**Автоматизированные тесты**

<https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#0>

Изучим автоматизированные тесты в Android и как они позволяют писать масштабируемые и надежные приложения. Вы также познакомитесь с разницей между логикой пользовательского интерфейса и бизнес-логикой, а также с тем, как тестировать обе. Наконец, вы научитесь писать и запускать автоматические тесты в Android Studio.

## [2. Получите стартовый код](https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#1)

Загрузите код:

[file\_downloadСкачать zip](https://github.com/google-developer-training/basic-android-kotlin-compose-training-tip-calculator/archive/refs/heads/main.zip)

## [3. Автоматизированные тесты](https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#2)

**Тестирование программного обеспечения** — это структурированный метод проверки вашего программного обеспечения, чтобы убедиться, что оно работает должным образом. Автоматизированное тестирование — это код, который проверяет правильность работы другого написанного вами фрагмента кода.

Тестирование — важная часть процесса разработки приложения. Последовательно выполняя тесты вашего приложения, вы можете проверить его корректность, функциональное поведение и удобство использования, прежде чем выпустить его публично.

Тестирование также дает возможность постоянно проверять существующий код по мере внесения изменений.

Хотя ручное тестирование почти всегда имеет место, тестирование в Android часто можно автоматизировать. На протяжении оставшегося курса вы сосредоточитесь на автоматических тестах для проверки кода приложения и функциональных требований самого приложения. В этой лабораторной работе вы изучите самые основы тестирования в Android. На более поздних занятиях вы изучите более продвинутые методы тестирования приложений Android.

Познакомившись с разработкой и тестированием Android-приложений для Android, вам следует регулярно писать тесты вместе с кодом вашего приложения. Создание теста каждый раз при создании новой функции в вашем приложении снижает вашу рабочую нагрузку в дальнейшем по мере роста вашего приложения. Это также дает вам удобный способ убедиться, что ваше приложение работает правильно, не тратя слишком много времени на тестирование приложения вручную.

Автоматизированное тестирование является неотъемлемой частью любой разработки программного обеспечения, и разработка Android не является исключением.

## ****Почему автоматизированные тесты важны****

На первый взгляд может показаться, что тесты в вашем приложении на самом деле не нужны, но тестирование необходимо в приложениях любого размера и сложности.

Чтобы расширить свою кодовую базу, вам необходимо тестировать существующую функциональность по мере добавления новых частей, что возможно только при наличии существующих тестов. По мере роста вашего приложения ручное тестирование требует гораздо больше усилий, чем автоматическое. Более того, как только вы начнете работать над приложениями в производстве, тестирование станет критически важным, если у вас большая база пользователей. Например, вы должны учитывать множество различных типов устройств, работающих под управлением разных версий Android.

В конце концов вы достигнете точки, когда автоматические тесты смогут учитывать большинство сценариев использования значительно быстрее, чем ручные тесты. Запуская тесты перед выпуском нового кода, вы можете внести изменения в существующий код, чтобы избежать выпуска приложения с неожиданным поведением.

Помните, что автоматические тесты — это тесты, выполняемые с помощью программного обеспечения, в отличие от ручных тестов, которые выполняются человеком, непосредственно взаимодействующим с устройством. Автоматизированное и ручное тестирование играют решающую роль в обеспечении приятного опыта пользователей вашего продукта. Однако автоматические тесты могут быть более точными и оптимизируют производительность вашей команды, поскольку для их запуска не требуется участие человека, и они могут выполняться гораздо быстрее, чем ручное тестирование.

## ****Тип автоматизированных тестов****

### **Локальные тесты**

Локальные тесты — это тип автоматизированного теста, который напрямую проверяет небольшой фрагмент кода, чтобы убедиться в его правильной работе. С помощью локальных тестов вы можете тестировать функции, классы и свойства. Локальные тесты выполняются на вашей рабочей станции, что означает, что они выполняются в среде разработки без необходимости использования устройства или эмулятора. Это необычный способ сказать, что на вашем компьютере выполняются локальные тесты. У них также очень низкие затраты на компьютерные ресурсы, поэтому они могут работать быстро даже при ограниченных ресурсах. Android Studio готова автоматически запускать локальные тесты.

### **Инструментальные испытания**

Для разработки Android инструментальный тест — это тест пользовательского интерфейса. Инструментальные тесты позволяют тестировать части приложения, которые зависят от API Android, а также API и сервисов его платформы.

В отличие от локальных тестов, тесты пользовательского интерфейса запускают приложение или его часть, моделируют взаимодействие с пользователем и проверяют, правильно ли отреагировало приложение. На протяжении всего курса тесты пользовательского интерфейса выполняются на физическом устройстве или в эмуляторе.

Когда вы запускаете инструментальный тест на Android, тестовый код фактически встроен в собственный пакет приложений Android (APK), как обычное приложение Android. APK — это сжатый файл, содержащий весь код и необходимые файлы для запуска приложения на устройстве или эмуляторе. Тестовый APK устанавливается на устройство или эмулятор вместе с обычным APK приложения. Затем тестовый APK тестирует APK приложения.

## [4. Напишите локальный тест](https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#3)

## ****Подготовьте код приложения****

Локальные тесты напрямую проверяют методы из кода приложения, поэтому тестируемые методы должны быть доступны классам и методам тестирования. Локальный тест в следующем фрагменте кода гарантирует, что метод calculateTip() работает правильно, но  в данный момент метод calculateTip() недоступен private и поэтому недоступен из теста. Удалите  обозначение private и сделайте его internal:

MainActivity.kt

**internal** fun calculateTip(amount: Double, tipPercent: Double = 15.0, roundUp: Boolean): String {  
    var tip = tipPercent / 100 \* amount  
    if (roundUp) {  
        tip = kotlin.math.ceil(tip)  
    }  
    return NumberFormat.getCurrencyInstance().format(tip)  
}

* В файле MainActivity.kt в строке перед методом calculateTip() добавьте аннотацию @VisibleForTesting:

**@VisibleForTesting**internal fun calculateTip(amount: Double, tipPercent: Double = 15.0, roundUp: Boolean): String {  
    var tip = tipPercent / 100 \* amount  
    if (roundUp) {  
        tip = kotlin.math.ceil(tip)  
    }  
    return NumberFormat.getCurrencyInstance().format(tip)  
}

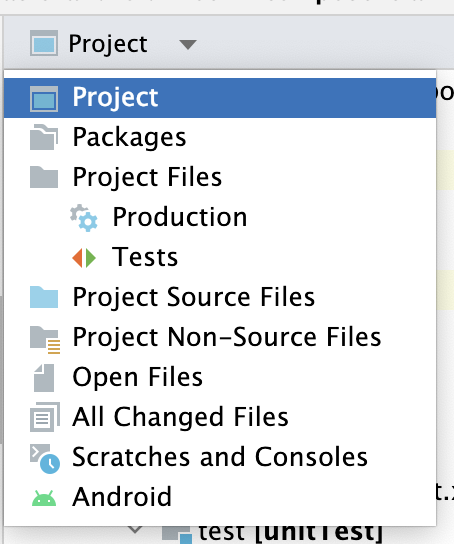
Это делает метод общедоступным, но указывает другим, что он доступен только в целях тестирования.

## Создайте тестовый каталог

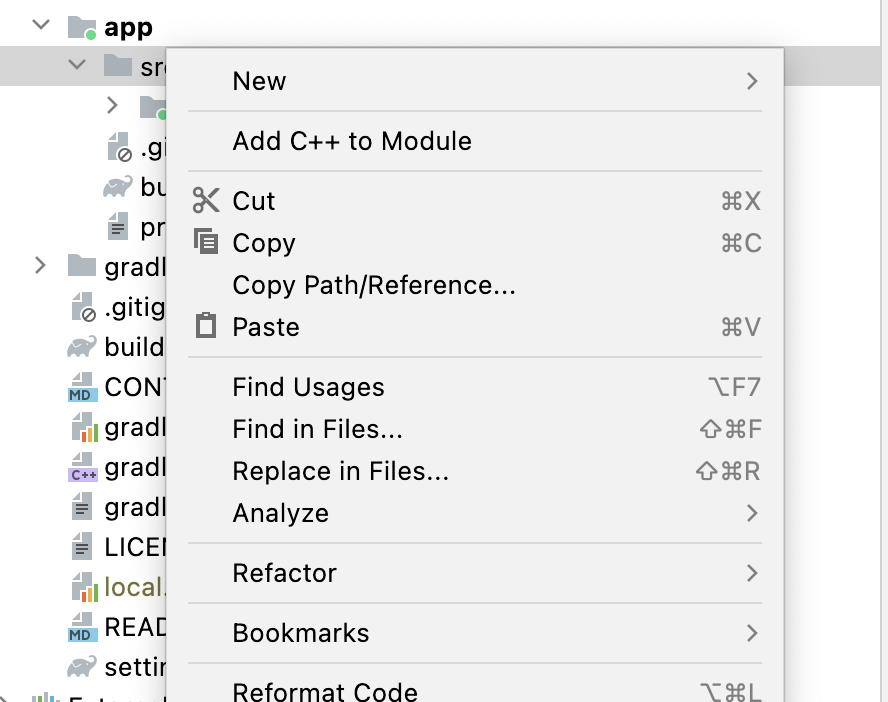
В проектах Android  это каталог test, в котором записываются локальные тесты.

Создайте **тестовый** каталог:

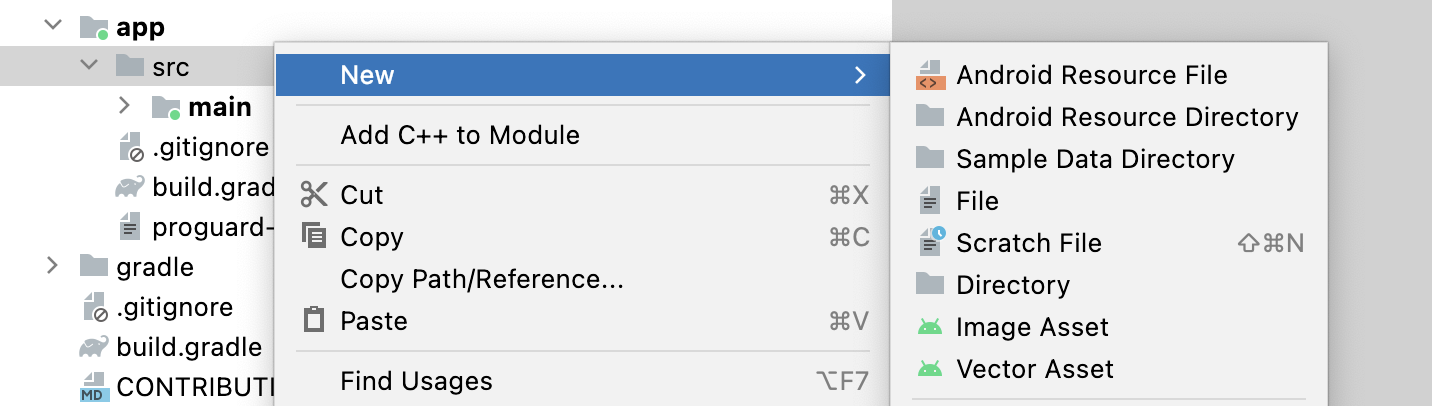
1. На вкладке **«Проект»** измените вид на «Project».



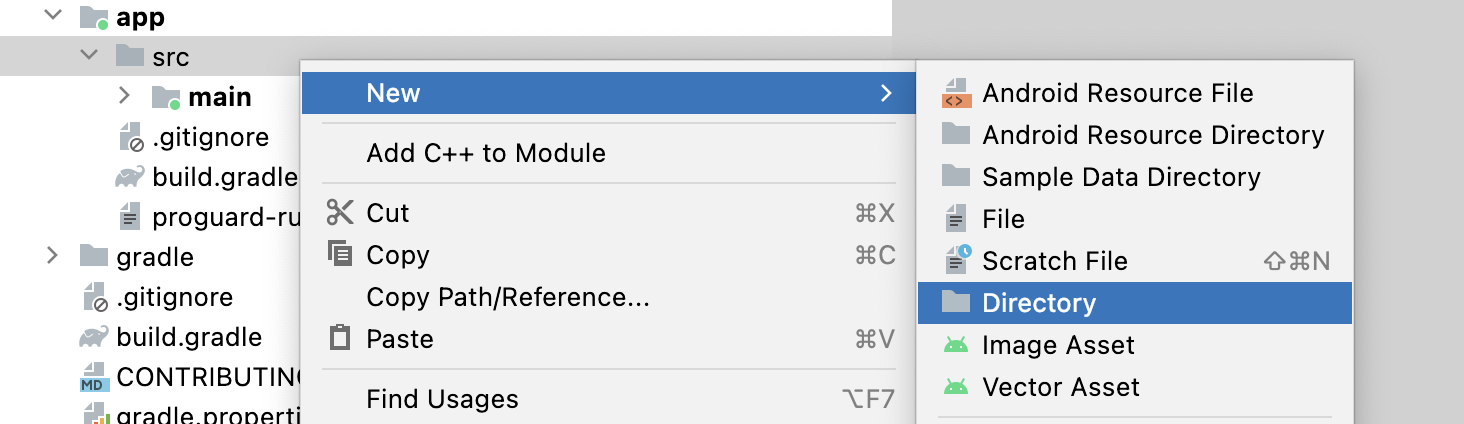
1. Щелкните правой кнопкой мыши каталог **src** .



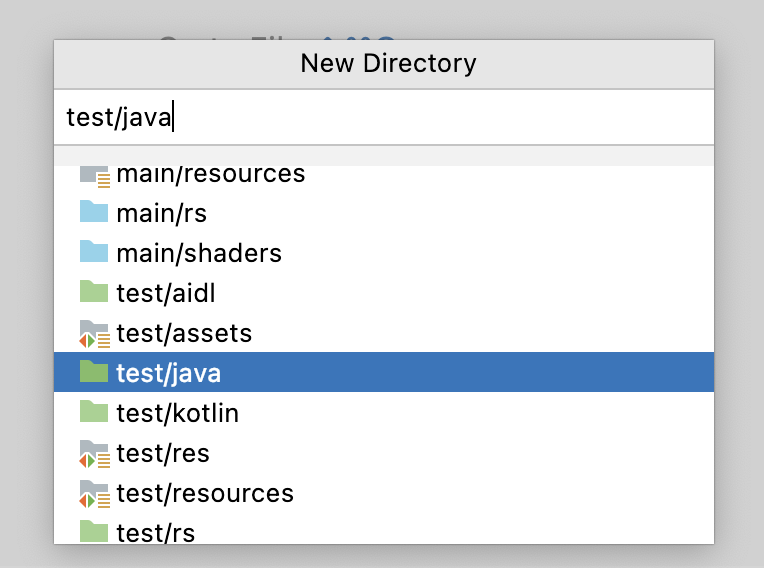
1. Выберите **New** .



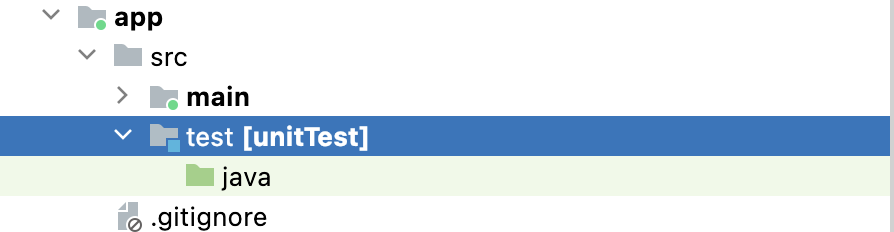
1. Выберите **Directory.**



1. В окне **«New Directory»** выберите **test/java** .



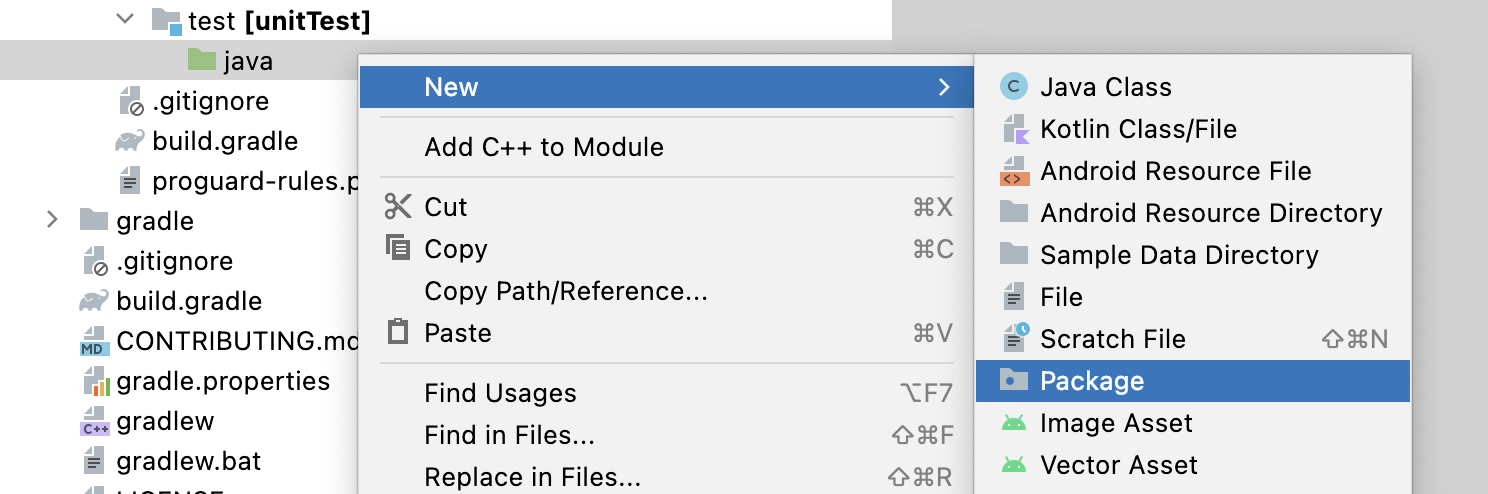
1. Нажмите клавишу **возврата** или **ввода** на клавиатуре. Тестовый каталог теперь можно увидеть на вкладке **« Project»** .



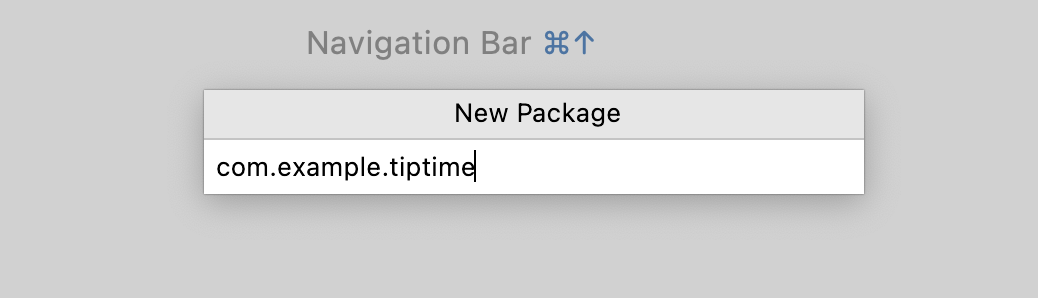
**Для тестового** каталога требуется структура пакета, идентичная структуре каталога, mainв котором находится код вашего приложения. Другими словами, так же, как код вашего приложения написан в пакете **main > java > com > example > Tiptime** , ваши локальные тесты будут написаны в **пакете test > java > com > example > Tiptime** .

Создайте эту структуру пакета в **тестовом** каталоге:

1. Щелкните правой кнопкой мыши каталог **test/java** и выберите **«New» > «Packege»** .



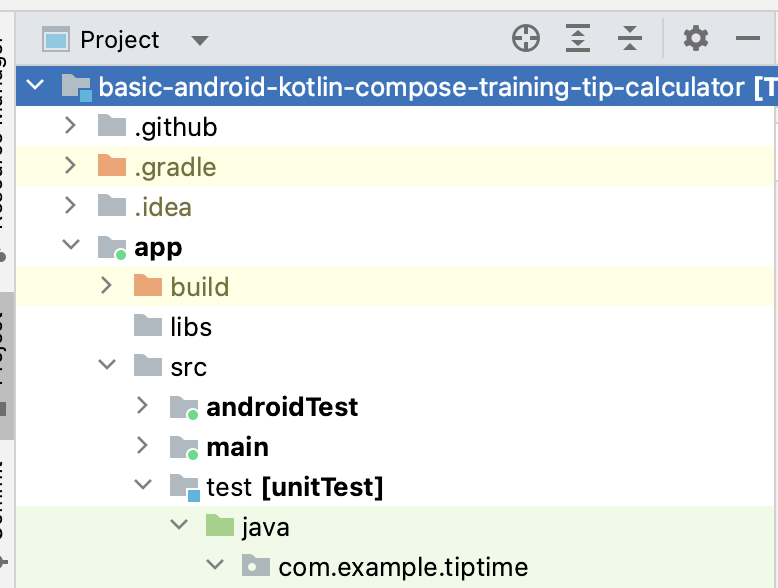
1. В окне **«New Package»** введите com.example.tiptime.



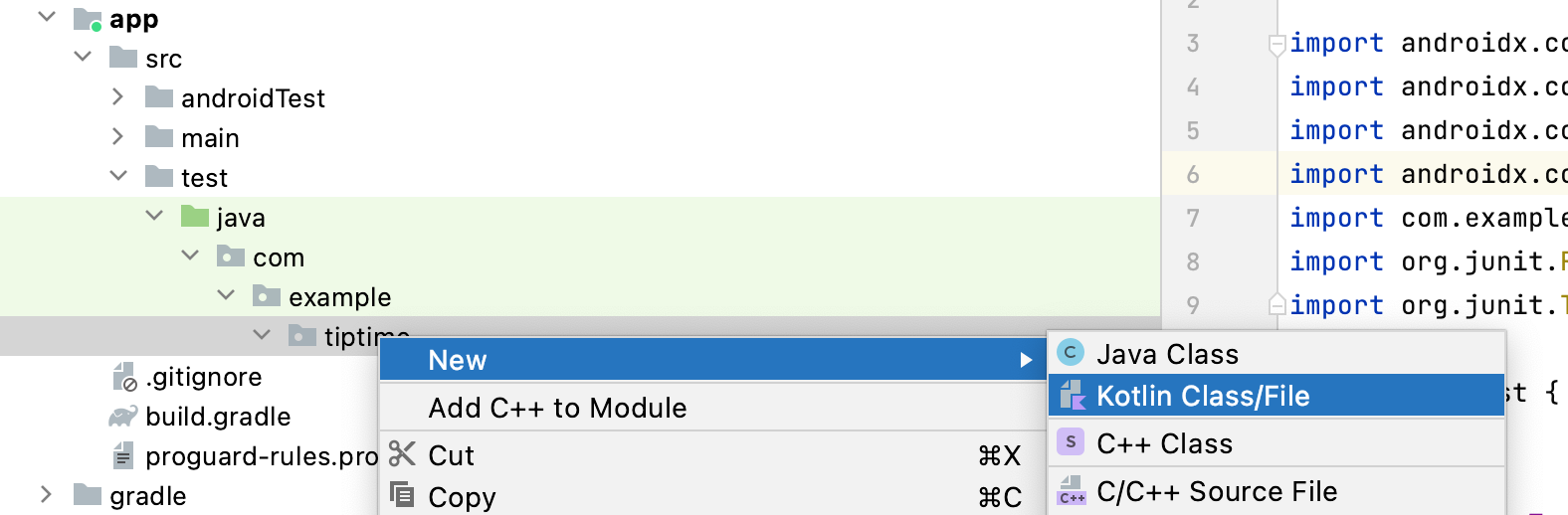
## ****Создайте тестовый класс****

Теперь, когда **тестовый** пакет готов, пришло время написать несколько тестов! Начните с создания тестового класса.

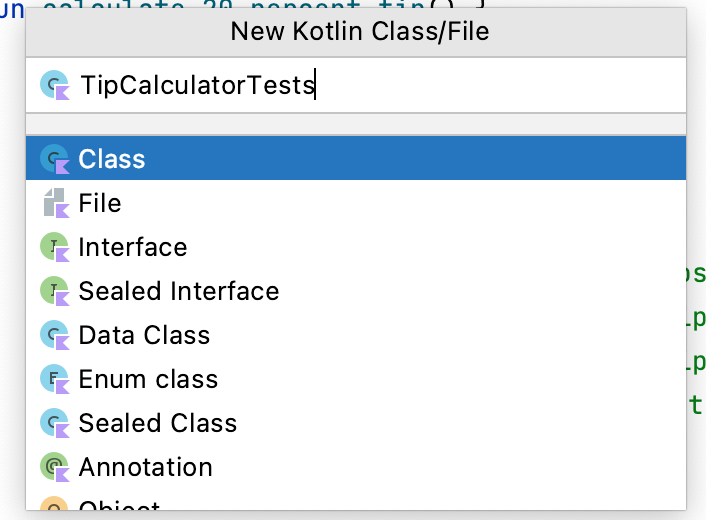
1. На вкладке **«Project»** нажмите **«App» > «src» > «test»** , а затем щелкните стрелку развертывания рядом с  каталогом test.



1. Щелкните правой кнопкой мыши каталог tiptime и выберите **«New» > «Kotlin Class/file»** .



1. Введите TipCalculatorTests в качестве имени класса.



## ****Написать тест****

Как упоминалось ранее, локальные тесты используются для тестирования небольших фрагментов кода в приложении. Основная функция приложения Tip Time — расчет чаевых, поэтому необходимо провести локальный тест, который обеспечит правильную работу логики расчета чаевых.

Для этого вам нужно напрямую вызвать функцию calculateTip(), как вы это сделали в коде приложения. Затем вы убедитесь, что значение, возвращаемое функцией, соответствует ожидаемому значению на основе значений, которые вы передали функции.

Есть несколько вещей, которые вам следует знать о написании автоматических тестов. Следующий список концепций применим к локальным и инструментальным тестам. Поначалу они могут показаться абстрактными, но к концу этой лаборатории вы познакомитесь с ними лучше.

* Пишите автоматизированные тесты в виде методов.
* Аннотируйте метод аннотацией @Test. Это позволяет компилятору узнать, что метод является тестовым, и запускает его соответствующим образом.

**Примечание** .  Аннотация @Test импортируется из пакета org.junit.Test с помощью оператора импорта import org.junit.Test.

* Убедитесь, что название четко описывает, что проверяет тест и каков ожидаемый результат.
* Методы тестирования не используют логику, как обычные методы приложения. Их не волнует, как что-то реализовано. Они строго проверяют ожидаемый результат для данного ввода. Другими словами, методы тестирования выполняют только набор инструкций, чтобы убедиться, что пользовательский интерфейс или логика приложения работают правильно. Вам пока не нужно понимать, что это значит, потому что вы увидите, как это выглядит позже, но помните, что тестовый код может сильно отличаться от кода приложения, к которому вы привыкли.
* Тесты обычно заканчиваются утверждением, которое используется для обеспечения выполнения заданного условия. Утверждения представляют собой вызов метода, в имени которого есть assert. Например:  утверждение assertTrue() обычно используется при тестировании Android. Операторы утверждения используются в большинстве тестов, но в реальном коде приложения они используются редко.

Напишите тест:

1. Создайте метод для проверки расчета чаевых в размере 20 % на сумму счета в 10 долларов. Ожидаемый результат этого расчета — 2 доллара.

import org.junit.Test  
  
class TipCalculatorTests {  
  
   @Test  
   fun calculateTip\_20PercentNoRoundup() {  
         
   }  
}

Возможно, вы помните, что метод calculateTip() из файла MainActivity.kt в коде приложения требует трех параметров. Сумма счета, процент чаевых и флажок округлять результат или нет.

fun calculateTip(amount: Double, tipPercent: Double, roundUp: Boolean)

Когда придет время вызвать этот метод из теста, эти параметры необходимо передать так же, как при вызове метода в коде приложения.

1. В методе calculateTip\_20PercentNoRoundup() создайте две константные переменные: переменную amount, которой присвоено 10.00, и переменную tipPercent, которой присвоено 20.00.

val amount = 10.00  
val tipPercent = 20.00

1. В коде приложения в файле MainActivity.kt обратите внимание на следующий код: сумма чаевых форматируется в зависимости от языкового стандарта устройства.

MainActivity.kt

...  
NumberFormat.getCurrencyInstance().format(tip)  
...

Тот же формат необходимо использовать при проверке ожидаемой суммы чаевых в тесте.

1. Создайте  переменную expectedTip, имеющую значение NumberFormat.getCurrencyInstance().format(2).

Переменная expectedTip позже  сравнивается с результатом метода calculateTip().  Таким образом тест гарантирует корректность работы метода. На последнем шаге вы присваиваете переменной amount значение 10.00, а переменной tipPercent — 20.00. Двадцать процентов от 10 равны 2, поэтому переменной expectedTip присваивается отформатированная валюта со значением 2. Помните, что метод calculateTip() возвращает форматированное значение String.

1. Вызовите метод calculateTip() с переменными amount и tipPercent и передайте аргумент false для округления.

В этом случае вам не нужно учитывать округление в большую сторону, поскольку ожидаемый результат не учитывает округление в большую сторону.

1. Сохраните результат вызова метода в постоянной  переменной actualTip.

До этого момента написание этого теста мало чем отличалось от написания обычного метода в коде приложения. Однако теперь, когда у вас есть значение, возвращаемое методом, который вы хотите протестировать, вы должны определить, является ли это значение правильным значением с помощью утверждения.

Создание утверждения обычно является конечной целью автоматического теста и не часто используется в коде приложения. В этом случае вы хотите убедиться, что  переменная actualTip равна переменной expectedTip. Для этого можно использовать метод assertEquals() из библиотеки JUnit.

Метод assertEquals() принимает два параметра — ожидаемое значение и фактическое значение. Если эти значения равны, утверждение и тест пройдены. Если они не равны, утверждение и тест не пройдены.

1. Вызовите этот метод assertEquals(), а затем передайте переменные expectedTip и actualTip в качестве параметров:

import org.junit.Assert.assertEquals  
import org.junit.Test  
import java.text.NumberFormat  
  
class TipCalculatorTests {  
  
    @Test  
    fun calculateTip\_20PercentNoRoundup() {  
        val amount = 10.00  
        val tipPercent = 20.00  
        val expectedTip = NumberFormat.getCurrencyInstance().format(2)  
        val actualTip = calculateTip(amount = amount, tipPercent = tipPercent, false)  
        assertEquals(expectedTip, actualTip)  
    }  
}

**Примечание** . В библиотеке имеется множество утверждений JUnit. Некоторые распространенные утверждения, с которыми вы можете столкнуться:

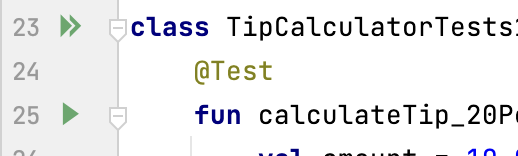
* assertEquals()
* assertNotEquals()
* assertTrue()
* assertFalse()
* assertNull()
* assertNotNull()
* assertThat()

Для получения дополнительной информации см. [Утверждение](https://developer.android.com/reference/junit/framework/Assert) .

## ****Запустить тест****

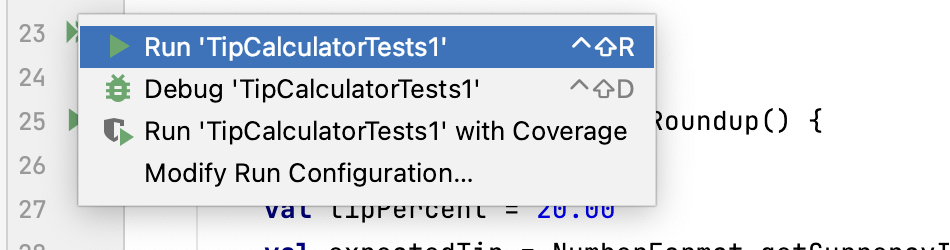
Теперь пришло время провести тест!

Возможно, вы заметили, что рядом с номером строки имени вашего класса и тестовой функции появляются стрелки. Вы можете щелкнуть эти стрелки, чтобы запустить тест. Если щелкнуть стрелку рядом с методом, вы запустите только этот метод тестирования. Если в классе есть несколько методов тестирования, вы можете щелкнуть стрелку рядом с классом, чтобы запустить все методы тестирования в этом классе.



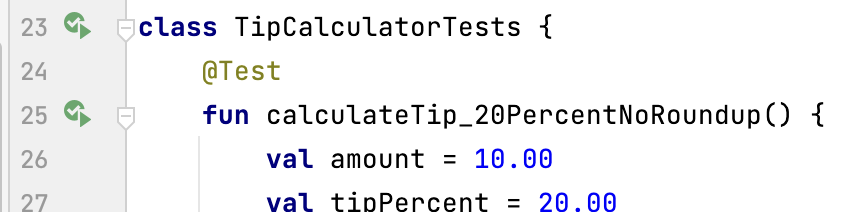
Запустите тест:

* Щелкните стрелки рядом с объявлением класса и выберите **«Run TipCalculatorTests»** .

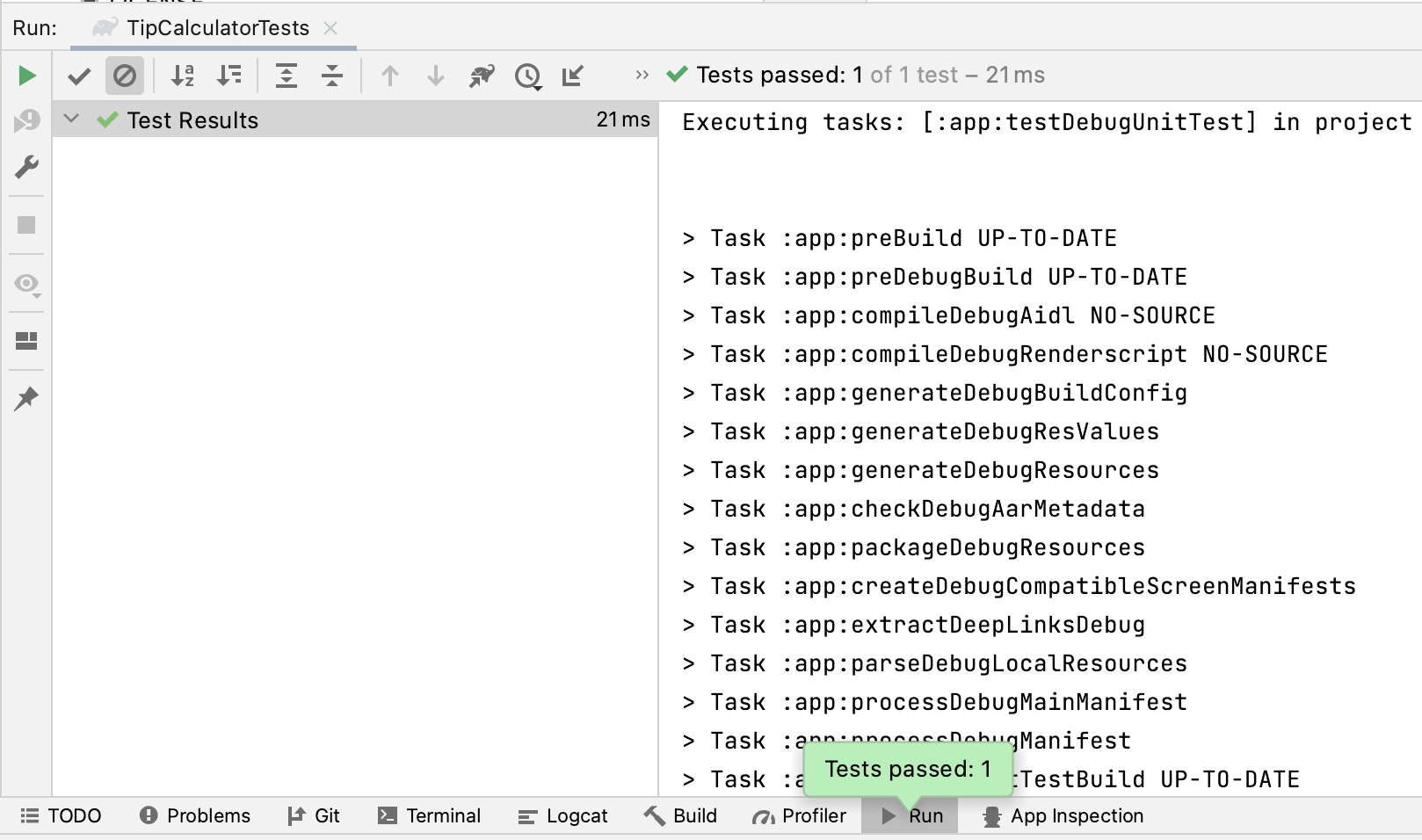


Вы должны увидеть следующие вещи:

* В желобе стрелки заменяются зеленой галочкой и треугольником . Это означает, что тест пройден.



* В нижней части панели **«Run»** вы увидите выходные данные.



* Признак того, что тесты пройдены.

## [5. Напишите инструментальный тест](https://developer.android.com/codelabs/basic-android-kotlin-compose-write-automated-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-write-automated-tests#4)

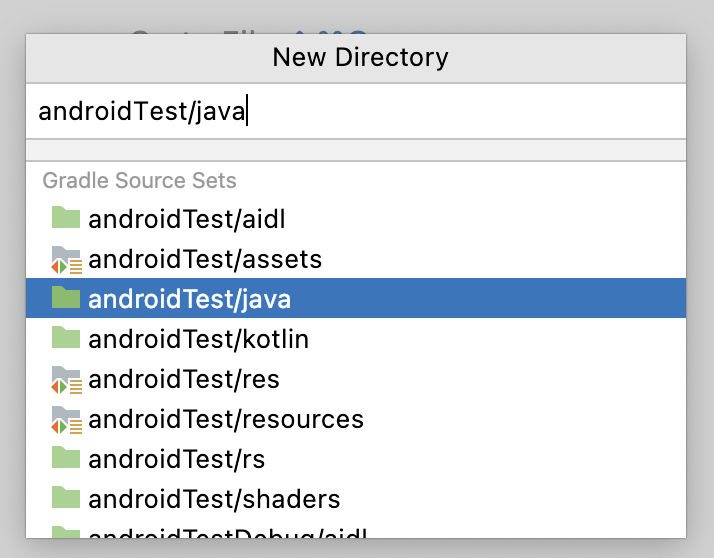
## Создайте каталог инструментов

Каталог инструментов создается аналогично локальному каталогу тестов.

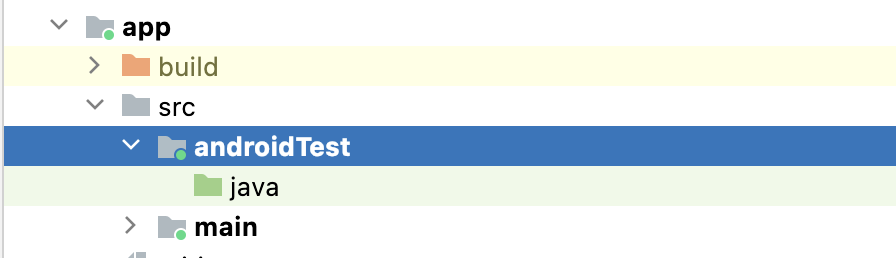
1. Щелкните правой кнопкой мыши каталог **src** и выберите **«New» > «Directory».**

## выбран пункт меню каталога

1. В окне **«New Directory»** выберите **androidTest/java** .



1. Введите клавишу **возврата** или **ввода** на клавиатуре. Каталог **androidTest** теперь можно увидеть на вкладке **«Project»** .



Точно так же, как каталоги main и test имеют одинаковую структуру пакетов, androidTest каталог должен содержать ту же самую структуру пакетов.

1. Щелкните правой кнопкой мыши папку **androidTest/java** и выберите **«New» > «Package».**
2. В окне **«New package»** введите com.example.tiptime.
3. Введите клавишу **возврата** или **ввода** на клавиатуре. Полную структуру пакета для androidTest каталога теперь можно увидеть на вкладке **«Project»** .

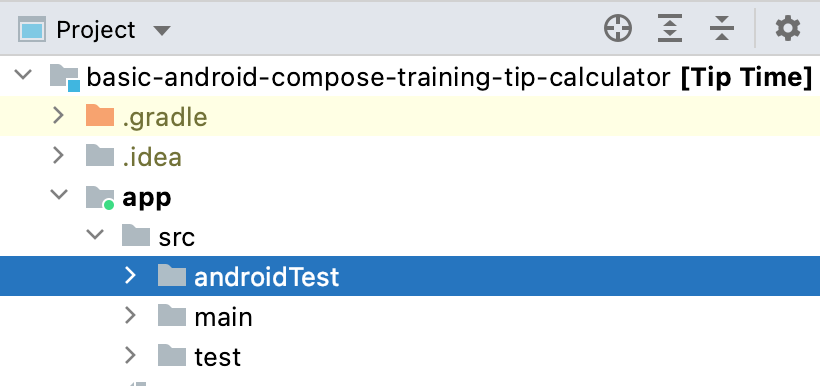
## ****Создайте тестовый класс****

В проектах Android каталог инструментальных тестов обозначается как androidTest каталог.

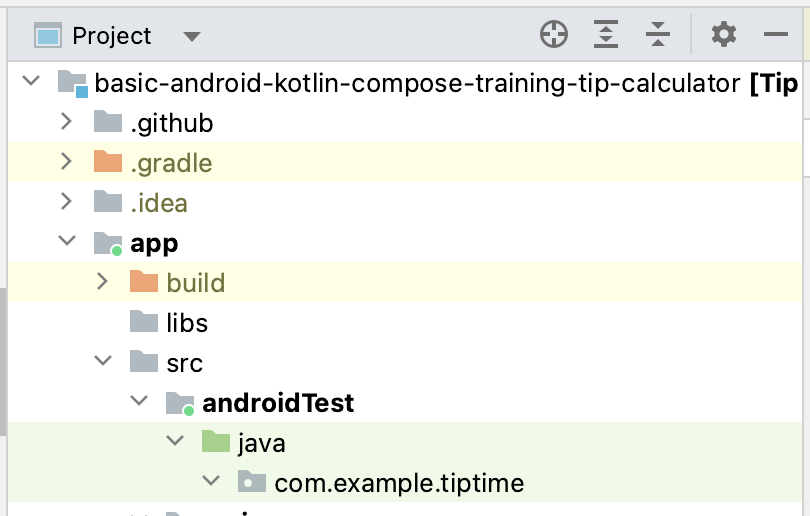
Чтобы создать инструментальный тест, вам необходимо повторить тот же процесс, который вы использовали для создания локального теста, но на этот раз создайте его внутри каталога androidTest.

Создайте тестовый класс:

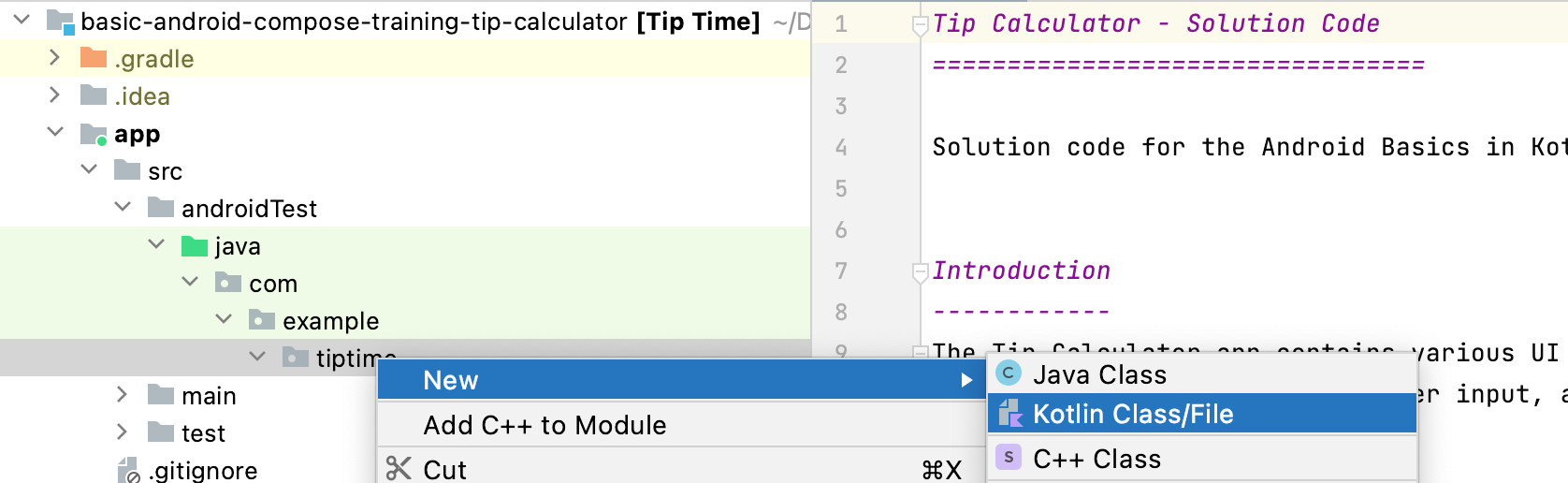
1. Перейдите в каталог androidTest на панели проекта.



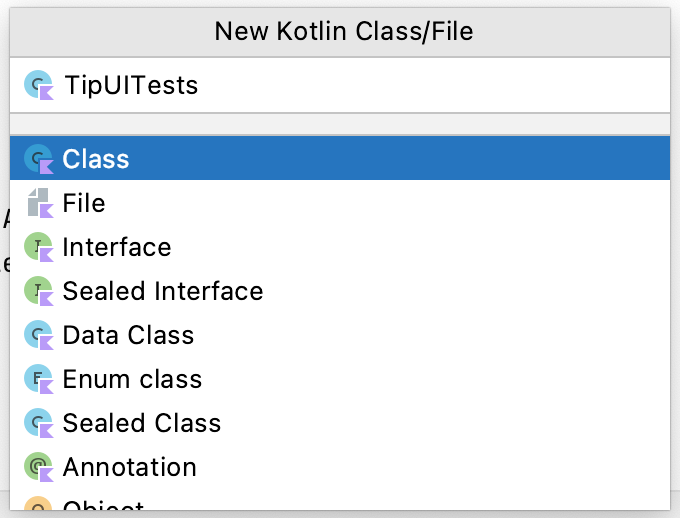
1. Нажимайте стрелки раскрытия рядом с каждым каталогом, пока не увидите нужный каталог tiptime.



1. Щелкните правой кнопкой мыши каталог tiptime и выберите **«New» > «Kotlin Class/file»** .



1. Введите TipUITests в качестве имени класса.



## Написать тест

Код инструментального тестирования сильно отличается от локального кода тестирования.

Инструментальные тесты проверяют реальный экземпляр приложения и его пользовательский интерфейс, поэтому содержимое пользовательского интерфейса должно быть установлено аналогично тому, как содержимое задается в  методе onCreate() файла MainActivity.kt, когда вы писали код для приложения Tip Time. Это необходимо сделать перед написанием всех инструментальных тестов для приложений, созданных с помощью Compose.

В случае с тестами приложения Tip Time вы переходите к написанию инструкций по взаимодействию с компонентами пользовательского интерфейса, чтобы процесс расчета чаевых тестировался через пользовательский интерфейс. Концепция инструментального теста на первый взгляд может показаться абстрактной, но не волнуйтесь! Этот процесс описан в следующих шагах.

Напишите тест:

1. Создайте переменную composeTestRule, соответствующую результату метода createComposeRule(),  и добавьте к ней аннотацию Rule:

import androidx.compose.ui.test.junit4.createComposeRule  
import org.junit.Rule  
  
class TipUITests {  
  
   @get:Rule  
   val composeTestRule = createComposeRule()  
}

1. Создайте метод calculate\_20\_percent\_tip() и аннотируйте его аннотацией @Test:

import org.junit.Test  
  
@Test  
fun calculate\_20\_percent\_tip() {  
}

Компилятор знает, что методы, помеченные аннотацией @Test в androidTest каталоге, относятся к инструментальным тестам, а методы, помеченные @Test аннотацией в test каталоге, относятся к локальным тестам.

1. В теле функции вызовите функцию composeTestRule.setContent(). Это устанавливает содержимое пользовательского интерфейса файла composeTestRule.
2. В теле лямбда-выражения функции вызовите функцию TipTimeTheme() с телом лямбда-выражения, которое вызывает функцию TipTimeLayout().

import com.example.tiptime.ui.theme.TipTimeTheme  
  
@Test  
fun calculate\_20\_percent\_tip() {  
    composeTestRule.setContent {  
        TipTimeTheme {  
           TipTimeLayout()  
        }  
    }  
}

Когда вы закончите, код должен выглядеть аналогично коду, написанному для установки содержимого метода onCreate() в файле MainActivity.kt. Теперь, когда содержимое пользовательского интерфейса настроено, вы можете написать инструкции для взаимодействия с компонентами пользовательского интерфейса приложения. В этом приложении вам необходимо проверить, что приложение отображает правильную сумму чаевых на основе введенной суммы счета и процента чаевых.

1. Доступ к компонентам пользовательского интерфейса можно получить как узлы через composeTestRule. Распространенный способ сделать это — получить доступ к узлу, содержащему определенный текст, с помощью метода onNodeWithText(). Используйте метод onNodeWithText() для доступа к TextFieldсоставной сумме счета:

import androidx.compose.ui.test.onNodeWithText  
  
@Test  
fun calculate\_20\_percent\_tip() {  
    composeTestRule.setContent {  
        TipTimeTheme {  
            TipTimeLayout()  
        }  
    }  
    composeTestRule.onNodeWithText("Bill Amount")  
}

Затем вы можете вызвать метод performTextInput() и передать текст, который вы хотите ввести, для заполнения  составного элемента TextField.

1. Заполните TextField сумму счета значением 10:

import androidx.compose.ui.test.performTextInput  
  
@Test  
fun calculate\_20\_percent\_tip() {  
    composeTestRule.setContent {  
        TipTimeTheme {  
            TipTimeLayout()  
        }  
    }  
    composeTestRule.onNodeWithText("Bill Amount")  
.performTextInput("10")  
}

1. Используйте тот же подход, чтобы заполнить OutlinedTextField процент чаевых значением 20:

@Test  
fun calculate\_20\_percent\_tip() {  
    composeTestRule.setContent {  
        TipTimeTheme {  
            TipTimeLayout()  
        }  
    }  
   composeTestRule.onNodeWithText("Bill Amount")  
.performTextInput("10")  
   composeTestRule.onNodeWithText("Tip Percentage").performTextInput("20")  
}

После того как все  составные элементы TextField заполнены, подсказка отображается в виде  составного элемента Text в нижней части экрана приложения.

Теперь, когда вы дали указание тесту заполнить эти  составные элементы TextField, вы должны убедиться, что  составной объект Text отображает правильную подсказку с утверждением.

В инструментальных тестах с помощью Compose утверждения можно вызывать непосредственно на компонентах пользовательского интерфейса. Доступно несколько утверждений, но в данном случае вы хотите использовать этот метод assertExists(). Ожидается, что составной  элемент Text, отображающий сумму чаевых, будет отображать: Tip Amount: $2.00.

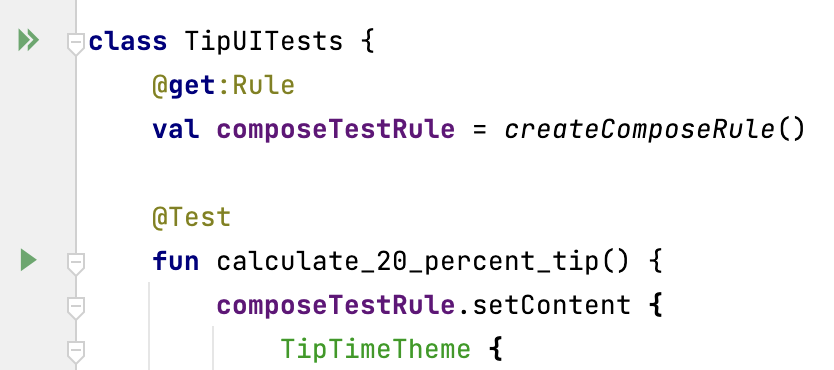
**Примечание** . Символ валюты может меняться в зависимости от языкового стандарта устройства. Использование ' $' является лишь примером. Валюта должна быть отформатирована в соответствии с локалью.

1. Сделайте утверждение, что узел с таким текстом существует:

import java.text.NumberFormat  
  
@Test  
fun calculate\_20\_percent\_tip() {  
    composeTestRule.setContent {  
        TipTimeTheme {  
            Surface (modifier = Modifier.fillMaxSize()){  
                TipTimeLayout()  
            }  
        }  
    }  
   composeTestRule.onNodeWithText("Bill Amount")  
      .performTextInput("10")  
   composeTestRule.onNodeWithText("Tip Percentage").performTextInput("20")  
   val expectedTip = NumberFormat.getCurrencyInstance().format(2)  
   composeTestRule.onNodeWithText("Tip Amount: $expectedTip").assertExists(  
      "No node with this text was found."  
   )  
}

## ****Запустить тест****

Процесс запуска инструментального теста такой же, как и для локального теста. Вы нажимаете стрелки в поле рядом с каждым объявлением, чтобы запустить отдельный тест или весь тестовый класс.



* Щелкните стрелки рядом с объявлением класса. Вы можете увидеть тесты, выполняемые на вашем устройстве или эмуляторе. Когда тест будет завершен, вы должны увидеть результат, который вы видите на этом изображении:

