Espresso. Продвинутая автоматизация тестирования Android

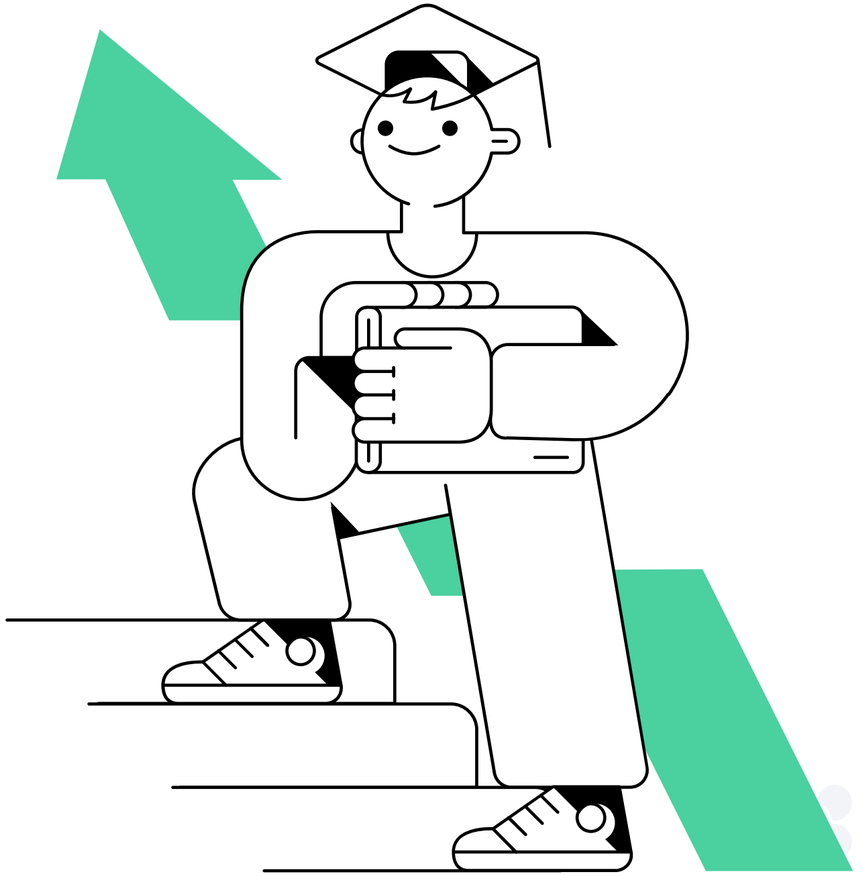
Кирилл Кузьмичев
Руководитель группы автоматизированного тестирования Яндекс.Такси

Кирилл Кузьмичёв

- Руководитель группы автоматизированного тестирования в Яндекс.Такси
- Нативное тестирование Android / iOS
- Нативное управление командой :)
- И даже мотоциклом

Цели занятия

1. Познакомимся с дополнительными библиотеками Espresso
2. Научимся проверять исходящие Intents приложения
3. Научимся дожидаться выполнения методов с использованием Idling Resources
4. Научимся реализовывать собственные ViewMatcher и ViewAssertions
5. Сможем собирать отчет и исправлять ошибки



План занятия

1. Проверка Intents
2. Работа с асинхронными операциями
3. Написание кастомных ViewMatcher
4. Написание кастомных ViewAssertions
5. Итоги знакомства с Espresso
6. Итоги
7. Домашнее задание
8. Дополнительные материалы

Проверка Intents

1

Проверка Intents

Intent - это объект для запроса действий от другого компонента приложения.
С его помощью можно:

1. Запускать Activity
2. Запускать Service
3. Передавать информацию между приложениями

Есть два типа Intent:

1. Explicit (Явные) - с указанием приложения, пакета или класса
2. Implicit (Неявные) - система выбирает приложение на основе передаваемых данных

Проверка Intents

Для проверки Intent в тестах, с помощью Espresso, необходимо добавить соответствующую зависимость в проект:

```
dependencies {
... // Стандартные зависимости не трогаем
   androidTestImplementation ‘androidx.test.espresso:espresso-intents:3.4.0’
}
```

А в тестах использовать метод intended() и IntentMatchers.

Например так:

```
@Test
public void testName() {
   …
   intended(
      hasData(“https://some.ru”) // IntentMatchers
   );
}
```

Проверка Intents. Способ №1

Есть два способа проверить Intent в классе тестов:

Первый - это использовать метод intended() внутри теста и IntentsTestRule.
Для этого необходимо задать вместо основного
“ActivityTestRule”-> “IntentsTestRule”

```
// Меняем
@Rule
public ActivityTestRule<MainActivity> activityTestRule =
   new ActivityTestRule<>(MainActivity.class);

// На
@Rule
public IntentsTestRule intentsTestRule =
   new IntentsTestRule(MainActivity.class);
```

Проверка Intents. Способ №2

Второй - использовать Intents.init() и Intents.release().

Intents.init() - записывает события Intent и должен вызываться до вызова Intent.
Intents.release() - “чистит” запись с Intents.

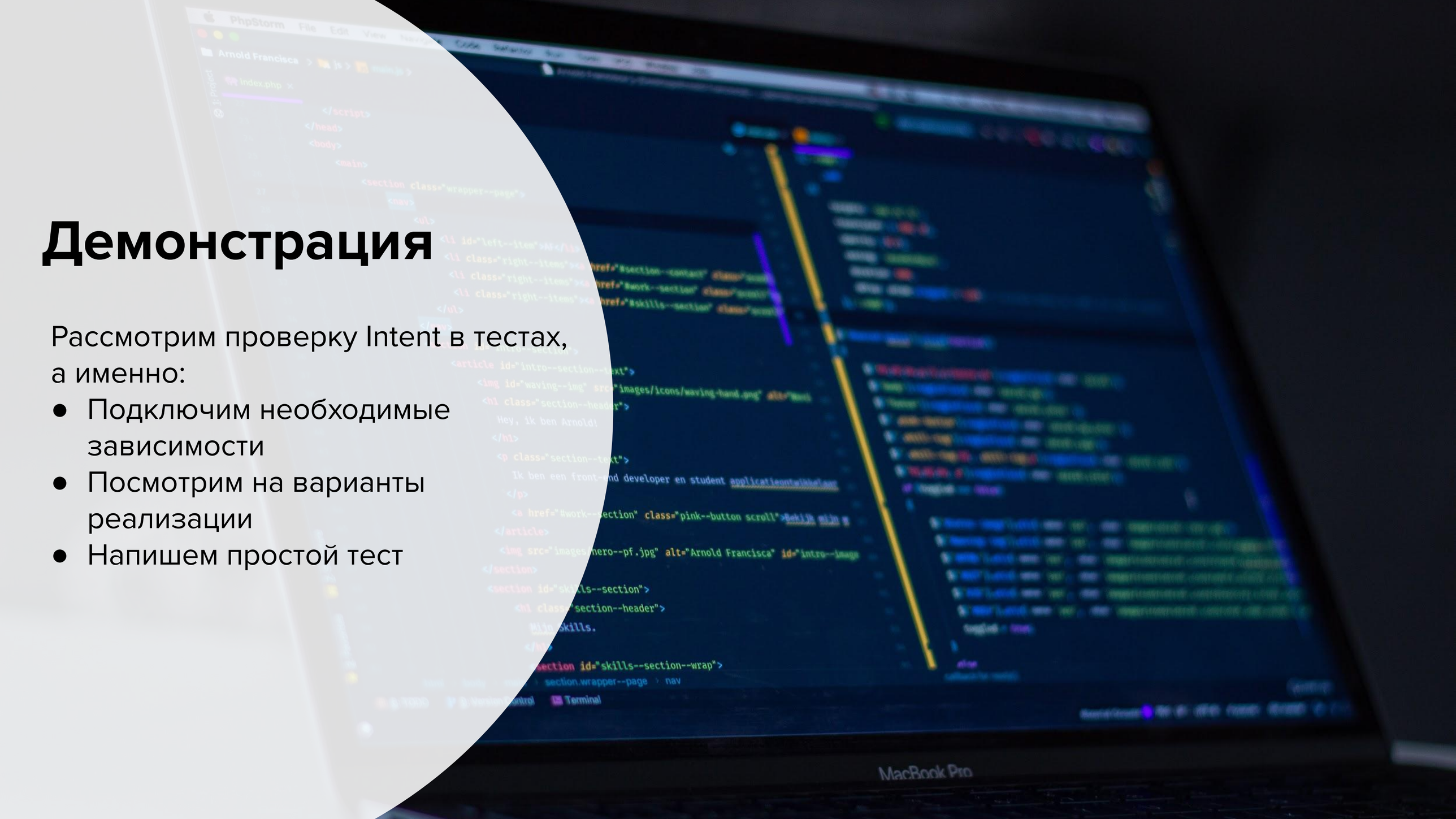
Если в первом варианте мы меняли “правило” для тестов и проверяли срабатывание Intent с помощью метода intended(). То во втором варианте нам необходимо указать Intents.init() перед вызовом Intent и Intents.release() по завершению проверок. Выглядеть это будет примерно так:

```
@Test
public void testName() {
   …
   Intents.init()
   item.perform(click()); // Для срабатывания Intent
   intended(hasData(“https://some.ru”)); // Проверка Intent
   Intents.release(); // Для “очистки” записей Intents
}
```

Демонстрация

Рассмотрим проверку Intent в тестах, а именно:

- Подключим необходимые зависимости
- Посмотрим на варианты реализации
- Напишем простой тест

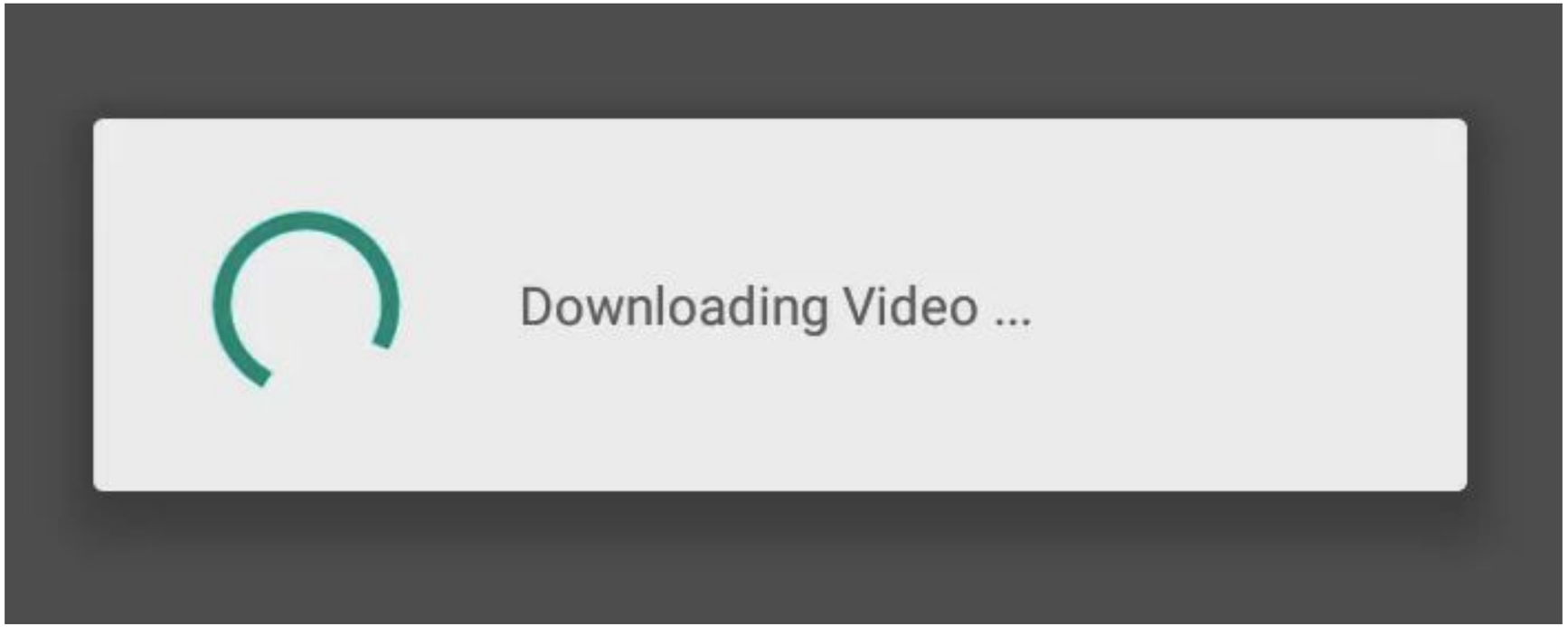


Работа с асинхронными операциями

2

Работа с асинхронными операциями

Асинхронные операции в Android применяются для выполнения трудозатратных операций в фоновом потоке. Такие как общение с сервером или загрузка картинок из хранилища. Обычно в случаях когда это занимает продолжительное время и информация не готова к отображению пользователю, применяется “заглушка”.



Работа с асинхронными операциями

Предположим, что нам надо проверить некий элемент в списке.
Но сам список иногда долго грузится и мы то видим заглушку, то нет.
Решить подобную задачу можно с помощью Thread.sleep() или через idling resources.
В случае с Thread.sleep() мы задаем время простоя между действиями в тесте.
И если за это время список прогрузился, то тест пройдет.

Такой подход имеет очевидные минусы:

1. Оптимальное время ожидания подбирается вручную для каждого случая
2. Список может не прогрузиться за отведенное время
3. Сильно растет время прохождения тестов

Работа с асинхронными операциями

Idling resources - позволяет дожидаться выполнения асинхронных операций в тестах. При этом выполнение теста будет продолжено после выполнения операций, а не по истечению времени ожидания.
По факту это реализация счетчика асинхронных операций и если он равен 0, то тест может продолжить выполнять свои шаги.

Для подключения Idling resources необходимо добавить следующую зависимость:

```
dependencies {
... // Стандартные зависимости не трогаем
   implementation 'androidx.test.espresso:espresso-idling-resources:3.4.0'
}
```

Работа с асинхронными операциями

Создадим в основной директории приложения/модуля необходимый для работы “счетчика” класс:

```
import androidx.test.espresso.idling.CountingIdlingResource;

public class EspressoIdlingResources() {
   private static final String RESOURCE = “GLOBAL” //Tag
   public static CountingIdlingResource idlingResource = new CountingIdlingResource(RESOURCE);

   public static void increment() {
      idlingResource.increment()
   }

   public static void decrement() {
      if(!idlingResource.isIdleNow()) {
         idlingResource.decrement();
         }
   }
}
```

Работа с асинхронными операциями

Основная “сложность” в использовании idling resource заключается в необходимости вызывать метод increment непосредственно перед асинхронной операцией и метод decrement по ее завершению.
В коде это выглядит примерно так:

```
 public class SomeClass() {

    public void someAsyncFunc() {
       EspressoIdlingResources.increment(); // Увеличили счетчик
       … // Сложные операции требующие время ;)
       EspressoIdlingResources.decrement(); // Уменьшили счетчик
    }
}
```

Работа с асинхронными операциями

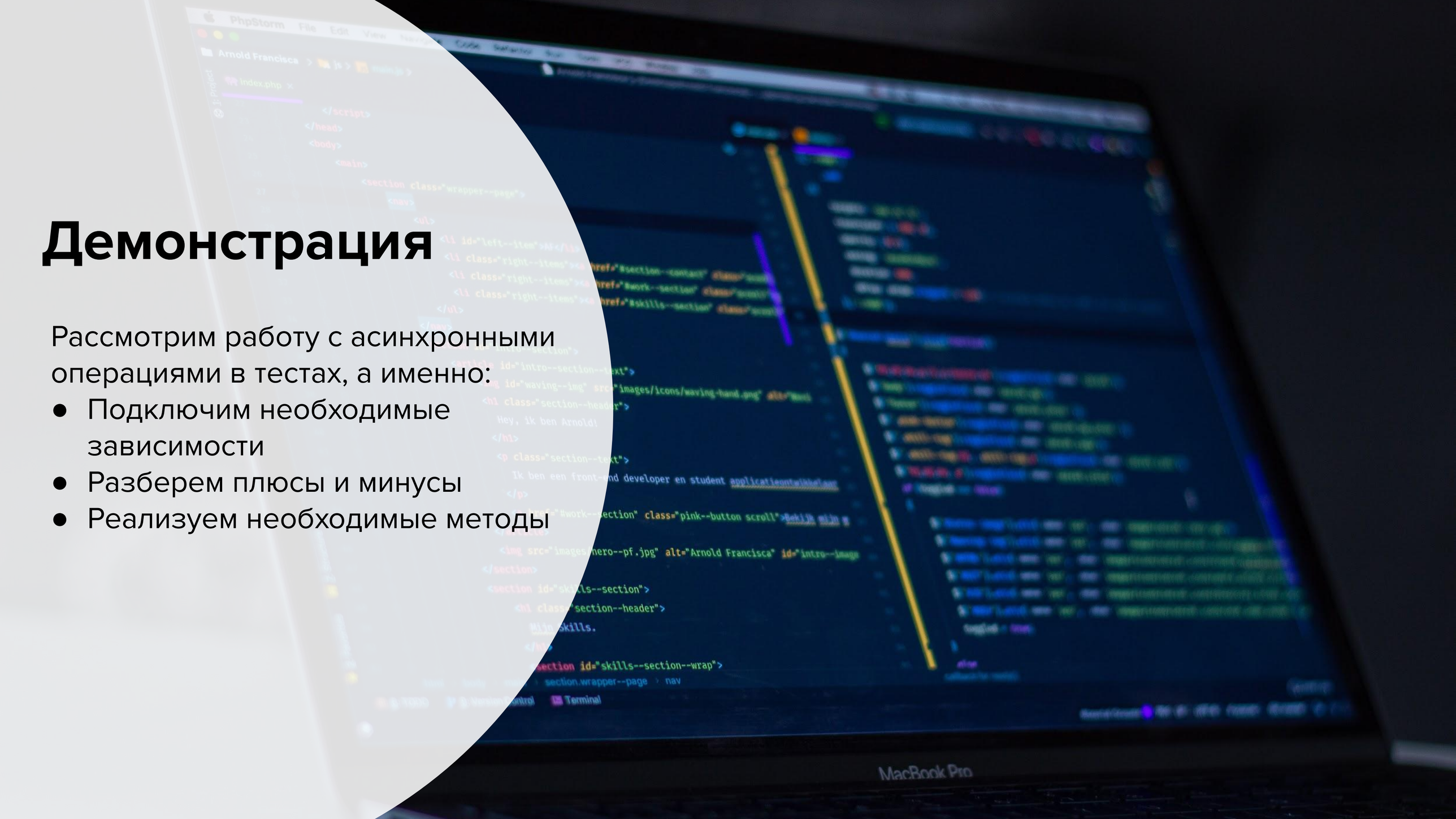
В тестах, для правильной работы, достаточно задать @Before и @After с подключением к “счетчику” и отключением от него:

```
public class SomeTestClass() {

   @Before // Выполняется перед тестами
   public void registerIdlingResources() { //Подключаемся к “счетчику”
      IdlingRegistry.getInstance().register(EspressoIdlingResources.idlingResource);
   }

   @After // Выполняется после тестов
   public void unregisterIdlingResources() { //Отключаемся от “счетчика”
      IdlingRegistry.getInstance().unregister(EspressoIdlingResources.idlingResource);
   }

… // В тесты добавлять ничего не надо

}
```

Демонстрация

Рассмотрим работу с асинхронными операциями в тестах, а именно:

- Подключим необходимые зависимости
- Разберем плюсы и минусы
- Реализуем необходимые методы



Написание кастомных ViewMatcher

3

Написание кастомных ViewMatcher

ViewMatcher отвечает за атрибуты элемента.
С его помощью проверяется состояние и задается уникальность элемента.
А если нужного ViewMatcher у нас нет, то мы можем с легкостью его написать.

Рассмотрим простой пример:
У нас есть RecyclerView с неким списком и нам необходимо проверить кол-во элементов в списке. Для этого создадим новый класс CustomViewMatcher и метод возвращающий Matcher<View> внутри.

Написание кастомных ViewMatcher

Выглядит это так:

```
imports … // Use org.hamcrest for Description and Matcher

public class CustomViewMatcher {

 public static Matcher<View> recyclerViewSizeMatcher(final int matcherSize) {
   return new BoundedMatcher<>(RecyclerView.class) {

   @Override
   public void describeTo(Description description) { // Доп. описание ошибки
      description.appendText("Item count: " + matcherSize)
    }

   @Override
   protected boolean matchesSafety(RecyclerView recyclerView) { // Проверка
      return matcherSize == recyclerView.getAdapter().getItemCount();
    }
   };
  }
}
```

Написание кастомных ViewMatcher

Разберем созданный класс подробнее:

- BoundedMatcher - абстрактный класс, содержащий в себе методы для простой реализации собственного Matcher.

- describeTo - метод позволяющий добавить дополнительное описание проверки элемента для лучшего восприятия ошибки, если проверка не прошла.

- matchesSafely - метод в котором проверяется соответствие атрибута элемента заданному значению.

В итоге, мы проверяем соответствие
matcherSize - ожидаемое нами кол-во элементов и
recyclerView.getAdapter().getItemCount(); - фактическое кол-во элементов

Написание кастомных ViewMatcher

Наш CustomViewMatcher можно использовать как для того, чтобы задать объект:

```
ViewInteraction recycleView =
onView(CustomViewMatcher.recyclerViewSizeMatcher(10)); // Ожидаемое кол-во элементов
```

Так и для его проверки:

```
ViewInteraction recycleView = onView(withId(R.id.recycler_view));
  recycleView.check(
     matches(CustomViewMatcher.recyclerViewSizeMatcher(10)) // Проверяем ожидаемое кол-во элементов
  );
```

Демонстрация

Рассмотрим реализацию собственных ViewMatcher, а именно:

- Напишем собственный ViewMatcher
- Разберем реализацию
- Применим в тесте

Написание кастомных ViewAssertions

4

Написание кастомных ViewAssertions

С помощью кастомного ViewMatcher мы смогли проверить кол-во элементов в RecyclerView. Но как проверить, что заданный элемент это RecyclerView?

Для этого можно написать кастомный ViewAssertion.
Создадим новый класс CustomViewAssertions и метод возвращающий ViewAssertion внутри.

Написание кастомных ViewAssertions

Выглядит это так:

```
imports …

public class CustomViewAssertions {

 public static ViewAssertion isRecyclerView() {
   return new ViewAssertion() {

   @Override
   public void check(View view, NoMatchingViewException noViewFoundException) {
      try {
         RecyclerView recyclerView = (RecyclerView) view;
      }
      catch (ClassCastException cce) {
         throw new IllegalStateException("This is not a RecyclerView");
      }
    }
   };
  }
}
```

Написание кастомных ViewAssertions

Метод check(), который мы переопределяем, включает в себя заданный в тесте элемент View и Exception на случай если элемент не найден.
Для проверки элемента, мы пытаемся привести view к типу RecyclerView и кидаем исключения если не вышло.

Таким образом мы выполняем необходимую нам проверку и контролируем сообщение в случае ошибки.
В тесте наша проверка выглядит следующим образом:

```
@Test
public void someTest() {
   ViewInteraction recyclerView = onView(withId(R.id.recycler_view));
   recyclerView.check(CustomViewAssertions.isRecyclerView());
}
```

Демонстрация

Рассмотрим реализацию собственных ViewAssertion, а именно:

- Напишем собственный ViewAssertion
- Разберем реализацию
- Применим в тесте

Итоги знакомства с Espresso

5

Итоги знакомства с Espresso

- Познакомились с Espresso и Allure.
- Научились добавлять необходимые зависимости, с учетом особенностей их применения
- Рассмотрели несколько вариантов запуска тестов
- Знаем как проверять Intent’ы и не зависеть от асинхронных операций
- Умеем делать красивые отчеты
- Можем добавлять необходимый нам функционал проверок и взаимодействия с элементами

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте в чате группы.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как приняты **все задачи**.

Дополнительные материалы

Пример хороших ссылок:

- Android Intents
- Android Idling resources

Задавайте вопросы и пишите отзыв о лекции!

Кирилл Кузьмичев