Тестируемый код

Что это и зачем

Содержание

1. Юнит тесты, как проверить работоспособность кода
2. Принципы SOLID, L & D

1. Unit-tests, как проверить работоспособность кода

Для начала давайте вспомним наши типичные задачи из джава раздела. Если помните мы писали класс треугольник и у него был по крайней мере 1 метод для проверки является ли он прямоугольным. Давайте быстро напишем код.

```
class Triangle(private val sideA: Int, private val sideB: Int, private val sideC: Int) {

    fun isRightTriangle(): Boolean {
        return sideA.square() + sideB.square() == sideC.square() ||
                sideA.square() + sideC.square() == sideB.square() ||
                sideC.square() + sideB.square() == sideA.square()
    }
}

private fun Int.square(): Int {
    return this * this
}
```

Для простоты я написал экстеншн функцию, вы можете использовать класс Math. Неважно.

Теперь, как мы ранее проверяли наш код? Мы писали в мейне вызовы метода и смотрели в консоль что там показывается. Примерно так

```
object Main {

    @JvmStatic
    fun main(args: Array<String>) {
        val triangle = Triangle( sideA: 3,  sideB: 4,  sideC: 5)
        print(triangle.isRightTriangle())
    }
}
```

```
n:  Main ×
/home/johnny/android-studio/jre/bin/java ...
true
Process finished with exit code 0
```

Кстати да, даже в AC можно написать psvm таким образом.

В чем минус такого подхода? Нам придется каждый раз запускать мейн метод и смотреть в консоль что там написано и самим проверять что там написано. Хотелось бы конечно чтобы это дело делалось само собой. Ведь если вы написали 1 метод то его легко проверить, но если их не 1, а 100? Ведь согласитесь, эта 1 проверка не говорит нам о том, что мы точно написали правильный код. Да, мы передали параметры для действительно прямоугольного треугольника, но мы забыли написать проверку для других параметров. А что если ваша функция всегда отвечает true на любые аргументы? Ачто если класс писали не вы и он прекомпилирован и вам недоступен исходный код? Давайте разбираться.

Краткий экскурс в тестирование

Итак, вы написали класс с 1 методом. Он возвращает булеан – true или false. Метод не принимает аргументов, но класс инициализируется посредством 3 чисел. Итого: чтобы быть уверенным в том, что вы написали правильный код вам нужно проверить все возможные ситуации. Числа бывают 3 видов – отрицательные, положительные и 0. Булеан 2 значения.

Значит вам нужно написать… К-комбинаторика

(+,+,+), (+,0,+), (+,-,+), (-,-,+)

(+,+,0), (+,0,0), (+,-,0), (-,-,0)

(+,+,-), (+,0,-), (+,-,-), (-,-,-)

Я решил не писать все кейсы, их реально много. Но и это не все. Кроме того что у нас 3 числа, у нас еще есть правило существования треугольника. Его конечно же можно написать в конструкторе и выбросить исключение. Но на это тоже нужны. Правило гласит – сумма любых 2 сторон должна быть больше третьей. Значит нам нужны тесты как на удовлетворение этих условий так и на неудовлетворение. Первых будет 3

a + b > c, a + c > b, b + c > a
а вот на неудовлетворение почти столько же

a + b <= c, a + c <=b, b + c <=a

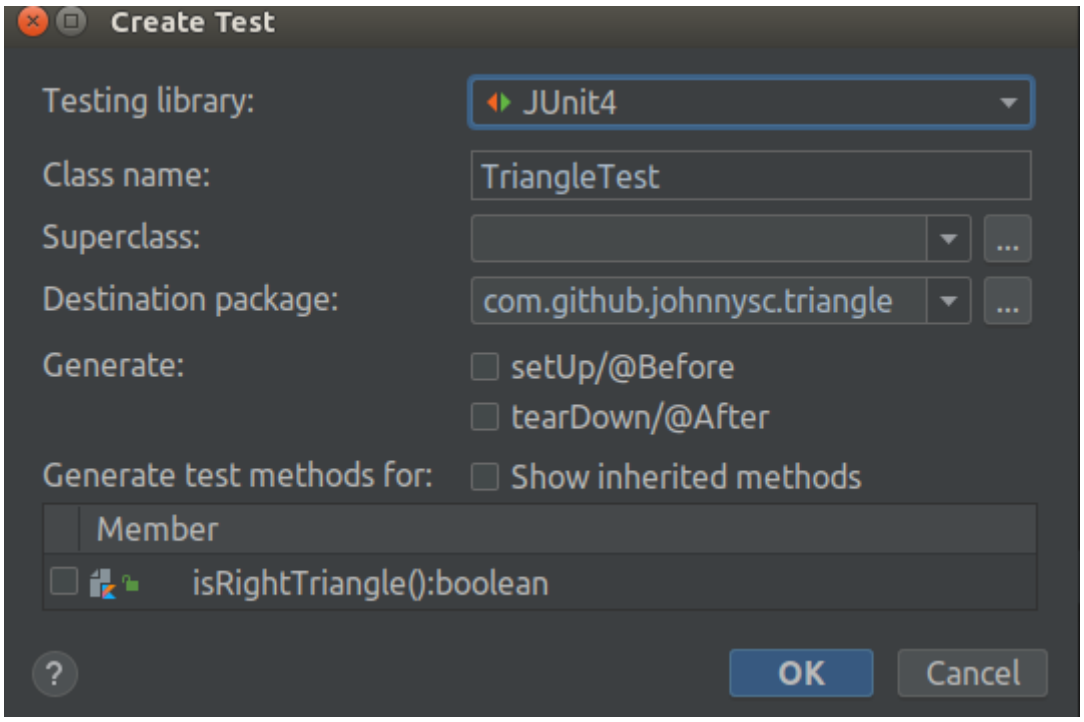
Ну и кроме всего этого у нас же проверка по теореме Пифагора. И здесь казалось бы 2 вероятности : или сумма квадратов 2 сторон равна квадрату третьей или же нет. Но ведь мы не знаем точно какие стороны катеты и надо проверить все. Так же надо будет проверить все ситуации для непрямоугольного.

Итак, дорогой мой друг, такая вот простая задача на проверку прямоугольного треугольника превратилась в ад. Если вы начнете писать тесты под это все то не закончите до конца дня.

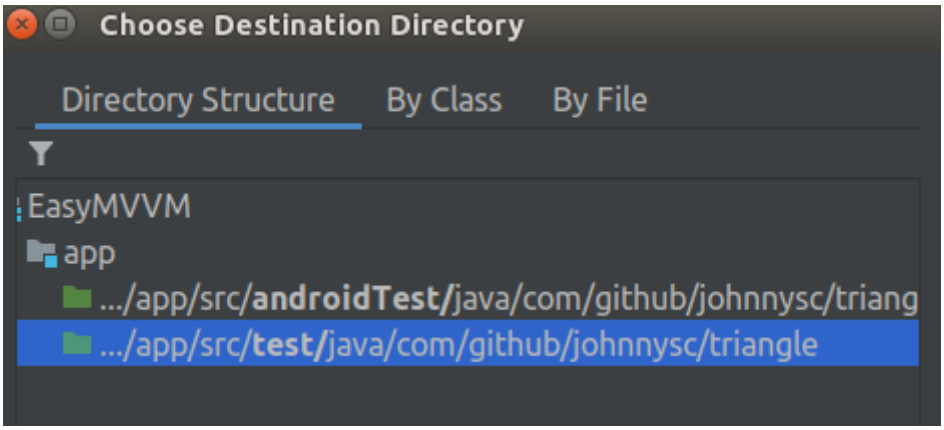
Звучит неплохо! Давайте займемся этим.

Итак, вы решили написать юнит-тесты. Для этого вам надо просто поставить курсор на имени класса и нажать Alt+Enter → create test

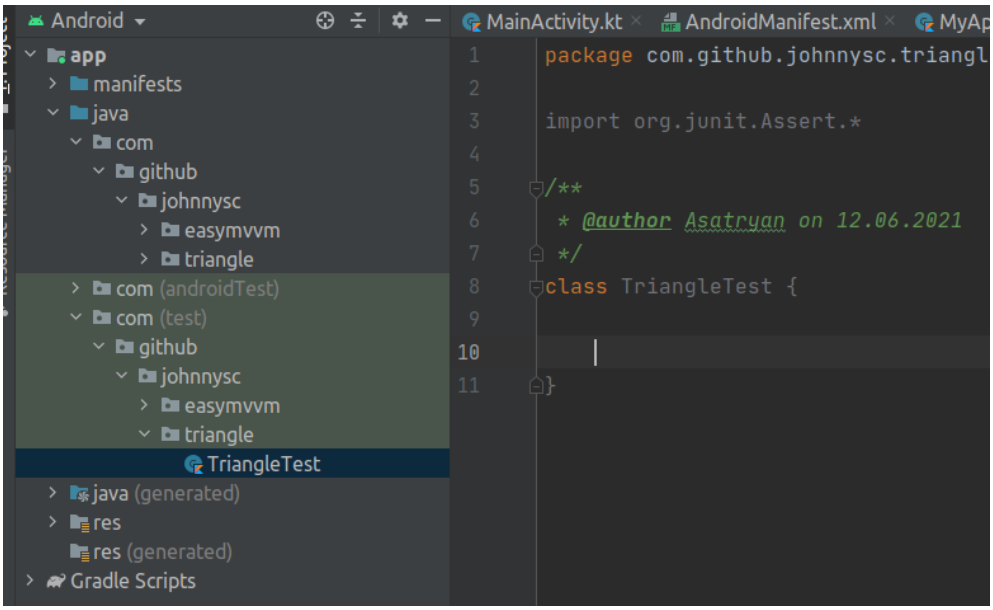
Вам откроется окно с выбором параметров. Давайте выберем Junit4



Пакет указываем такой смотрим что будет. Жмем ок.

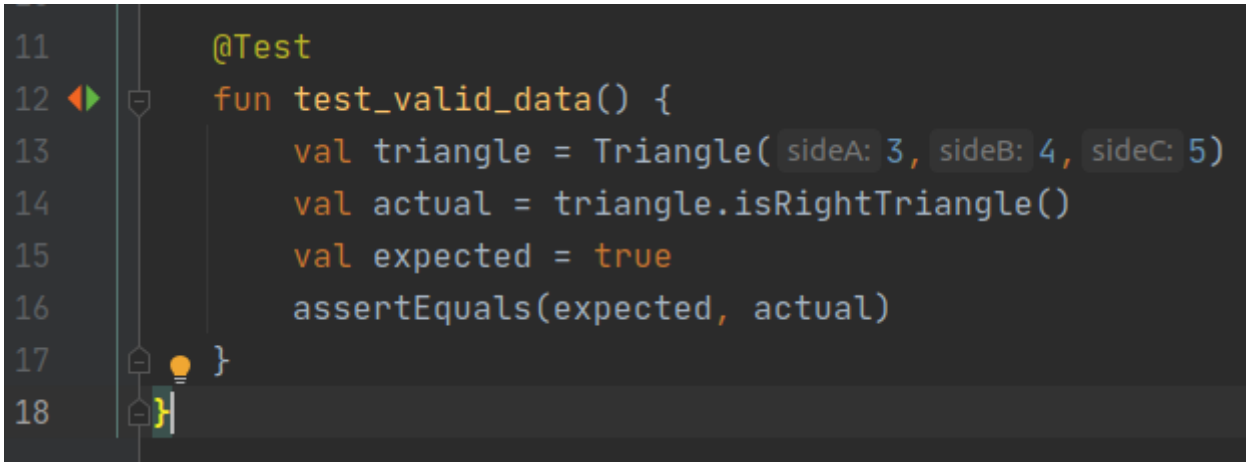


Здесь выбираем test потому что юнит тесты проверяют простой джава/котлин код. Пакет androidTest для юай тестов где мы запускаем приложение и робот проходит сценарий за нас (да, мы пройдем это в иной лекции). Жмем ок



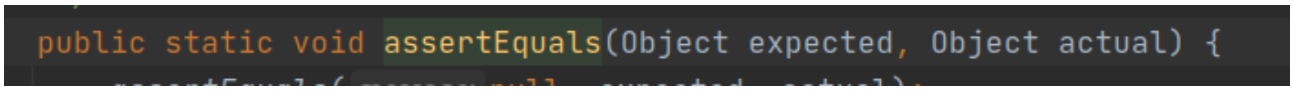
Как видите АС создало класс в такой же иерархии пакетов. Теперь мы можем написать наш юнит тест. Для этого нужно понимать суть юнит тестирования

Суть юнит теста в том, что вы создаете ожидаемый результат, получаете с помощью вашего класса и метода фактический (актуальный результат) и сравниваете их. Давайте конкретно на треугольнике.



1. Мы создаем обьект, 2. получаем актуальный результат. 3 создаем ожидаемый. 4. сравнить

Вы не запутаетесь в методе, потому что у него есть подсказка



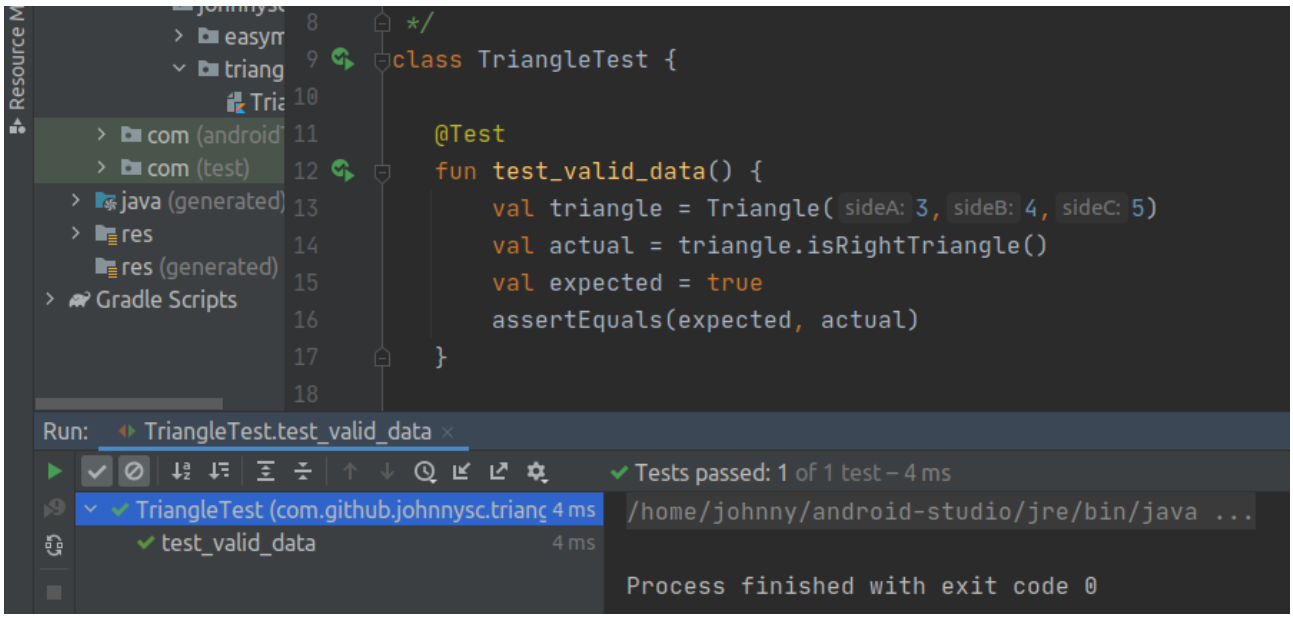
Мы сраниваем ожидаемое с актуальным. Вот и вся суть юнит теста как будто бы. Но такой тест просто ни о чем. Мы написали лишь 1 кейс, позитивный и с 1 набором данных. Да, метод работает так как мы ожидаем. Он проверил что 3 4 и 5 создали прямоугольный треугольник. А как же непрямоугольный? А как же кейсы когда передали положительные числа, которые не удовлетворяют правилу труегольника, например 1, 2, 3. По сути да, если мы напишем тест с аргументами 1, 2, 3 то результат будет как и ожидалось. Но ведь у нас в методе квадраты сторон. А значит если мы передадим -3, -4, -5 то тоже будет true.

Пара слов о TDD – Test Driven Development. Суть в том, что вы пишете тесты сначала и потом код. ЧТО?????? Да, не как все нормальные люди, а наоборот. Если бы мы следовали этой методологии, то написали бы сначала все тесты, потом только код.

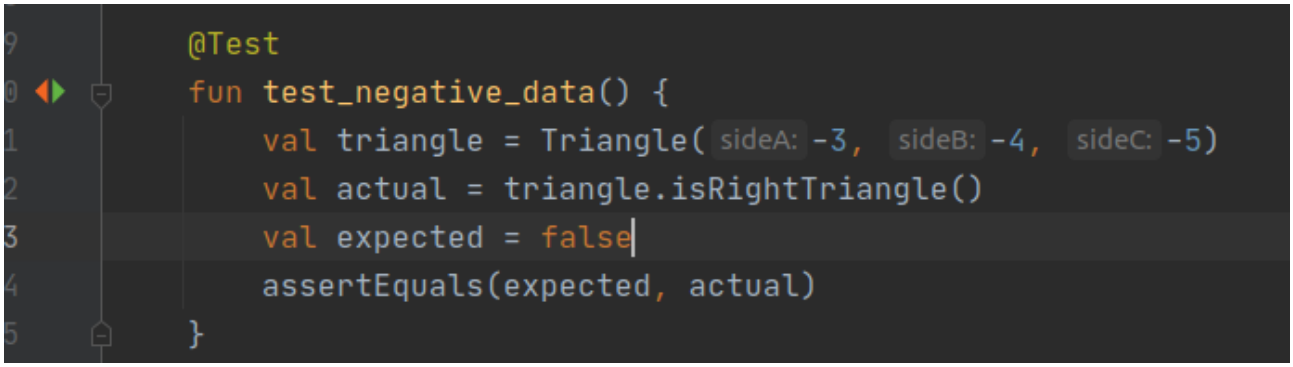
Так, подождите, мы же не запустили тесты. Так же как мейн метод, но можно слева от метода. Заметьте, метод становится тестируемым если написать @Test над ним.

Нажимаем на треугольник и вуаля. Тест пройден. Значок сменился на зеленый. Действительно, если мы имеем треугольник со сторонами 3 4 и 5, то он прямоугольный.

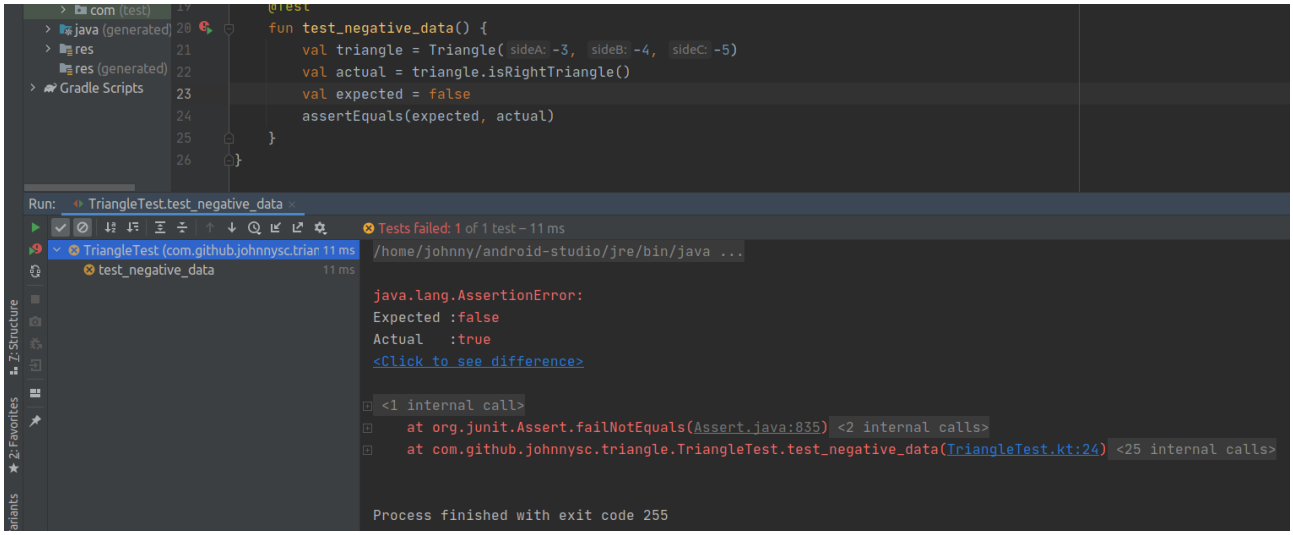
А теперь напишем то же самое с отрицательными сторонами



Так должен выглядеть метод примерно. Заметьте, что имена методов для теста я написал в снейк_кейс. Просто для камелКейса немного нечитаемо.



Запустим тест? Он не должен пройти – т.е. результат прохождения должен быть ошибкой



Ожидалось false, а было true. Кстати, методы тестового кода тоже можно дебажить!

Ладно, мы поняли что наш код кривой. Тест не проходит. Значит нам надо улучшить код.

Заметили что слева от теста значок красный?

```
class Triangle(private val sideA: Int, private val sideB: Int, private val sideC: Int) {

    init {
        if (sideA <= 0 || sideB <= 0 || sideC <= 0) {
            throw IllegalArgumentException("triangle sides cannot be non-positive")
        }
    }
```

Написали выброс исключения в инит блоке. Мы не будем сейчас говорить о том насколько это хороший или плохой подход. Пусть будет так. Теперь, прогоним тест.

```
Tests failed: 1 of 1 test – 13 ms
/home/johnny/android-studio/jre/bin/java ...

java.lang.IllegalArgumentException: triangle sides cannot be non-positive

    at com.github.johnnysc.triangle.Triangle.<init>(Triangle.kt:12)
    at com.github.johnnysc.triangle.TriangleTest.test_negative_data(TriangleTest.kt:21) <25 internal calls>
```

Тест не прошел, просто потому что он не дошел до последней линии где была проверка. Как же нам написать тест, в котором выбрасывается исключение? Легко!

```
java (generated)
res
res (generated)
Gradle Scripts

19
20      @Test(expected = IllegalArgumentException::class)
21      fun test_negative_data() {
22          val triangle = Triangle( sideA: -3,  sideB: -4,  sideC: -5)
23          val actual = triangle.isRightTriangle()
24          val expected = false
25          assertEquals(expected, actual)
26      }
27  }

Run:  TriangleTest.test_negative_data
Tests passed: 1 of 1 test – 3 ms
TriangleTest (com.github.johnnysc.trianc 3 ms
    test_negative_data 3 ms
/home/johnny/android-studio/jre/bin/java ...

Process finished with exit code 0
```

Прямо после аннотации Test пишем что ожидаем выброса исключения такого класса.

Ладно, мы написали уже 2 теста. Один на положительный кейс, второй на отрицательный кейс. Что еще? Еще надо учесть что числа просто не создают треугольник сами по себе.

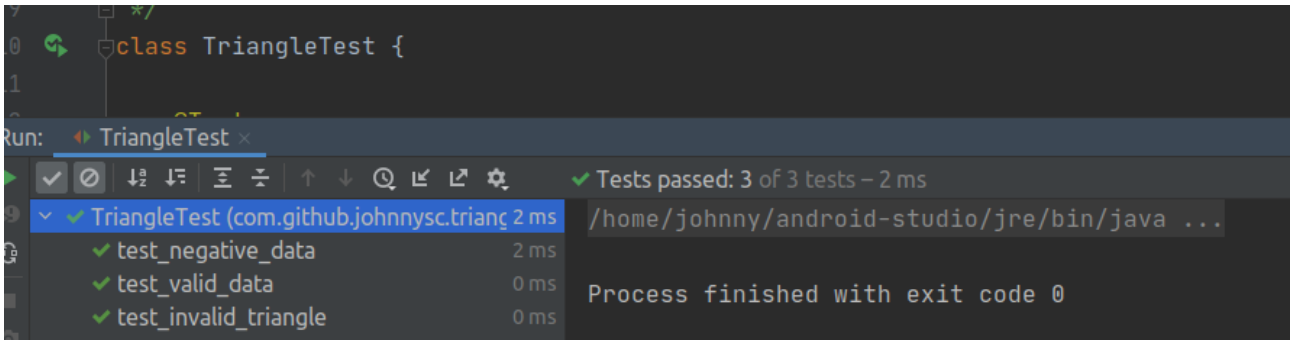
```
    @Test(expected = IllegalArgumentException::class)
    fun test_invalid_triangle() {
        val triangle = Triangle( sideA: 1,  sideB: 2,  sideC: 3)
        val actual = triangle.isRightTriangle()
        val expected = false
        assertEquals(expected, actual)
    }
}
```

Давайте сначала напишем тест и потом уже код. На самом деле в этом тестовом методе достаточно было и первой линии. Запустим код и тест не должен пройти

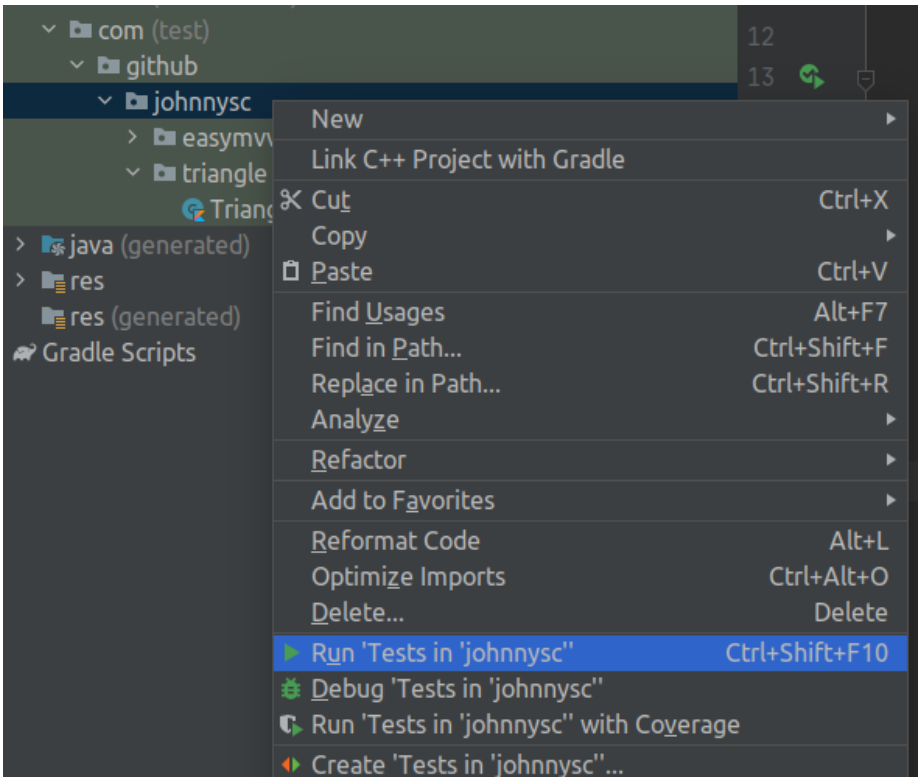
Так и есть. Но если убрать то, что мы ожидаем исключение то тест пройдет.

```
init {
    if (sideA <= 0 || sideB <= 0 || sideC <= 0) {
        throw IllegalArgumentException("triangle sides cannot be non-positive")
    }
    if (sideA + sideB <= sideC || sideB + sideC <= sideA || sideA + sideC <= sideB) {
        throw IllegalArgumentException("triangle sides cannot be non-positive")
    }
}
```

Теперь вернемся к нашему тесту и запустим его еще раз. И он прошел. И здесь у вас встанет вопрос. А не поломали ли мы другие тесты? Их уже много и я бы не хотел случайно сломать один из них. Было бы классно прогнать сразу все тесты. Это легко! Жмем иконку слева от имени класса теста



Вуаля. Если же у вас много классов тестовых, то можно все запустить из дерева.



Ладно. Мы проверили негативные числа, когда позитивные не образуют треугольник, на этом все? Нет конечно же. А как же существующие треугольники с позитивными сторонами, но просто не прямоугольные?

```
@Test
fun test_not_right_triangles() {
    val triangles = listOf(
        Triangle( sideA: 3,  sideB: 4,  sideC: 6),
        Triangle( sideA: 3,  sideB: 5,  sideC: 6),
        Triangle( sideA: 3,  sideB: 4,  sideC: 4),
        Triangle( sideA: 3,  sideB: 3,  sideC: 5),
    )
    val actuals = triangles.map { it.isRightTriangle() }
    val expected = false
    actuals.forEach { actual ->
        assertEquals(expected, actual)
    }
}
}
```

В идеале вы должны написать нечто подобное. Список треугольников которые валидные, но непрямоугольные. То же самое и касается теста прямоугольных. У нас там 1 кейс – 3, 4,5.

Кароче, написание юнит тестов по истине нужная штука и я вам докажу это ниже, когда мы перейдем от абстрактных и ненужных треугольников к нашему андроид приложению.

А зачем ты тогда 8 страниц рассказывал про тесты треугольника, спросите меня вы?

А чтобы вы понимали суть. По истине правильные юнит тесты проверяют функциональность. Если бы мы писали тесты ради тестов (эту фразу вы услышите на плохой работе, запомните ее и если услышите, то перед вами непрофессионалы однозначно, бегите оттуда), то мы бы просто написали бы 1 или 2 метода тестовых где передали бы 3, 4, 5 и например 3, 4, 6 и все. Вы всегда можете сказать (да и не только вы) – никто не будет передавать нули или отрицательные числа в конструктор треугольника. Ну ладно, это не так конечно, но все же. А что насчет правила треугольника? Треугольник со сторонами 1, 2, 3 физически не может существовать. Значит надо написать тесты на этот кейс и исправить код, если ваш метод просто вернет false, а не бросит исключение.

Запомните, раз и навсегда: если метод или конструктор принимает какие-либо аргументы, то в них можно будет передать все что угодно!

Если у вас число, то тут несколько вариантов – отрицательное, положительное, ноль и еще 2: максимальное и минимальное. Помните мы говорили о том, что если перемножить 2 числа int то они могут стать double и надо отдавать результат другого типа. Ибо перемножив миллиард на миллиард вы выйдете за рамки int. На самом деле легко запомнить правила. Для каждого аргумента берите крайние значения – максимум, минимум, ноль.

Если у вас строка например – берите пустую, пробел, длинную строку, которая содержит только буквы на латинице, на кириллице, смешанное, только цифры, только символы (~!@#$%^&*()_+<>,./) и тогда ваши тесты будут действительно хорошими и проверят все что можно и нельзя. Не ждите что кто-то вызовет ваш метод с правильными значениями. Всегда проверяйте все что можно и нельзя и имейте решение на все случаи жизни.

2. Принципы SOLID, L & D

Ладно, вернемся к андроид. Но перед тем как мы перейдем к тестам на тот код который мы писали в предыдущей лекции я расскажу вам про принципы SOLID, сразу про 2 из них.
L – Liskov substitution principle принцип подстановки Лисков. Он говорит о том, что нам нужно работать с такими классами (интерфейсами) чтобы работоспособность класса не зависела от конкретных реализаций. Обьясню ниже.

D – dependency inversion principe – принцип инверсии зависимостей – класс должен получать зависимые классы извне. Что это такое тоже обьясню ниже.

На одном примере сразу оба принципа. Погнали

Давайте пока отойдем от наших вьюмоделей и моделей из предыдущей лекции и просто напишем класс в котором работает другой класс. Итак, плохой код. Не делайте так

```
class Doer {

    private var mainThingDone = false
    private val logger = LoggingTool()

    fun doMain() {
        if (!mainThingDone) {
            logger.log("main thing done")
            mainThingDone = true
        }
    }
}

class LoggingTool {

    fun log(message: String) {
        Log.d(javaClass.canonicalName, message)
    }
}
```

Итак, у нас есть класс Doer и у него 1 метод, там что-то делается. Заметьте, один раз лишь за его жизнь. Второй раз оно не будет вызвано. Так же у него внутри есть некий логер, который

записывает в лог что-то. Ок, почему это плохой код? Давайте начнем с того, что любой код может называться хорошим если он : масштабируемый, поддерживаемый и читаемый. Проще говоря у него есть юнит-тесты по которым можно понять как оно должно работать. Давайте попробуем написать юнит тест на этот плохой класс. Но метод не принимает аргументов и он не возвращает ничего! Но в нем есть логика. Нам нужно проверить что код внутри действительно вызовется 1 раз. Проблема номер 1 – логер. Он класс без интерфейса.

Запомните раз и навсегда: в классе не должно быть публичных методов без override.

Кроме этого он создается в классе Doer в поле. Т.е. даже если мы напишем интерфейс логера и базовую реализацию, то все равно мы не сможем поменять его. Ладно. Давайте по порядку.

```
class LoggingTool : Logging {

    override fun log(message: String) {
        Log.d(javaClass.canonicalName, message)
    }
}

interface Logging {

    fun log(message: String)
}
```

Ну окей. Написали интерфейс и класс его наследует. Что дальше? Во-первых мы следуем принципу Лисков. Реализацию логинга можно заменить. Уже хорошо. Но этот принцип не работает без DIP инверсии зависимостей. Перепишем класс Doer.

```
class Doer(private val logger: Logging) {

    private var mainThingDone = false

    fun doMain() {
        if (!mainThingDone) {
            logger.log("main thing done")
            mainThingDone = true
        }
    }
}
```

Вот это и есть инверсия зависимостей. Класс получает свои зависимостей извне. Мы не инстанциируем обьекты внутри классов! Мы получаем обьекты снаружи. И нам неважно

какая реализация будет, для этого есть интерфейс. Теперь наш класс Doer удовлетворяет 2 принципам из SOLID (ну он еще конечно удовлетворяет S ибо делает 1 вещь – main method)

Теперь мы можем написать юнит тест на этот класс. Как? Ведь метод по прежнему не возвращет ничего а лишь вызывает метод другого класса. Да, но теперь этот класс у нас интерефейсом приходит в конструктор и его можно подменить (Лисков).

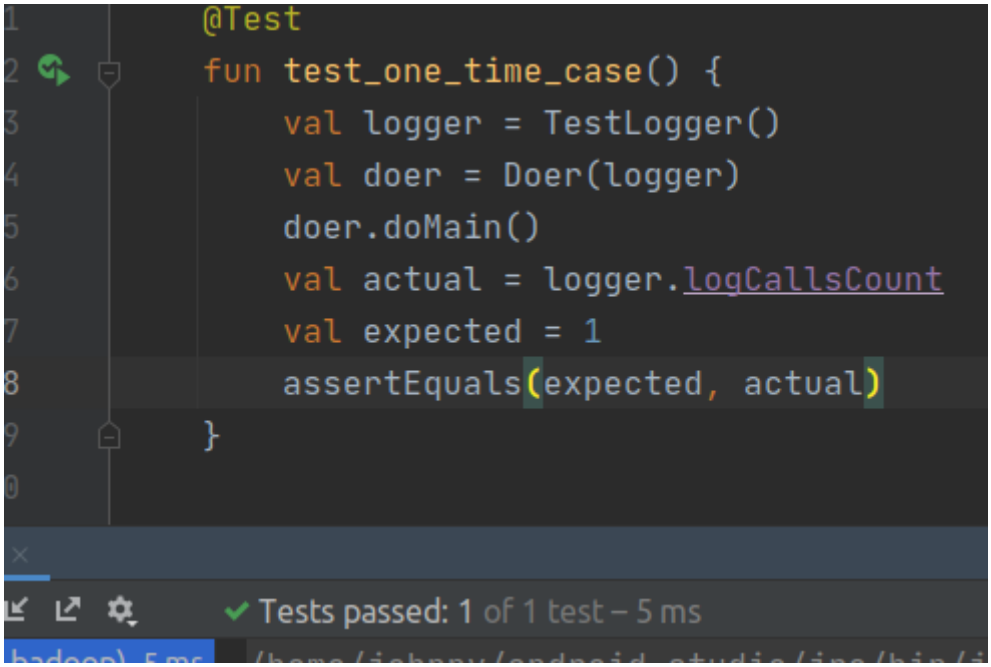
```
class DoerTest {

    @Test
    fun test_one_time_case() {
        val logger = TestLogger()
        val doer = Doer(logger)
    }

    private inner class TestLogger : Logging {
        var logCallsCount = 0

        override fun log(message: String) {
            logCallsCount++
        }
    }
}
```

Итак, мы написали тестовую реализацию логера, в нем мы просто будет считать сколько раз вызывался метод. Пишем тест теперь – создаем логер и даем его нашему классу Doer.



```
    @Test
    fun test_one_time_case() {
        val logger = TestLogger()
        val doer = Doer(logger)
        doer.doMain()
        val actual = logger.logCallsCount
        val expected = 1
        assertEquals(expected, actual)
    }
```

Tests passed: 1 of 1 test – 5 ms

Вызываем у класс единственный метод и проверяем что у вложенного логера метод вызвался 1 раз. Как видите тест прошел.

Ладно, теперь напишем второй тест – что мы вызываем метод 2 раза. Тогда логер должен сработать лишь 1 раз. Верно? Давайте

```
    @Test
    fun test_two_times_case() {
        val logger = TestLogger()
        val doer = Doer(logger)
        doer.doMain()
        doer.doMain()
        val actual = logger.logCallsCount
        val expected = 1
        assertEquals(expected, actual)
    }
```

Еще раз – мейн метод у нас остался каким он и был. В нем проверка что метод вызывался ранее. Можете подебажить.

```
fun doMain() {
    if (!mainThingDone) {  mainThingDone: true
        logger.log("main thing done")
        mainThingDone = true
    }
```

Это второй раз мы входим в метод и переменная уже поменяла свое значение.

И давайте еще раз посмотрим на наши методы в тесте. Как видите мы 2 раза написали один и тот же код – создали логер и дали его классу. Явное нарушение DRY – Don’t Repeat Yourself. Я считаю что этот принцип не менее важен как и все из SOLID. Но нам нужно чтобы перед каждым тестом создавались новые инстансы. Можно просто вынести в поля класса

```
class DoerTest {

    private val logger = TestLogger()
    private val doer = Doer(logger)

    @Test
    fun test_one_time_case() {
        doer.doMain()
        val actual = logger.logCallsCount
        val expected = 1
        assertEquals(expected, actual)
    }
```

Но у нас могут быть ситуации, когда нам нужно перед тестом проинициализировать обьекты, а после тестов почистить их. Например если вы пишете тесты на базу данных. Во время теста положили данные и после теста нужно их удалить из базы. Для этого есть 2 аннотации в Junit

Здесь я просто занулил переменную, но она просто число. А если например у вас база данных, то в методе clear с аннотацией After можно закрыть ее или удалить тестовые данные.

Можете поставить брейкпойнты(бряки в простонародии) и посмотреть как идет код – Before, Test, After. Если вам интересно насчет других возможностей Junit просто погуглите какие еще аннотации есть.

Теперь можем вернуться к нашим классам из предыдущей лекции и покрыть тестами модель.

Для начала уберем ненужное логирование. Если помните то модель запускала таймер и каждую секунду увеличивала число и отдавала в колбек. Так же она сохраняла значение в кеш и брала его если таковое имелось, но не шла в кеш если текущее значение не было равно начальному. Значит нам нужно проверить весь функционал.

Итак, первый тест кейс – когда класс только создали и запустили. Мы должны будем увидеть в колбеке 0. Правильно? Нет. Мы должны получить значение из кеша. Значит мы должны написать тестовую имплементацию датасорса таким образом, чтобы в него можно было положить значение и так же взять его, но не записывая в файл как это делает SharedPreferences. Если помните то этот класс записывает в файл данные и читает их и для теста мы написали обертку чтобы в тестах как раз и не писать тестовый контекст. В конце лекции мы все же попробуем это сделать и вы увидете как это страшно.

```
class Model(private val dataSource: DataSource) {

    private var timer: Timer? = null
    private val timerTask
        get() = object : TimerTask() {
            override fun run() {
                count++
                callback?.updateText(count.toString())
            }
        }
    private var callback: TextCallback? = null
    private var count = -1

    fun start(textCallback: TextCallback) {
        callback = textCallback
        if (count < 0)
            count = dataSource.getInt(COUNTER_KEY)
        timer = Timer()
        timer?.scheduleAtFixedRate(timerTask, delay: 0, period: 1000)
    }

    fun stop() {
        dataSource.saveInt(COUNTER_KEY, count)
        timer?.cancel()
        timer = null
    }

    companion object {
        private const val COUNTER_KEY = "counterKey"
    }
}
```

Начнем с тестового колбека. Просто напишем класс и сохраним текст который придет в него

```
private class TestCallback : TextCallback {
    var text = ""
    override fun updateText(str: String) {
        text = str
    }
}
```

Не смущайтесь того, что у нас переменная открыта. Это тестовый класс и здесь можно упрощать все что можно и нельзя.

Также напишем наш тестдатасорс. Я не хочу создавать мапу для 1 числа и просто сделаю

```
private class TestDataSource : DataSource {
    private var int: Int = Int.MIN_VALUE
    override fun saveInt(key: String, value: Int) {
        int = value
    }
    override fun getInt(key: String) = int
}
```

Это конечно же немного неправильно, потому что мы игнорируем ключ, но вы можете написать мапу и класть в нее и брать из нее. У нас в классе кладется и берется по 1 ключу и потому в конкретном кейсе нет необходимости в мапе. Вот если бы клали минимум по 2 разным ключам, тогда бы да, нужна была бы мапа. И давайте наконец напишем наш тест.

```
@Test
fun test_start_with_saved_value() {
    val testDataSource = TestDataSource()
    val model = Model(testDataSource)
    val callback = TestCallback()
    testDataSource.saveInt("", 5)
    model.start(callback)
    val actual = callback.text
    val expected = "5"
    assertEquals(expected, actual)
}
```

Инитим тестовые обьекты, кладем в кеш 5, стартуем модель и проверяем что после вызова будет 5 строкой. И… тест не прошел! Почему же? А может потому что мы в классе создаем таймер? И запускаем его пусть и без задержки (первый аргумент delay). Наш тестовый код переходит от одной линии ко второй намного быстрей чем успевает проинициализироваться таймер. Как решить эту проблему? Ну вы уже знаете. Мы не должны вызывать конструкторы внутри класса в идеальном случае. Но давайте сначала я вам покажу плохое решение. Не смущайтесь кстати этой фразы - во-первых вам нужно решение, а плохое оно или хорошее уже вопрос другой. Чтобы наш таймер успел проницализироваться просто добавим задержку в потоке тестового метода. Если помните у нас есть класс Thread и у него есть метод sleep где можно поставить на паузу выполнение дальнейшего кода. Нам достаточно 10 миллисекунд как видите и тест прошел!

```
Thread.sleep( millis: 10)
val actual = callback.text
val expected = "5"
assertEquals(expected, actual)
```

Ладно, чуть позже рассмотрим как обойти это. А пока давайте протестируем следующий кейс. Нам нужно запустить таймер и например спустя 2 секудны остановить и проверить что записалось значение. Напишем код и там где нужна будет задержка в 2 секунды вы знаете как

```
@Test
fun test_stop_after_2_seconds() {
    val testDataSource = TestDataSource()
    val model = Model(testDataSource)
    val callback = TestCallback()
    testDataSource.saveInt("", 0)
    model.start(callback)
    Thread.sleep( millis: 2010)
    val actual = callback.text
    val expected = "2"
    assertEquals(expected, actual)
}
```

В кеш я положу ноль, стартану таймер, через 2 секунды (плюс 10 миллисекун для инициализации) я увижу там 2. Но мы не проверили что в кеш записали значение! Добавим кода в этот же метод

```
model.stop()
val savedCountActual = testDataSource.getInt( key: "")
val savedCountExpected = 2
assertEquals(savedCountExpected, savedCountActual)
```

И здесь нам нужно остановится. У нас проблема в том, что мы дергаем колбек перед или после того как инкрементим. Так же у нас таймер срабатывает сразу. Итого у нас 2 проблемы. Первая если мы в колбек пишем значение и потом инкрементим количество то наши тесты не проходят по логике, потому что видим мы одно, а счет уже увеличился. А если мы в таймер пишем Delay 1000 то тесты вовсе не проходят. Почему? А вот черт его знает (нам и не нужно узнавать!). За 10 миллисекунд успевал инициализироваться таймер который без задержки. А с задержкой и вовсе не работает, просто потому что нужно увеличить ее. Кароче. Слишком много сложностей. Давайте уже перепишем код!

Запомните, это все костыли – когда вы пишете задержки потому что не успела проинициализироваться переменная. Вы должны понимать что делаете явную фигню.

Еще раз: исходя из принципов SOLID L & D мы не должны порождать обьекты внутри классов а получать их в конструкторе и не их самих, а интерфейсы чтобы можно было подменить. Давайте напишем некий интерфейс и класс в котором будет таймер использоваться.

```
interface TimeTicker {

    fun start(callback: Callback, period: Long = 1000)

    fun stop()

    interface Callback {

        fun tick()
    }
}
```

2 простых метода – старт и стоп. И простой колбек который просто уведомит о том, что прошло нужное количество миллисекунд. Чем проще ваши методы и классы тем лучше.

Теперь напишем реализацию с таймером

```
class TimerTicker : TimeTicker {

    private var callback: TimeTicker.Callback? = null
    private var timer: Timer? = null
    private val timerTask
        get() = object : TimerTask() {
            override fun run() {
                callback?.tick()
            }
        }

    override fun start(callback: TimeTicker.Callback, period: Long) {
        this.callback = callback
        timer = Timer()
        timer?.scheduleAtFixedRate(timerTask, delay: 0, period)
    }

    override fun stop() {
        callback = null
        timer?.cancel()
        timer = null
    }
}
```

И здесь уже отдельно ото всего мира можно создавать таймер и стартовать его и стопить.

Мы не привязаны ни к чему и можем переиспользовать его и заменять реализации.

Отредактируем теперь нашу модель (п.с. занулите колбек, я забыл в методе stop)

```
class Model(
    private val dataSource: DataSource,
    private val timeTicker: TimeTicker
) {

    private val tickerCallback
        get() = object : TimeTicker.Callback {
            override fun tick() {
                count++
                callback?.updateText(count.toString())
            }
        }
    private var callback: TextCallback? = null
    private var count = -1

    fun start(textCallback: TextCallback) {
        callback = textCallback
        if (count < 0)
            count = dataSource.getInt(COUNTER_KEY)
        timeTicker.start(tickerCallback)
    }

    fun stop() {
        dataSource.saveInt(COUNTER_KEY, count)
        timeTicker.stop()
    }

    companion object {
        private const val COUNTER_KEY = "counterKey"
    }
}
```

Теперь мы не создаем ничего внутри. Все на откуп реализации. Мы работаем лишь с

интерфейсом. Пусть вас не смущает что у нас 2 колбека. Один из таймера получает событие, второе это отдаем вьюмодели или кому нужно значение уже в строковом виде.

Для начала давайте отредактируем наш код в классе Application. Ведь у нас новая зависимость в конструктор модели

```
override fun onCreate() {
    super.onCreate()
    viewModel = ViewModel(Model(CacheDataSource( context: this), TimerTicker()))
}
```

Ну и давайте запустим проект и проверим что все работает как и прежде.

Да, все работает. А теперь вернемся к тестам и отредактируем их.

```
private class TestTimeTicker : TimeTicker {

    private var callback: TimeTicker.Callback? = null

    var state = 0

    override fun start(callback: TimeTicker.Callback, period: Long) {
        this.callback = callback
        state = 1
    }

    override fun stop() {
        callback = null
        state = -1
    }

    fun tick(times: Int) {
        for (i in 0 until times)
            callback?.tick()
    }
}
```

Мы написали тестовый тикер, в который добавили метод для ручного вызова метода у колбека. Теперь нам не нужно ждать 2 секунды или 2000000 секунд чтобы проверить функционал. Теперь мы можем имитировать работу таймера легко и просто. Давайте посмотрим теперь на сами методы. В первом мы стартовали с сохраненным значением. Уберем оттуда Thread.sleep и вообще запомните – если вы написали это, то у вас проблемы.

Итак, предположим ранее у нас прошло 5 секунд и было сохранено число 5. Теперь мы создали модель и стартуем – но мы не можем сразу проверить перменную count ибо она приватная. Потому нам нужно сделать это через таймер. Мы имитируем что прошла 1 секунда и проверяем что имеем 6 строкой.

```kotlin
@Test
fun test_start_with_saved_value() {
    val testDataSource = TestDataSource()
    val timeTicker = TestTimeTicker()
    val model = Model(testDataSource, timeTicker)
    val callback = TestCallback()
    testDataSource.saveInt("", 5)
    model.start(callback)
    timeTicker.tick( times: 1)
    val actual = callback.text
    val expected = "6"
    assertEquals(expected, actual)
}
```

Как видите никаких пауз потока не нужно. Давайте теперь отредактируем второй тест.

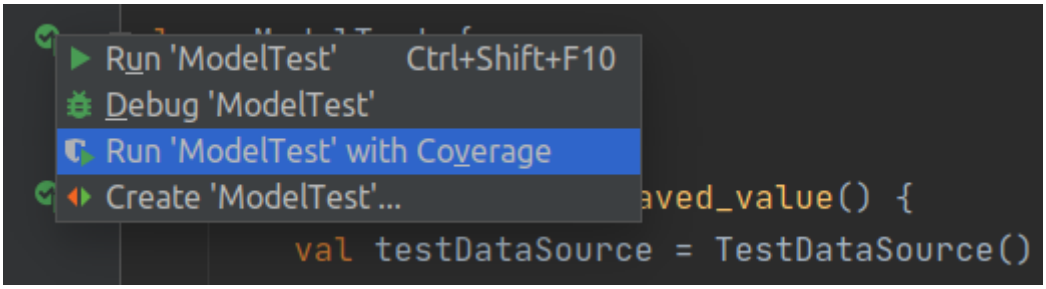
```kotlin
@Test
fun test_stop_after_2_seconds() {
    val testDataSource = TestDataSource()
    val timeTicker = TestTimeTicker()
    val model = Model(testDataSource, timeTicker)
    val callback = TestCallback()
    testDataSource.saveInt("", 0)
    model.start(callback)
    timeTicker.tick( times: 2)
    val actual = callback.text
    val expected = "2"
    assertEquals(expected, actual)

    model.stop()
    val savedCountActual = testDataSource.getInt( key: "")
    val savedCountExpected = 2
    assertEquals(savedCountExpected, savedCountActual)
}
```

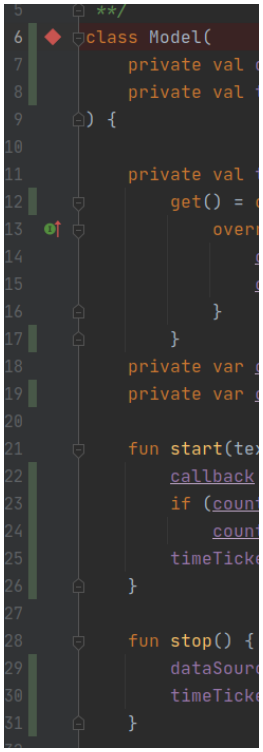
Здесь же мы с помощью одного метода имитируем прохождение 2 секунд и проверяем что все правильно работает. Также вызываем стоп и проверяем что сохранилось значение в кешдатасорс. И самое удивительное – мы не запускаем никаких реальных таймеров и не

ждем нисколько секунд. Так же не пишем ни в какие файлы значения. Но все равно мы уверены в нашем функционале. Мы точно знаем что все работает на тестовых реализациях и уверены что на базовых реализациях тоже будет работать так же. Ведь в этом и суть принципа подстановки Liskov – все работает вне зависимости от их реализаций.

И на последок давайте я вам расскажу про покрытие кода тестами. Часто на проектах есть правило – должно быть покрытие минимум 80 процентов кода. Ок, как посчитать? У нас есть слева от класса возможность запуска тестов с покрытием.

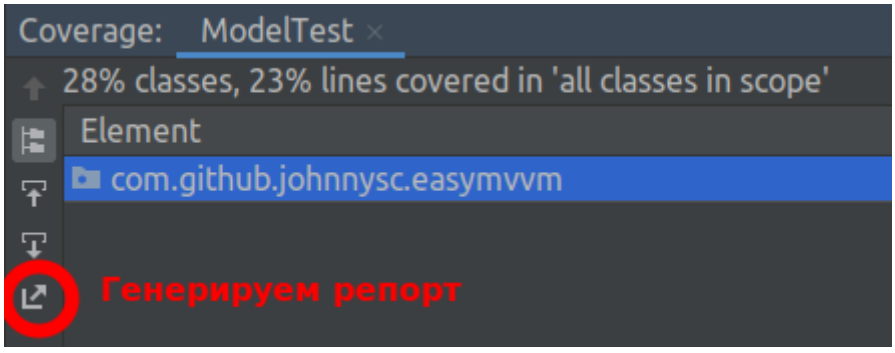


Давайте запустим тесты с подсчетом покрытия.



Как видите слева от кода класса мы видим зеленые полоски. Это значит что когда мы запускали тестовые методы то он проходил по всем этим линиям. Если бы например не вызывали метод Stop то и там не было бы зеленых полос. А были бы красные. Можете удалить второй тест и проверить это. А еще справа у нас высвечивается статистика покрытия.

(п.с. здесь все нормально, ничего не поплыло)



Но здесь неинформативно. Нажмите на кнопку генерации репорта и увидете следующее

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	80% (4/ 5)	80% (8/ 10)	90.9% (20/ 22)

Я если честно не понимаю до конца как AC считает процент покрытия, но похоже на правду, хотя по сути должно быть 100 процетов. Но вам не нужно заботиться об этом. На крупных проектах есть нормальный инструмент подсчета покрытия кода тестами. А пока просто смотрите слева от кода есть ли зеленая полоса или нет. Ну и нужно думать своей головой – все ли кейсы вы проверили. Мы вроде все проверили – 1. было значение в кеше и оттуда считали, положили в кеш и проверили спустя 2 секудны что все правильно работает.

Стоп! Нет же. Мы не проверили всю цепочку. Мы же стартуем, потом стоп и потом продолжаем. Нам нужен полноценный тест кейс на весь сценарий

```
@Test
fun test_start_after_stop() {
    val testDataSource = TestDataSource()
    val timeTicker = TestTimeTicker()
    val model = Model(testDataSource, timeTicker)
    val callback = TestCallback()
    testDataSource.saveInt("", 10)
    model.start(callback)
    timeTicker.tick( times: 2)
    val actual = callback.text
    val expected = "12"
    assertEquals(expected, actual)

    model.stop()
    val savedCountActual = testDataSource.getInt( key: "")
    val savedCountExpected = 12
    assertEquals(savedCountExpected, savedCountActual)

    model.start(callback)
    timeTicker.tick( times: 3)
    val actualText = callback.text
    val expectedText = "15"
    assertEquals(expectedText, actualText)
}
```

И мы не закончили. Самое главное – когда приложение умирает. Но как нам это протестировать? Что значит смерть приложения? Что все инстансы обнулились и при повторном запуске мы просто возьмем лишь сохраненное значение. В нашем случае можем воспользоваться переменной и заново создать модель с новыми обьектами. Но не забыть проставить значение сохраненное.

Оставляю вам это задание. Просто создайте новые инстансы типа val newModel, newDataSource, newTicker но не забудьте перед смертью занулить старые ссылки и после создания датасорса засетить значение из переменной.
Val savedNumber = dataSource.getInt(“”)
dataSource = null

val newDataSource = TestDataSource().apply { saveInt(“”, savedNumber) }

И как обещал в конце : что если бы мы не написали наш класс ДатаСорс и просто передавали контекст, а не его в нашу модель? Тогда бы нам нужна была тестовая реализация контекста.

```
128    private class TestContext : Context() {...}
640    //end
```

Я схлопнул чтобы вы понимали масштаб трагедии – более 500 линий кода. Там внутри все методы без реализаций

```
128  private class TestContext : Context() {
129      override fun getAssets(): AssetManager {
130          TODO( reason: "Not yet implemented")
131      }
132
133      override fun getResources(): Resources {
134          TODO( reason: "Not yet implemented")
135      }
```

Просто напишите класс который наследуется от контекста и вы увидете все методы, которые нужно написать чтобы использовать тестовую реализацию класса.

Именно поэтому очень не рекомендуется писать классы, которые принимают в конструктор контекст – это монструозный обьект! И я уже вроде как говорил что это – God Object андроид фреймворка который дает доступ ко всему что связано с приложением – к ресурсам, к пакетам типа assets (туда можно класть файлики всякие). И самое интересное что даже если вы решите написать тестовый контекст, то у вас будет цепочка проблем – начнем с ресурсов

```
 */
public class Resources {
    /**
```

Да, это класс без интерфейса и даже не наследуется от абстрактного класса – написать тестовые ресурсы невозможно. Занавес!

Даже первый метод контекста посмотрите

```
 */
public final class AssetManager implements AutoCloseable {
    private static final String TAG = "AssetManager";
```

Финальный класс! И так далее.

Просто запомните – не имейте дел с классом Context в своих тестах! Пишите обертки.