[2. Introdução](https://developer.android.com/codelabs/android-basics-kotlin-write-unit-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-four&hl=pt-br" \l "1)

Agora que você já criou códigos para Android, podemos introduzir os códigos de teste. Primeiro, você vai aprender a teoria sobre testes, depois, os testes gerados automaticamente em projetos Androids vão ser apresentados e, por último, vai criar seus próprios testes para o app Dice Roller. Esta lição abrange um conteúdo extenso, mas não se preocupe. Estude este material quantas vezes achar necessário. O processo para aprender sobre testes leva bastante tempo e exige muito treinamento. Não desanime se você não pegar o jeito de primeira.

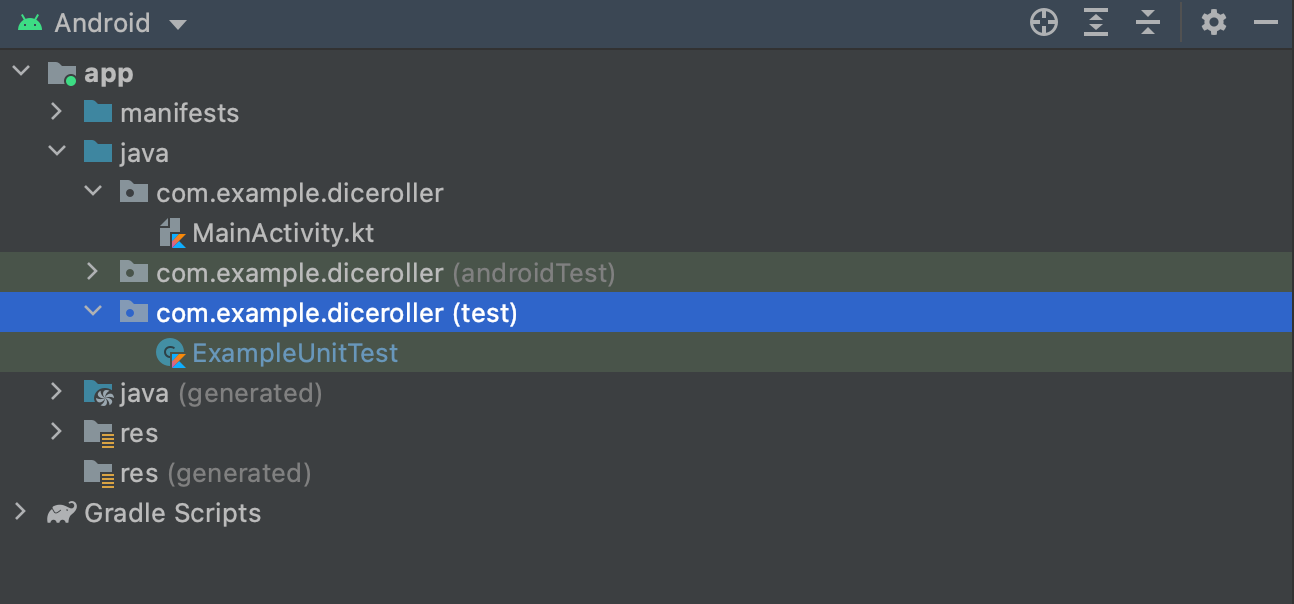
Por que os testes são importantes?

No início, pode parecer que você não precisa realizar testes no app. Quando o app é pequeno e tem funções limitadas, é fácil fazer testes manuais para determinar se tudo está funcionando da forma correta. Entretanto, à medida que o app cresce, a realização de testes manuais passa a exigir mais esforços do que a criação de testes automatizados. Além disso, ao trabalhar em apps de nível profissional, os testes passam a ser essenciais quando se tem uma grande base de usuários. É preciso considerar diversos tipos de dispositivos com diferentes versões do Android. Por fim, chega um momento em que os testes automatizados conseguem dar conta da maioria dos casos de uso de forma significativamente mais rápida do que os testes manuais. Quando você executa testes antes de lançar um código novo, é possível fazer modificações no código para evitar lançar um app com comportamentos inesperados. Os testes automatizados são executados por software, diferente dos manuais, que são realizados por uma pessoa que interage diretamente com um dispositivo. Os testes automatizados e manuais têm um papel essencial para garantir uma experiência agradável aos usuários do produto. No entanto, os testes automatizados podem ser mais precisos. Eles também otimizam a produtividade da equipe, porque não precisam de uma pessoa para serem feitos e porque podem ser executados de forma muito mais rápida do que um teste manual.

Uma análise mais detalhada dos testes de unidade

Neste codelab, vamos nos concentrar nos testes de unidade. Os testes de instrumentação vão ser abordados mais adiante. Para começar, vamos analisar os testes gerados ao criar um app Android no Android Studio. Você também vai ter uma experiência prática com os testes e aprender a criar códigos de teste.

Em um programa de aprendizagem anterior, você aprendeu onde encontrar os arquivos de origem para testes. Os testes de unidade ficam sempre localizados no diretório test:



1. Abra o arquivo app/build.gradle e observe as dependências. Você vai ver algumas dependências marcadas como testImplementation e androidTestImplementation, que correspondem aos testes de unidade e de instrumentação, respectivamente. É importante observar:

app/build.gradle

testImplementation 'junit:junit:4.12'

A biblioteca JUnit aciona os testes de unidade e permite marcar um código como sendo de teste para que ele possa ser compilado e executado de forma a testar o código de apps.

1. No diretório test, abra o arquivo ExampleUnitTest.kt.

Você vai ver um teste de unidade de exemplo, como este:

ExampleUnitTest.kt

class ExampleUnitTest {  
   @Test  
   fun addition\_isCorrect() {  
       assertEquals(4, 2 + 2)  
   }  
}

Embora você tenha adicionado código ao app Dice Roller, é provável que não tenha programado nenhum teste. O único teste existente é gerado por um código genérico criado automaticamente pelo Android Studio. Esse é um teste arbitrário que serve como um marcador de posição para testes mais relevantes que precisam ser programados pelo desenvolvedor. No momento, esse bloco de código testa somente se 2 + 2 = 4. É claro que isso é sempre verdade. Vamos analisar com mais atenção o que está acontecendo:

* As funções de teste precisam ter a anotação @ Test, importada da biblioteca org.junit.test. Considere anotações como tags de metadados para uma parte do código que pode mudar a forma como o código é compilado. Nesse caso, a anotação @Test informa ao compilador que o método a seguir é um teste, o que permite que ele seja executado como tal.

Após a anotação, há uma declaração de função. Neste caso, a função addition\_isCorrect(). Nessa função, a função assertEquals() declara que um valor esperado precisa ser igual a um valor real recebido pela lógica de negócios. Os métodos de declaração são o objetivo final de um teste de unidade. Em última instância, é importante declarar que um resultado recebido do código está em um estado específico. Se o estado do resultado corresponder ao estado esperado, o teste vai ser aprovado. Se o estado do resultado não corresponder ao estado esperado, o teste vai falhar. Nesse caso, o código está comparando dois valores. Portanto, o método assertEquals() considera dois parâmetros: um valor esperado e um valor real. Fazendo jus ao nome, o valor esperado é aquele que você espera de um resultado específico. Neste caso, 4. O valor real representa o resultado gerado de uma parte do código. Geralmente, uma parte do código do próprio app seria testada. Nesse caso, estamos testando apenas uma parte arbitrária do código, por exemplo, 2 + 2. Por fim, execute o teste para ver o que acontece.

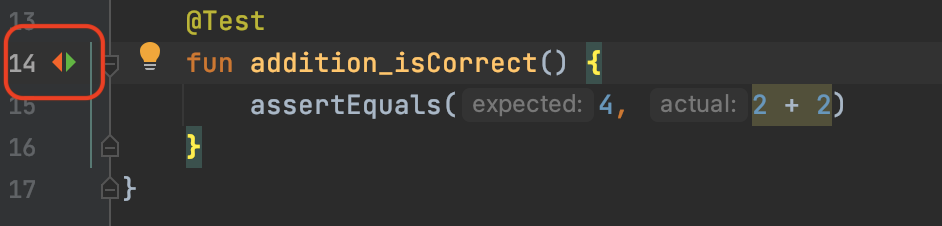
**Observação**: há muitas declarações na biblioteca JUnit. Veja algumas declarações comuns que você pode encontrar:

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

For more information, see [Assert](https://developer.android.com/reference/junit/framework/Assert?hl=pt-br).

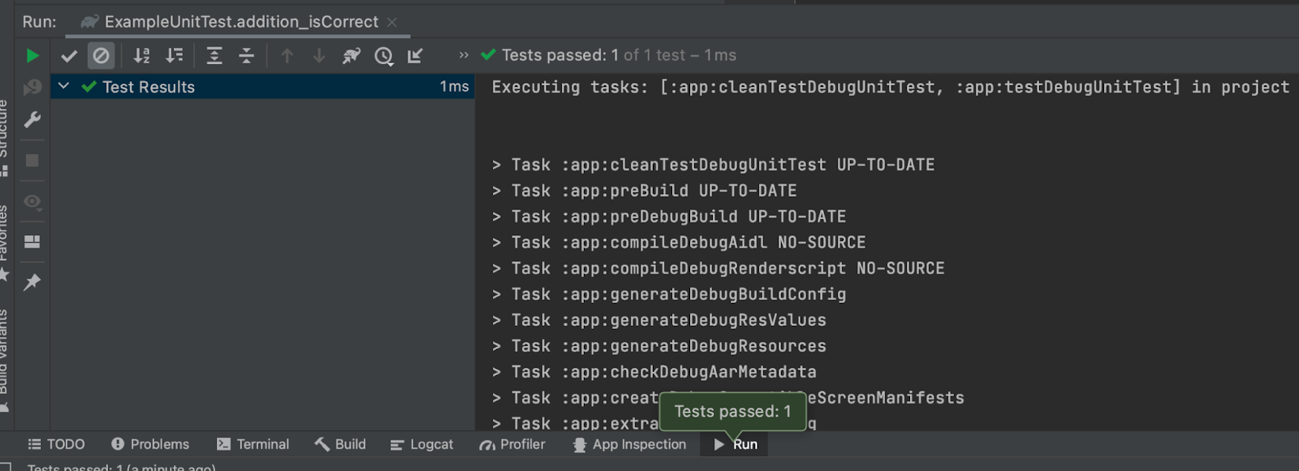
Há várias maneiras de executar testes no Android Studio, que vão ser abordadas mais tarde. Por enquanto, vamos simplificar.

1. Ao lado da declaração do método addition\_isCorrect, clique nas setas e selecione **Run ‘ExampleUnitTest.addition\_isCorrect'**.

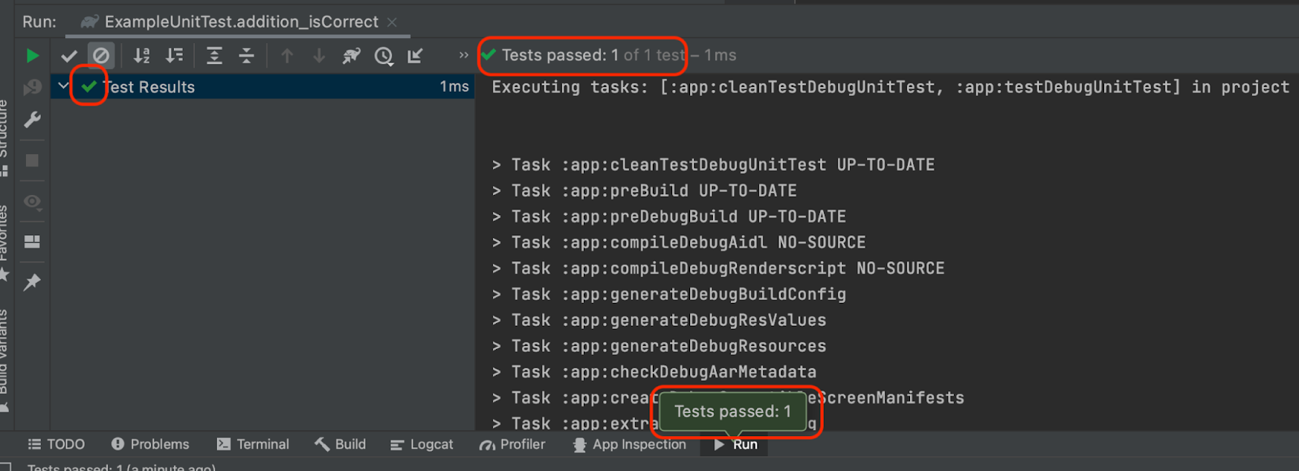


Isso é chamado de teste positivo. Em outras palavras, a declaração é afirmativa. 2 + 2 é igual a 4. Como alternativa, criaríamos um teste negativo que faz uma declaração negativa. Por exemplo, 2 + 2 não é igual a 5.

No painel **Run**, você vai ver algo parecido com esta captura de tela:



Há várias indicações de que o teste foi concluído, como marcas de seleção verde e o número de testes aprovados.

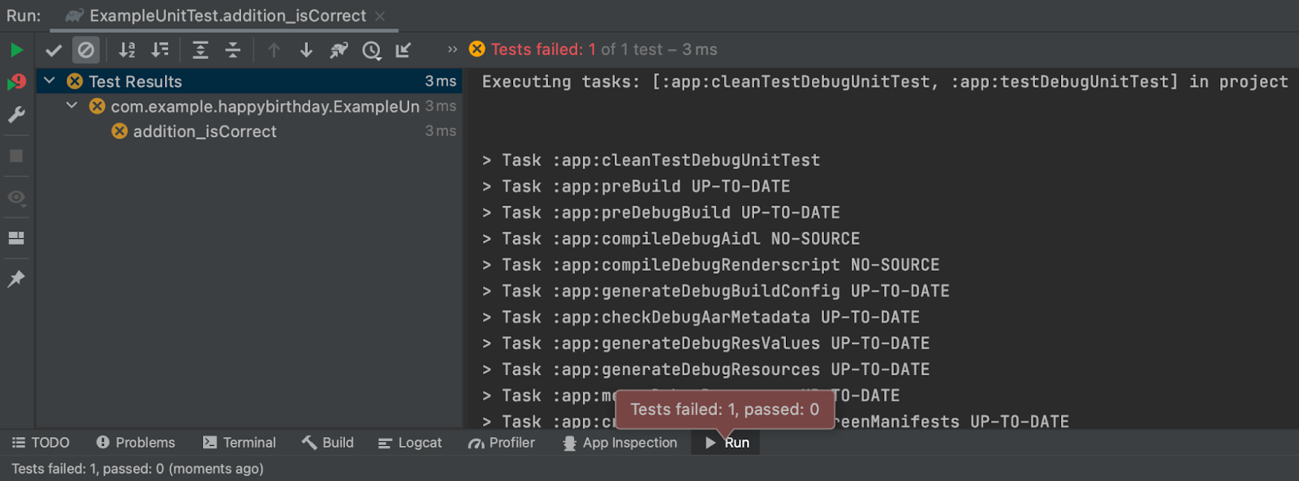


1. Modifique o teste para ver o que acontece quando há uma falha. Mude 2 + 2 para 2 + 3 e execute o teste novamente. Por enquanto, você está apenas fazendo experimentos com o código gerado para ganhar experiência com os testes. Essas mudanças não têm relevância para as funções do app Dice Roller.

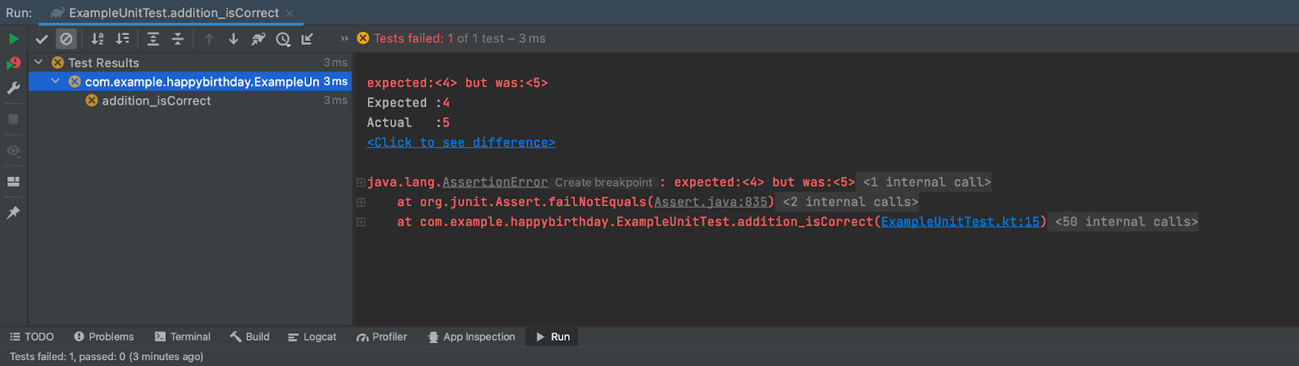
ExampleUnitTest.kt

class ExampleUnitTest {  
   @Test  
   fun addition\_isCorrect() {  
       assertEquals(4, 2 + 3)  
   }  
}

Depois de executar o teste modificado, você vai ver algo assim:



O texto em vermelho indica uma falha no teste. No menu de resultados do teste, clique nos itens para ver uma mensagem de erro indicando o motivo da falha.



Nesse caso, a mensagem indica que a declaração falhou porque esperava um resultado 4, mas o valor real foi 5. Isso faz sentido, já que você mudou o valor real para 2 + 3, mas deixou o valor esperado como 4. Também é possível ver a linha em que o teste falhou. Neste caso, a falha se encontra na linha 15, marcada como ExampleUnitTest.kt:15.

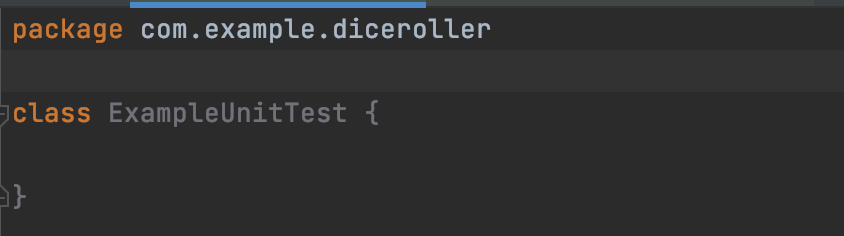
1. Para uma análise mais minuciosa, mude o valor esperado de 4 para 5 e execute o teste novamente. O teste vai ser aprovado quando o valor esperado corresponder ao resultado real do código em questão.

[3. Criar seu primeiro teste de unidade](https://developer.android.com/codelabs/android-basics-kotlin-write-unit-tests?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-four&hl=pt-br#2)

Agora que já se familiarizou com os testes de unidade, você pode criar um teste de unidade próprio que seja mais relevante para o app Dice Roller.

Como já observado, a principal função do app Dice Roller tem como base um gerador de números aleatórios. Infelizmente, é muito difícil testar geradores de números aleatórios, porque não é possível ter certeza do resultado de um número gerado aleatoriamente. O objetivo desse teste é garantir que, ao jogar o dado ou chamar o método roll na classe dice, você receba um número adequado. O teste criado verifica apenas se a saída do gerador de números aleatórios é um número dentro do intervalo especificado para o gerador.

1. No arquivo ExampleUnitTest.kt, exclua o método de teste gerado e as instruções de importação. O arquivo vai ficar assim:



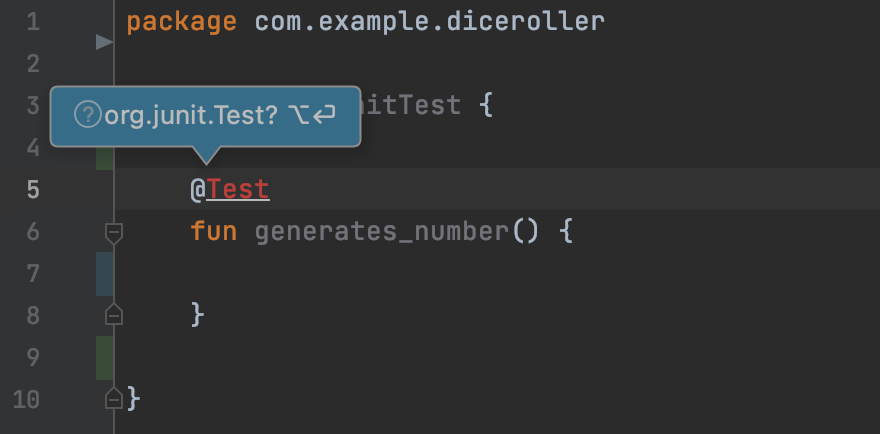
1. Crie uma função generates\_number():

ExampleUnitTest.kt

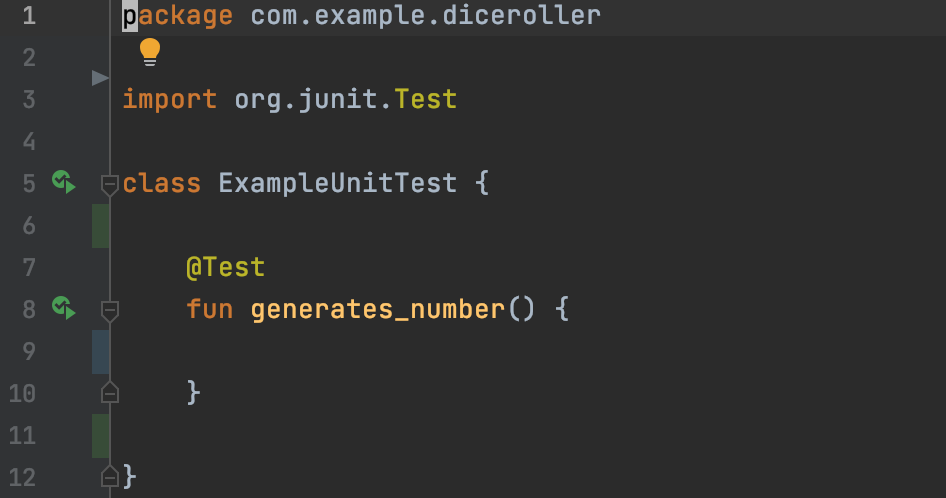
fun generates\_number() {  
}

1. Adicione a anotação @Test ao método generates\_number(). Quando você tentar chamar @Test, o texto vai ficar vermelho. Isso ocorre porque não é possível encontrar a declaração dessa anotação e, portanto, é necessário a importar. Isso pode ser feito de forma automática ao pressionar Control+Enter (ou Options+Return no Mac).

Se você clicar na linha de código, vai aparecer uma solicitação de importação:



Também é possível copiar e colar o arquivo import org.junit.Test depois do nome do pacote, mas antes da declaração de classes. Agora, o código vai ficar assim:



1. Crie uma instância do objeto Dice.

ExampleUnitTest.kt

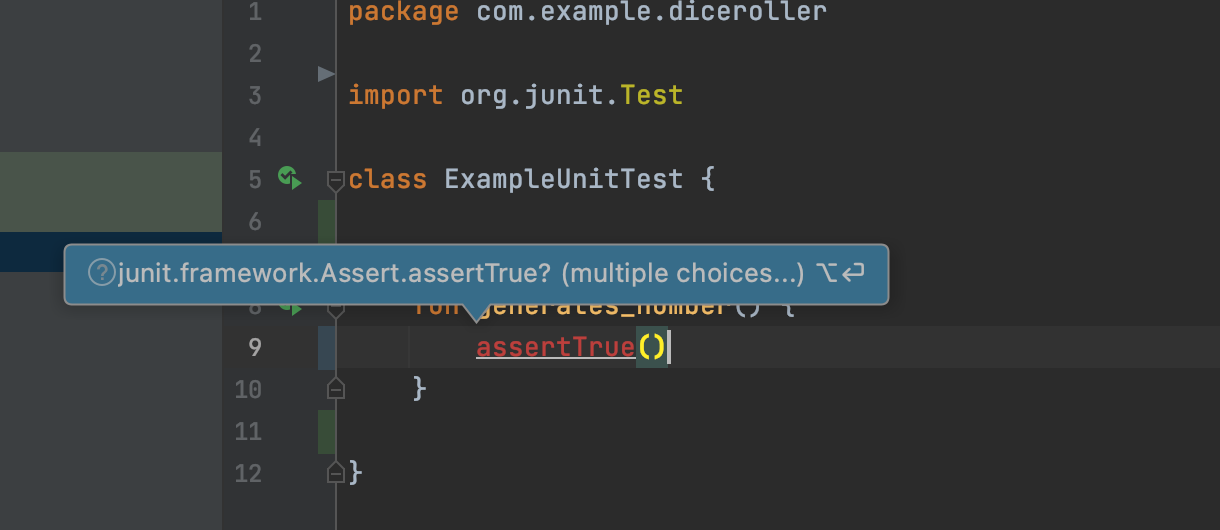
@Test  
fun generates\_number() {  
   val dice = Dice(6)  
}

1. Em seguida, chame o método roll() nessa instância e armazene o valor retornado.

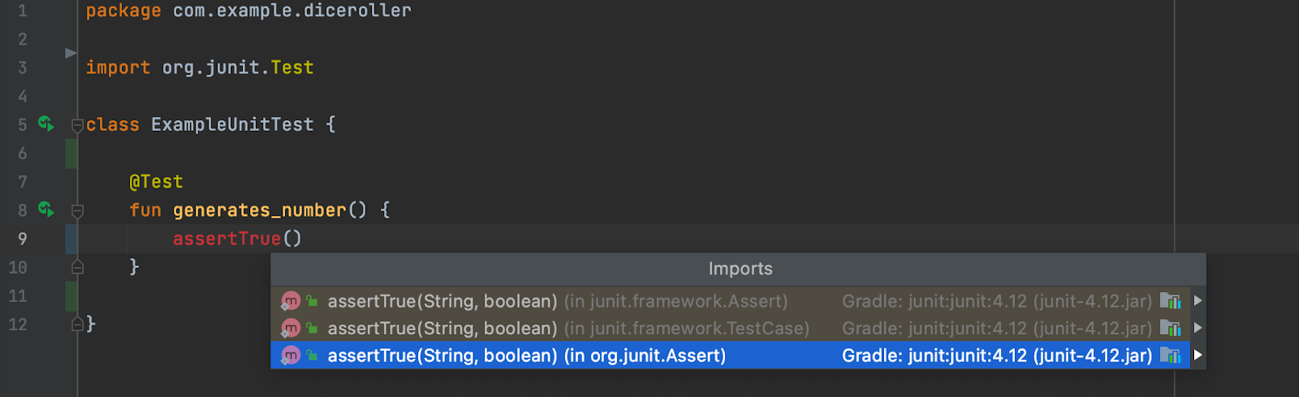
ExampleUnitTest.kt

@Test  
fun generates\_number() {  
   val dice = Dice(6)  
   val rollResult = dice.roll()  
}

1. Por fim, faça uma declaração real. Em outras palavras, é necessário declarar que o método retornou um valor dentro do limite de número de lados transmitido. Nesse caso, o valor precisa ser maior que 0 e menor que 7. Para fazer isso, use o método [assertTrue()](https://developer.android.com/reference/junit/framework/Assert?hl=pt-br" \l "assertTrue(java.lang.String,%20boolean)" \t "_blank). Quando você tenta chamar o método assertTrue() da primeira vez, o texto fica vermelho. Isso ocorre porque não é possível encontrar a declaração desse método. Dessa forma, é necessário o importar, de modo semelhante ao que foi feito com a anotação.



Ele pode ser importado de forma automática, como apresentado anteriormente. Observe, no entanto, que desta vez você tem várias opções. Neste caso, você pode selecionar a opção do pacote org.junit.Assert:

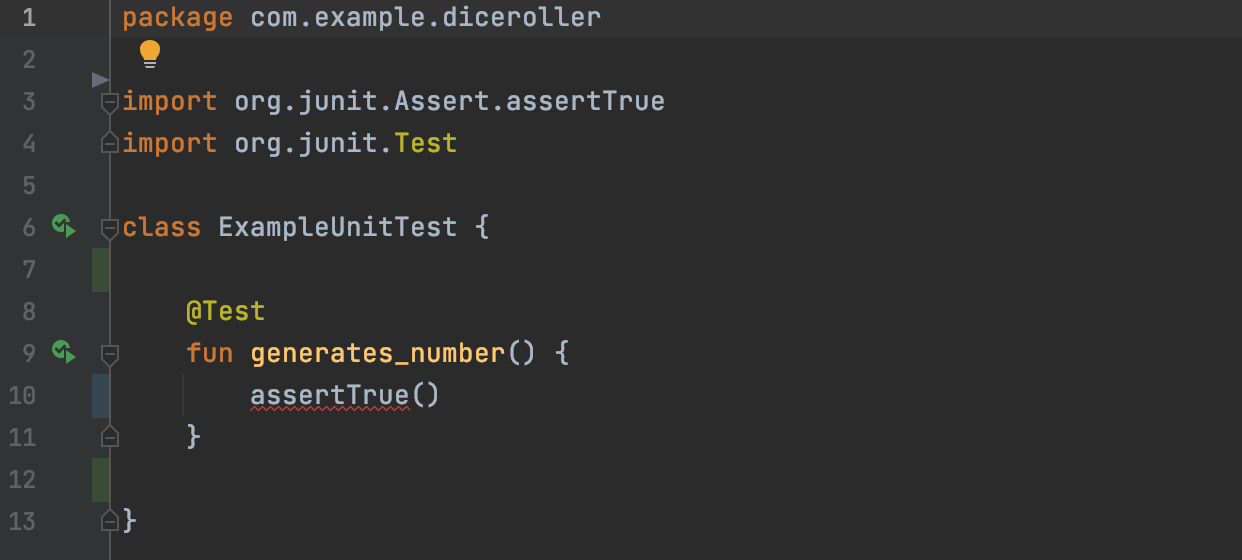


Também é possível colar o código depois da instrução de importação para a anotação de teste:

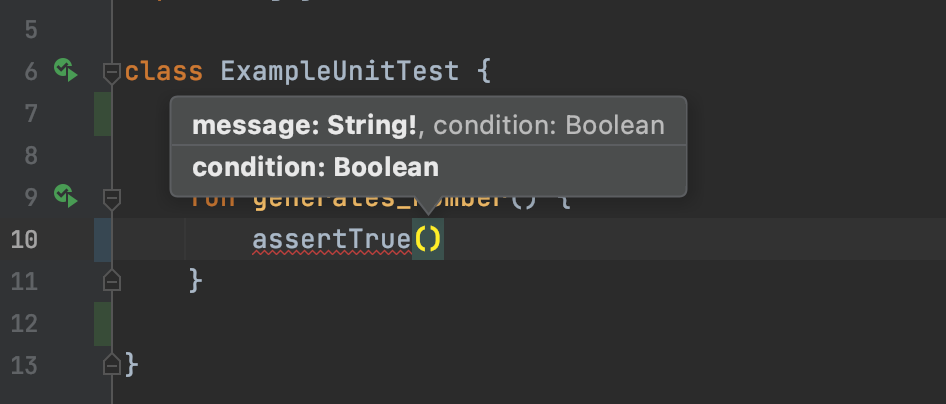
ExampleUnitTest.kt

import org.junit.Assert.assertTrue

O código vai ficar assim:



Ao colocar o cursor entre os parênteses e pressionar Control+P (ou Command+P no Mac), você vai ver uma dica que mostra quais parâmetros o método usa:



O método assertTrue() usa dois parâmetros, uma String e um Boolean. Se a declaração falhar, a string vai ser uma mensagem exibida no console. O valor booleano é uma instrução condicional. Defina a mensagem como:

ExampleUnitTest.kt

"The value of rollResult was not between 1 and 6"

Conforme mencionado anteriormente, testar números aleatórios é um desafio porque, devido à natureza da aleatoriedade, o valor não pode ser previsto. Só é possível garantir que o valor esteja dentro de um intervalo específico. Defina o parâmetro de condição como:

ExampleUnitTest.kt

rollResult in 1..6

O código vai ficar assim:

ExampleUnitTest.kt

@Test  
fun generates\_number() {  
   val dice = Dice(6)  
   val rollResult = dice.roll()  
   assertTrue("The value of rollResult was not between 1 and 6", rollResult in 1..6)  
}

1. Clique nas setas ao lado da função e selecione **Run ‘ExampleUnitTest.generates\_number()'**.

Se o resultado for semelhante ao snippet de código anterior, o teste vai ser aprovado.

1. Opcional: para praticar mais, modifique o dado de forma que ele tenha somente quatro ou cinco lados sem mudar a declaração para ver o teste falhar.