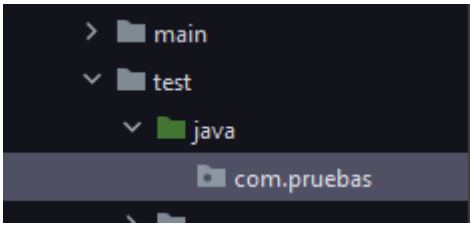
AEREOPUERTOS JUAN

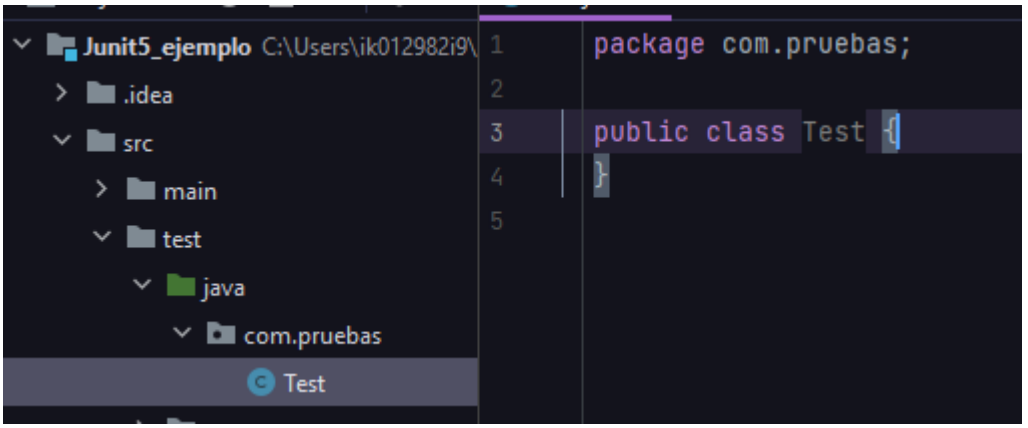
Ametz Segovia

Después de descomprimir y abrir el fichero ***Junit5_ejemplo.rar*** con InteillJ, realizar las siguientes tareas:

1. Dentro de la carpeta test – java crear un nuevo paquete: com.pruebas



2. Crear un conjunto de Test (clase Java).



3. Crear un método de Test para probar el método count() de la clase SmartPhoneServiceImpl mediante aserciones (assert), en concreto:
3.1 Que el método count() no devuelve NULL.
3.2 Que el valor que devuelve el método count() es mayor que 0.
3.3 Que el valor que devuelve el método count() es justo 3.

```
public class Pruebas {
    @Test
        public void testCount(){
            SmartPhoneServiceImpl s1 = new SmartPhoneServiceImpl();
            Assertions.assertNotNull(s1.count(), message: "No NULL");
            Assertions.assertTrue( condition: s1.count() > 0,  message: "Mayor que 0");
            Assertions.assertEquals( expected: 3, s1.count(), message: "3");

    }
```

4. Ejecutar estos tests y crear un caso el que alguno de los tests falle.

```
C:\Users\ik012982i9\.jdks\openjdk-19\bin\java.exe ...
inicializando SmartPhoneServiceImpl

org.opentest4j.AssertionFailedError: 3 ==> expected: <3> but was: <4>
Expected :3
Actual   :4
<Click to see difference>
```

5. Crear otro método de Test para probar el método findOne(Long id), en concreto:
5.1 Que el método al recibir un ID nulo, lanza una excepción de tipo IllegalArgumentException.

```
@Test
public void testFindOne(){
    SmartPhoneServiceImpl s1 = new SmartPhoneServiceImpl();
    Long id = null;
    assertThrows(IllegalArgumentException.class, () -> {
        s1.findOne(id);
    });
}
```

6. Juntar las aseciones 3.1, 3.2 eta 3.3 en una sola (mediante assertAll) y probar cada una ellas se ejecuta de manera independiente.

```
@Test
public void countSmartphone(){
    SmartPhoneServiceImpl s1 = new SmartPhoneServiceImpl();
    assertAll(() -> assertNotNull(s1.count(), message: "Este valor es nulo"));
    assertAll(() -> assertTrue( condition: s1.count() > 0,  message: "Este numero debe ser mayor que 0
    assertAll(() ->assertEquals( expected: 3, s1.count(), message: "El numero debe ser 3"));

}
```

7. Explicar para qué se utilizan las funciones lambdas.
Las funciones lambda son funciones anónimas que se utilizan en programación para definir una pequeña pieza de código que se puede pasar como argumento a otras funciones o métodos. Estas funciones son útiles en situaciones en las que necesitas crear una función temporal para realizar una tarea específica y no quieres crear una función separada completa.
8. Asignar un nombre a cada método de Test y al conjunto de Tests (único).

```
public void testCount(){
```

```
public void testFindOne(){
```

9. Investiga las opciones @BeforeEach y @AfterEach en una clase Test. ¿Para qué sirven? ¿Cuál es su uso? Demuestra que has probado ambas opciones (capturas de pantalla).
En JUnit, las anotaciones @BeforeEach y @AfterEach se utilizan para ejecutar

métodos antes y después de cada método de prueba en una clase de prueba.

```
@Before
public void pruebaBefore() {
    System.out.println("Inicio");
}
```

```
@After
public void ending() {
    System.out.println("Final");
}
```

```
Inicio
inicializando SmartPhoneServiceImpl
Final
Inicio
inicializando SmartPhoneServiceImpl
Final
```

10. Investiga las opciones @BeforeAll y @AfterAll en una clase Test. ¿Para qué sirven? ¿Cuál es su uso? Demuestra que has probado ambas opciones (capturas de pantalla).
Las anotaciones @BeforeAll y @AfterAll son anotaciones de JUnit que se utilizan para marcar métodos que se ejecutarán antes de que se ejecute cualquier prueba en una clase o después de que se hayan ejecutado todas las pruebas en una clase.

```
@BeforeAll
public void pruebaBefore() {
    System.out.println("Inicio");
}
```

```
@AfterAll
public void ending() {
    System.out.println("Final");
}
```

11. Investiga la opción @TestMethodOrder. ¿Para qué sirve? ¿Cuál es su uso? Demuestra que lo has probado (captura de pantalla)
La anotación @TestMethodOrder es una anotación de JUnit 5 que permite especificar el orden en el que se ejecutan los métodos de prueba en una clase de prueba. Esta anotación se utiliza para controlar el orden de ejecución de las pruebas y asegurarse de que se ejecuten en un orden específico.

```java
@Order(2)
public void testCount(){
    SmartPhoneServiceImpl s1 = new SmartPhoneServiceImpl();
    assertAll(() -> assertNotNull(s1.count(), message: "Este valor es nulo"));
    assertAll(() -> assertTrue( condition: s1.count() > 0,  message: "Este numero debe ser mayor que 0"));
    assertAll(() ->assertEquals( expected: 3, s1.count(), message: "El numero debe ser 3"));
    System.out.println("1");
}
/*
public void testCount(){
        SmartPhoneServiceImpl s1 = new SmartPhoneServiceImpl();
        Assertions.assertNotNull(s1.count(),"No NULL");
        Assertions.assertTrue(s1.count() > 0, "Mayor que 0");
        Assertions.assertEquals(3, s1.count(),"3");
}
*/
@Test
@Order(1)
public void testFindOne(){
    SmartPhoneServiceImpl s1 = new SmartPhoneServiceImpl();
    Long id = null;
    assertThrows(IllegalArgumentException.class, () -> {
        s1.findOne(id);
    });
    System.out.println("2");
}
```

```
inicializando SmartPhoneServiceImpl
2
inicializando SmartPhoneServiceImpl
1
```