2.10. Отладка и тестирование приложений

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 2.10. Отладка и тестирование приложений

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:47

Оглавление

2.10.1 Введение в отладку и тестирование

Программы всегда содержали ошибки, содержат и будут их содержать. Как правило, последняя найденная ошибка на самом деле является предпоследней. Независимо от обстоятельств код, создаваемый разработчиками программного обеспечения, далеко не всегда работает так, как задумано. В некоторых случаях все идет совершенно не по плану. В подобных ситуациях необходимо выяснить, почему так происходит. При этом вместо многочасового изучения кода в поисках ошибок гораздо проще и эффективнее будет использовать средства тестирования и отладки.

Отладка — это процесс определения и устранения причин ошибок. Чтобы понять, где возникла ошибка, приходится: узнавать текущие значения переменных; выяснять, по какому пути выполнялась программа. Отладка включает поиск дефекта и его исправление, причем поиск дефекта и его понимание обычно составляют 90% работы. В некоторых проектах процесс отладки занимает до 50% общего времени разработки. Существуют две взаимодополняющие технологии отладки.

1) Отладочный вывод и логирование (от англ. log — журнал событий, протокол). Текущее состояние программы логируется с помощью расположенных в критических точках программы операторов вывода.

2) Использование отладчиков. Отладчики позволяют пошагового выполнять программы оператор за оператором и отслеживать состояние переменных.

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

Тестирование - это проверка того, что программа или код работают правильно и надежно в различных условиях: вы «тестируете» свой код, предоставляя входные данные, стандартные правильные входные данные, преднамеренно неправильные входные данные, граничные значения, изменяющуюся среду, По сути, мы можем сказать, что вы пытаетесь обнаружить ошибки и в конечном итоге «отладить» их в процессе тестирования.

Говорят, что «ошибка» произошла, когда ваша программа при исполнении не ведет себя так, как должна. То есть он не дает ожидаемого результата или результатов. Любая попытка найти источник этой ошибки, найти способы исправить поведение и внести изменения в код или конфигурацию, чтобы исправить проблему, может быть названа отладкой. Тестирование - это то, где вы проверяете, что программа или код работают правильно и надежно в различных условиях: вы «тестируете» свой код, предоставляя входные данные, стандартные правильные входные данные, преднамеренно неправильные входные данные, граничные значения, изменяющуюся среду, По сути, мы можем сказать, что вы пытаетесь обнаружить ошибки и в конечном итоге «отладить» их в процессе тестирования.

Помимо основной папки с классами структуре проектов, созданных с помощью Android Studio, вы наверняка замечали папки androidTest и test. В небольших проектах разработчик в состоянии контролировать логику приложения, предсказывать слабые места приложения и т.д. По мере роста сложности проекта обходится без инструментов тестирования становиться довольно проблематично.

Приведем ряд преимуществ тестирования программного обеспечения:

- *Экономия ресурсов*. Без грамотного подхода к тестированию количество ресурсов, необходимых для поддержания проекта в долгосрочной перспективе значительно больше, чем затраты на него.
- *Безопасность*. При командной работе обеспечивается безопасность кода: разные разработчики с течением времени меняют один и тот же фрагмент кода, при этом наличие тестов делает это более безопасным, так как никто не сможет что-то «испортить», не узнав об этом.
- *Улучшенная архитектура*. При тестировании приложений их создают обычно с использованием шаблонов проектирования, чтобы сделать приложения максимально простым и тестируемым.
- *Качество кода*. Приложение в меньшей степени подвержено сбоям в работе так как тесты помогают писать более надёжный код.
- *Улучшенный рефакторинг*. Наличие хороших тестов позволяет модифицировать определённый код, проверяя, что тесты все ещё успешно проходят.

Приведем пример как помогло бы тестирование в случае проверки метода сложения двух чисел. Для метода заводится тест на сложение двух контрольных чисел, пусть будет 5 и 4. Когда разработчик внесет правки в метод (например, перепутает знак сложения со знаком вычитания), то запускается тестирование метода. Теперь если вместо 9 (5 + 4) метод выдаст 1 (5 - 4), то тест не провален и можно будет искать проблему в нужном месте. Так и работают тесты - проверка на соответствие к ожидаемому результату.

По одной из классификаций тесты делятся на две категории - локальные (Unit Testing) и инструментальные (UI Testing). В данном разделе подробнее поговорим про отладку приложений и Unit-тестирование в Android.

2.10.2 Отладка в Android приложениях

Отладочный вывод и логирование

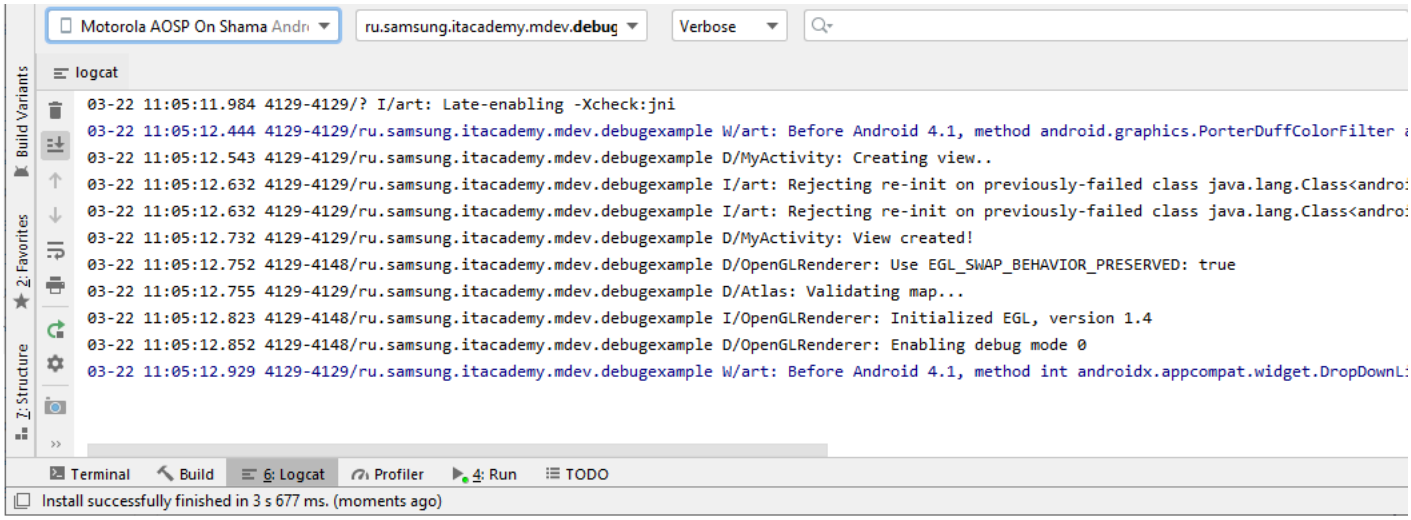
Логирование основано на включении в программу дополнительных отладочных выводов в местах, где меняются значения переменных (узловые точки). Например, если вы пишите консольную программу, то для ее отладки можно выводить промежуточные значения переменных. Зачастую это помогает найти ошибки в коде и исправить их. В Android для логирования есть специальный класс `android.util.Log`.

Создадим новый проект. В созданном проекте открываем файл MainActivity.kt. Чтобы добавить логирование, вставим вызовы метода `Log.d()` до и после `setContentView()`. Android Studio будет подсвечивать Log.d красным цветом, так как класс Log не импортирован. Для быстрого импорта наведите на Log нажмите Alt + Enter.

Код активности будет выглядеть следующим образом

```
class MainActivity : AppCompatActivity() {
    private val LOG_TAG = "MyActivity"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(LOG_TAG, "Creating view..")
        setContentView(R.layout.activity_main)
        Log.d(LOG_TAG, "View created!")

    }
}
```

Наш код выведет информацию о подгрузке макета для текущей Activity. Для просмотра этой информации нужно запустить проект и открыть вкладку с LogCat. На вкладке LogCat видны наши сообщения, а также куча других сообщений, которые создаются различными модулями и программами.



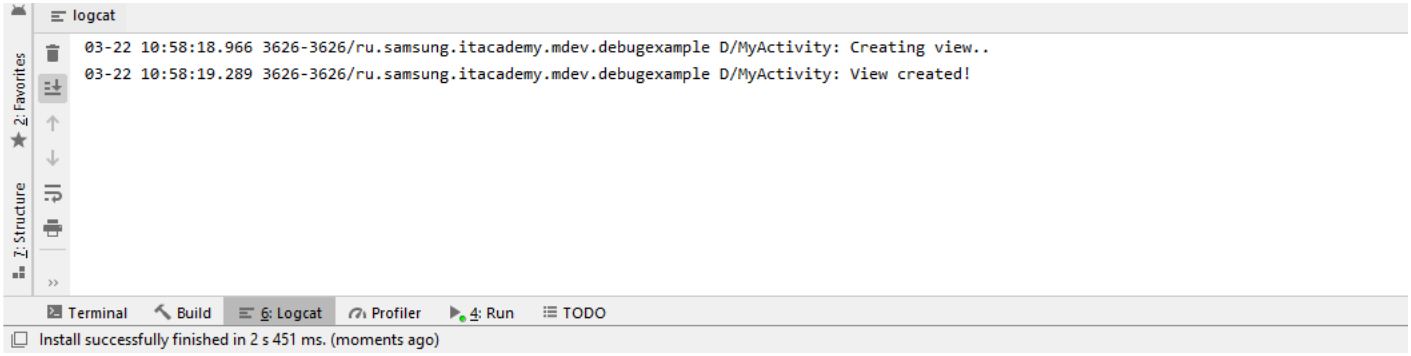
Класс Log разбивает сообщения по категориям в зависимости от важности. Для этого используются специальные методы, которые легко запомнить по первым буквам, указывающим на категорию:

- `Log.e()` – ошибки (error);
- `Log.w()` – предупреждения (warning);
- `Log.i()` – информация (info);
- `Log.d()` – отладка (debug);
- `Log.v()` – подробности (verbose).

В первом параметре представленных методов класса `Log` используется строка, называемая тегом. В качестве тега обычно задают имя класса, название библиотеки или название приложения. Обычно принято объявлять глобальную статическую строковую переменную, и уже в любом месте вашей программы вы вызываете нужный метод записи в Log с этим тегом, например:

```
Log.d(LOG_TAG, "View created!")
```

Данный тег мы можем применить для фильтрации сообщения в LogCat. Для этого нужно добавить фильтр по тегу. В Android Studio нажмем в поле поиска и введем наш тег:



Если в коде использовать метод `Log.e()`, то строчка будет выделена красным цветом. Также мы можем отображать сообщения по уровням: VERBOSE, DEBUG, INFO, WARN, ERROR и ASSERT. Если выбрать уровень сообщений ERROR, то будут выводиться сообщения, сгенерированные с уровнем ERROR и ASSERT. Если выбрать VERBOSE, то будут выводиться все сообщения.

Как правило, в серьезных приложениях в режиме тестирования постоянно логируется информация об обращении к сторонним сервисам API, обращении к базе данных, при возникновении нестандартных ситуаций и т. д. При выкладывании приложения в маркеты рекомендуется отключать всю отладочную информацию.

Существенный минус данного метода отладки — для получения информации нужно заранее в тексте программы проставить вызовы логирования с соответствующими данными. Но существует и другой метод отладки, позволяющий отлаживать программу на лету, — использование отладчика.

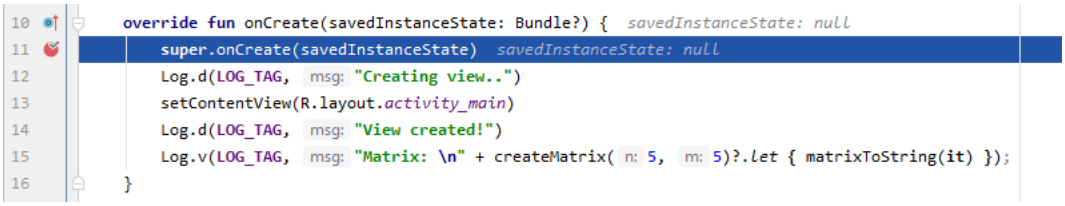
Использование встроенного отладчика

С помощью встроенного отладчика Android Studio мы можем отлаживать программы во время их выполнения.

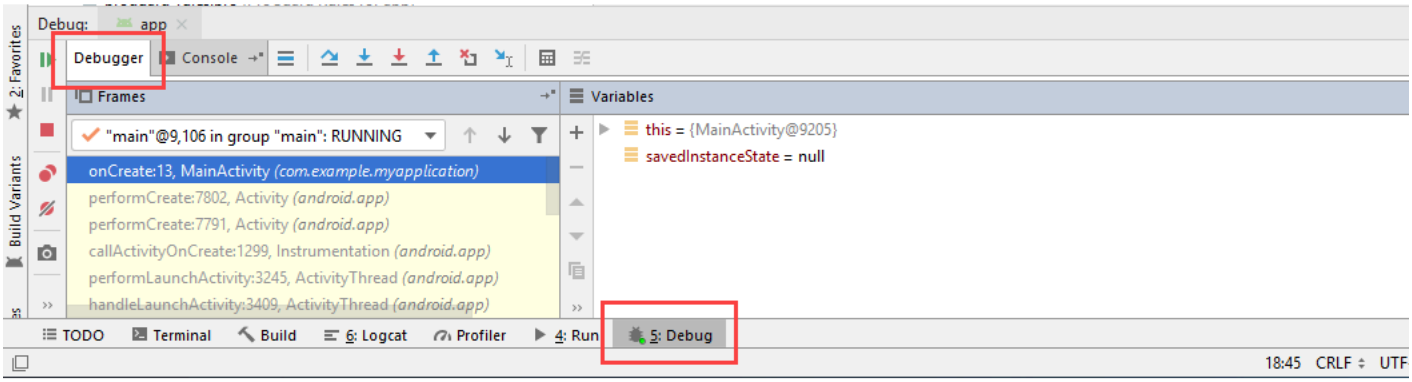
Предположим мы написали программу, которая генерирует двумерный массив и переводит его в строку.

```
class MainActivity : AppCompatActivity() {
    private val LOG_TAG = "MyActivity"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(LOG_TAG, "Creating view..")
        setContentView(R.layout.activity_main)
        Log.d(LOG_TAG, "View created!")
        Log.v(LOG_TAG, "Matrix: \n" + createMatrix(5, 5)?.let { matrixToString(it) });
    }

    fun createMatrix(n: Int, m: Int): Array<IntArray>? {
        val array = Array(n) { IntArray(m) }
        for (i in array.indices) {
            for (j in array[i].indices) {
                array[i][j] = (Math.random() * 10).toInt()
            }
        }
        return array
    }

    fun matrixToString(array: Array<IntArray>): String? {
        var result = ""
        for (i in array.indices) {
            for (element in array[i]) {
                result += "$element "
            }
            result += "\n"
        }
        return result
    }
}
```

В методе onCreate установим точку останова (breakpoint). Для этого нужно поместить курсор на нужную строчку метода и щелкнуть левой кнопкой мыши слева от номера строки.



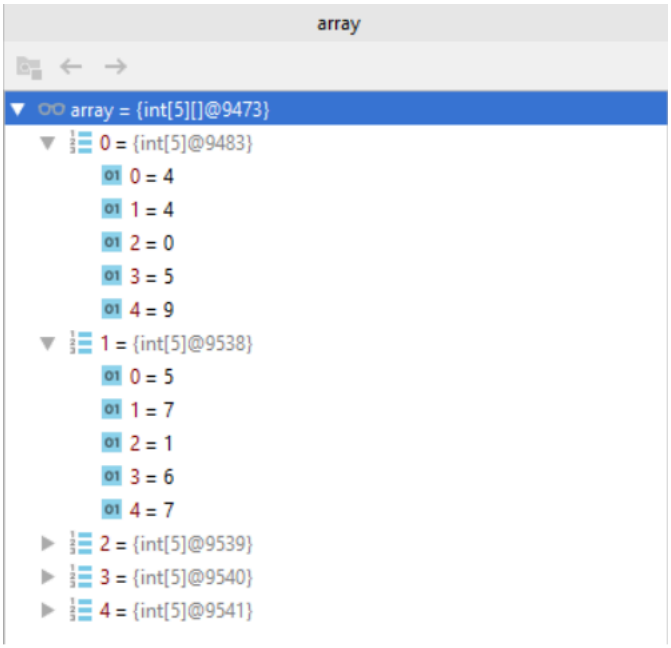
На строке кода, где установлен breakpoint, появится кружок. Теперь запустим выполнение в режиме отладки. Для этого нужно нажать на кнопку с жучком, либо выбрать из меню *Run -> Debug 'app'*, либо нажать на клавиатуре Shift + F9. После запуска приложения Android Studio автоматически откроет вид отладки.



Перед вами откроется много разных окон, но мы остановимся на самых важных из них: окно с кодом программы и окно Variables. В окне кода одна из строчек подсвечена синим цветом. Так выделена текущая строчка, на которой приостановилось выполнение программы. В окне Variables показаны переменные текущей области видимости:

Добавим точки останова в начало методов createMatrix и matrixToString и запустим отладку (Resume Program). Выполнение программы остановится на строке с красной точкой в методе createMatrix. В окне Variables можно увидеть значения параметров m и n.

Нажимая на кнопку Step Over, выполняем программу шаг за шагом. Заметим, что в окне Variables появилась переменная array. По мере выполнения массив наполняется элементами. Если навести курсор на массив, мы увидим значения его элементов.



Если нажать на Step Out, выполнение программы остановится после выхода из текущего метода, то есть к тому методу, который вызвал данный метод. В нашем случае выполнение программы вернулось в метод onCreate. Нажимаем на Resume Program, и выполнение переходит в метод matrixToString. Здесь также можно пошагово выполнять команды, чтобы посмотреть, как наполняется строковая переменная result.

Данный инструмент отладки отлично подходит для просмотра значений переменных и просмотра пути выполнения программы. Можно на ходу добавлять, убирать точки останова, задавать условия останова. Но данный метод не может полностью заменить метод вывода отладочной информации. Чаще всего в обычных ситуациях разработчики используют логи для важной информации, а при возникновении необходимости отладки используют инструменты отладки.

Проект с кодом приложения с использованием логов и встроенного отладчика можно посмотреть в приложении DebugExample.

2.10.3 Модульное тестирование Android приложение.

С усложнением программного проекта растет и потенциальное количество ошибок в нем. При этом увеличивается не только количество строк кода проекта, но и, к примеру, появляются новые разработчики, которые заменяют старых или работают совместно (каждый разработчик разрабатывает свою часть проекта). Для добавления нового функционала зачастую приходится переделывать старый код. Ошибки неизбежны.

В связи со всем вышесказанным в промышленной разработке программ используют модульное тестирование. Идея модульного тестирования заключается в написании тестов для всех нетривиальных методов, что позволяет оперативно проверять корректность уже протестированного кода после очередной его модификации. Цель модульного тестирования — исключить из поиска ошибок отдельные части программы путем их автоматической проверки на заранее написанных тестах.

Unit Testing

Локальные модульные тесты (unit-тесты) проверяют работу метода, класса, компонента. По сути вы тестируете код, который можно проверить без применения устройства или эмулятора. Подобные тесты находятся в папке Test проекта. Для создания для юнит-тестов можно использовать следующие инструменты:

- JUnit
- Mockito
- PowerMock

В данном разделе остановимся на фреймворке JUnit. В build.gradle приложения имеется строка для компиляции юнит-тестов.

```
testImplementation 'junit:junit:4.12'
```

Введение в JUnit

JUnit – это фреймворк, предназначенный для тестирования программ, разработанных с использованием технологии Java. Ключевая идея фреймворка – “сначала тесты, потом код”. Поэтому сначала необходимо определить результат работы того или иного раздела приложения и написать тесты, которые проверяют идентичность результата с требуемым, а только после этого написать сам код, который и будет тестироваться. Такой подход увеличивает эффективность работы разработчика и позволяет писать более стабильный код, особенно при работе в команде над большими проектами.

Тестовый случай (Test Case) в юнит тестировании – это код, который проверяет работу другого кода (класса, метода и т.п.) в соответствии с заданными требованиями. Тестовый случай характеризуется известными входными данными и ожидаемым выводом программы. При разработке необходимо предусмотреть как минимум, два тестовых случая для каждого требования – отрицательный и положительный.

Давайте напишем несколько тестов, чтобы увидеть на практике, что именно мы можем протестировать и как выглядит код тестирования.

Пример

В качестве тестируемого приложения возьмем простейшее приложения для сложения двух чисел. В разметке укажем два EditText, кнопку с операцией сложения и TextView с результатом.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/a"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="a ="
        android:inputType="textPersonName"
        tools:layout_editor_absoluteX="13dp"
        tools:layout_editor_absoluteY="12dp" />

    <EditText
        android:id="@+id/b"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="b ="
        android:inputType="textPersonName" />

    <Button
        android:id="@+id/add"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="a + b" />

    <TextView
        android:id="@+id/text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="a + b = " />

    <TextView
        android:id="@+id/res"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

В приложении создадим класс Calculator, который выполняет сложение чисел. Для простоты будем использовать тип Int.

```
class Calculator {
    fun add(a: Int, b: Int): Int {
        return a + b
    }
}
```

В данном коде нет фрагментов характерных только для Android-приложений, следовательно можно создать локальные тесты, которые будут тестировать этот класс.

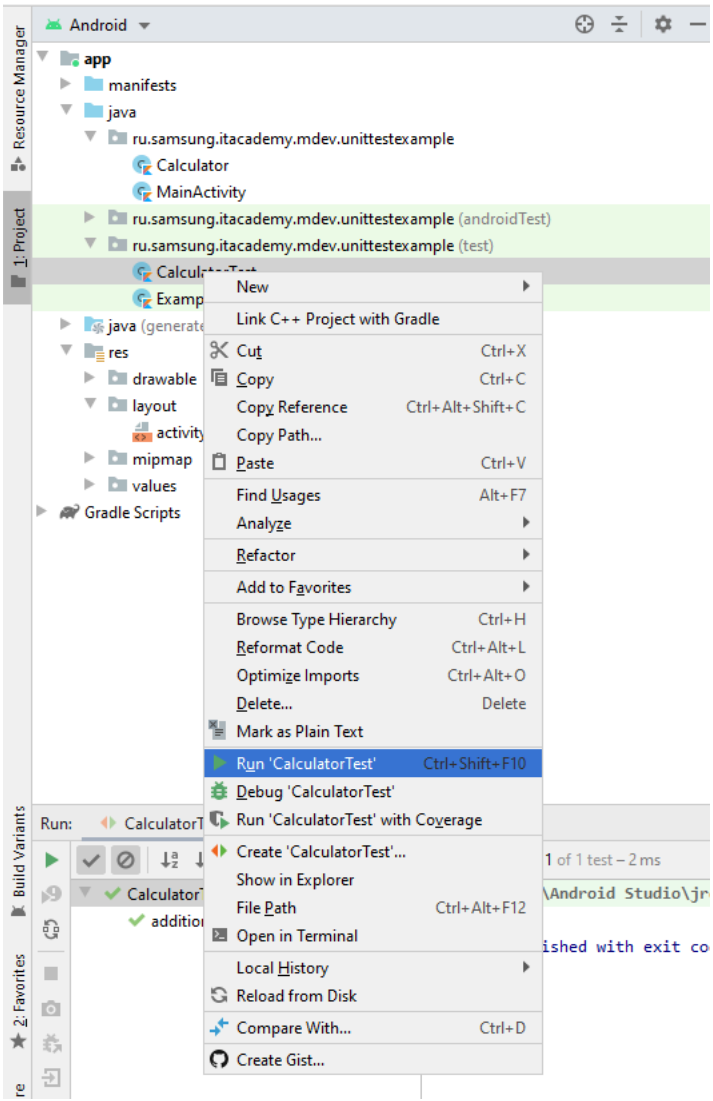
В папке **Test** создадим файл *CalculatorTest.kt* и добавим в него следующий код, импортировав необходимые библиотеки.

```
import org.junit.Assert
import org.junit.Before
import org.junit.Test

class CalculatorTest {
    private var calculator: Calculator? = null
    @Before
    fun setUp() {
        calculator = Calculator()
    }

    @Test
    fun addition() {
        Assert.assertEquals(3, calculator!!.add(1, 2).toLong())
    }

    @After
    fun tearDown() {
        calculator = null
    }
}
```

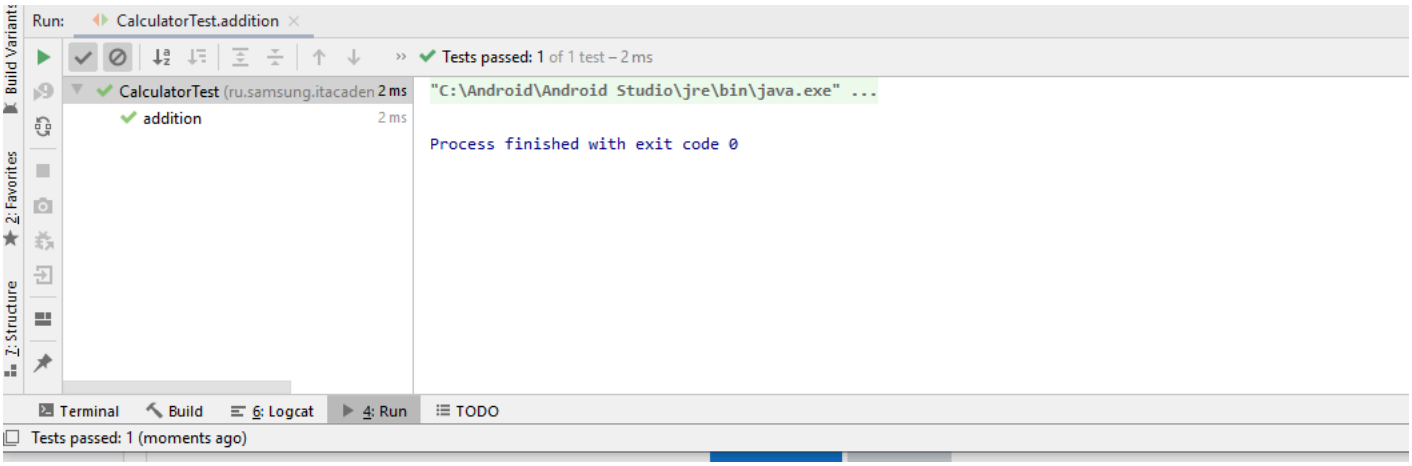
Любой метод с аннотацией «@Test» в JUnit считается отдельным тестом. Метод addition будет вызван при запуске тестирования для проверки работы метода add класса Calculator.

И еще два важных метода – это методы, помеченные аннотациями Before и After. Код метода с аннотацией Before будет выполняться перед выполнением каждого тестового метода. Соответственно, код с аннотацией After будет выполняться после каждого тестового метода. Эти методы нужны для того, чтобы подготовить какие-то параметры или объекты к тестам (например, вынести тестируемый объект в поле класса и инициализировать его в методе setUp вместо того, чтобы выполнять инициализацию в каждом тестовом методе) или же очистить ресурсы после окончания тестового метода.

Чтобы запустить модульный тест по умолчанию, выберите CalculatorTest в студии Android, щелкните правой кнопкой мыши по нему и затем нажмите « Run CalculatorTest», как показано ниже



Запустив тест, мы получим сообщение о том, что тест успешно пройден

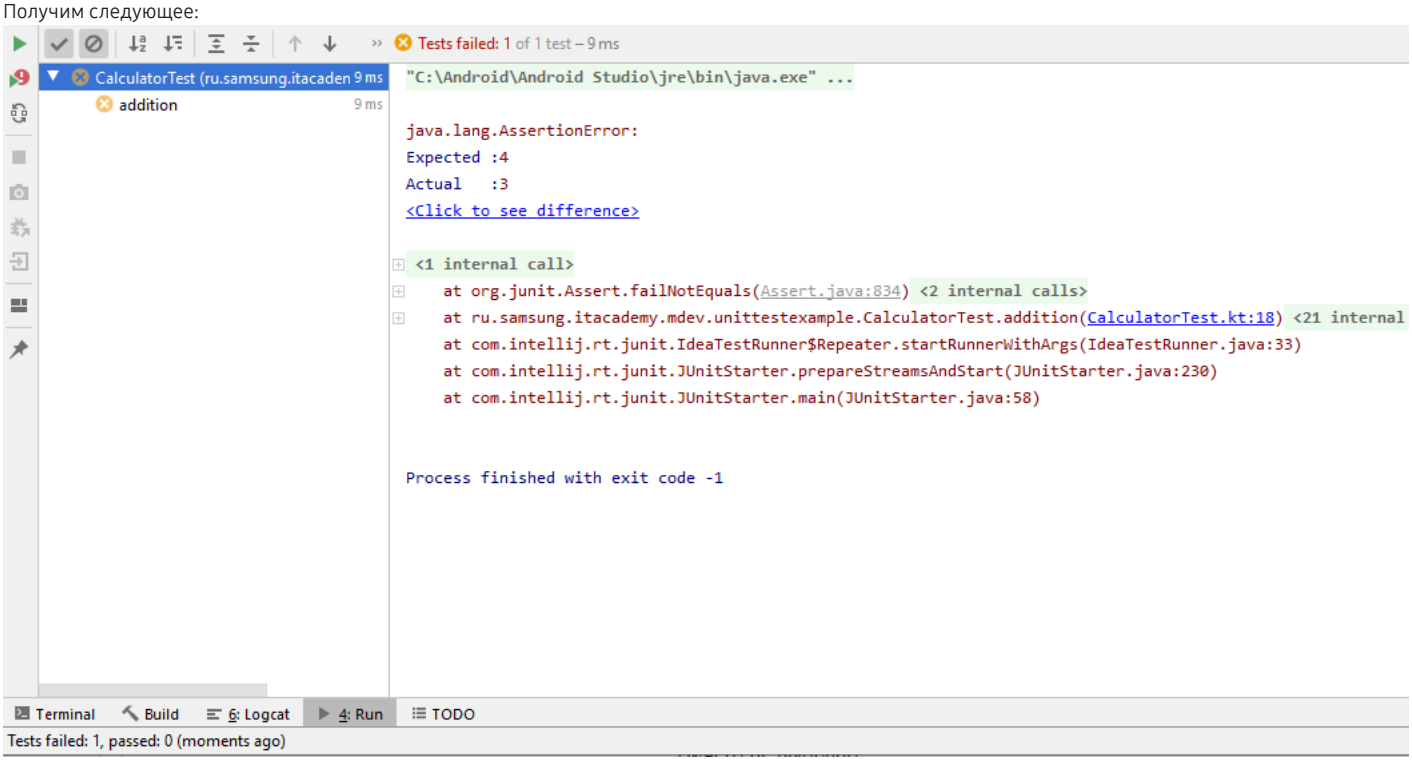


Таким образом тест проверил метод Calculator.add со значениями 1 и 2, получил 3, затем сравнил с ожидаемым значением (3) и выяснил, что результаты совпали. Таким образом программа работает так, как мы от нее ожидали.

Если поменять код и вместо цифры 3 поставить цифру 4:

```
@Test
fun addition() {
    Assert.assertEquals(4, calculator!!.add(1, 2).toLong())
}
```

Получим следующее:

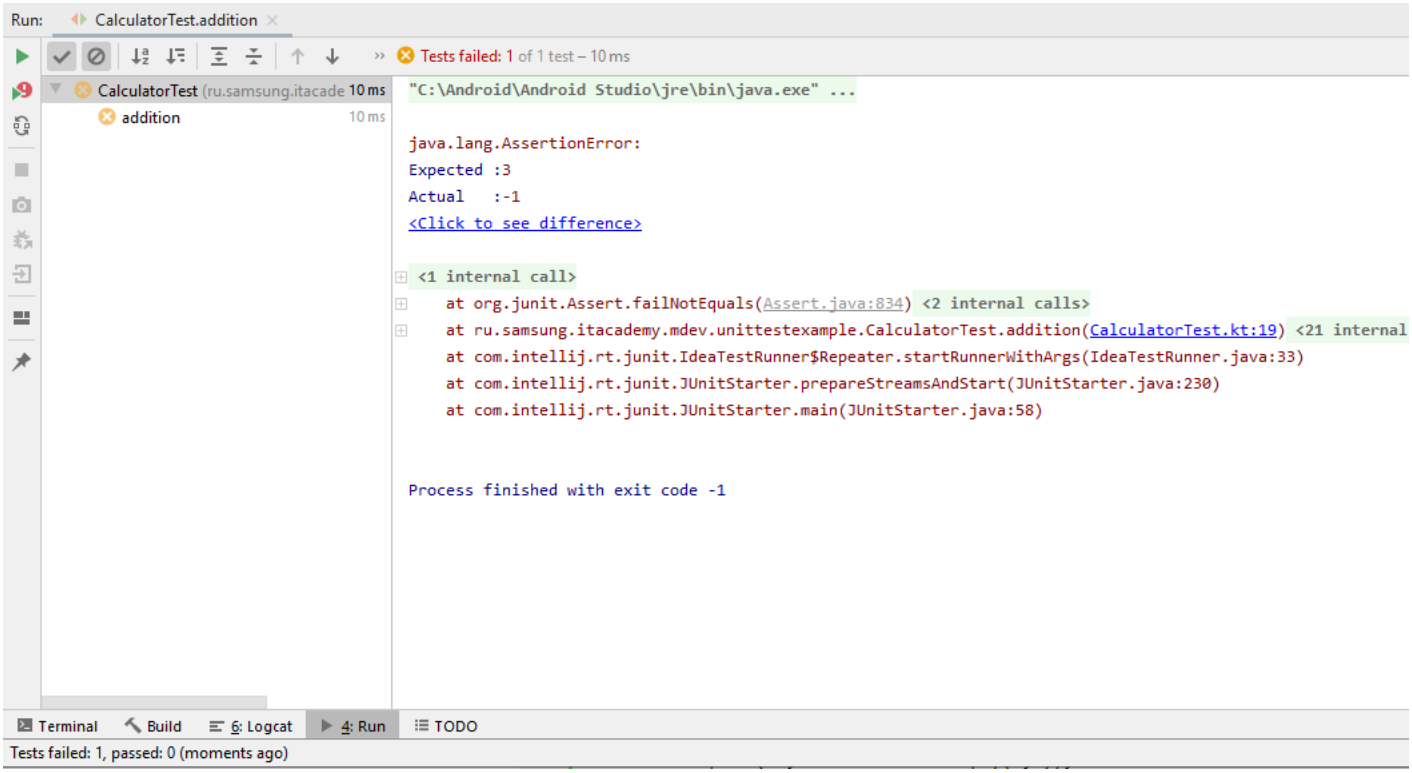


При запуске теста нам не понадобилось Android устройство. Тест выполнился на компьютере, в Java-машине.

Откроем класс Calculator и изменим знак "+" на "-" в методе add.

```
fun add(a: Int, b: Int): Int {
    return a - b
}
```

После запуска теста появилось сообщение о том, что ожидалось значение 3, в результате -1. Таким образом метод сработал не так, как мы ожидали, а это означает, что в нем появилась ошибка.



Кроме метода addition, мы можем в тестовом классе создать и другие @Test методы для тестирования. Так, можно настроить тест на ожидаемое исключение используя параметр expected:

```
@Test(expected = NullPointerException::class)
    fun nullStringTest() {
        val str: String? = null
        assertTrue(str!!.isEmpty())
}
```

В таком случае тест выполнится, т.к. это исключение мы ожидали. Если поменяем исключение, например на IOException, тест будет завален. Для длинных операций можно также указать параметр timeout и установить значение в миллисекундах. Если метод не выполнится в течение заданного времени, тест будет считаться проваленным:

```
@Test(timeout = 1000)
fun requestTest() {}
```

Код с проектом примера можно посмотреть в проекте UnitTestExample

2.10.4 Другие виды тестирования

Разработка программных проектов не ограничивается тестированием отдельных компонентов системы. После модульного тестирования следует интеграционное. *Интеграционное тестирование* — это тестирование не отдельных компонентов системы, а результата их взаимодействия между собой в какой-либо среде.

Упор делается именно на тестировании взаимодействия. Так как интеграционное тестирование производится после модульного, то все проблемы, обнаруженные в процессе объединения модулей, скорее всего, связаны с особенностями их взаимодействия.

Вначале описывается план тестирования, подготавливаются тестовые данные, создаются и исполняются тест-кейсы (пошаговые действия для тестирования определенного функционала системы). Найденные ошибки исправляют и снова запускают тестирование. Цикл повторяется до тех пор, пока взаимодействие всех компонентов не будет работать без ошибок.

Для того чтобы автоматизировать интеграционное тестирование, используются системы непрерывной интеграции (англ. continuous Integration System, CIS). CIS проводит мониторинг исходных кодов. Как только разработчики выкладывают обновление кода, выполняются различные проверки и модульные тесты. Далее проект компилируется и проходит интеграционное тестирование. Выявленные ошибки включаются в отчет тестирования.

Автоматические интеграционные тесты выполняются сразу после внесения изменений. Это существенно сокращает время поиска и устранения ошибок.

Следующий этап тестирования — системное тестирование, которое разделяется на 2 этапа.

- *Альфа-тестирование* — имитация реальной работы с системой разработчика, либо реальная работа с системой, ограниченной кругом потенциальных пользователей.
- *Бета-тестирование* — в некоторых случаях выполняется распространение предварительной версии для некоторой группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Кроме тестирования по этапам выделяют также классификацию по объектам тестирования:

1. Тестирование удобства пользования продуктом.
2. Оценка уязвимости программного обеспечения к различным атакам.
3. Проверка перевода пользовательского интерфейса, документации и сопутствующих файлов программного обеспечения на различных языках.
4. Проверка скорости работы системы под определенной нагрузкой. В тестировании производительности выделяют следующие направления:

- *Нагрузочное тестирование*. С помощью нагрузочного тестирования обычно оценивают поведение программы под заданной ожидаемой нагрузкой. Такой нагрузкой зачастую выступает, например, ожидаемое число одновременно работающих пользователей, которые совершают определенное количество действий за заданный интервал времени.
- *Стресс-тестирование*. С его помощью обычно исследуют пределы пропускной способности программы. Такое тестирование проводят для определения надежности системы во время значительных нагрузок. Стресс-тестирование позволяет ответить на вопрос о производительности системы в случае значительного превышения ожидаемого максимума нагрузки. Стресс-тестирование позволяет определить «узкие места» системы, которые вероятнее всего приведут к сбоям системы в пиковой ситуации. Проверка правильности работы программы осуществляется на большом количестве случайно сгенерированных данных. Мотивация данного тестирования — предпочтительнее быть готовым к обработке экстремальных условий системы, чем ожидать отказа системы, тем более когда стоимость отказа системы в экстремальных условиях может быть очень велика.

Для лучшей иллюстрации рассмотрим виды тестирования по аналогии с производством техники, например, телефонов.

Завод закупает множество комплектующих — от винтиков до печатных плат. Очевидно, что качество готовой продукции напрямую зависит от любого из компонентов телефона. Поэтому контроль входного качества компонентов очень важен.

Все начинается с простейших тестов — веса и размера компонентов. Из партии деталей выбирают часть для тестов. Если это новый поставщик или деталь ранее не тестировалась, то проверка проходит самым тщательным образом — устойчивость к агрессивной среде, влажность, прочностные характеристики, рентген-снимки на наличие скрытых дефектов и так далее. В результате тестирования фабрика решает, будет ли она работать с данным поставщиком. Это все модульное тестирование.

После сборки телефона вставляется сим-карта и начинается проверка его работоспособности в целом: по заданным сценариям проверяются функции телефона. Это — интеграционное тестирование.

Аппараты могут проходить и более тщательную проверку:

- время работы телефона в различных режимах;
- деградация аккумулятора с количеством циклов разряда/заряда;
- работа процессора при максимальной нагрузке.

В тестовой лаборатории могут присутствовать помещения для проверки в условиях различных температур и влажности — имитация климатических зон. Кроме этого, есть специальные камеры «старения» для получения реального эффекта старения за максимально короткий срок.

Механическая прочность экрана испытывается металлическими шариками, которые падают на него с определенной высоты. На других автоматах телефон поднимают на определенную высоту и роняют на металлическую поверхность.

Все описанное специальное оборудование применяется для нагрузочного и стресс-тестирования. Зачастую эти понятия считают синонимами, но это не совсем верно. Нагрузочное тестирование — это когда для проверки воссоздают предполагаемые нормальные условия эксплуатации. Когда же производится проверка в условиях сверхвысоких (выходящих за пределы обычного использования) нагрузок, то это уже стресс-тестирование.