CURSO DE PROGRAMACIÓN FULL STACK

J Unit

Objetivos de la Guía

En esta guía aprenderemos a:

- Comprender el concepto de pruebas unitarias.
- Familiarizarse con JUnit.
- Configuración previa y posterior a las pruebas.
- Personalizar el comportamiento de ejecución.
- Ignorar y categorizar pruebas.
- Aplicar las mejores prácticas de pruebas unitarias.

¿QUE ES JUnit?

Una prueba unitaria es una técnica de prueba que se utiliza en el desarrollo de software para verificar el funcionamiento individual y aislado de una unidad de código, generalmente un método o una función. El objetivo principal de una prueba unitaria es garantizar que cada unidad funcione correctamente de manera independiente antes de integrarla con otras partes del sistema.

Estructura de pruebas unitarias

Las pruebas unitarias suelen seguir una estructura común que incluye los siguientes elementos:

Configuración (Setup): En esta etapa, se prepara el entorno necesario para ejecutar la prueba. Esto puede incluir la creación de objetos, la inicialización de variables y la configuración de cualquier estado necesario para el método que se va a probar.

Ejecución: Aquí se invoca el método o la función que se está probando con los datos de entrada adecuados. Es el paso en el cual se realiza la llamada al código que se desea evaluar.

Verificación (Assertion): En esta fase se verifica si el resultado obtenido por el método bajo prueba coincide con el resultado esperado. Se utilizan aserciones (assertions) para comparar los valores esperados con los valores obtenidos.

Limpieza (Teardown): En esta etapa, se realiza cualquier limpieza necesaria después de la ejecución de la prueba. Esto puede incluir la liberación de recursos, la restauración de estados previos o la eliminación de objetos creados durante la configuración.

Importancia de pruebas unitarias

Las pruebas unitarias son importantes por varias razones:

Detección temprana de errores: Las pruebas unitarias permiten identificar y corregir errores en las unidades de código a medida que se desarrollan. Esto ayuda a prevenir la propagación de errores en otras partes del sistema y facilita su resolución antes de que se vuelvan más complejos y costosos de corregir.

Mantenibilidad del código: Las pruebas unitarias facilitan el mantenimiento del código a lo largo del tiempo. Al tener un conjunto de pruebas que se ejecutan automáticamente, se puede realizar cambios o mejoras en el código con mayor confianza, ya que las pruebas actuarán como un mecanismo de verificación continua.

Documentación viva: Las pruebas unitarias proporcionan una forma de documentación viva y actualizada del comportamiento esperado de las unidades de código. Sirven como una guía clara y concisa para comprender cómo se supone que deben funcionar las diferentes partes del sistema.

Facilita la colaboración: Las pruebas unitarias permiten que varios desarrolladores trabajen en paralelo en un proyecto, ya que proporcionan un medio para verificar rápidamente si los cambios realizados en una unidad de código afectan negativamente a otras partes del sistema.

Mejora la calidad del software: Al garantizar que cada unidad de código funcione correctamente individualmente, las pruebas unitarias contribuyen a mejorar la calidad general del software. Ayudan a reducir los errores y las fallas en el sistema, lo que se traduce en un software más robusto y confiable.

En resumen, las pruebas unitarias son pruebas automatizadas que se centran en verificar el correcto funcionamiento de las unidades de código de forma individual. Proporcionan beneficios como la detección temprana de errores, facilitar el mantenimiento del código, actuar como documentación viva, mejorar la colaboración entre desarrolladores y elevar la calidad del software en general.

Sintaxis

Anotaciones:

● @Test: Esta anotación se utiliza para marcar un método como una prueba unitaria. Los métodos anotados con @Test deben tener una firma pública y no devolver un valor.

● @Before: Esta anotación se utiliza para marcar un método que se ejecutará antes de cada prueba. Puedes utilizar este método para realizar la configuración necesaria antes de cada prueba.

● @After: Esta anotación se utiliza para marcar un método que se ejecutará después de cada prueba. Puedes utilizar este método para realizar la limpieza o restauración de estados después de cada prueba.

● @BeforeClass: Esta anotación se utiliza para marcar un método que se ejecutará una vez antes de todas las pruebas en la clase. Puedes utilizar este método para realizar configuraciones que sean comunes a todas las pruebas.

● @AfterClass: Esta anotación se utiliza para marcar un método que se ejecutará una vez después de todas las pruebas en la clase. Puedes utilizar este método para realizar tareas de limpieza o liberación de recursos después de todas las pruebas.

● @Ignore: Esta anotación se utiliza para marcar una prueba para ser ignorada. La prueba no se ejecutará y se marcará como pasada automáticamente.

Assertions (Aserciones):

● assertEquals(expected, actual): Compara si el valor expected es igual al valor actual.

● assertTrue(condition): Verifica si una condición dada es verdadera.

● assertFalse(condition): Verifica si una condición dada es falsa.

● assertNull(object): Verifica si un objeto dado es nulo.

● assertNotNull(object): Verifica si un objeto dado no es nulo.

● assertSame(expected, actual): Verifica si el valor expected es el mismo objeto que el valor actual.

● assertNotSame(unexpected, actual): Verifica si el valor unexpected no es el mismo objeto que el valor actual.

¿Cómo escribo la prueba?

Anotación @Test: La anotación @Test se coloca justo encima del método de prueba para indicar que ese método es una prueba unitaria. Es una forma de marcar el método como parte del conjunto de pruebas.

```
@Test
public void testAddition() {
    // Código de prueba
}
```

Creación de objetos y configuración: Dentro del método de prueba, puedes crear objetos o instancias de la clase que deseas probar. Esto implica la creación de un objeto de la clase y, opcionalmente, la configuración de su estado inicial.

```
@Test
public void testAddition() {
    Calculator calculator = new Calculator();
    // Configuración adicional si es necesaria
}
```

Llamada al método bajo prueba: Luego de crear los objetos necesarios, puedes llamar al método o función que deseas probar. Puedes pasar los argumentos necesarios al método y almacenar el resultado en una variable si es necesario.

```java
@Test
public void testAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.add(2, 3);
    // Almacenar el resultado si es necesario
}
```

Verificación del resultado: Una vez que se ha llamado al método bajo prueba y se ha obtenido un resultado, puedes utilizar las aserciones (assertions) de JUnit para verificar si el resultado es el esperado. Las aserciones comparan el resultado obtenido con el resultado esperado y muestran un mensaje de error si no coinciden.

```java
@Test
public void testAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.add(2, 3);
    assertEquals(5, result);
}
```

Configuraciones previas

La anotación @BeforeClass en JUnit se utiliza para marcar un método que se ejecuta una vez antes de todas las pruebas en una clase de pruebas. Algunos beneficios de realizar configuraciones previas utilizando @BeforeClass son los siguientes:

Configuración común: El método anotado con @BeforeClass permite realizar configuraciones que son comunes a todas las pruebas en la clase. Por ejemplo, puedes crear instancias de objetos, establecer conexiones con bases de datos, cargar datos de prueba, inicializar variables estáticas, entre otros.

Ahorro de tiempo y recursos: Al ejecutar una configuración previa una vez antes de todas las pruebas, se evita la repetición de la misma configuración en cada prueba individual. Esto ahorra tiempo de ejecución y recursos, especialmente si la configuración es costosa o requiere mucho tiempo.

Estado compartido: Si hay algún estado o datos que deben ser compartidos entre varias pruebas, la configuración previa con @BeforeClass puede ser útil. Puedes preparar el estado compartido en el método @BeforeClass y acceder a él desde diferentes pruebas en la clase.

Mayor legibilidad y mantenibilidad: Al colocar la configuración común en un método anotado con @BeforeClass, mejora la legibilidad del código de prueba. Es más claro y conciso que repetir la misma configuración en cada prueba. Además, si necesitas modificar o actualizar la configuración, solo necesitas hacerlo en un lugar, lo que facilita el mantenimiento del código.

Aislamiento de pruebas: La configuración previa con @BeforeClass ayuda a mantener la independencia y el aislamiento de las pruebas unitarias. Al realizar la configuración una vez antes de

todas las pruebas, cada prueba puede centrarse en verificar un comportamiento específico sin depender del estado creado por otras pruebas.

En resumen, la anotación @BeforeClass en JUnit se utiliza para realizar configuraciones previas que son comunes a todas las pruebas en una clase. Esto proporciona beneficios como configuraciones compartidas, ahorro de tiempo y recursos, mayor legibilidad y mantenibilidad del código, y aislamiento de pruebas.

Veamoslo en nuestro ejemplo:

```
@BeforeClass
public static void setUp() {
    calculator = new Calculator();
    // Configuración adicional si es necesaria
}
```

En este ejemplo, hemos agregado el método setUp() anotado con @BeforeClass. Este método se ejecutará una vez antes de que se ejecuten todas las pruebas en la clase. Dentro del método setUp(), configuramos el estado inicial necesario, en este caso, la creación de una instancia de la clase Calculator. También puedes realizar configuraciones adicionales si es necesario.

Luego, las pruebas individuales, como testAddition() y testSubtraction(), pueden acceder al objeto calculator creado en el método setUp() y realizar las pruebas necesarias en cada uno de ellos.

```
@Test
public void testAddition() {
    int result = calculator.add(2, 3);
    assertEquals(5, result);
}

@Test
public void testSubtraction() {
    int result = calculator.subtract(5, 3);
    assertEquals(2, result);
}
```

La anotación @BeforeClass es útil cuando tienes configuraciones que son comunes a todas las pruebas en la clase y que solo necesitan ejecutarse una vez antes de todas las pruebas. Esto evita la necesidad de repetir la misma configuración en cada método de prueba individual.

Recuerda que el método anotado con @BeforeClass debe ser estático para que se ejecute correctamente antes de las pruebas.

Configuraciones posteriores

Los métodos @After y @AfterClass en JUnit se utilizan para realizar tareas de limpieza y restauración después de las pruebas. A continuación, se explican por qué son necesarios y qué sucede si no se utilizan:

@After:

Propósito: El método anotado con @After se ejecuta después de cada prueba individual. Sirve para realizar tareas de limpieza específicas que sean necesarias después de la ejecución de cada prueba.

Necesidad: Es útil para liberar recursos, cerrar conexiones a bases de datos, eliminar archivos temporales, revertir cambios en el estado, entre otras tareas de limpieza.

Importancia: Sin @After, no se realizarían las tareas de limpieza necesarias después de cada prueba. Esto podría llevar a una acumulación de recursos no liberados o a un estado incorrecto que podría afectar a otras pruebas y a la precisión de los resultados.

@AfterClass:

Propósito: El método anotado con @AfterClass se ejecuta después de todas las pruebas en una clase. Se utiliza para realizar tareas de limpieza o restauración que sean comunes a todas las pruebas en la clase.

Necesidad: Es útil para liberar recursos compartidos, cerrar conexiones globales, realizar una limpieza final o restaurar estados iniciales.

Importancia: Si no se utiliza @AfterClass, las tareas de limpieza o restauración que sean necesarias después de todas las pruebas no se realizarían automáticamente. Esto podría provocar una acumulación de recursos no liberados, conexiones abiertas o un estado incorrecto para futuras ejecuciones.

En resumen, los métodos @After y @AfterClass son necesarios para realizar tareas de limpieza y restauración después de las pruebas. Son importantes para garantizar una ejecución limpia, prevenir problemas de acumulación de recursos y mantener el estado adecuado para pruebas futuras. Si no se utilizan, no se realizarán automáticamente las tareas de limpieza necesarias, lo que puede llevar a problemas de recursos no liberados, conexiones abiertas o un estado incorrecto que podría afectar a las pruebas posteriores.

```
    // Otras pruebas...

    @After
    public void tearDown() {
        // Limpieza después de cada prueba, si es necesaria
    }

    @AfterClass
    public static void tearDownClass() {
        // Limpieza después de todas las pruebas en la clase, si es necesari
    }
}
```

Otras anotaciones importantes

@Ignore:

Propósito: La anotación @Ignore se utiliza para marcar una prueba para ser ignorada. Indica que la prueba no se debe ejecutar.

Uso: Simplemente anota el método de prueba con @Ignore y se omitirá durante la ejecución de las pruebas.

Ejemplo:

```
import org.junit.Ignore;
import org.junit.Test;

@Ignore
@Test
public void ignoredTest() {
    // Código de la prueba ignorada
}
```

@RunWith:

Propósito: La anotación @RunWith se utiliza para personalizar el comportamiento de ejecución de las pruebas. Permite especificar un corredor (runner) personalizado para ejecutar las pruebas en lugar del corredor predeterminado de JUnit.

Uso: Anota la clase de prueba con @RunWith y especifica la clase del corredor personalizado como argumento.

Ejemplo:

```
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)
public class CustomRunnerTest {
    // Métodos de prueba
}
```

@Category:

Propósito: La anotación @Category se utiliza para categorizar las pruebas. Permite agrupar pruebas en categorías para poder ejecutar subconjuntos de pruebas en función de esas categorías.

Uso: Define una interfaz o una clase de categoría y anota las pruebas con @Category para asignarlas a una o varias categorías.

Ejemplo:

```java
import org.junit.Test;
import org.junit.experimental.categories.Category;

public interface FastTests { /* Categoría de pruebas rápidas */ }
public interface SlowTests { /* Categoría de pruebas lentas */ }

public class CategorizedTest {

    @Category(FastTests.class)
    @Test
    public void fastTest() {
        // Código de prueba rápida
    }

    @Category(SlowTests.class)
    @Test
    public void slowTest() {
        // Código de prueba lenta
    }
}
```

Estas anotaciones adicionales en JUnit brindan flexibilidad y personalización en las pruebas unitarias. Puedes utilizar @Ignore para excluir pruebas temporales, @RunWith para utilizar corredores personalizados y @Category para ejecutar subconjuntos de pruebas en función de categorías específicas. Estas anotaciones te permiten adaptar las pruebas a tus necesidades y mejorar la organización y el control de ejecución de las pruebas.

EJERCICIOS DE APRENDIZAJE

Antes de comenzar con esta guía, les damos algunas recomendaciones:

Como de ahora en más **TODOS LOS EJERCICIOS QUE REALICES DEBERÁN TENER SUS PROPIAS PRUEBAS** al resolver los siguientes ejercicios procura utilizar todos las anotaciones que puedas para comprender cómo funcionan.

1. Calculadora de Descuentos:

Crea una clase DiscountCalculator que calcule el descuento aplicado a un producto.

Escribe pruebas unitarias para verificar que el cálculo del descuento se realiza correctamente para diferentes escenarios (por ejemplo, descuento del 10%, descuento máximo, sin descuento, etc.).

2. Conversión de Temperatura:

Crea una clase TemperatureConverter que convierta entre diferentes unidades de temperatura (por ejemplo, Celsius, Fahrenheit, Kelvin).

Escribe pruebas unitarias para asegurarte de que la conversión entre las diferentes unidades se realiza correctamente y produce los resultados esperados.

3. Validador de Contraseñas:

Crea una clase PasswordValidator que verifique la fortaleza de una contraseña según ciertas reglas (por ejemplo, longitud mínima, presencia de caracteres especiales, letras mayúsculas, etc.).

Escribe pruebas unitarias para asegurarte de que el validador de contraseñas funcione correctamente para diferentes escenarios, incluyendo contraseñas válidas e inválidas.

4. Gestor de Tareas:

Crea una clase TaskManager que permita agregar, eliminar y listar tareas.

Escribe pruebas unitarias para verificar que el gestor de tareas realiza las operaciones correctamente, como agregar una tarea, eliminar una tarea existente y listar las tareas disponibles.

5. Validador de Fechas:

Crea una clase DateValidator que valide la corrección de una fecha (por ejemplo, si es una fecha válida en el calendario gregoriano).

Escribe pruebas unitarias para asegurarte de que el validador de fechas detecte correctamente fechas válidas e inválidas, teniendo en cuenta diferentes casos, como años bisiestos.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por último, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Generador de Números Aleatorios:

   Crea una clase RandomNumberGenerator que genere números aleatorios dentro de un rango específico.

   Escribe pruebas unitarias para asegurarte de que el generador de números aleatorios produce resultados dentro del rango esperado y cumple con las propiedades de aleatoriedad.

2. Verificador de Palíndromos:

   Crea una clase PalindromeChecker que verifique si una cadena es un palíndromo (se lee igual de adelante hacia atrás y viceversa).

   Escribe pruebas unitarias para asegurarte de que el verificador de palíndromos detecte correctamente las cadenas que son palíndromos y las que no lo son.

3. Generador de Contraseñas Seguras:

   Crea una clase SecurePasswordGenerator que genere contraseñas seguras con ciertos requisitos (por ejemplo, longitud mínima, combinación de caracteres, etc.).

   Escribe pruebas unitarias para asegurarte de que el generador de contraseñas seguras cumpla con los requisitos establecidos y produzca contraseñas seguras.

4. Analizador de Texto:

   Crea una clase TextAnalyzer que analice un texto y proporcione estadísticas sobre él (por ejemplo, número de palabras, número de frases, frecuencia de palabras, etc.).

   Escribe pruebas unitarias para asegurarte de que el analizador de texto proporcione los resultados correctos y maneje diferentes escenarios, como texto vacío o texto con caracteres especiales.

DESAFÍO EXTRA

Calculadora de Matrices

Crea una clase MatrixCalculator que realice operaciones matemáticas con matrices, como suma, resta, multiplicación, determinante, inversa, etc.

Implementa correctamente los algoritmos necesarios para cada operación.

Escribe pruebas unitarias exhaustivas para cada operación matemática, verificando que los resultados obtenidos sean correctos según las reglas matemáticas correspondientes.

Asegúrate de cubrir diferentes tamaños de matrices, matrices invertibles y no invertibles, y situaciones especiales, como la multiplicación de matrices no compatibles.

También puedes incluir pruebas para comparar los tiempos de ejecución de las operaciones en función del tamaño de las matrices.