Curso de desarrollo de software

Actividad preliminar individual

Tiempo estimado : 90 minutos

JUnit en IntelliJ

En este tutorial, aprenderás cómo configurar JUnit para tus proyectos, crear pruebas y ejecutarlas para ver si tu código funciona correctamente.
Contiene solo los pasos básicos para comenzar. Si quieres saber más sobre JUnit, consulta la documentación oficial.
Para obtener más información sobre las funciones de prueba de IntelliJ IDEA, consulta otros temas de esta sección. Puedes elegir seguir el tutorial usando Maven o Gradle.

Crea un proyecto

1. En el menú principal, selecciona File | New | Project
2. Selecciona New Project. Especifica el nombre del proyecto, por ejemplo, junit-tutorial.
3. Selecciona Maven como herramienta de compilación. En Language, selecciona Java.
4. En la lista de JDK, selecciona el JDK que deseas utilizar en tu proyecto.
   Si el JDK está instalado en su computadora, pero no está definido en el IDE, selecciona Add JDK y especifique la ruta al directorio principal de JDK.
   Si no tienes el JDK necesario en tu computadora, selecciona Download
   JDK. 5. Haz clic en Create.

Agregar dependencias

Para que nuestro proyecto usa las funciones de JUnit, debemos agregar JUnit como una dependencia.

1. Abre pom.xml en el directorio raíz de tu proyecto.
   Para navegar rápidamente a un archivo, presione Ctrl+Shift+N e ingresa su
   nombre. 2. En pom.xml, presione Alt+Insert, selecciona Add dependency.
3. En la ventana de herramientas que se abre, escribe org.junit.jupiter:junit-jupiter en el campo de búsqueda.
   Localiza la dependencia necesaria en los resultados de búsqueda y haz clic en Add. 4.
   Ahora necesitamos aplicar los cambios en el script de compilación. Presiona Ctrl+Shift+O o haz Load Maven Changes en la notificación que aparece en la esquina superior derecha del editor.

El procedimiento anterior muestra la forma 'manual' para que sepa qué sucede detrás de escena y dónde configura el framework de prueba. Sin embargo, si recién comienza a escribir pruebas, IntelliJ IDEA detectará automáticamente si falta la dependencia y le pedirá que la agregue.

Escribir código de aplicación

Agreguemos un código que estaremos probando.

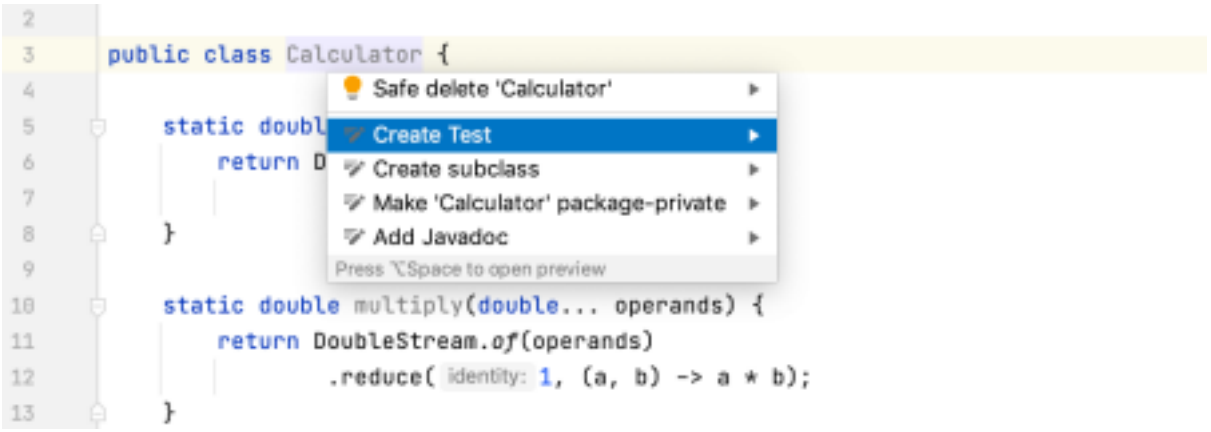
1. En la ventana de la herramienta Project Alt+1, ir a src/main/java y crea un archivo Java llamado Calculator.java.
2. Pega el siguiente código en el archivo:

```
import java.util.stream.DoubleStream;

public class Calculator {

        static double add(double... operands) {
            return DoubleStream.of(operands)
                    .sum();
        }

        static double multiply(double... operands) {
            return DoubleStream.of(operands)
                    .reduce(1, (a, b) -> a * b);
        }
    }
```

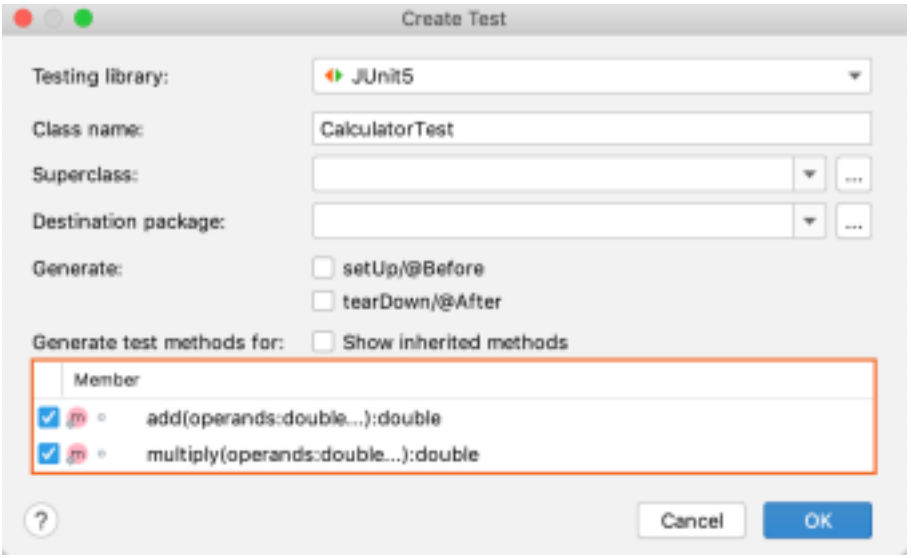
Creación de pruebas

Ahora vamos a crear una prueba. Una prueba es una pieza de código cuya función es verificar si otra pieza de código está funcionando correctamente. Para realizar la verificación, llama al método probado y compara el resultado con el resultado esperado predefinido. Un resultado esperado puede ser, por ejemplo, un valor de retorno específico o una excepción.

1. Coloca el signo de intercalación en la declaración de la clase Calculator y presiona Alt+Enter. Como alternativa, haga clic con el botón derecho en él y selecciona. Show Context Actions. En el menú, selecciona Create Test.



2. Selecciona los dos métodos de clase que vamos a probar.



3. El editor te lleva a la clase de prueba recién creada. Modifica la prueba add() de la siguiente manera:

```
@Test
@DisplayName("Suma de dos números")
void add() {
    assertEquals(4, Calculator.add(2, 2));
}
```

Esta sencilla prueba verificará si nuestro método add suma correctamente 2 y 2. La anotación **@DisplayName** especifica un nombre más conveniente e informativo para la prueba.

4. Ahora, ¿qué sucede si deseas agregar varias aserciones en una sola prueba y ejecutarlas todas independientemente de si algunas de ellas fallan? Hagámoslo para el método **multiply()**:

```
@Test
@DisplayName("Multiplica dos números")
void multiply() {
    assertAll(() -> assertEquals(4, Calculator.multiply(2, 2)),
    () -> assertEquals(-4, Calculator.multiply(2, -2)),
```

```
    () -> assertEquals(4, Calculator.multiply(-2, -2)),
    () -> assertEquals(0, Calculator.multiply(1, 0)));
}
```

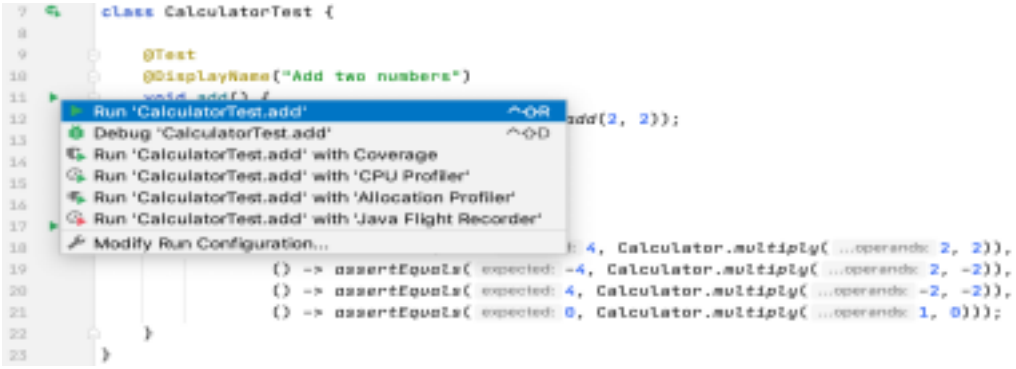
El método **assertAll()** toma una serie de aserciones en forma de expresiones lambda y garantiza que todas estén verificadas. Esto es más conveniente que tener varias afirmaciones individuales porque siempre verá un resultado granular en lugar del resultado de la prueba completa.

Para navegar entre la prueba y el código que se está probando, usa el atajo

Ctrl+Shift+T. Ejecución de pruebas y ver sus resultados

Después de configurar el código para la prueba, podemos ejecutar las pruebas y averiguar si los métodos probados funcionan correctamente.

- Para ejecutar una prueba individual, haga clic en Run.



- Para ejecutar todas las pruebas en una clase de prueba, haga clic en la declaración de clase de prueba y selecciona Run.
  Puedes ver los resultados de la prueba en la ventana de la herramienta Run.

IntelliJ IDEA oculta las pruebas aprobadas de forma predeterminada. Para verlos, asegúrese de que la opción Show Passed esté habilitada en la ventana de la herramienta Run.
Ejercicio: Presenta este procedimiento completo al instructor de

clase. **Aplicando todo desde cero**

Para nuestro primer ejemplo, crearemos una clase Calculator muy simple que suma dos números. Nuestra calculadora, que se muestra en la siguiente lista, proporciona una API a los clientes que no contiene una interfaz de usuario. Para probar su funcionalidad, primero crearemos nuestras propias pruebas de Java puro y luego pasaremos a JUnit 5.

```
public class Calculator {
    public double add(double number1, double number2) {
            return number1 + number2;
    }
}
```

Aunque no se muestra la documentación, el propósito previsto del método add(double, double) de Calculator es tomar dos dobles y devolver la suma como un doble. El compilador puede decirle que el código se compila, pero también debe asegurarse de que funcione en tiempo de ejecución. Un principio fundamental de las pruebas unitarias es: “Cualquier función del programa sin una prueba automatizada simplemente no existe”.

El método de add representa una función central de la calculadora. Tienes un código que supuestamente implementa la función. Lo que falta es una prueba automatizada que demuestre que la implementación funciona.

El programa de prueba puede pasar valores conocidos al método y ver si el resultado coincide con las expectativas. También puedes volver a ejecutar el programa más tarde para asegurarse de que el método sigue funcionando a medida que crece la aplicación. Entonces, ¿cuál es el programa de prueba más simple posible que podría escribir?

Pregunta 1: ¿Qué pasa con este programa CalculatorTest?

```
public class CalculatorTest {
   public static void main(String[] args) {
            Calculator calculator = new Calculator();
              double result = calculator.add(10, 50);
             if (result != 60) {
                    System.out.println("Bad result: " + result);
                  }
            }
     }
```

De forma directa en el main, se instancia un objeto calculator de la clase Calculator y crea una variable donde se almacena el resultado del método add de calculator, que al recibir dos parámetros devuelve la suma de esos dos números, en este caso: 10 y 50, como sabemos el valor del resultado, 60, agrega una sentencia condicional que compara ese resultado del método con el valor que conocemos y en caso el resultado no sea el mismo se imprimirá un mensaje de “Bad result” mostrando el resultado que tuvo el método.

La siguiente lista muestra un programa **CalculatorTest** ligeramente mejor.

```
public class CalculatorTest {
 private int nbErrors = 0;
   public void testAdd() {
      Calculator calculator = new Calculator();
      double result = calculator.add(10, 50);
       if (result != 60) {
           throw new IllegalStateException("Mal resultado: " + result);
```

```
    }
 }
public static void main(String[] args) {
     CalculatorTest test = new CalculatorTest();
    try {
        test.testAdd();
   }
  catch (Throwable e) {
        test.nbErrors++;
      e.printStackTrace();
}
if (test.nbErrors > 0) {
        throw new IllegalStateException("Hay " + test.nbErrors + " error(s)");
  }
  }
}
```

Pregunta 2: ¿Qué pasa con este programa CalculatorTest?

En general, sigue con la lógica de la primera solo que crea una función testAdd() que se encarga de comparar un resultado esperado conocido con el obtenido por el método add, característico de las pruebas. Inicialmente, se inicializa una variable que pueda contar los errores encontrados durante la prueba y en caso de que el valor esperado no sea igual al obtenido se lanza una excepción en la que se muestra el valor del resultado del método antecedido de “Mal resultado”.
En el main se prueba el testAdd creado mediante try y gracias al catch se señala el bloque de instrucciones a intentar (lo de testAdd) , y especifica las respuesta al producirse una excepción cuando falle la prueba y, también, en caso se hayan encontrado errores se muestra también como excepción la cantidad que hay.

Ahora que has visto una aplicación simple y tus pruebas, puedes ver que incluso esta pequeña clase y sus pruebas pueden beneficiarse del pequeño código esqueleto que creó para ejecutar y administrar los resultados de las pruebas. Pero a medida que una aplicación se vuelve más complicada y las pruebas se vuelven más complicadas, continuar creando y manteniendo un framework de prueba personalizado se convierte en una carga.

A continuación, damos un paso atrás y observamos el caso general de un framework de pruebas unitarias.

Entendiendo los frameworks de pruebas unitarias

Las pruebas unitarias tienen varias mejores prácticas que los frameworks deben seguir. Las mejoras aparentemente menores en el programa CalculatorTest en el listado anterior resaltan tres reglas que deben seguir todos los frameworks de pruebas unitarias:

- Cada prueba unitaria debe ejecutarse independientemente de todas las demás pruebas unitarias.
- El framework debe detectar y reportar errores prueba por prueba.
- Debe ser fácil definir qué pruebas unitarias se ejecutarán.

El programa de prueba "ligeramente mejor" se acerca a seguir estas reglas, pero aún se queda corto.

Para que cada prueba de unidad sea verdaderamente independiente, por ejemplo, cada una debe ejecutarse en una instancia de clase diferente.

Adición de pruebas unitarias

Facilitar la adición de pruebas (la tercera regla de la lista anterior) suena como otra buena regla para un framework de pruebas unitarias. El código de soporte que realiza esta regla (mediante registro o reflexión) no sería trivial, pero valdría la pena.
Tendrías que hacer mucho trabajo por adelantado, pero ese esfuerzo valdría la pena cada vez que agregas una nueva prueba.

Afortunadamente, el equipo de JUnit te ha ahorrado el problema. El framework JUnit ya admite métodos de descubrimiento. También admite el uso de una instancia de clase diferente y una instancia de cargador de clase para cada prueba, además de informar los errores prueba por prueba. El equipo ha definido tres objetivos discretos para el framework:

- El framework debe ayudarnos a escribir pruebas útiles
- El framework debe ayudarnos a crear pruebas que conserven su valor a lo largo del tiempo.
- El framework debe ayudarnos a reducir el costo de escribir pruebas mediante la reutilización del código.

Configurando JUnit

Para usar JUnit para escribir las pruebas de su aplicación, necesitas conocer sus dependencias. Trabajaremos con JUnit 5, la última versión del framework. La versión 5 del framework de prueba es modular; ya no puede simplemente agregar un archivo jar a la ruta de clase de compilación de tu proyecto y tu ruta de clase de ejecución.

De hecho, a partir de la versión 5, la arquitectura ya no es monolítica. Además, con la introducción de anotaciones en Java 5, JUnit también pasó a usarlas. JUnit 5 se basa en gran medida en anotaciones, lo que contrasta con la idea de extender una clase base para todas las clases de prueba y usar convenciones de nomenclatura para todos los métodos de prueba para que coincidan con el patrón **text XYZ,** como se hizo en versiones anteriores.

Nota: JUnit 5 representa la próxima generación de JUnit. Utilizarás las capacidades de programación introducidas a partir de Java 8; podrás construir pruebas de forma modular y jerárquica; y las pruebas serán más fáciles de entender, mantener y ampliar.

Para administrar las dependencias de JUnit 5 de manera eficiente, es lógico trabajar con la ayuda de una herramienta de compilación. Por ahora, usaremos Maven, una herramienta de compilación muy popular.

Lo que necesitas saber ahora son las ideas básicas detrás de Maven: configurar su proyecto a través del archivo pom.xml, ejecutar el comando mvn clean install y comprender los efectos del comando.

Para poder ejecutar pruebas desde el prompt del sistema, asegúrate de que tu archivo de configuración pom.xml incluya una dependencia del proveedor JUnit para el complemento **Maven Surefire**. Así es como se ve esta dependencia.

```
<build>
  <plugins>
   <plugin>
<artifactId>maven-surefire-plugin</artifactId>
```

```
      <version>2.22.2</version>
       </plugin>
</plugins>
</build>
```

Para ejecutar las pruebas, la carpeta bin del directorio Maven debe estar en la ruta del sistema operativo. También debes configurar la variable de entorno JAVA_HOME en tu sistema operativo para que apunte a la carpeta de instalación de Java Además, tu versión de JDK debe ser al menos 8, según lo exige JUnit 5.

Probando con JUnit

JUnit tiene muchas características que facilitan la escritura y ejecución de pruebas. Veremos estas características en funcionamiento:

- Instancias de clase de prueba separadas y cargadores de clase para cada prueba de unidad para evitar efectos secundarios
- Anotaciones JUnit para proporcionar métodos de inicialización y limpieza de recursos: @BeforeEach, @BeforeAll, @AfterEach y @AfterAll (a partir de la versión 5); y @Before, @BeforeClass, @After y @AfterClass (hasta la versión 4)
- Una variedad de métodos de afirmación que facilitan la verificación de los resultados de sus pruebas
- Integración con herramientas populares como Maven y Gradle, así como entornos de desarrollo integrado (IDE) populares como Eclipse, NetBeans e IntelliJ

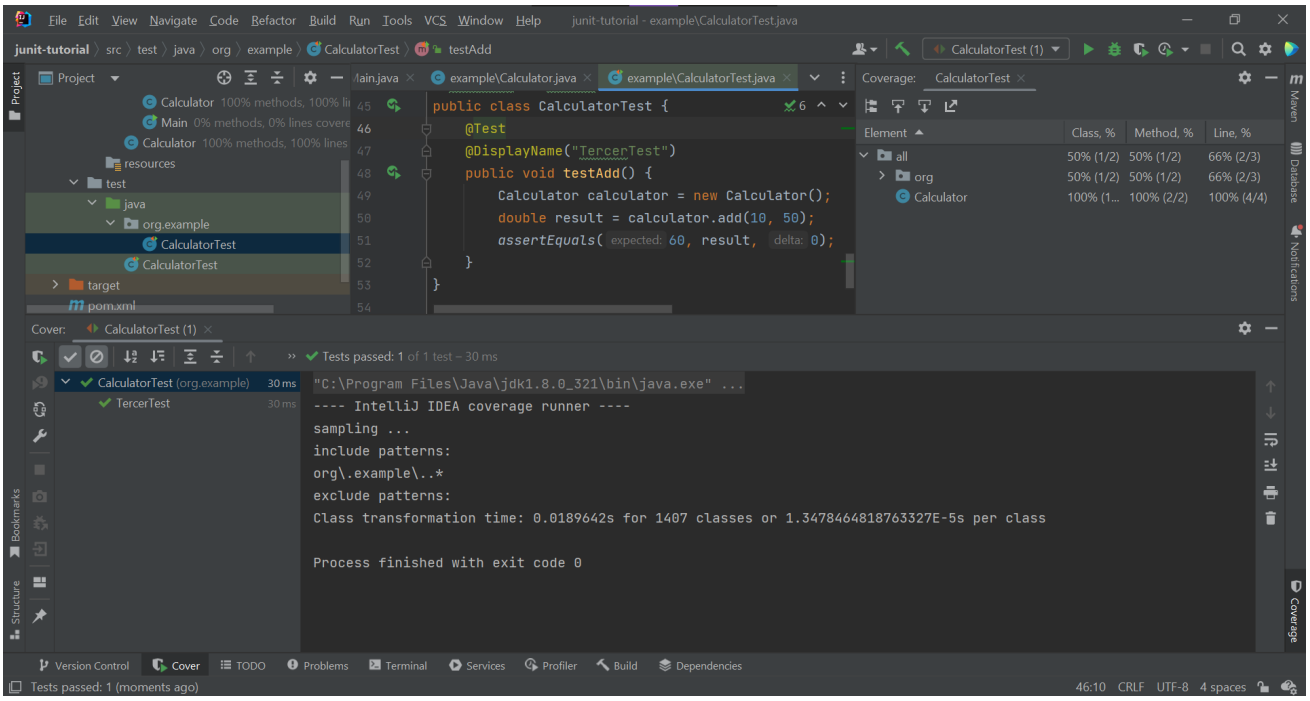
Sin más preámbulos, la siguiente lista muestra cómo se ve la prueba de la calculadora simple cuando se escribe con JUnit.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest {
  @Test
    public void testAdd() {
          Calculator calculator = new Calculator();
          double result = calculator.add(10, 50);
          assertEquals(60, result, 0);
          }
}
```

Pregunta 3: ¿Qué pasa con este programa CalculatorTest?

Del mismo modo, se pone a prueba el método Add y la prueba resulta ser pasada correctamente al igual que en los dos anteriores casos, lo resaltante es la simplificación de código al momento de hacer un uso del método assertEquals para comparar el valor esperado con el obtenido. En general, se ve más legible el código gracias a hacer uso de JUnit y está más apegado a las características de las pruebas en cuanto a que puede ser ejecutado independientemente de las demás pruebas. A continuación, se puede ver el test ejecutado con cobertura.

Como observación, al momento de obtener una prueba fallida, se puede ver el valor esperado y el actual al momento de presentar la falla.



En 5, el framework JUnit comienza a brillar. Para verificar el resultado de la prueba, llama a un método assertEquals, que se importó con una importación estática en la primera línea de la clase. El Javadoc para el método assertEquals es:

```
/**
* Assert that expected and actual are equal within the non-negative delta.
* Equality imposed by this method is consistent with Double.equals(Object)
* and Double.compare(double, double). */
public static void assertEquals(
    double expected, double actual, double delta)
```

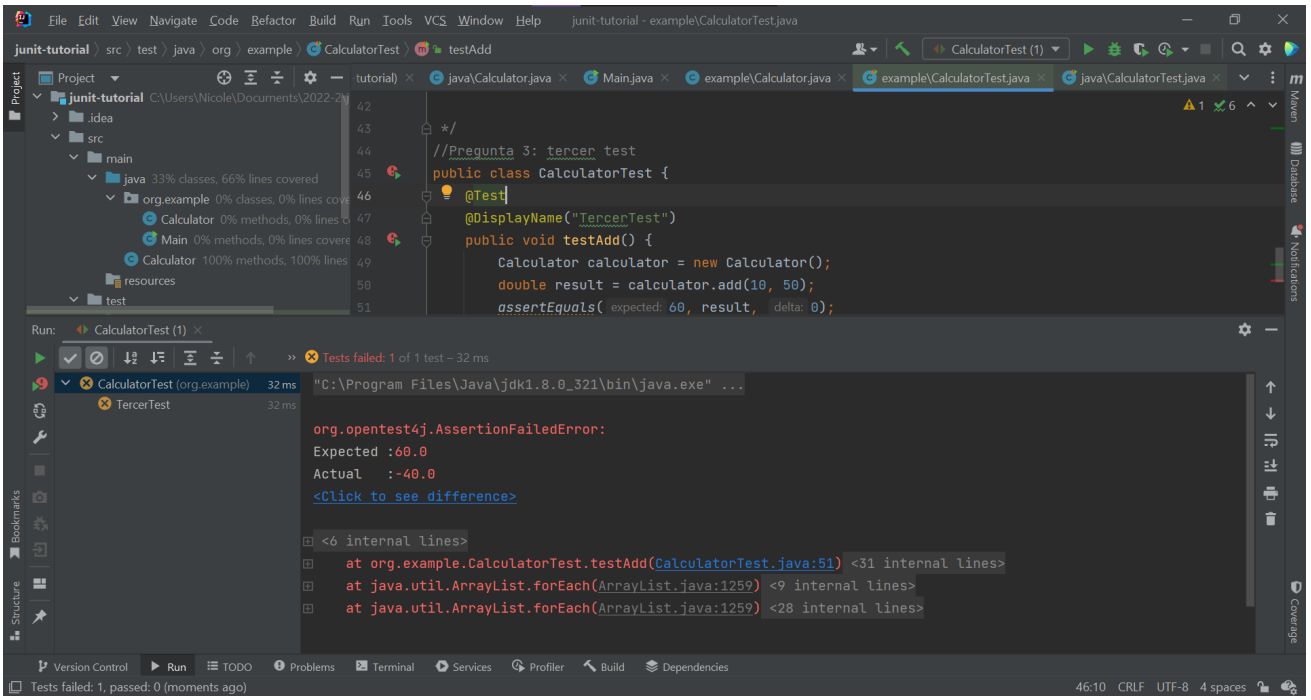
Pregunta 4: En el listado anterior, pasas estos parámetros a **assertEquals** :
expected = 60
actual = result
delta = 0
Describe lo que sucede en este caso.

Como sabemos la respuesta que se espera al utilizar el método Add() con los parámetros 10 y 50, es el valor que toma expected: 60 y actual toma el valor del resultado al usar el método, el delta es el indicador de error que señala la diferencia máxima esperada entre el valor expected y actual para el cual ambos todavía se pueden considerar como iguales, por así decirlo, es como la tolerancia. Vemos esos parámetros en el assertEquals de la tercera prueba y apreciamos que pasa la prueba.

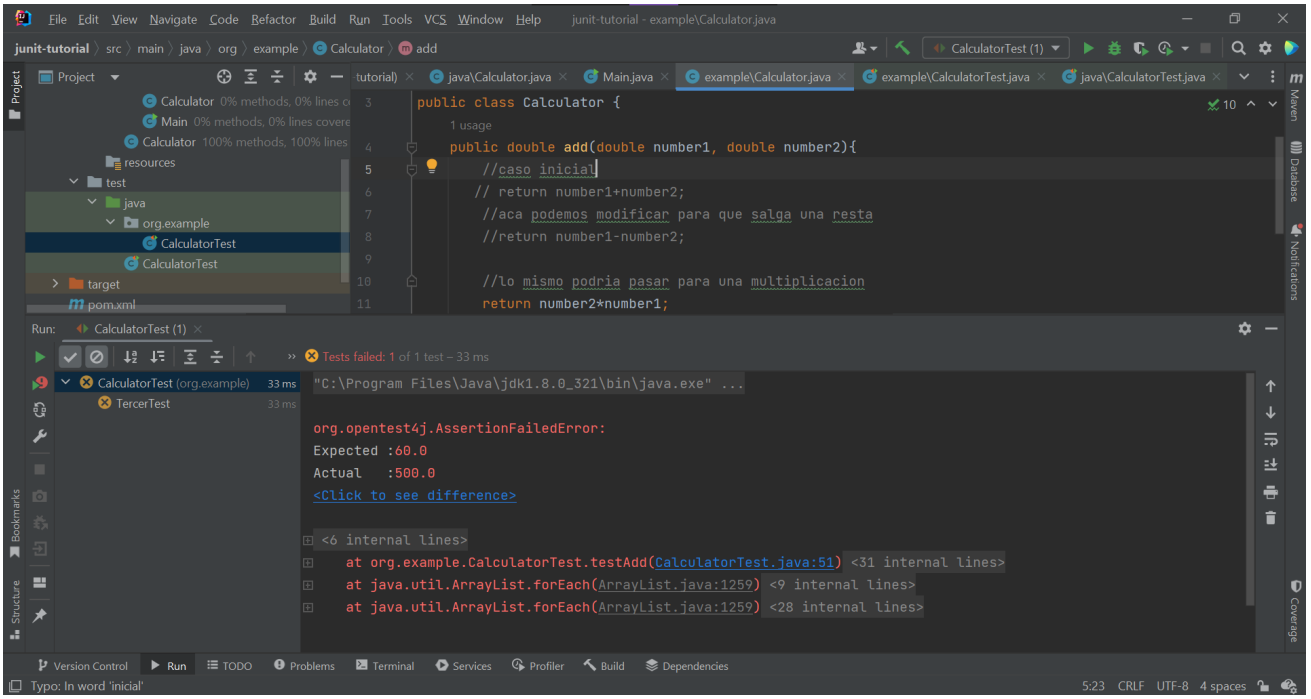
Lo notable de la clase JUnit CalculatorTest del listado anterior es que el código es más fácil de escribir que el primer programa CalculatorTest. Además, puedes ejecutar la prueba automáticamente a través del marco JUnit.

Pregunta 5: Puedes modificar la clase Calculator para que tenga un error; por ejemplo, en lugar de sumar los números, los resta. Luego puedes ejecutar la prueba y ver cómo se ve el resultado cuando falla una prueba.

Al cambiar la clase Calculator haciendo que reste los números en lugar de sumarlos:



También podríamos hacer lo mismo con otras operaciones elementales



Pregunta 6: Revisa el siguiente vídeo IntelliJ IDEA. Adding a Project to Git and GitHub y sube el proyecto realizado en esta actividad con un nombre adecuado.