks<WeatherForecast> {
```

```
inner class WeatherForecastLoaderCallbacks :
    LoaderManager.LoaderCallbacks<WeatherForecast> {

   override fun onCreateLoader(id: Int, args: Bundle?):
                                        Loader<WeatherForecast> {

        return WeatherForecastLoader(applicationContext)
   }
}
```

```
inner class WeatherForecastLoaderCallbacks :
    LoaderManager.LoaderCallbacks<WeatherForecast> {

    override fun onCreateLoader(id: Int, args: Bundle?):
                                    Loader<WeatherForecast> {

        return WeatherForecastLoader(applicationContext)
    }

    override fun onLoadFinished(loader: Loader<WeatherForecast>?,
                                data: WeatherForecast) {

        temperatureTextView.text = data.temp.toString();
    }
}
```

```
class WeatherForecastLoader(context: Context)
            : AsyncTaskLoader<WeatherForecast>(context) {

    override fun loadInBackground(): WeatherForecast {
        try {
            Thread.sleep(5000)
        } catch (e: InterruptedException) {
            return WeatherForecast("", 0F, "")
        }

        return WeatherForecast("Saint-Petersburg", 20F, "Sunny")
    }
}
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    ...
    val weatherForecastLoader = WeatherForecastLoaderCallbacks()

    loaderManager
      .initLoader(forecastLoaderId,
            Bundle(),
            weatherForecastLoader)
}
```

How does the loader gets reused?

How does the loader gets reused?

How does the loader gets reused?

— LoaderManager saves all loaders internally

How does the loader gets reused?

— LoaderManager saves all loaders internally

— Activity saves all LoaderManagers inside NonConfigurationInstances

How does the loader gets reused?

- LoaderManager saves all loaders internally
- Activity saves all LoaderManagers inside NonConfigurationInstances
- On activity recreation NonConfigurationInstances got passed to a new activity

Minuses

Minuses

— Still a lot of boilerplate

Minuses

— Still a lot of boilerplate

— Complex interfaces and abstract classes

Minuses

— Still a lot of boilerplate
— Complex interfaces and abstract classes
— Loaders are Android API - you can’t create libs in pure Java

Thread pool executors FTW!

```
public class Background {

    private val mService = ScheduledThreadPoolExecutor(5)

    fun execute(runnable: Runnable): Future<*> {
        return mService.submit(runnable)
    }

    fun <T> submit(runnable: Callable<T>): Future<T> {
        return mService.submit(runnable)
    }
}
```

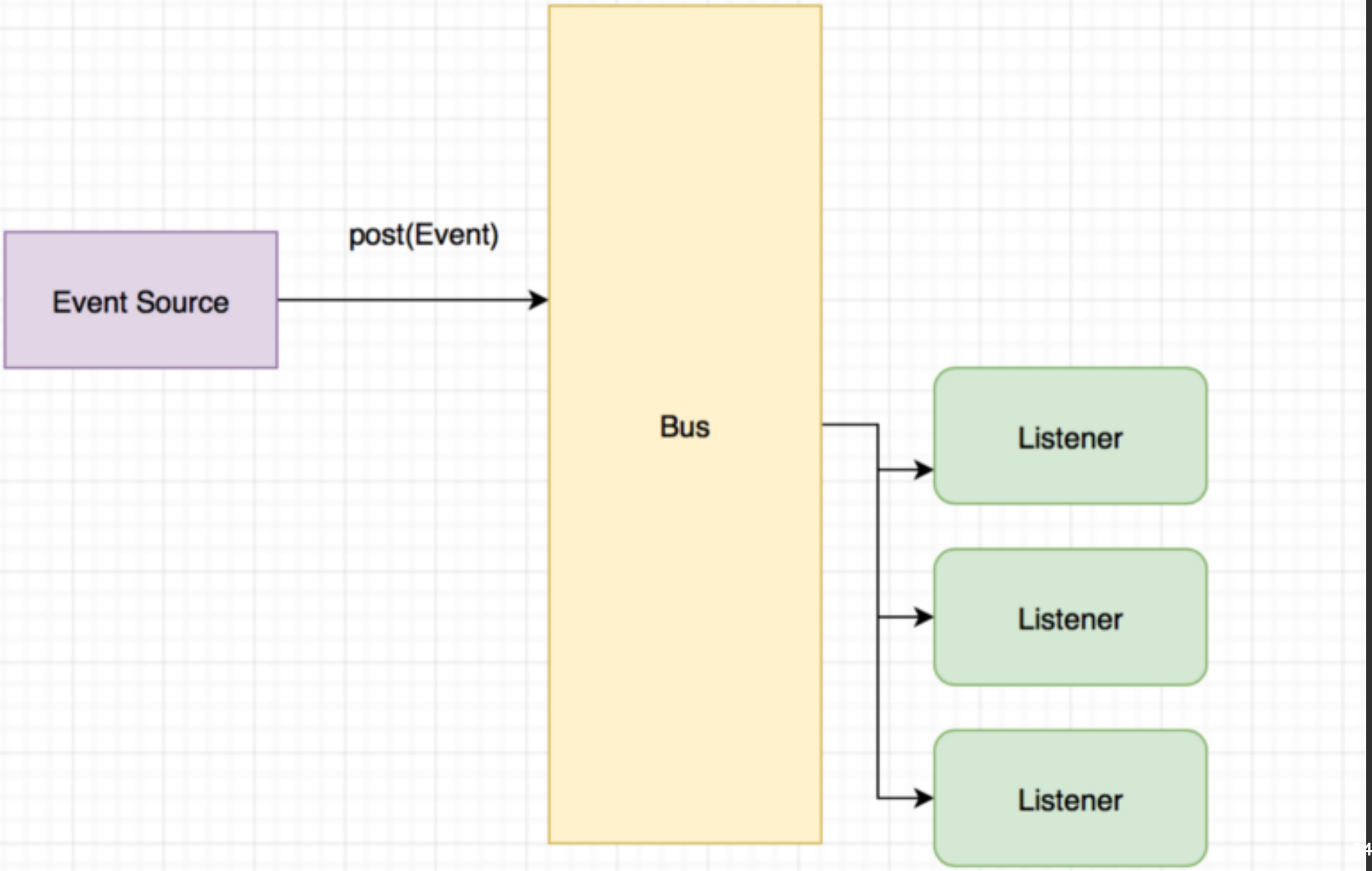
```
public class Background {

    …
    private lateinit var mUiHandler: Handler

    public fun postOnUiThread(runnable: Runnable) {
        mUiHandler.post(runnable)
    }
}
```

But how will out UI know it should be changed?

Event bus!


Event Source
post(Event)
Bus
Listener
Listener
Listener

Event bus implementations

Event bus implementations

— Google guava

Event bus implementations

— Google guava

— Otto

Event bus implementations

— Google guava
— Otto
— Greenbot eventbus

```
public class Background {

    private lateinit val mEventBus: Bus

    fun postEvent(event: Any) {
        mEventBus.post(event)
    }
}
```

```
mBackground.execute(Runnable {
    try {

        initDatabaseInternal()
        mBackground.post(DatabaseLoadedEvent())

    } catch (e: Exception) {
        Log.e("Failed to init db", e)
    }
```

```
public class SplashActivity : Activity() {

    @Subscribe
    fun on(event: DatabaseLoadedEvent) {
        progressBar.setVisibility(View.GONE)
        showMainActivity()
    }

    override fun onStart() {
        super.onStart()
        eventBus.register(this)
    }

    override fun onStop() {
        eventBus.unregister(this)
        super.onStop()
    }

}
```

What is the problem here?

UI is modified from the background thread!

```
public class Background {

    fun postEventOnUiThread(event: Any) {

        mUiHandler.post(Runnable {
            mEventBus.post(event)
        })

    }
}
```

Minuses

Minuses

— The posting code knows which thread is needed on the listeners code

Minuses

— The posting code knows which thread is needed on the listeners code

— The actual projects begin to be unmaintanable with EventBus

Next step: RxJava!

```
interface ApiClientRx {

    fun login(auth: Authorization)
        : Single<GithubUser>
    fun getRepositories
        (reposUrl: String, auth: Authorization)
        : Single<List<GithubRepository>>

}
```

```
private fun attemptLoginRx() {
    showProgress(true)
    apiClient.login(auth)
            .flatMap {
                    user -> apiClient.getRepositories(user.repos_url, auth)
            }
            .map { list -> list.map { it.full_name } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doFinally { showProgress(false) }
            .subscribe(
                    {
                        list -> showRepositories(this@LoginActivity, list)
                    },
                    {
                        error -> Log.e("TAG", "Failed to show repos", error)
                    }
            )
}
```

```
private fun attemptLoginRx() {
    showProgress(true)
    apiClient.login(auth)
            .flatMap {
                    user -> apiClient.getRepositories(user.repos_url, auth)
            }
            .map { list -> list.map { it.full_name } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doFinally { showProgress(false) }
            .subscribe(
                    {
                        list -> showRepositories(this@LoginActivity, list)
                    },
                    {
                        error -> Log.e("TAG", "Failed to show repos", error)
                    }
            )
}
```

```
private fun attemptLoginRx() {
    showProgress(true)
    apiClient.login(auth)
            .flatMap {
                    user -> apiClient.getRepositories(user.repos_url, auth)
            }
            .map { list -> list.map { it.full_name } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doFinally { showProgress(false) }
            .subscribe(
                    {
                        list -> showRepositories(this@LoginActivity, list)
                    },
                    {
                        error -> Log.e("TAG", "Failed to show repos", error)
                    }
            )
}
```

```
private fun attemptLoginRx() {
    showProgress(true)
    apiClient.login(auth)
            .flatMap {
                    user -> apiClient.getRepositories(user.repos_url, auth)
            }
            .map { list -> list.map { it.full_name } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doFinally { showProgress(false) }
            .subscribe(
                    {
                        list -> showRepositories(this@LoginActivity, list)
                    },
                    {
                        error -> Log.e("TAG", "Failed to show repos", error)
                    }
            )
}
```

```
private fun attemptLoginRx() {
    showProgress(true)
    apiClient.login(auth)
            .flatMap {
                    user -> apiClient.getRepositories(user.repos_url, auth)
            }
            .map { list -> list.map { it.full_name } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doFinally { showProgress(false) }
            .subscribe(
                    {
                        list -> showRepositories(this@LoginActivity, list)
                    },
                    {
                        error -> Log.e("TAG", "Failed to show repos", error)
                    }
            )
}
```

Threading?

```
private fun attemptLoginRx() {
    showProgress(true)
    apiClient.login(auth)
            .flatMap {
                    user -> apiClient.getRepositories(user.repos_url, auth)
            }
            .map { list -> list.map { it.full_name } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doFinally { showProgress(false) }
            .subscribe(
                    {
                        list -> showRepositories(this@LoginActivity, list)
                    },
                    {
                        error -> Log.e("TAG", "Failed to show repos", error)
                    }
            )
}
```

Pluses

Pluses

— Standard de-facto, 65% of new projects will use RxJava

Pluses

— Standard de-facto, 65% of new projects will use RxJava

— Powerful API, lots of operators

Pluses

— Standard de-facto, 65% of new projects will use RxJava

— Powerful API, lots of operators

— Opensource and community driven library

Pluses

— Standard de-facto, 65% of new projects will use RxJava
— Powerful API, lots of operators
— Opensource and community driven library
— Adapters for popular libraries like Retrofit

Pluses

— Standard de-facto, 65% of new projects will use RxJava
— Powerful API, lots of operators
— Opensource and community driven library
— Adapters for popular libraries like Retrofit
— Relatively easy to provide unit-tests

Minuses

Minuses

— Touch learning curve

Minuses

— Touch learning curve

— Operators to learn

Minuses

— Touch learning curve

— Operators to learn

— Callback style

Minuses

— Touch learning curve

— Operators to learn

— Callback style

— Memory overhead

Minuses

— Touch learning curve

— Operators to learn

— Callback style

— Memory overhead

— Unrelated stacktraces

What’s next?

What’s next?

Kotlin coroutines!

```
private fun attemptLogin() {
    launch(UI) {
        val auth = BasicAuthorization(login, pass)
        try {
            showProgress(true)
            val userInfo = login(auth).await()
            val repoUrl = userInfo.repos_url
            val list = getRepositories(repoUrl, auth).await()
            showRepositories(
                    this@LoginActivity,
                    list.map { it -> it.full_name }
            )
        } catch (e: RuntimeException) {
            Toast.makeText(this@LoginActivity, e.message, LENGTH_LONG).show()
        } finally {
            showProgress(false)
        }
    }
}
```

```
private fun attemptLogin() {
    …
    launch(UI) {
        val auth = BasicAuthorization(login, pass)
        try {
            showProgress(true)
            val userInfo = login(auth).await()
            val repoUrl = userInfo.repos_url
            val list = getRepositories(repoUrl, auth).await()
            showRepositories(this@LoginActivity, list.map { it -> it.full_name })
        } catch (e: RuntimeException) {
            Toast.makeText(this@LoginActivity, e.message, LENGTH_LONG).show()
        } finally {
            showProgress(false)
        }
    }
}
```

Suspension

Pluses

Pluses

— Non-blocking approach

Pluses

— Non-blocking approach

— Async code, sync styled

Pluses

— Non-blocking approach

— Async code, sync styled

— Language tools instead of operators

Pluses

— Non-blocking approach
— Async code, sync styled
— Language tools instead of operators
— Learning curve is almost straight-line

Pluses

— Non-blocking approach
— Async code, sync styled
— Language tools instead of operators
— Learning curve is almost straight-line
— Unit-tests are easy

Minuses

Minuses

— Fresh thing, status: experimental

Minuses

— Fresh thing, status: experimental

— Library adapters are not ready(retrofit?)

Minuses

— Fresh thing, status: experimental

— Library adapters are not ready(retrofit?)

— Not a replacement for Rx

Summary

Summary

— Android went long path

Summary

— Android went long path

— The modern tools are RxJava and Coroutines

Summary

— Android went long path

— The modern tools are RxJava and Coroutines

— But legacy is still there, hence the talk

Contacts

Contacts

— https://github.com/Kotlin/kotlinx.coroutines 

Contacts

— https://github.com/Kotlin/kotlinx.coroutines 💻

— https://twitter.com/vvsevolodovich 🐦

Contacts

— https://github.com/Kotlin/kotlinx.coroutines 💻

— https://twitter.com/vvsevolodovich 🐦

— https://medium.com/@dzigorium