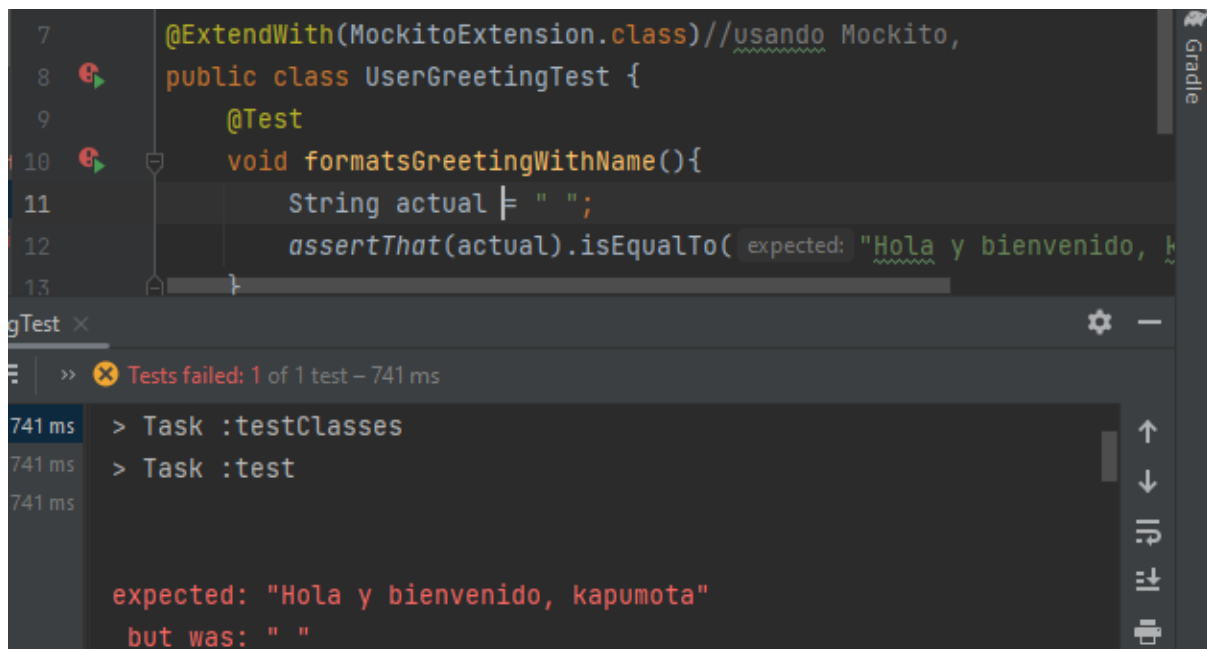


## Escribir un stub con Mockito

2) Este es el uso estándar de los frameworks JUnit y AssertJ como hemos visto antes.



```
7      @ExtendWith(MockitoExtension.class) // usando Mockito,
8      public class UserGreetingTest {
9
10         @Test
11         void formatsGreetingWithName(){
12             String actual = " ";
13             assertThat(actual).isEqualTo( expected: "Hola y bienvenido, kapumota", but was: " ")
14         }
15     }
```

Test failed: 1 of 1 test – 741 ms

741 ms > Task :testClasses

741 ms > Task :test

741 ms

expected: "Hola y bienvenido, kapumota"

but was: " "

**Problema:** ¿Qué sucede si ejecutas la prueba ahora?.

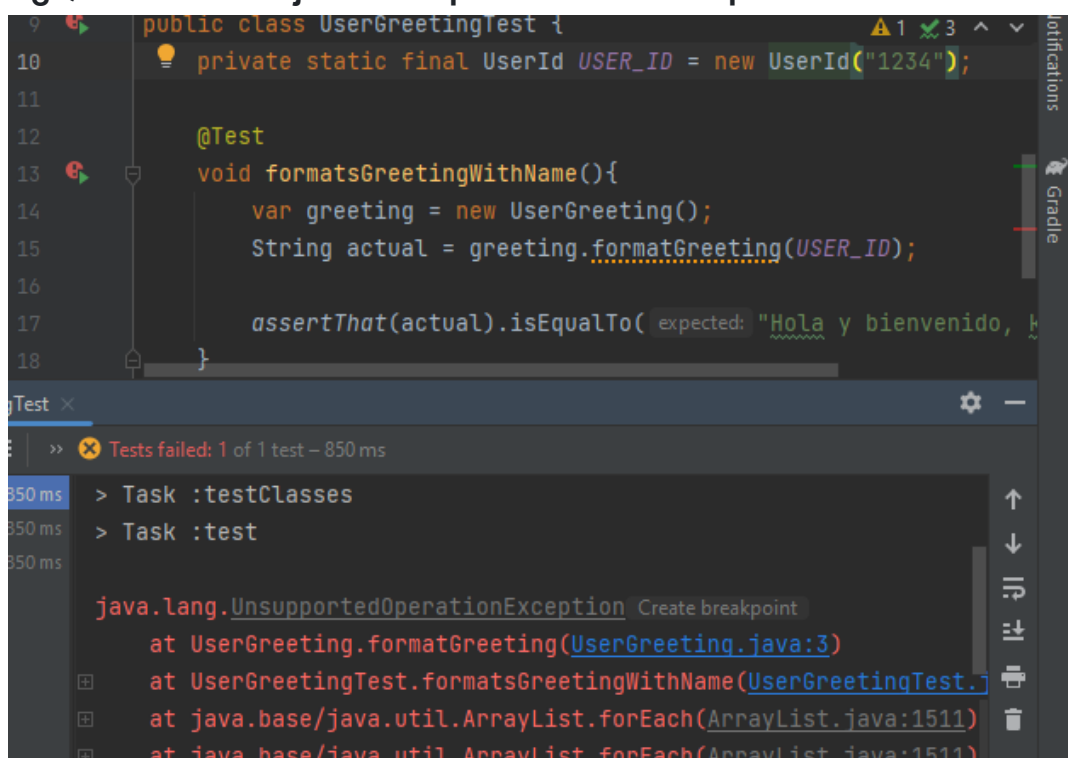
La prueba falla, porque no concuerda lo actual con lo esperado.

3) Elimina el SUT, la clase que queremos escribir, con un paso Act:

4) Agrega un esqueleto de clase `UserGreeting`:

5) Agrega un esqueleto de clase `UserId`:

**Problema:** ¿Qué sucede si ejecutas la prueba ahora?. Explica la salida.



```
9      public class UserGreetingTest {
10          private static final UserId USER_ID = new UserId("1234");
11
12          @Test
13          void formatsGreetingWithName(){
14              var greeting = new UserGreeting();
15              String actual = greeting.formatGreeting(USER_ID);
16
17              assertThat(actual).isEqualTo( expected: "Hola y bienvenido, kapumota", but was: " ")
18          }
19      }
```

Test failed: 1 of 1 test – 850 ms

850 ms > Task :testClasses

850 ms > Task :test

850 ms

java.lang.UnsupportedOperationException Create breakpoint

at UserGreeting.formatGreeting(UserGreeting.java:3)

at UserGreetingTest.formatsGreetingWithName(UserGreetingTest.java:15)

at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)

at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)

Me lanza una excepcion, como estoy instanciando a la clase `UserGreeting` y de ella su metodo `formatGreeting` aun no esta implementado (solo hay una excepci3n).

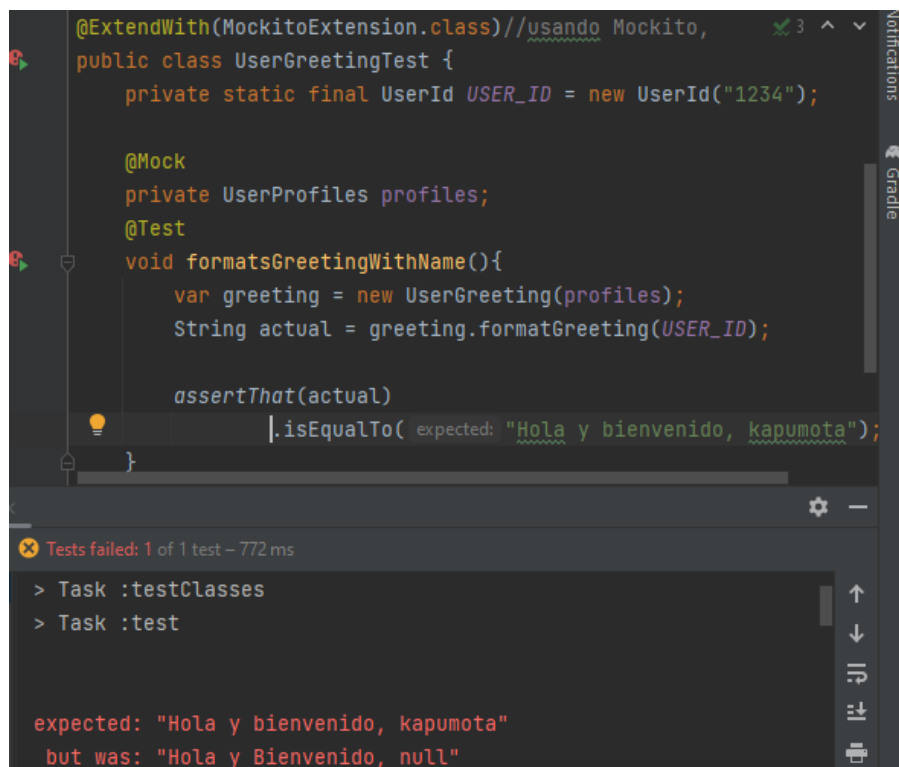
6) Otra decisi3n de dise1o a capturar es que la clase `UserGreeting` depender3 de una interfaz `UserProfiles`. Necesitamos crear un campo, crear el esqueleto de la interfaz e inyectar el campo en un nuevo constructor para el SUT

7) Agrega comportamiento al m3todo `formatGreeting()`:

8) Agrega `fetchNicknameFor()` a la interfaz `UserProfiles`.

9) Ejecute la prueba. 3Qu3 sucede?.

10) Agrega la anotaci3n `@Mock` al campo `profiles`:



```
@ExtendWith(MockitoExtension.class) // usando Mockito,
public class UserGreetingTest {
    private static final UserId USER_ID = new UserId("1234");

    @Mock
    private UserProfiles profiles;
    @Test
    void formatsGreetingWithName(){
        var greeting = new UserGreeting(profiles);
        String actual = greeting.formatGreeting(USER_ID);

        assertThat(actual)
            .isEqualTo( expected: "Hola y bienvenido, kapumota");
    }
}
```

Tests failed: 1 of 1 test - 772 ms

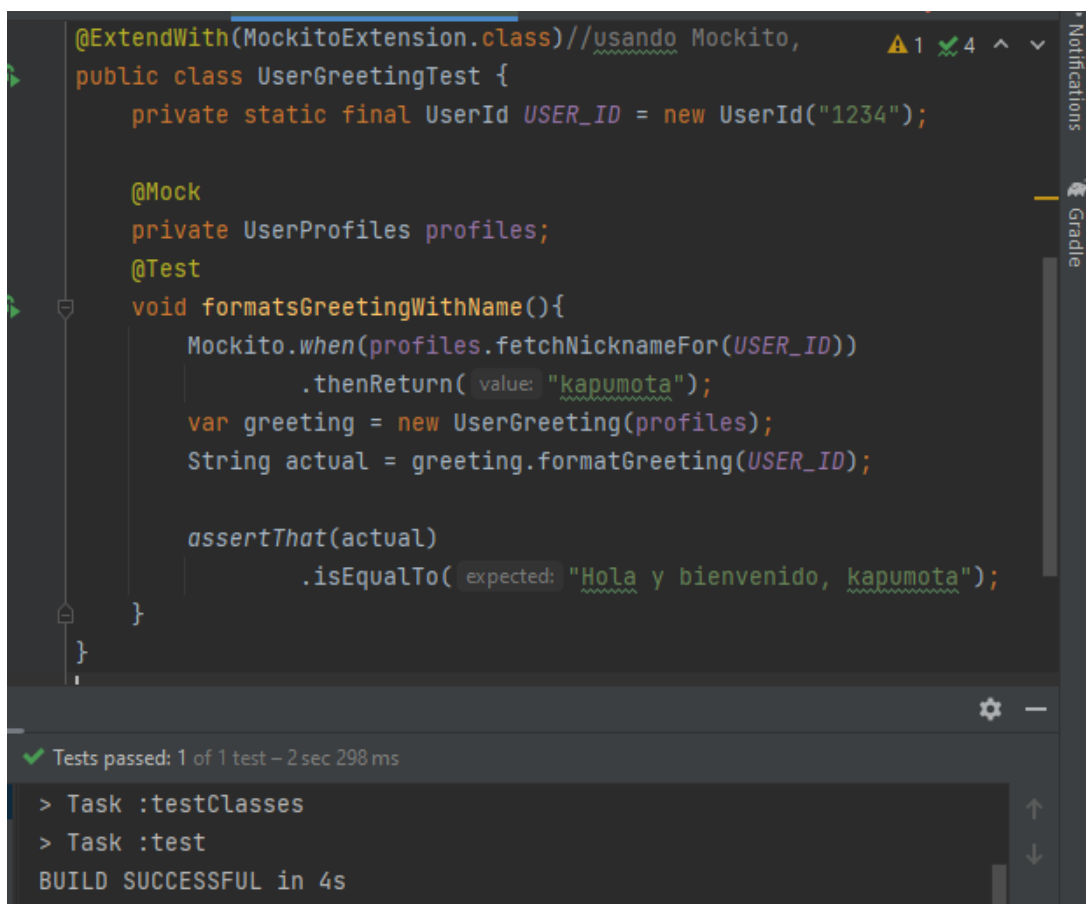
> Task :testClasses  
> Task :test

expected: "Hola y bienvenido, kapumota"  
but was: "Hola y Bienvenido, null"

**Problema:** 3Qu3 sucede si ejecutas la prueba ahora?. Explica la salida.

La prueba falla, pero ya no lanza la excepcion, falla porque no coincide lo esperado con lo actual.

## 11) Configura @Mock para devolver los datos del stub correctos para la prueba



```
@ExtendWith(MockitoExtension.class) // usando Mockito,
public class UserGreetingTest {
    private static final UserId USER_ID = new UserId("1234");

    @Mock
    private UserProfiles profiles;

    @Test
    void formatsGreetingWithName(){
        Mockito.when(profiles.fetchNicknameFor(USER_ID))
            .thenReturn( value: "kapumota");
        var greeting = new UserGreeting(profiles);
        String actual = greeting.formatGreeting(USER_ID);

        assertThat(actual)
            .isEqualTo( expected: "Hola y bienvenido, kapumota");
    }
}
```

✓ Tests passed: 1 of 1 test – 2 sec 298 ms

> Task :testClasses  
> Task :test  
BUILD SUCCESSFUL in 4s

En la clase **UserGreeting** vemos que usamos el metodo `fetchNicknameFor(id)` que pertenece a la clase **UserProfiles**, pero en nuestra prueba no nos importa como funciona esa clase, solo nos importa **UserGreeting**, por eso lo simulamos usando **Mockito.when().thenReturn()**, asi cuando le pasamos el `id=1234` me retorne “kapumota”.

## 12) ¿Qué sucede si vuelves a ejecutar la prueba?.

Como vemos en la imagen anterior, la prueba es exitosa. Porque ya definimos el comportamiento del objeto doble de prueba(stub) `profiles`.

Ahora a la variable **actual** le estamos pasando `USER_ID`, pero por la simulacion tiene que devolverme el “String definido” + “kapumota”.

## Escribiendo un mock con Mockito

**Mockito es una biblioteca** de pruebas unitarias en Java que se utiliza para simular objetos y comportamientos en pruebas unitarias. Tanto los stubs como los mocks son conceptos clave en Mockito, pero se utilizan en diferentes contextos y tienen propósitos ligeramente diferentes.

1. **Stub: Un stub (o burlador)** es un objeto que proporciona respuestas predefinidas a las llamadas de métodos durante las pruebas. Se utiliza para simular el comportamiento de dependencias externas o colaboradores de un objeto bajo prueba. Un stub generalmente se utiliza para proporcionar respuestas predefinidas a métodos que no son el foco de la prueba actual. Con Mockito, se pueden crear stubs utilizando métodos como **when()** y **thenReturn()** para especificar el comportamiento deseado del stub.
2. **Mock: Un mock (o falso)** es similar a un stub, pero con la capacidad adicional de verificar interacciones entre el objeto bajo prueba y el objeto simulado. A diferencia de un stub, un mock puede generar una excepción si se llama a un método no esperado durante la prueba. Los mocks se utilizan para asegurarse de que el objeto bajo prueba interactúe correctamente con sus dependencias y colaboradores. Mockito ofrece métodos como **verify()** para verificar las interacciones entre objetos simulados y el objeto bajo prueba.