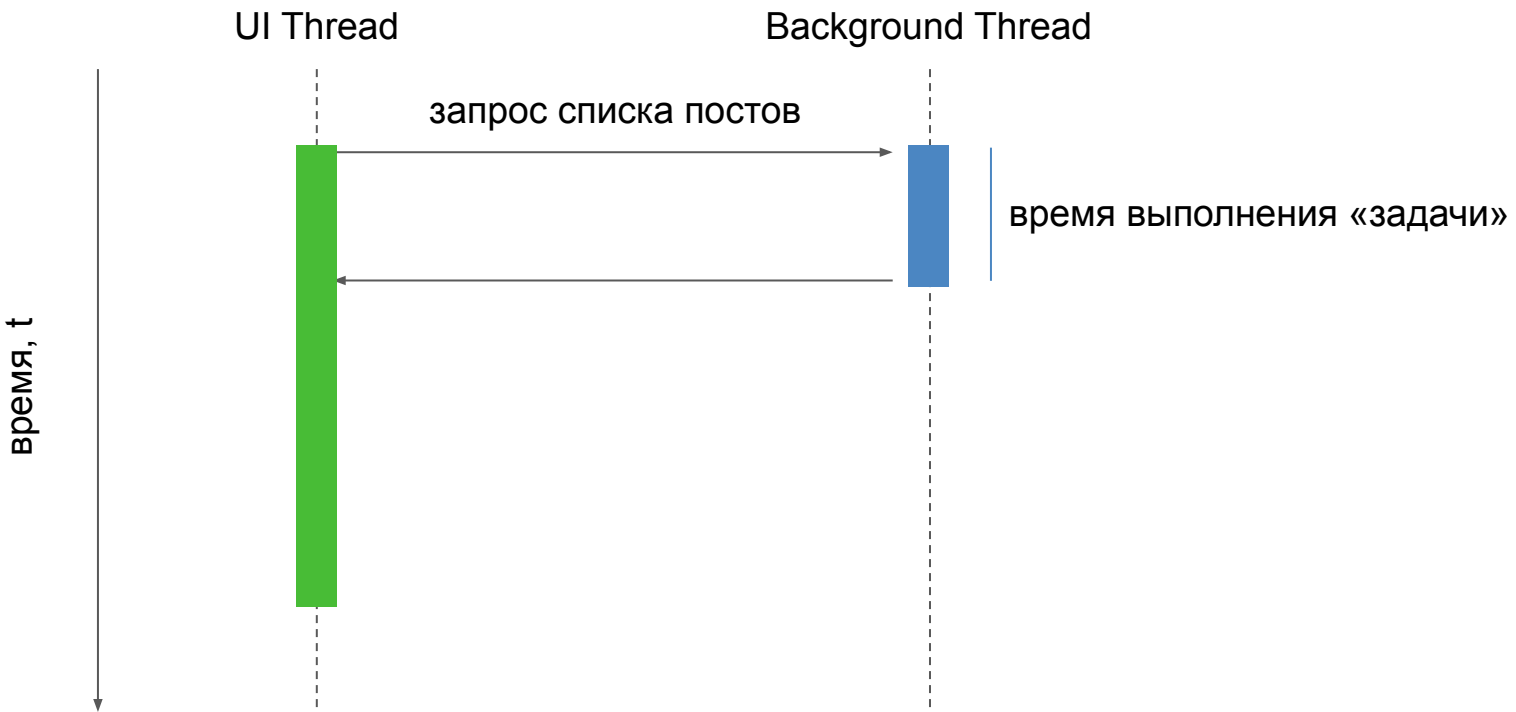
План занятия

1. Android Multithreading
2. Image Loaders
3. Итоги
4. Домашнее задание

ANDROID MULTITHREADING

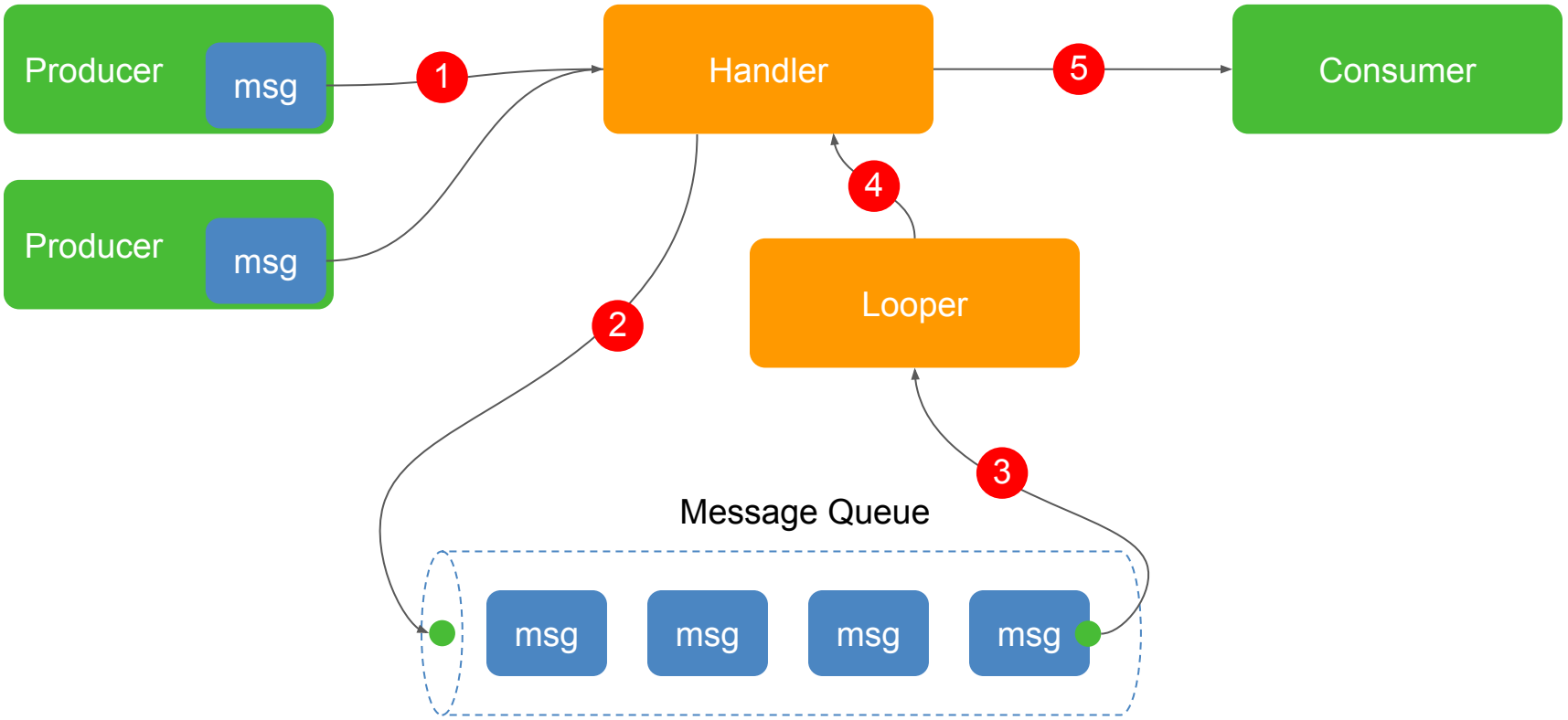
ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

Из рассмотренных нами лекций следует, что ключевым вариантом взаимодействия между потоками в Android является передача «задачи» из UI Thread в Background Thread с необходимостью получения результата:



MESSAGE PASSING

Android предлагает собственный механизм взаимодействия между потоками, который основывается на передаче сообщений:



MESSAGE PASSING

- Looper — диспетчер сообщений, который может быть использован только одним Consumer Thread
- Handler — обработчик поступающих сообщений (кладёт сообщения в MessageQueue), откуда их забирает Looper
- Message — сообщение, которое будет обработано Consumer'ом

AVATARS

К сожалению, в Android нет встроенного механизма для отображения изображений по URL.

Мы должны «вручную» скачать изображение, преобразовать его в один из типов, которые понимает ImageView и отобразить.

Давайте на этом небольшом примере разберём, как работает вся связка с предыдущих слайдов.

AVATARS

```
class MainActivity : AppCompatActivity() {
    private val worker = WorkerThread().apply { start() }
    private val urls = listOf("netology.jpg", "sber.jpg", "tcs.jpg")
    private var index = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.load.setOnClickListener { it: View!
            if (index == urls.size)
                index = 0
            }
            worker.download( url: "http://10.0.2.2:9999/avatars/${urls[index++]}")
        }
    }
}
```

AVATARS

```
class WorkerThread : Thread() {
    private lateinit var handler: Handler
    private val client = OkHttpClient.Builder()
        .connectTimeout( timeout: 30, TimeUnit.SECONDS)
        .build()

    override fun run() {...}

    fun download(url: String) {
        println("pass to queue: $url")
        val message = handler.obtainMessage().apply { this: Message
            data = bundleOf( ...pairs: "url" to url)
        }
        handler.sendMessage(message)
    }
}
```

исполняется не в UI Thread

handler.obtainMessage — способ получить объект Message для отправки с помощью дальнейшего вызова sendMessage.

LOOPER

```
override fun run() {
    Looper.prepare()
    // just for demo: don't use !! & as
    handler = Handler(Looper.myLooper()!!) {...}
    Looper.loop()
}
```

```
/** Initialize the current thread as a looper.
  * This gives you a chance to create handlers that then reference
  * this looper, before actually starting the loop. Be sure to call
  * {@link #loop()} after calling this method, and end it by calling
  * {@link #quit()}.
  */
public static void prepare() {
    prepare( quitAllowed: true);
}

private static void prepare(boolean quitAllowed) {
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper(quitAllowed));
}
```

```
/**
 * Run the message queue in this thread. Be sure to call
 * {@link #quit()} to end the loop.
 */
public static void loop() {
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }
```

THREAD LOCAL

Если заглянуть в класс Looper, то увидим:

```
// sThreadLocal.get() will return null unless you've called prepare().
@UnsupportedAppUsage
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
@UnsupportedAppUsage
private static Looper sMainLooper;  // guarded by Looper.class
```

ThreadLocal — это переменные, копии которых уникальны для каждого потока. Т.е. несмотря на то, что поля static final, у каждого потока будет своя, независимая от других потоков копия переменной (они не будут пересекаться, это не Shared State).

THREAD LOCAL

Мы обязаны вызывать prepare, чтобы создать Looper для нашего потока и вызывать loop, чтобы начать цикл обработки сообщений.

По завершению работы мы должны вызывать quit (в одном из Lifecycle методов Activity).

Looper.myLooper возвращает ссылку на Looper:

```
/**
 * Return the Looper object associated with the current thread.  Returns
 * null if the calling thread is not associated with a Looper.
 */
public static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}
```

```
override fun run() {
    Looper.prepare()
    // just for demo: don't use !! & as
    handler = Handler(Looper.myLooper()!!) { message ->
        try {
            val url = message.data["url"] as String
            println("loading: $url")

            val request = Request.Builder()
                .url(url)
                .build()

            client.newCall(request)
                .execute()
                .body?.use { it: ResponseBody
                    println("loaded: $url")
                    println(it.contentType())
                    println(it.contentLength())
                }
        } catch (e: IOException) {
            e.printStackTrace()
        }
        return@Handler true
    }
    Looper.loop()
}
```

Приложение работает (мы специально поставили 5-секундную задержку на сервере, чтобы было видно, в каком порядке поступают задачи и выполняются), но как передать полученный bitmap обратно в основной поток?

Проще всего «перевернуть» картину и не в наш поток отправлять сообщения, а из нашего потока в UI Thread.

ACTIVITY

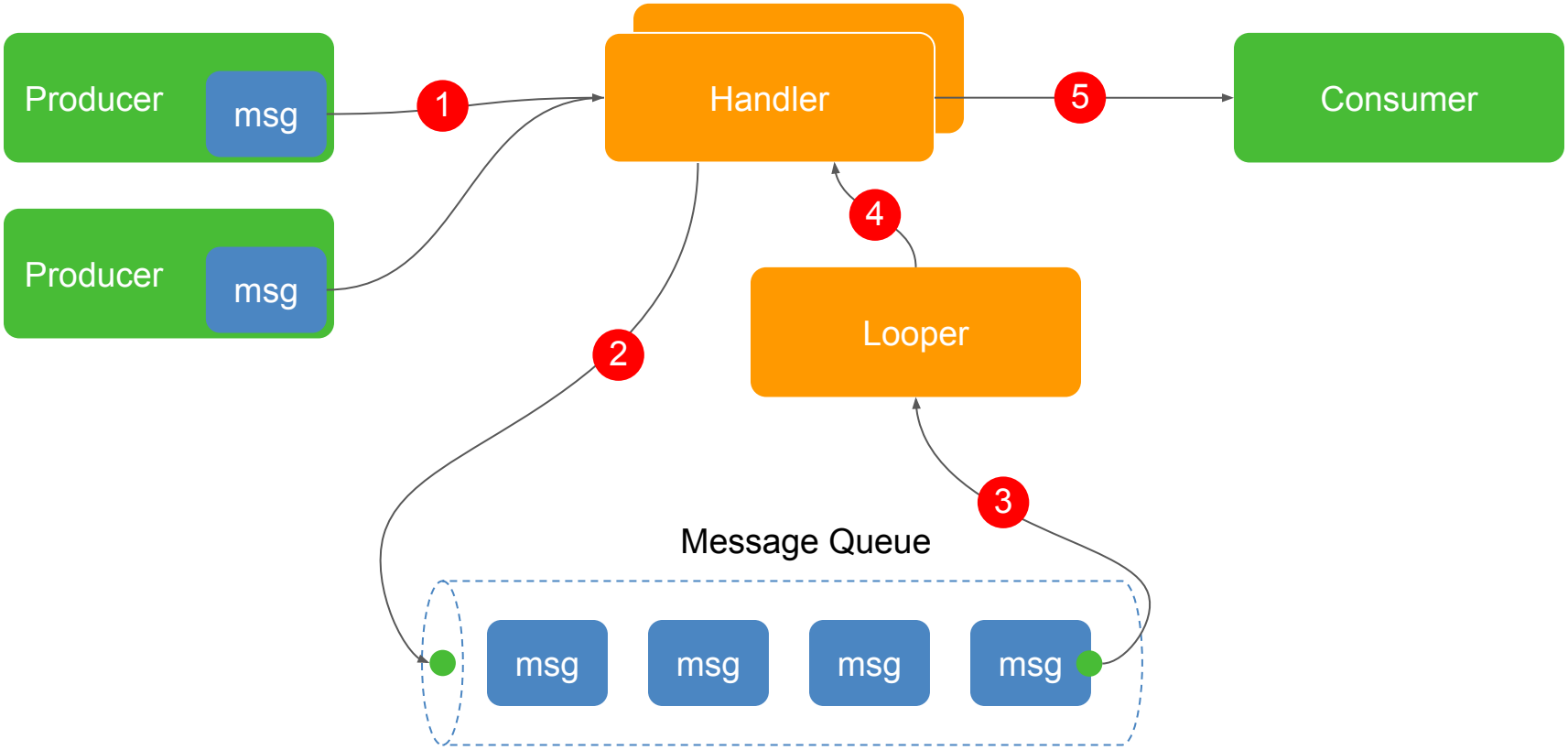
Для этого нам понадобится Handler для Activity:

```
// we must have a handler before the FragmentController is constructed
@UnsupportedAppUsage
final Handler mHandler = new Handler();
```

Но, как вы видите, в терминах Java он package private, поэтому нам не доступен.

HANDLER

Давайте вернёмся к нашей диаграмме: на самом деле Handler'ов может быть несколько, а Looper для потока только один:



```kotlin
val handler = Handler(Looper.getMainLooper()) { message ->
    // only for demo
    val bitmap = message.data["image"] as Bitmap
    binding.image.setImageBitmap(bitmap)
    return@Handler true
}

binding.load.setOnClickListener { it: View!
    if (index == urls.size) {...}

    val url = "http://10.0.2.2:9999/avatars/${urls[index++]}"
    val request = Request.Builder()
        .url(url)
        .build()

    client.newCall(request)
        .enqueue(object : Callback {
            override fun onResponse(call: Call, response: Response) {
                response.body?.use { it: ResponseBody
                    val bitmap = BitmapFactory.decodeStream(it.byteStream())
                    val message = handler.obtainMessage().apply { this: Message
                        data = bundleOf( ...pairs: "image" to bitmap)
                    }
                    handler.sendMessage(message)
                }
            }

            override fun onFailure(call: Call, e: IOException) {...}
        })
}
```

LOOPER

Для UI Thread'a Looper создаётся самой системой и хранится внутри Looper в отдельном поле:

```
/**
 * Returns the application's main looper, which lives in the main thread of the application.
 */
public static Looper getMainLooper() {
    synchronized (Looper.class) {
        return sMainLooper;
    }
}
```

RUNONUITHREAD

Конечно же, в Android есть и более простой способ сделать это:

```
client.newCall(request)
    .enqueue(object : Callback {
        override fun onResponse(call: Call, response: Response) {
            response.body?.use { it: ResponseBody
                val bitmap = BitmapFactory.decodeStream(it.byteStream())
                this@MainActivity.runOnUiThread {
                    binding.image.setImageBitmap(bitmap)
                }
            }
        }

        override fun onFailure(call: Call, e: IOException) {...}
    })
```

RUNONUITHREAD

«Под капотом» мы увидим всё то же самое + возможность в Message класть Runnable в поле callback:

```
public final void runOnUiThread(Runnable action) {
    if (Thread.currentThread() != mUiThread) {
        mHandler.post(action);
    } else {
        action.run();
    }
}
public final boolean post(@NonNull Runnable r) {
   return  sendMessageDelayed(getPostMessage(r), delayMillis: 0);
}

private static Message getPostMessage(Runnable r) {
    Message m = Message.obtain();
    m.callback = r;
    return m;
}
```

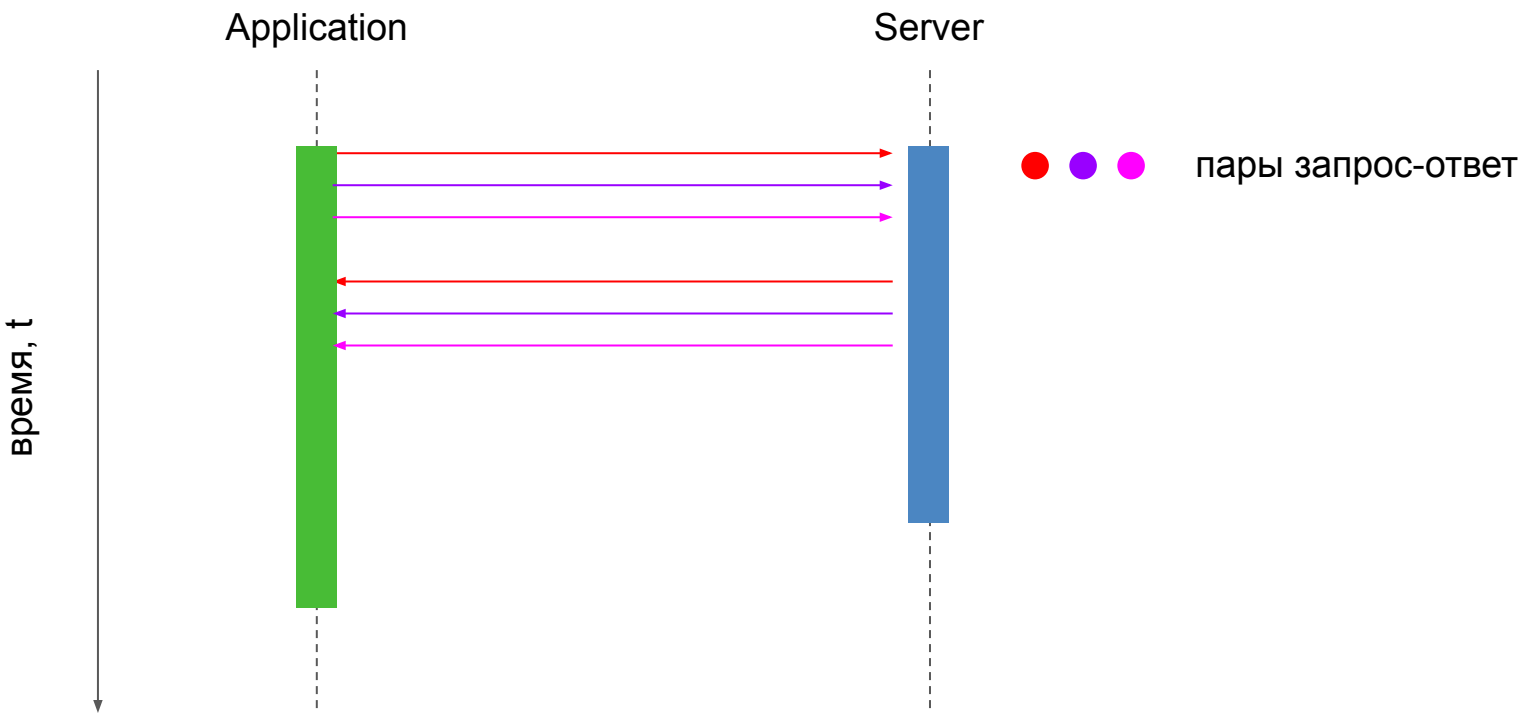
Кстати говоря, со ViewModel всё примерно так же:

```
protected void postValue(T value) {
    boolean postTask;
    synchronized (mDataLock) {
        postTask = mPendingData == NOT_SET;
        mPendingData = value;
    }
    if (!postTask) {
        return;
    }
    ArchTaskExecutor.getInstance().postToMainThread(mPostValueRunnable);
}
```

```
@Override
public void postToMainThread(Runnable runnable) {
    if (mMainHandler == null) {
        synchronized (mLock) {
            if (mMainHandler == null) {
                mMainHandler = createAsync(Looper.getMainLooper());
            }
        }
    }
    //noinspection ConstantConditions
    mMainHandler.post(runnable);
}
```

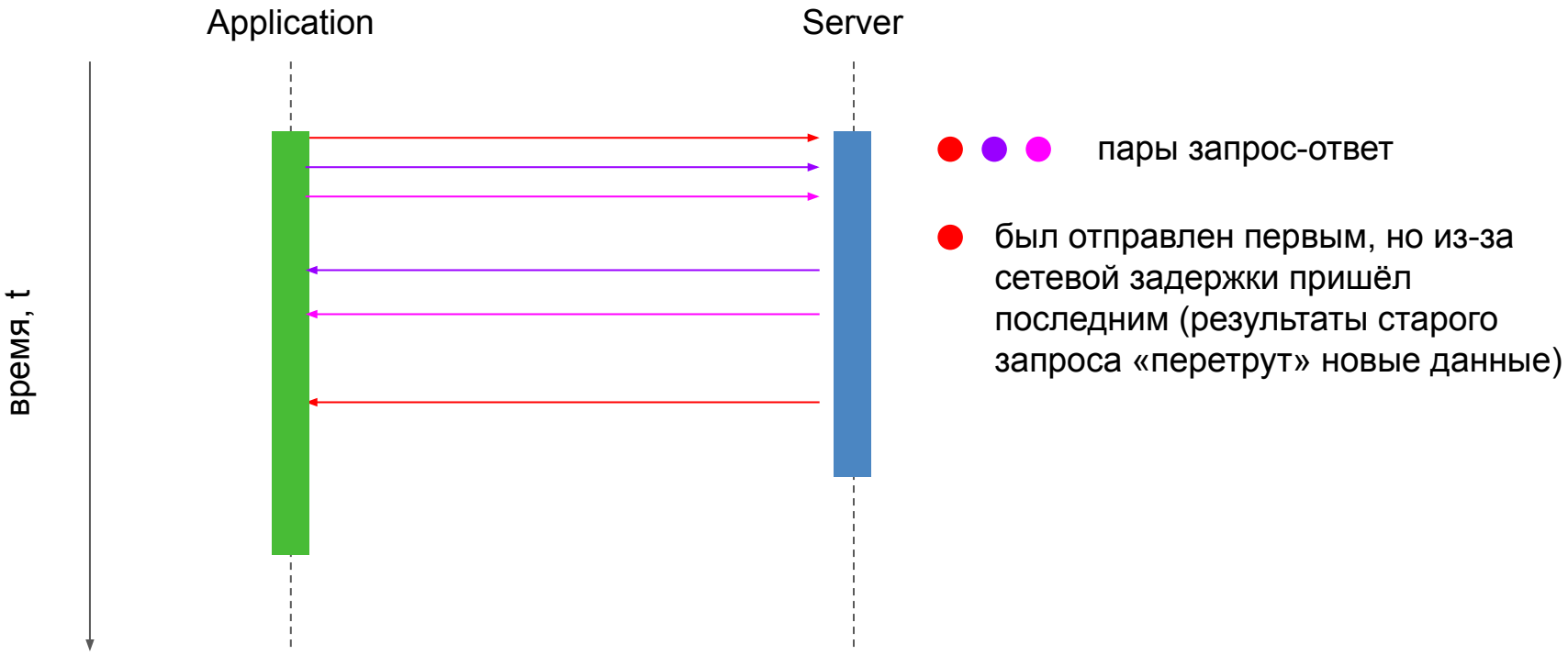
NETWORK LATENCY

А теперь попробуем «покликать» на кнопке много раз. Что происходит? Изображения продолжают загружаться, но быстро меняют друг друга:



NETWORK LATENCY

Обратите внимание: мы поставили фиксированную задержку в 5 секунд на сервере. В реальной жизни задержка может быть разная для разных запросов и мы получим типичную ситуацию гонки:



RECYCLERVIEW

? Вопрос: как вы думаете, какие проблемы возникнут при использовании загрузки рассмотренным нами способом в RecyclerView?

IMAGE LOADERS

IMAGE LOADERS

Конечно же, «вручную» каждый раз писать загрузчик изображений — плохая идея, тем более, что существующие инструменты предлагают уже готовые решения:

1. Загрузка в одну строку
2. Thread Pool'ы
3. Интеграция с жизненным циклом и RecyclerView
4. Кеширование
5. Манипуляции с изображениями (обрезка, центрирование, масштабирование)
6. Плейсхолдеры и fallback изображения (показываемые в случае ошибки) и т.д.

IMAGE LOADERS

Самые популярные на сегодняшний день:

1. Glide
2. Picasso
3. Fresco (не рассматриваем)

В зависимости от предпочтений команды используют один из трёх (хотя Glide используется чаще).

ЗАВИСИМОСТИ

```
implementation "com.github.bumptech.glide:glide:$glide_version"
implementation "com.facebook.fresco:fresco:$fresco_version"
implementation "com.squareup.picasso:picasso:$picasso_version"
```

GLIDE

Начинается всё с того, что вы используете Glide.with:

```
Glide.with
```

```
with(view: View)                                RequestManager
with(context: Context)                          RequestManager
with(activity: Activity)                        RequestManager
with(fragment: Fragment)                        RequestManager
with(activity: FragmentActivity)                RequestManager
with(fragment: Fragment)                        RequestManager
Press Enter to insert, Tab to replace
```

Чаще всего (особенно в RecyclerView) вы будете использовать вариант с view (который сам и достанет контекст):

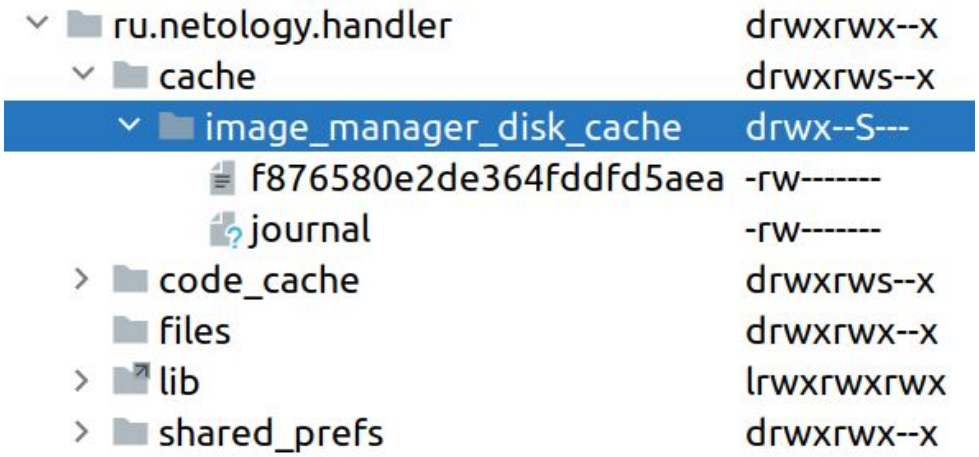
```
@NonNull
public static RequestManager with(@NonNull View view) {
  return getRetriever(view.getContext()).get(view);
}
```

GLIDE

```
private val urls = listOf("netology.jpg", "sber.jpg", "tcs.jpg", "404.png")
private var index = 0

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.load.setOnClickListener { it: View!
        if (index == urls.size) {
            index = 0
        }

        val url = "http://10.0.2.2:9999/avatars/${urls[index++]}"
        Glide.with(binding.image)
            .load(url)
            .placeholder(R.drawable.ic_loading_100dp)
            .error(R.drawable.ic_error_100dp)
            .timeout( timeoutMs: 10_000)
            .into(binding.image)
    }
}
```

GLIDE

При этом изображение автоматически будет кэшироваться:



GLIDE

Как это работает — типичный шаблон Singleton:

```
private static volatile Glide glide;
/**
 * Get the singleton.
 *
 * @return the singleton
 */
@NonNull
public static Glide get(@NonNull Context context) {
  if (glide == null) {
    GeneratedAppGlideModule annotationGeneratedModule =
        getAnnotationGeneratedGlideModules(context.getApplicationContext());
    synchronized (Glide.class) {
      if (glide == null) {
        checkAndInitializeGlide(context, annotationGeneratedModule);
      }
    }
  }

  return glide;
}
```

APPLICATION CONTEXT

Когда в Android запускается приложение, происходит следующая цепочка событий:

1. Создаётся процесс
2. Создаётся UI Thread
3. Создаётся экземпляр класса Application (который является контекстом)
4. Создаётся экземпляр запускаемого компонента (например, Activity, если запускаем через Launcher)

APPLICATION CONTEXT

Экземпляр класса Application существует на протяжении всей жизни приложения, поэтому многие инструменты используют его для работы (дабы избежать утечек памяти*):

> Return the context of the single, global Application object of the current process. This generally should only be used if you need a Context whose lifecycle is separate from the current context, that is tied to the lifetime of the process rather than the current component.

Примечание*: про утечки памяти (memory leaks) мы будем говорить отдельно.

GLIDE

```
@NonNull
Glide build(@NonNull Context context) {
  if (sourceExecutor == null) {
    sourceExecutor = GlideExecutor.newSourceExecutor();
  }

  if (diskCacheExecutor == null) {
    diskCacheExecutor = GlideExecutor.newDiskCacheExecutor();
  }
```

Где в качестве Executor'ов, как вы уже наверное догадались, те самые ThreadPoolExecutor'ы (см. следующий слайд).

GLIDE

```
/** Builds a new {@link GlideExecutor} with any previously specified options. */
public GlideExecutor build() {
  if (TextUtils.isEmpty(name)) {
    throw new IllegalArgumentException(
        "Name must be non-null and non-empty, but given: " + name);
  }
  ThreadPoolExecutor executor =
      new ThreadPoolExecutor(
          corePoolSize,
          maximumPoolSize,
          /*keepAliveTime=*/ threadTimeoutMillis,
          TimeUnit.MILLISECONDS,
          new PriorityBlockingQueue<Runnable>(),
          new DefaultThreadFactory(name, uncaughtThrowableStrategy, preventNetworkOperations));

  if (threadTimeoutMillis != NO_THREAD_TIMEOUT) {
    executor.allowCoreThreadTimeOut( value: true);
  }

  return new GlideExecutor(executor);
}
```

ПРОЦЕСС ЗАГРУЗКИ

А теперь разберём сам процесс загрузки, чтобы понять, какие проблемы уже решены за нас.

Q: Зачем это нужно? Разве недостаточно просто использовать инструмент?

A: Понять, как устроено и как правильно* работать с жизненным циклом и поведением Android можно только путём разбора исходников и чужих библиотек.

КЛЮЧЕВЫЕ МОМЕНТЫ

1. В RecyclerView View переиспользуется, поэтому нельзя загружать в переиспользуемое View результаты ответа от старого запроса
2. У всего есть жизненный цикл — нужно учитывать это
3. Кэш — запись в него и чтение из него

КЛЮЧЕВЫЕ МОМЕНТЫ

```
return into(
    glideContext.buildImageViewTarget(view, transcodeClass),
    /*targetListener=*/ targetListener: null,
    requestOptions,
    Executors.mainThreadExecutor());
```

1. buildImageViewTarget — обёртка для ImageView
2. requestOptions — опции запроса, например, таймаут, placeholder и т.д.
3. Executors.mainThreadExecutor — Executor для выполнения callback'a:

```
private static final Executor MAIN_THREAD_EXECUTOR =
    new Executor() {
      private final Handler handler = new Handler(Looper.getMainLooper());

      @Override
      public void execute(@NonNull Runnable command) { handler.post(command); }
    };
```

ЗАПРОС

Дальше самое интересное:

```
requestManager.clear(target);
target.setRequest(request);
requestManager.track(target, request);
```

У View есть поле tag, в который можно положить любой Object, так вот туда Glide и кладёт Request (и при необходимости вычищает):

```
/**
 * The view's tag.
 * {@hide}
 *
 * @see #setTag(Object)
 * @see #getTag()
 */
@UnsupportedAppUsage
protected Object mTag = null;
```

```
/**
 * Sets the tag associated with this view. A tag can be used to mark
 * a view in its hierarchy and does not have to be unique within the
 * hierarchy. Tags can also be used to store data within a view without
 * resorting to another data structure.
 *
 * @param tag an Object to tag the view with
 *
 * @see #getTag()
 * @see #setTag(int, Object)
 */
public void setTag(final Object tag) {
    mTag = tag;
}
```

GLIDE

А дальше уже идёт вызов соответствующих потоков и выполнение запроса:

```
if (hasResource) {
  // Acquire early so that the resource isn't recycled while the Runnable below is still sitting
  // in the executors queue.
  incrementPendingCallbacks( count: 1);
  callbackExecutor.execute(new CallResourceReady(cb));
} else if (hasLoadFailed) {
  incrementPendingCallbacks( count: 1);
  callbackExecutor.execute(new CallLoadFailed(cb));
} else {
  Preconditions.checkArgument(!isCancelled, message: "Cannot add callbacks to a cancelled EngineJob");
}
```

GLIDE

Ключевое для нас: понимание общей механики построения архитектуры:

- Thread Safe синглтон для доступа из любого участка приложения
- Thread Pool для выполнения запросов
- Handler/Looper для общения с UI-тредом

PICASSO

Посмотрим на альтернативную реализацию с помощью Picasso:

```
Picasso.get()
    .load(url)
    .error(R.drawable.ic_error_100dp)
    .into(binding.image);
```

Picasso, к сожалению, не поддерживает векторные placeholder.

PICASSO

```
public static Picasso get() {
  if (singleton == null) {
    synchronized (Picasso.class) {
      if (singleton == null) {
        if (PicassoProvider.context == null) {
          throw new IllegalStateException("context == null");
        }
        singleton = new Builder(PicassoProvider.context).build();
      }
    }
  }
  return singleton;
}
```

PICASSO

```
this.dispatcherThread = new DispatcherThread();
this.dispatcherThread.start();
Utils.flushStackLocalLeaks(dispatcherThread.getLooper());
this.handler = new DispatcherHandler(dispatcherThread.getLooper(), dispatcher: this);

void dispatchSubmit(Action action) {
  handler.sendMessage(handler.obtainMessage(REQUEST_SUBMIT, action));
}
```

HANDLERTHREAD

Где DispatcherThread — это:

```
static class DispatcherThread extends HandlerThread {
  DispatcherThread() {
    super( name: Utils.THREAD_PREFIX + DISPATCHER_THREAD_NAME, THREAD_PRIORITY_BACKGROUND);
  }
}
```

А HandlerThread — это встроенный в Android класс:

```
/**
 * A {@link Thread} that has a {@link Looper}.
 * The {@link Looper} can then be used to create {@link Handler}s.
 * <p>
 * Note that just like with a regular {@link Thread}, {@link #start()} must still be called.
 */
public class HandlerThread extends Thread {
    int mPriority;
    int mTid = -1;
    Looper mLooper;
    private @Nullable Handler mHandler;
```

HANDLERTHREAD

Далее это всё попадает в PicassoExecutorService на исполнение:

```
class PicassoExecutorService extends ThreadPoolExecutor {
  private static final int DEFAULT_THREAD_COUNT = 3;

  PicassoExecutorService() {
    super(DEFAULT_THREAD_COUNT, DEFAULT_THREAD_COUNT, keepAliveTime: 0, TimeUnit.MILLISECONDS,
        new PriorityBlockingQueue<Runnable>(), new Utils.PicassoThreadFactory());
  }
```

ИТОГИ

ИТОГИ

Сегодня мы посмотрели на вопросы организации асинхронного взаимодействия в Android и рассмотрели, как оно (взаимодействие) организуется в промышленных библиотеках:

1. Architecture Components
2. Glide и Picasso

На первом этапе вы будете чаще использовать уже готовые решения, но про Handler и Looper вас обязательно спросят на собеседовании.

deprecated

ASYNCTASK

В старых статьях, книгах и документации вы можете встретить использование AsyncTask для выполнения фоновых задач:

> Deprecated Use the standard `java.util.concurrent` or Kotlin concurrency utilities ↗ instead.

```
@Deprecated
public abstract class AsyncTask<Params, Progress, Result> {
```

Данный класс объявлен устаревшим и не рекомендуется к использованию.