JUnit: Pruebas parametrizadas

Las pruebas parametrizadas en JUnit 5 permiten ejecutar un mismo método de prueba con diferentes conjuntos de datos de entrada, lo que facilita la verificación del comportamiento de un método o función bajo varios escenarios. Este enfoque mejora la cobertura de pruebas y garantiza que el código sea robusto y funcione correctamente en diversas condiciones.

¿Por qué utilizar pruebas parametrizadas?

Las pruebas parametrizadas son especialmente útiles cuando necesitamos verificar que un método funcione correctamente para una variedad de entradas sin escribir múltiples métodos de prueba. En lugar de repetir el mismo código con datos diferentes, las pruebas parametrizadas permiten reutilizar el mismo método de prueba para distintas entradas. Esto mejora la eficiencia del proceso de testing, facilita la detección de errores en diversas condiciones y asegura que el código sea más confiable.

Cómo crear una prueba parametrizada en JUnit 5

Para crear una prueba parametrizada, se utiliza la anotación `@ParameterizedTest` en lugar de `@Test`. A continuación, se proporcionan los datos de entrada utilizando alguna de las siguientes anotaciones de origen de datos:

.

Anotación	Descripción
`@ValueSource`	Especifica una lista de valores de un solo tipo para ser utilizados como argumentos en la prueba

```java
@ParameterizedTest

@ValueSource (ints = {2, 3, 5, 7, 11, 13, 17, 19})
void testEsPrimo(int number) {
    assertEquals(Primos.esPrimo(number), "El número debería ser primo");
}
```

En este caso, la anotación `@ValueSource` pasa diferentes valores de tipo `int` (números primos) a la prueba. Cada uno de estos valores es utilizado para ejecutar el método `testEsPrimo`. Esto permite probar el método `esPrimo()` con diversos números y asegurarse de que funciona correctamente para cada uno.

@MethodSource	Hace referencia a un método que proporciona los parámetros para la prueba.

```
@ParameterizedTest
@MethodSource("provideStrings")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

// Este método debe ser estático y no debe tener parámetros. static
String[] provideStrings() {
    return new String[]{"manzana", "banana"};
}
```

En este caso, la anotación `@MethodSource` hace referencia al método `provideStrings`, que devuelve un `Stream` de valores (en este caso, frutas). Estos valores son luego utilizados como entradas para el método de prueba `testWithExplicitLocalMethodSource`. La ventaja es que el conjunto de datos puede generarse dinámicamente, lo cual ofrece más flexibilidad que la simple lista de valores.

@CsvSource	Permite proporcionar valores para una prueba parametrizada en formato de cadena de texto CSV (Valores separados por comas).

```
class CalculadoraTest {
    @ParameterizedTest
    @CsvSource({"1, 2, 3", "2, 3, 5", "3, 5, 8"})
    void testSuma(int num1, int num2, int expectedResult) {
assertEquals(expectedResult, Calculadora.suma(num1, num2), "La suma debería ser correcta");
    }}
```

La anotación `@CsvSource` te permite pasar varios conjuntos de datos en formato CSV. En el ejemplo, los tres pares de números se usan como entradas para el método de prueba `testSuma`, verificando que la suma de dos números dé como resultado el valor esperado. Cada conjunto de datos se pasa como una cadena de texto separada por comas, y los parámetros se mapean automáticamente a los parámetros del método de prueba.

@CsvFileSource	Similar a `@CsvSource`, pero los valores se leen de un archivo CSV en lugar de ser proporcionados directamente en la anotación.

```
@ParameterizedTest
@CsvFileSource(resources = "/test/day-week-data.csv", numLinesToSkip = 1) // suponiendo que tienes un archivo llamado day-week-data.csv en tu carpeta de test
void testWithCsvFileSource(String dayWeek, Integer expectedResult) {
    Integer result = getNumberDay(dayWeek);
    assertEquals(expectedResult, result);
}
```

La anotación `@CsvFileSource` permite cargar los datos de prueba desde un archivo CSV. En este ejemplo, los datos se leen de un archivo llamado `day-week-data.csv`, y la primera línea del archivo se omite gracias a `numLinesToSkip = 1`. Cada fila del archivo será utilizada como un conjunto de parámetros para las pruebas. Esto es útil cuando se tienen grandes cantidades de datos de entrada.

Consideraciones adicionales:

Excepciones en pruebas parametrizadas: En algunos casos, los datos de entrada pueden ser inválidos y el método bajo prueba debe lanzar una excepción. En esos casos, puedes usar `@TestFactory` o `@MethodSource` con una combinación de `assertThrows()` para validar que las excepciones se manejan correctamente en diferentes situaciones.

Ejemplo:

```
 @ParameterizedTest
@ValueSource(strings = {"", "   ", "invalid"})
void testEmailInvalido(String email) {
      assertThrows(IllegalArgumentException.class, () -> new Usuario("Juan", email, 25));
}
```

Pruebas con objetos: Si deseas realizar pruebas con objetos más complejos, como instancias de clases o listas, puedes usar las anotaciones `@MethodSource` o `@CsvSource` para proporcionar estos objetos como parámetros. Esto es

especialmente útil cuando los datos de entrada no son simples valores literales, sino que involucran entidades o estructuras más complejas.

Ejemplo con objetos:

```
@ParameterizedTest
@MethodSource("provideUsuarios")
void testEmailValido(Usuario usuario) {
     assertTrue(usuario.emailValido(), "El correo debería ser
válido");
}

// Método estático que proporciona los objetos de prueba
static List<Usuario> provideUsuarios() {
    return List.of(
        new Usuario("Carlos", "carlos@example.com", 30),
        new Usuario("Ana", "ana@example.com", 28)
    );
}
```

Explicación:

1. **`@MethodSource("provideUsuarios")`**: La anotación `@MethodSource` hace referencia a un método estático llamado `provideUsuarios`, que proporciona una lista de objetos `Usuario`.
2. **Método `provideUsuarios()`**: Este método devuelve una lista de instancias de la clase `Usuario` que se utilizarán como parámetros para la prueba. Cada usuario es probado con el método `testEmailValido()`.
3. **Prueba parametrizada**: Al ejecutar la prueba, JUnit invocará el método `testEmailValido` con cada uno de los objetos proporcionados por `provideUsuarios`, y verificará si el correo de cada usuario es válido usando el método `emailValido()` de la clase `Usuario`.

Conclusión

Las pruebas parametrizadas son una herramienta poderosa para mejorar la eficiencia de las pruebas y garantizar que tu código sea robusto frente a una variedad de entradas. Ya sea utilizando valores literales, métodos fuente o archivos

CSV, JUnit 5 facilita la creación y ejecución de pruebas con diferentes conjuntos de datos. ¡Explora las distintas opciones y aprovecha al máximo las pruebas parametrizadas en tu desarrollo!

💡**DOCUMENTACIÓN OFICIAL:**

Existen más formas de realizar pruebas parametrizadas que involucran el uso de objetos, las cuales se pueden explorar en la documentación de JUnit 5 en el apartado de pruebas parametrizadas: **Documentación de JUnit 5 - Pruebas parametrizadas**