

Отладка. Юнит-тестирование

Нечасто в работе программиста случается так, что написанная программа сразу начинает работать так как задумывалось. В остальных случаях приходится находить и исправлять допущенные ошибки.

Ошибки бывают *синтаксические* – например, опечатки в названии метода или пропущенная скобка – такие ошибки обнаруживает компилятор и с помощью среды разработки сигнализирует об этом:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val etNumber = findViewById<EditText>(R.id.numbe)
    etNumber.~
}
```

Такие ошибки исправляются просто: нужно найти место ошибки, которое показывает среда разработки, понять суть проблемы и внести необходимые изменения.

Гораздо сложнее исправлять *логические* ошибки в программе, ведь она компилируется и даже запускается (хоть и не всегда), но работает не так как планировалось. И тогда разработчик приложения приступает к отладке (англ. debug, debugging, букв. *обезжучивание*, см. знаменитую историю про мотылька, который нарушил работу одного из первых компьютеров, и дал английское название процессу отладки программ).

Логирование

Если есть подозрение что в каком-то участке программы происходит что-то не то – например, значения переменных не соответствуют ожиданиям, то можно не останавливая выполнение программы вывести значения этих переменных, или любой другой поясняющий текст, в лог. Для этого используется класс `Log`, у которого есть ряд методов для вывода информации:

```
val count = 5
Log.d("Lab11", "Значение count: $count")
Log.e("Lab11", "Значение count: $count")
Log.i("Lab11", "Значение count: $count")
Log.v("Lab11", "Значение count: $count")
Log.w("Lab11", "Значение count: $count")
Log.wtf("Lab11", "Значение count: $count")
```

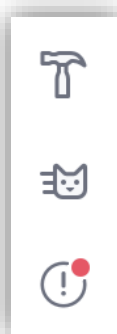
Могут использоваться следующие методы:

- `d` – отладочная информация (уровень `DEBUG`)
- `e` – информация об ошибке (уровень `ERROR`)
- `i` – просто информация (уровень `INFO`)
- `v` – подробная информация (уровень `VERBOSE`)

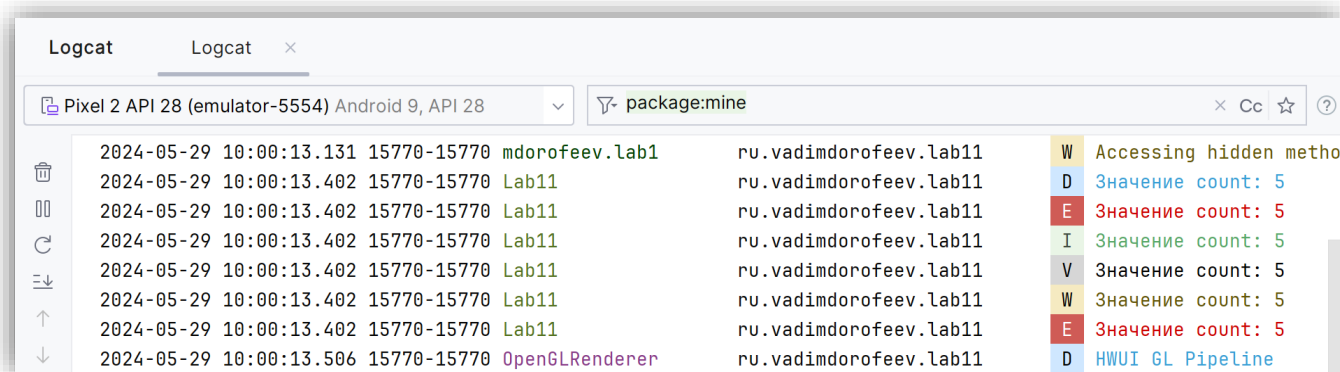
- **w** – предупреждение (уровень **WARN**)
- **wtf** (англ. What a Terrible Failure) – ошибка, которая по логике вообще не должна была произойти, в зависимости от конфигурации системы может даже прервать работу программы

Уровни означают важность информации, они идут в порядке убывания: **ERROR** → **WARN** → **INFO** → **DEBUG** → **VERBOSE**. В большинстве случаев на уровень важности можно не обращать внимания, просто использовать подходящий по логике метод.

Для просмотра логов в Android Studio предусмотрен инструмент Logcat. Его можно активировать через меню (View → Tool Windows → Logcat) или щёлкнуть значок Logcat в виде кошачьей мордочки на левой панели Android Studio:



Панель Logcat содержит не только записи, которые выводит программа, но и разнообразные системные сообщения, поэтому для поиска своих записей используются теги, это первый аргумент методов класса **Log**. В примере выше используется тег **Lab11**, но лучше использовать что-то ещё более уникальное. Выглядит лог следующим образом:



В верхней левой части панели в выпадающем списке можно выбрать устройство, на котором запущена программа. Обычно оно уже выбрано, но иногда приходится переключать его вручную.

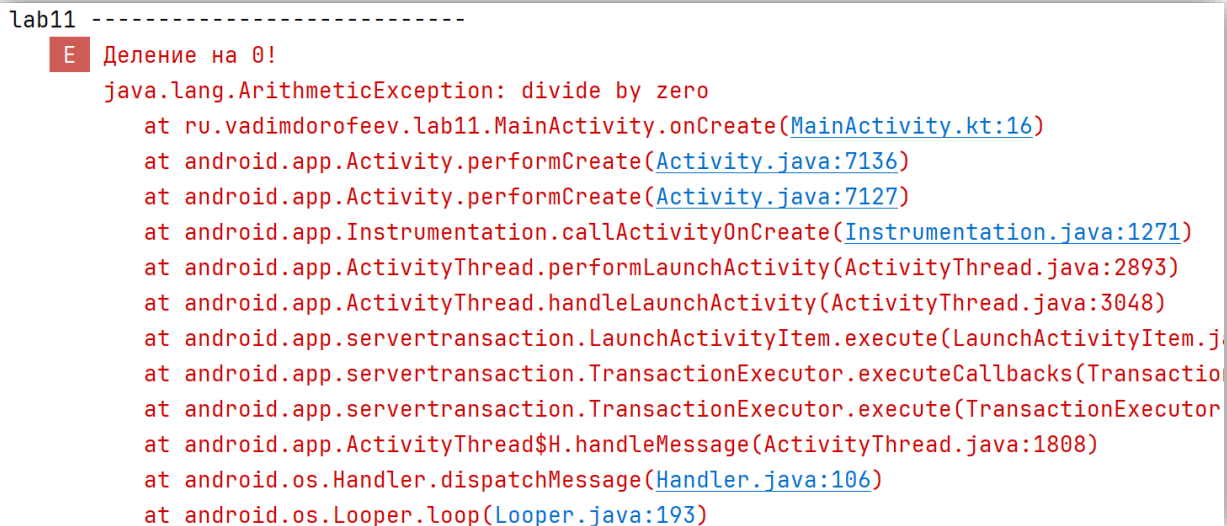
В правой верхней части панели находится фильтр. С его помощью можно оставить в списке только те записи, которые интересуют в данный момент. Например, можно набрать там слова «значение» – и останутся только те строки, которые выводила программа в примере выше.

В самом логе выводятся строки, которые были отправлены с помощью класса [Log](#). В зависимости от использованного метода ([d](#), [i](#), [e](#), ...) слева выводится цветная буква, чтобы было проще искать строки.

Помимо текстового сообщения и значения переменных в лог можно выводить и информацию об ошибке, если её удалось перехватить. В следующем примере происходит деление на 0, но падения программы не происходит, потому что используется блок перехвата исключений [try-catch](#). Информация об ошибке записывается в переменную [ex](#), которая затем выводится в лог:

```
try {  
    val c = 1 / 0  
}  
catch (ex: Exception) {  
    Log.e("Lab11", "Деление на 0!", ex)  
}
```

В логе появится само сообщение, а затем большой список вызовов методов:



The screenshot shows a log entry for 'lab11' with a red 'E' icon indicating an error. The message is 'Деление на 0!'. Below the message is a stack trace in red text, listing the sequence of method calls that led to the exception. The top of the stack trace is 'java.lang.ArithmeticException: divide by zero' at 'ru.vadimdorofeev.lab11.MainActivity.onCreate(MainActivity.kt:16)'. Other methods in the stack include 'android.app.Activity.performCreate', 'android.app.Instrumentation.callActivityOnCreate', 'android.app.ActivityThread.performLaunchActivity', 'android.app.ActivityThread.handleLaunchActivity', 'android.app.servertransaction.LaunchActivityItem.execute', 'android.app.servertransaction.TransactionExecutor.executeCallbacks', 'android.app.servertransaction.TransactionExecutor.execute', 'android.app.ActivityThread\$H.handleMessage', 'android.os.Handler.dispatchMessage', and 'android.os.Looper.loop'.

```
lab11 -----  
E Деление на 0!  
java.lang.ArithmeticException: divide by zero  
    at ru.vadimdorofeev.lab11.MainActivity.onCreate(MainActivity.kt:16)  
    at android.app.Activity.performCreate(Activity.java:7136)  
    at android.app.Activity.performCreate(Activity.java:7127)  
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)  
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2893)  
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3048)  
    at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.j  
    at android.app.servertransaction.TransactionExecutor.executeCallbacks(Transactio  
    at android.app.servertransaction.TransactionExecutor.execute(TransactionExecuto  
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1808)  
    at android.os.Handler.dispatchMessage(Handler.java:106)  
    at android.os.Looper.loop(Looper.java:193)
```

Чтобы понять где именно случилась ошибка, нужно найти файл, который входит в проект – в данном случае это файл [MainActivity.kt](#), после двоеточия указана строка, где произошла ошибка - 16. Можно просто нажать мышкой на синий текст [MainActivity.kt:16](#) и среда разработки откроет нужный файл и строку.

Точки останова

Вывод информации в лог не всегда помогает понять проблему, и обычно требует много времени и усилий – дополнить код нужными строками, разбираться что именно вывелось в лог и т. д. Вместо этого можно просто остановить программу на нужной строке и посмотреть значения переменных на месте, или пошагово выполнить фрагмент программы. Для этого в нужной строке нужно либо щёлкнуть на левой полоске, либо выбрать в меню [File](#) → [Run](#) → [Toggle Breakpoint](#) → [Line Breakpoint](#). На левой полоске появится красная точка, она означает что программа остановится в этом месте:

```

15      val a = 16
16      val b = 4
17      val c = a - b * 4
18      Log.i( tag: "Lab11", msg: "Значение: $c")
19  }

```

Однако, если запустить программу обычным образом, то точка останова (англ. Breakpoint) не сработает. Для отладки программу нужно запускать либо через меню Run → Debug 'app', или нажатием кнопки с жуком на панели запуска в верхней части Android Studio:



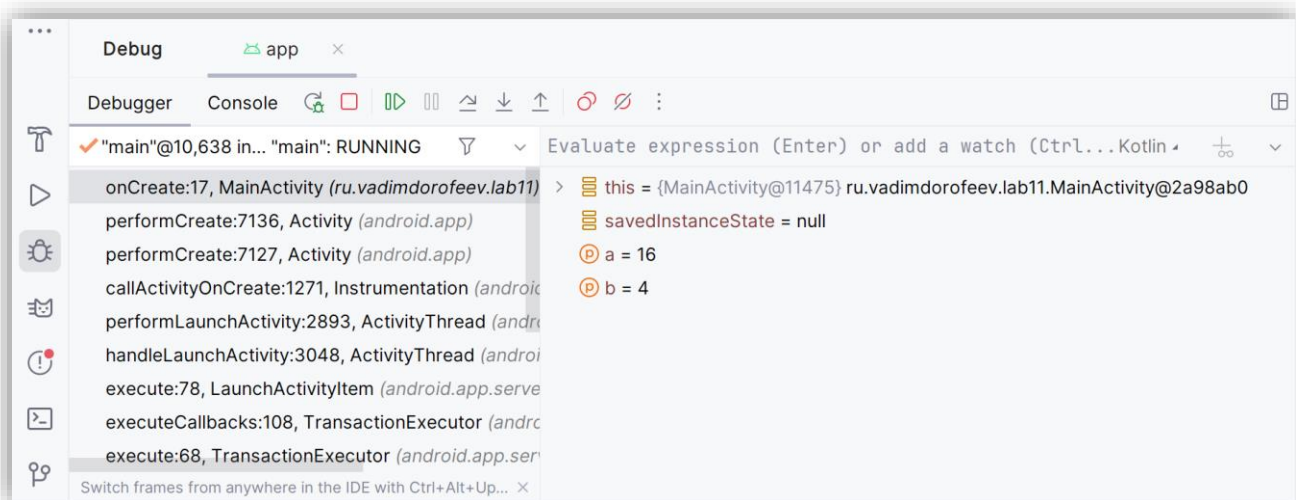
Когда программа прервётся на точке останова, Android Studio отобразит справа значения переменных, которые определены в этой области:

```

15      val a = 16    a: 16
16      val b = 4    b: 4
17      val c = a - b * 4    a: 16    b: 4
18      Log.i( tag: "Lab11", msg: "Значение: $c")

```

В окне Debug в нижней части Android Studio можно посмотреть стек вызовов и полный список переменных:



Если слева от переменной стоит значок > (как у переменной `this` на скриншоте), то переменная представляет собой объект, и нажатием на значок можно развернуть его и посмотреть свойства.

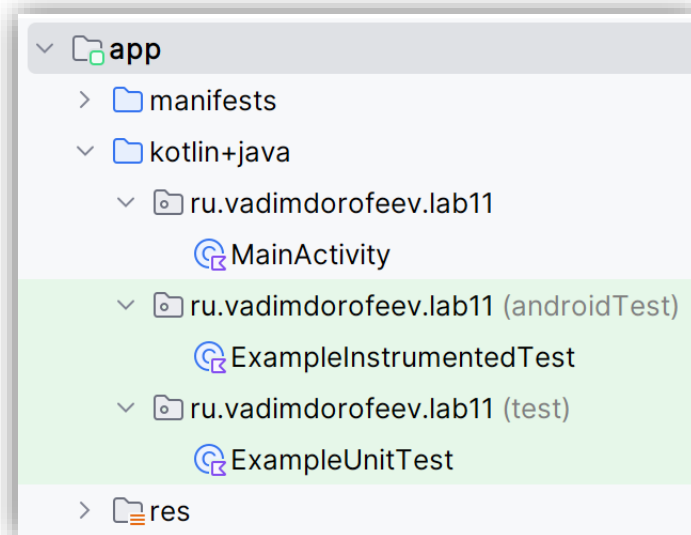
Используя кнопки на панели Debug (или пункты меню Run → Debugging Actions, или горячие клавиши, что ещё удобнее) можно проходить программу по шагам и смотреть как изменяются значения переменных:

- Step Over – выполнить выделенную строку; если в строке есть вызов метода, то он выполнится целиком
- Step Into – выполнить выделенную строку; если в строке есть вызов метода, то отладчик перейдёт в этот метод и его тоже можно будет выполнять по шагам
- Step Out – отладчик выполнит текущий метод до конца, перейдёт к родительскому методу, и остановится строке, которая следует после вызова метода
- Resume Program – выполнение программы будет продолжено обычным образом, не по шагам

Юнит-тестирование

Android Studio поддерживает не только обычную отладку, но и автоматизированное тестирование. Это может быть полезно в тех случаях, когда код достаточно сложный, и в него планируется вносить изменения, которые могут его сломать и привести к неверной работе. Чтобы этого избежать, пишется ряд тестов, которые вызывают код с разными параметрами, и проверяют правильность результата. Тесты стараются писать таким образом, чтобы они максимально покрывали всю область допустимых значений параметров, т. е. не только обычные, но и предельные, а иногда и заведомо неверные, чтобы убедиться, что код работает правильно и в этих ситуациях.

Тесты в Android Studio не смешиваются с обычным кодом, для них выделены отдельные ветки в дереве проектов, они имеют такое же название, как и основная ветка, но справа помечены спецификаторами (`androidTest`) и (`test`):



Ветка (`test`) служит для тестирования логики программы, тесты в этой ветке не используют фреймворк Android и могут запускаться прямо в Android Studio. Ветка (`androidTest`) используется для тестирования той части логики, которую невозможно проверить локально – например, интерфейс программы, эти тесты запускаются в эмуляторе.

Тестирование логики

Рассмотрим тестирование логики на примере. Пусть есть класс `Geometry`, который содержит геометрические методы, в частности, метод `distance` для определения расстояния между точками на плоскости по их координатам:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Класс выглядит следующим образом:

```
class Geometry {
    fun distance(x1: Float, y1: Float, x2: Float, y2: Float): Float {
        return sqrt((x2 - x1).pow(2) + (y2 - y1).pow(2))
    }

    fun isWithinRange(x: Float, y: Float, radius: Float, xt: Float, yt: Float): Boolean {
        return distance(x, y, xt, yt) <= radius
    }
}
```

Создадим в ветке `(test)` класс `GeometryUnitTest`:

```
class GeometryUnitTest {
    private var g: Geometry? = null

    @Before
    fun init() {
        g = Geometry()
    }

    @Test
    fun distance_isCorrect() {
        assertEquals(1f, g?.distance(1f, 1f, 2f, 1f))
        assertEquals(1f, g?.distance(1f, 1f, 1f, 2f))
    }

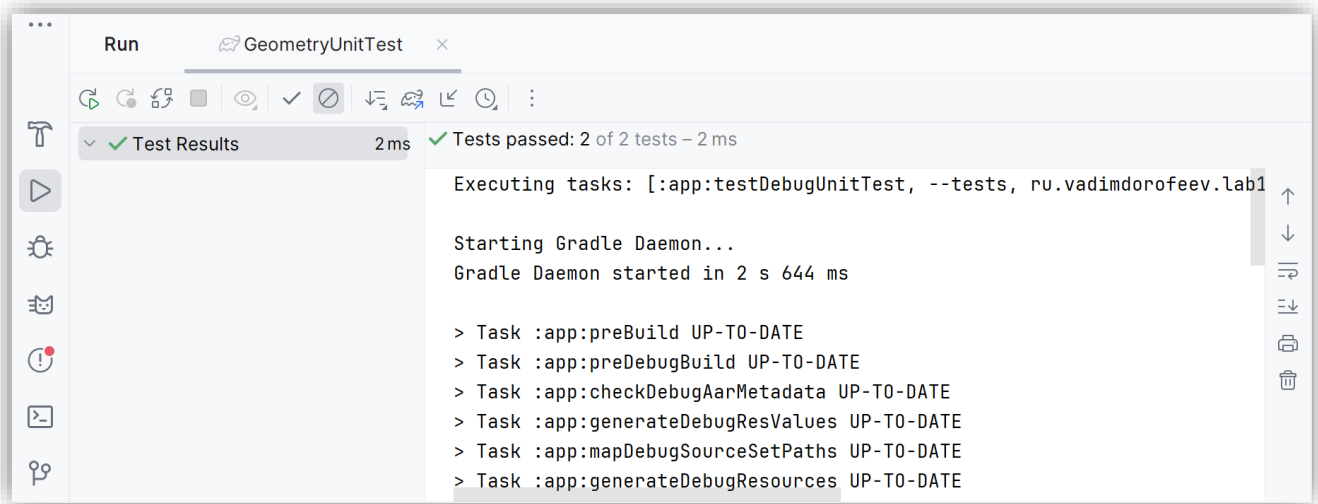
    @Test
    fun isWithinRange_isCorrect() {
        assertTrue(g?.isWithinRange(0f, 0f, 5f, 1f, 1f) == true)
        assertFalse(g?.isWithinRange(0f, 0f, 5f, 8f, 7f) == true)
    }

    @After
    fun finalize() {
        g = null
    }
}
```

Все тесты должны содержать аннотацию `@Test`, она показывает, что этот метод должен быть запущен в процессе тестирования. Кроме аннотации `@Test` предусмотрены также аннотация `@Before` – такой метод будет запускаться перед каждым тестом, в нём можно

инициализировать новые объекты или привести данные в исходное состояние, и `@After` – он, соответственно, будет запускаться после каждого теста.

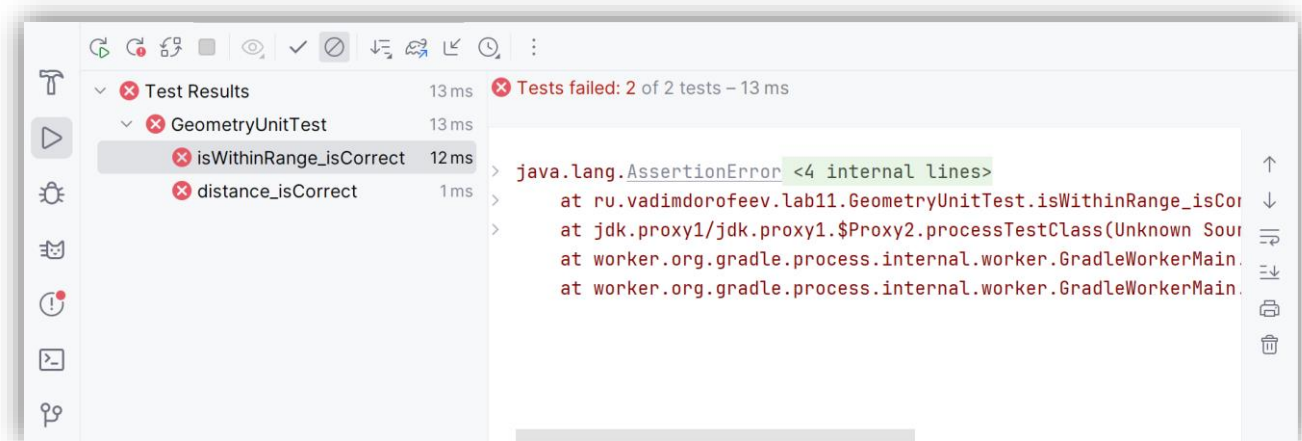
Чтобы запустить тестирование, нужно в дереве проекта в ветке `(test)` найти файл `GeometryUnitTest.kt`, щёлкнуть по нему правой кнопкой мыши и выбрать пункт `Run 'GeometryUnitTest'`. Несколько секунд будет проводиться компиляция и запуск тестов, после чего в панели `Run` будут выведены результаты:



Все тесты успешно пройдены. Попробуем внести изменение в метод `distance` в классе `Geometry`, намеренно испортив вычисления, заменим знак `+` на знак `-`:

```
fun distance(x1: Float, y1: Float, x2: Float, y2: Float): Float {
    return sqrt((x2 - x1).pow(2) - (y2 - y1).pow(2))
}
```

Запустим тесты ещё раз, теперь результат не такой радостный:



Если открыть код с тестами, то методы класса, которые вызвали ошибки, будут отмечены слева красными значками, а конкретные `assert`-методы внутри них будут подчёркнуты:


```

17      @Test
18      fun distance_isCorrect() {
19          assertEquals( expected: 1f, q?.distance( x1:
20          assertEquals( expected: 1f, q?.distance( x1:
21      }
22
23      @Test
24      fun isWithinRange_isCorrect() {
25          assertTrue(q?.isWithinRange( x: 0f, y: 0f,
26          assertFalse(q?.isWithinRange( x: 0f, y: 0f,
27      }

```

Таким образом можно регулярно проверять правильность кода, периодически запуская тесты и убеждаясь в правильности логики программы.

Для проверки правильности логики можно использовать следующие методы:

- `assertEquals` – сравнивает два значения, они должны быть равны; используется сравнение с помощью метода `equals` у сравниваемых объектов
- `assertNotEquals` – сравнивает два значения, они должны быть **не** равны; используется сравнение с помощью метода `equals` у сравниваемых объектов
- `assertSame` – сравнивает два значения, они должны быть равны; используется простое сравнение `==`
- `assertNotSame` – сравнивает два значения, они должны быть **не** равны; используется простое сравнение `==`
- `assertArrayEquals` – сравнивает два массива, они должны быть равны
- `assertTrue` – параметр должен иметь значение `true`
- `assertFalse` – параметр должен иметь значение `false`
- `assertNull` – параметр должен иметь значение `null`
- `assertNotNull` – параметр должен иметь значение **не** `null`

Методы `assertEquals` и `assertSame` похожи, и может возникнуть сомнение: когда какой метод нужно использовать. Для простых значений (целые числа, строки и т. д.) оба метода работают одинаково, можно использовать любой. А вот если сравниваются объекты, то разница уже есть: метод `assertSame` проверит что обе ссылки ссылаются на один и тот же объект, если объекты разные (пусть даже у них полностью одинаковые свойства), то равенства не будет. Метод же `assertEquals` для сравнения вызывает метод `equals` у одного из объектов. Если в объекте переопределён этот метод, то объект может сравнить свои свойства со свойствами другого объекта и решить, одинаковое ли наполнение у обоих объектов или нет.

Задание

Выберите какую-либо область, в которой можно делать расчёты: геометрия, физика, химия, экономика, ...

Разработайте класс, содержащий несколько методов для каких-либо вычислений в выбранной области.

Разработайте тесты для проверки правильности работы каждого из этих методов, на обычных значениях, и на граничных значениях.

Запустите тесты, убедитесь, что всё считается правильно. Зафиксируйте это в отчёте скриншотами и описанием.

Допустите намеренную ошибку в одном или нескольких методах исходного класса. Запустите тесты и убедитесь, что все неверные вычисления приводят к падению соответствующих тестов. Зафиксируйте это в отчёте скриншотами и описанием какие именно тесты и `assert`-методы вызвали ошибки и почему.