



Testing

JUnit y Mockito

{desafío}
latam_



***Implementar tests unitarios
y de instrumentación para la
verificación del buen
funcionamiento de los
componentes de un
proyecto Android.***

- Unidad 1:
Acceso a datos en Android
- Unidad 2:
Consumo de API REST
- Unidad 3:
Testing
- Unidad 4:
Distribución del aplicativo Android



Te encuentras aquí



¿Qué aprenderás en esta sesión?

- *JUnit y Mockito*

¿Has escuchado el
término “Mock” antes?
¿Lo relacionas con algo?

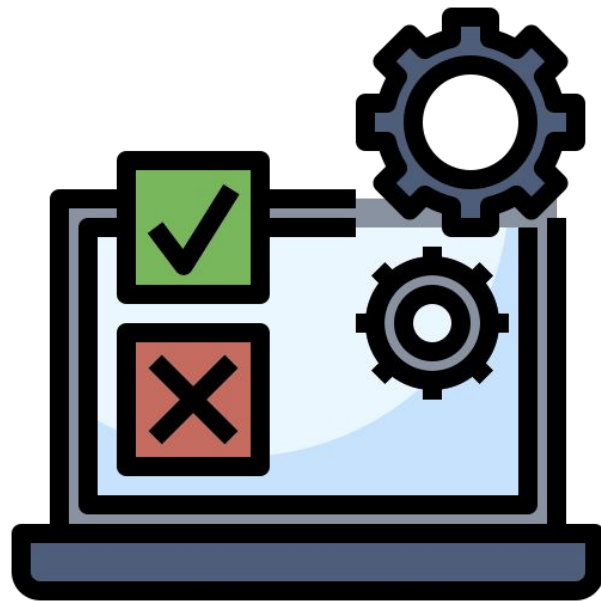


`/* Qué es JUnit y Mockito */`

¿Qué es JUnit?

JUnit es un marco de prueba de unidad de código abierto popular para Java (y otros lenguajes basados en JVM como Kotlin) que permite a los desarrolladores escribir y ejecutar pruebas automatizadas.

Proporciona un conjunto de anotaciones, aserciones y ejecutores de pruebas que facilitan la escritura y organización de pruebas, así como herramientas para informar los resultados de las pruebas y depurar fallas.



¿Qué es JUnit?

- Las pruebas JUnit generalmente se escriben como métodos dentro de una clase, y cada método prueba una pieza específica de código (como una función o un método de clase) y usa la anotación `@Test` para indicar que es un método de prueba. JUnit también proporciona otras anotaciones, como `@Before` y `@After`, que permiten a los desarrolladores configurar y desmontar accesorios de prueba (como crear objetos simulados o configurar una base de datos de prueba) antes y después de ejecutar cada método de prueba.
- En general, JUnit es un marco de prueba poderoso y ampliamente utilizado que ha ayudado a revolucionar la forma en que se prueba y desarrolla el software en el ecosistema de Java.

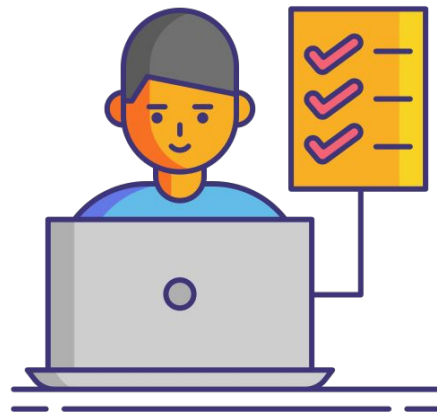
¿Qué es Mockito?

- Mockito es un marco de simulación de código abierto popular para Java (y otros lenguajes basados en JVM como Kotlin) que permite a los desarrolladores crear y usar objetos simulados en sus pruebas.
- Un objeto simulado es un objeto falso que imita el comportamiento de un objeto real, pero se puede personalizar para simular escenarios o entradas específicas.
- Mockito proporciona un conjunto de API y anotaciones que facilitan la creación y el uso de objetos simulados en las pruebas. Permite a los desarrolladores definir el comportamiento de un objeto simulado utilizando métodos como `when()`, que especifica lo que debe hacer el objeto simulado cuando se llama a un determinado método, y `thenReturn()`, que especifica qué valor debe devolver el objeto simulado.

¿Qué es Mockito?

Mockito también proporciona herramientas para verificar que se hayan producido ciertos métodos o interacciones con un objeto simulado durante una ejecución de prueba, utilizando métodos como `verify()` y `times()`. Esto permite a los desarrolladores asegurarse de que su código interactúe con los objetos correctamente y como se espera.

En general, Mockito ayuda a aislar el código bajo prueba de las dependencias externas y proporciona una herramienta poderosa para simular escenarios e insumos específicos durante la prueba.



Diferencias entre JUnit y Mockito

JUnit y Mockito, ambos son frameworks de prueba, pero tienen diferentes propósitos, más bien, están diseñados para usarse juntos.

JUnit es un marco de pruebas unitarias que se utiliza para escribir y ejecutar pruebas automatizadas para código Java o Kotlin en nuestro caso.

Mockito, por otro lado, es un marco de simulación que se usa para crear y usar objetos simulados en las pruebas.

Si bien JUnit y Mockito tienen diferentes propósitos, a menudo se usan juntos en las pruebas de Java o Kotlin. JUnit se usa para escribir y ejecutar las pruebas, mientras que Mockito se usa para crear y usar objetos simulados para aislar el código que se está probando y simular escenarios específicos. Al combinar estos dos marcos, los desarrolladores pueden escribir pruebas más efectivas y eficientes que ayuden a garantizar la calidad de su código.

Demostración

"Test unitario usando JUnit y Mockito"



JUnit y Mockito

Supongamos que tenemos una clase **Calculator** simple que realiza operaciones aritméticas básicas:

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



JUnit y Mockito

Para probar la clase **Calculator**, podemos crear una nueva clase de prueba de Kotlin e importar **JUnit** y **Mockito**:

```
import org.junit.Test
import org.junit.Assert.assertEquals
import org.mockito.Mock
import org.mockito.Mockito.`when`
import org.mockito.MockitoAnnotations
```

A continuación, podemos definir nuestra clase de prueba y crear una instancia de **Calculator** para probar:

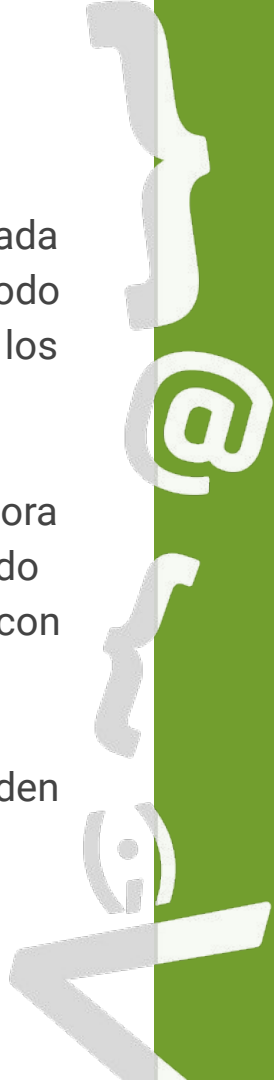


```
class CalculatorTest {  
  
    @Mock  
    lateinit var calculator: Calculator  
  
    @Before  
    fun setup() {  
        MockitoAnnotations.initMocks(this)  
    }  
  
    @Test  
    fun testAddition() {  
        `when`(calculator.add(2, 3)).thenReturn(5)  
        val result = calculator.add(2, 3)  
        assertEquals(result, 5)  
    }  
  
    @Test  
    fun testSubtraction() {  
        `when`(calculator.subtract(7, 4)).thenReturn(3)  
  
        val result = calculator.subtract(7, 4)  
        assertEquals(result, 3)  
    }  
}
```



JUnit y Mockito - Explicación

- En este ejemplo, usamos Mockito para crear una instancia de calculadora simulada y definir el comportamiento de sus métodos de suma y resta mediante el método `when()`. Luego llamamos a estos métodos y verificamos que los resultados son los que esperamos usando el método `assertEquals()` de JUnit.
- Ten en cuenta que también tenemos que inicializar la instancia de la Calculadora simulada usando la anotación `@Mock` y el método `MockitoAnnotations.initMocks(this)` en el método `setup()`, que se anota con `@Before`.
- Este es solo un ejemplo simple, pero demuestra cómo JUnit y Mockito se pueden usar juntos para probar el código de Android escrito en Kotlin.



Demostración

"JUnit y Mockito, resultados"



Supongamos que tenemos la siguiente clase y queremos crear su correspondiente Unit Test, pero esta vez queremos mostrar los resultados de esos test, podemos hacer lo siguiente:

```
class Calculator {  
    fun add(a: Int, b: Int): Int = a + b  
  
    fun subtract(a: Int, b: Int): Int = a - b  
  
    fun multiply(a: Int, b: Int): Int = a * b  
  
    fun divide(a: Int, b: Int): Int {  
        if (b == 0) {  
            throw IllegalArgumentException("Cannot divide by  
zero")  
        }  
        return a / b  
    }  
}
```



La clase de Test, muy similar al ejemplo anterior, pero con algunas funciones más:

```
class CalculatorTest {  
  
    private lateinit var calculator:  
    Calculator  
  
    @Before  
    fun setUp() {  
        calculator = Calculator()  
    }  
  
    @Test  
    fun testAdd() {  
        val result = calculator.add(2,  
3)        assertEquals(5, result)  
    }  
}
```

{desafío}
latam_

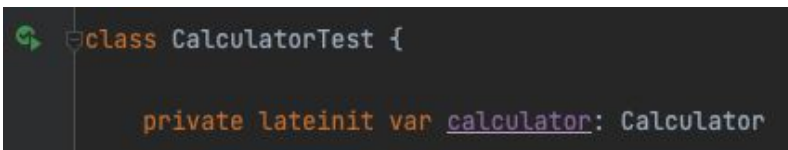
```
@Test  
fun testSubtract() {  
    val result = calculator.subtract(5, 3)  
    assertEquals(2, result)  
}  
  
@Test  
fun testMultiply() {  
    val result = calculator.multiply(2, 3)  
    assertEquals(6, result)  
}  
  
@Test  
fun testDivide() {  
    val result = calculator.divide(6, 3)  
    assertEquals(2, result)  
}  
  
@Test(expected = IllegalArgumentException::class)  
fun testDivideByZero() {  
    calculator.divide(6, 0)  
}  
}
```

Vimos anteriormente que los test los puedes ejecutar a través de una línea de comando:

```
user@machine UnitTest % ./gradlew test

BUILD SUCCESSFUL in 2s
45 actionable tasks: 7 executed, 38 up-to-date
user@machine UnitTest %
```

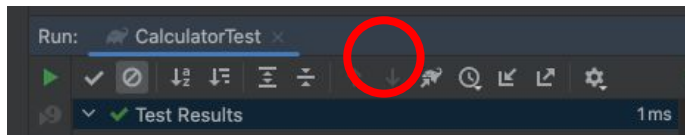
También puedes ejecutar los test presionando la flecha verde al lado de la clase o del Test que quieres ejecutar:



```
class CalculatorTest {

    private lateinit var calculator: Calculator
```

Finalmente, puedes generar un reporte de los test ejecutados y sus resultados, presionando el icono de Gradle “Open Gradle Test Report”:



El resultado será algo similar al siguiente reporte:

Class com.desafiolatam.unittest.CalculatorTest

all > [com.desafiolatam.unittest](#) > CalculatorTest

5	0	0	0.001s
tests	failures	ignored	duration

100%
successful

Tests

Test	Duration	Result
testAdd	0s	passed
testDivide	0s	passed
testDivideByZero	0.001s	passed
testMultiply	0s	passed
testSubtract	0s	passed

Simular el entorno en el que se ejecuta una función es parte fundamental de los Unit Test ¿Puedes decirnos por qué?





Próxima sesión...

- *Anotaciones*
- *True(assert library)*
- *Test de integración*
- *Room test*
- *Espresso IdlingResources*
- *Otras alternativas para testing*

{desafío}
latam_

*Academia de
talentos digitales*

