|  |  |  |
| --- | --- | --- |
| entornos de desarrollo | | CASO PRACTICO 1UD3 |
|  | | |
|  | | |
|  |  | |
| alumno cesur 24/25  Alejandro Muñoz de la Sierra | PROFESOR  Diego Tinedo Rodríguez | |

# introduccion

Este caso práctico se ocupa de hacer pruebas unitarias con JUnit. El propósito es verificar que las clases matemáticas simples operan bien en Java. Vamos a crear tres clases: Resta, Multiplicación y División, cada una con sus variables enteras y métodos para sus operaciones. Luego, haremos pruebas unitarias para confirmar que esos métodos funcionan correctamente.

En el desarrollo de aplicaciones web, es muy importante que los cálculos matemáticos se realicen bien. Si algo va mal, puede causar fallos grandes en todo el sistema. Para evitar problemas, usamos pruebas unitarias. Estas pruebas nos dejan examinar cada método por separado antes de integrarlo en el proyecto final. En este caso práctico, vamos a hacer pruebas unitarias para operaciones matemáticas simples en Java, utilizando JUnit en Eclipse. Esto nos ayudará a garantizar que todo funcione bien desde el comienzo.



# 01

# configuracion de junit

Para verificar si JUnit está bien puesto en nuestro proyecto, hacemos unos pasos. Primero, clic derecho en el proyecto y elegimos Propiedades. En la ventana que aparece, vamos a Java Build Path y veremos varias pestañas. Aquí, elegimos Libraries y hacemos clic en Add Library.

En la siguiente pantalla, seleccionamos JUnit y pulsamos Next. Asegurémonos de elegir la versión correcta, usualmente JUnit 5, aunque se puede elegir otra si es necesario. Después, verificamos que la opción esté en Classpath. Este es el lugar correcto porque Eclipse lo usará para gestionar lo que necesitamos para nuestro código.

Ahora, ¿por qué usamos Classpath en vez de Modulepath? Porque Classpath es el método tradicional para manejar librerías en proyectos Java. Es bueno para proyectos como los que hacemos en DAM, ya que JUnit funciona bien aquí. Modulepath solo es útil cuando trabajamos con módulos de Java desde la versión 9. Si no usamos módulos (lo cual es común en proyectos más simples), Classpath siempre será mejor.

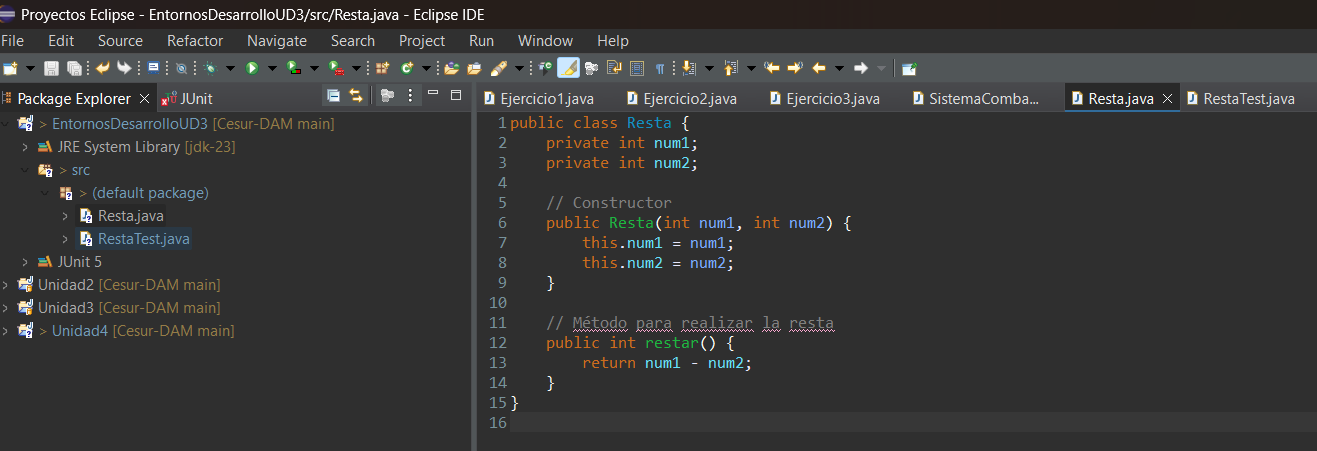
En resumen: al configurar JUnit, solo lo agregamos al Classpath. Y si alguna vez tienes dudas entre Classpath y Modulepath, recuerda que en la mayoría de los casos, sobre todo en entornos académicos o proyectos pequeños, Classpath será la mejor opción.

# 02

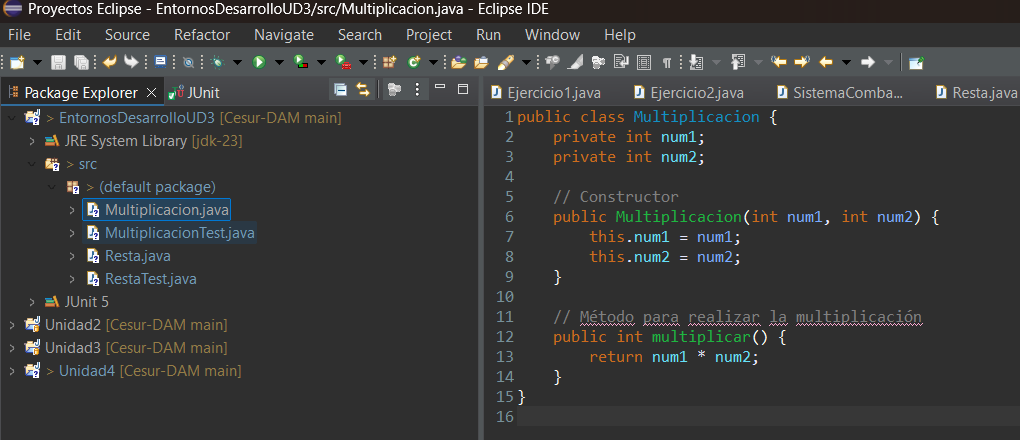
Implementación de las clases en Java

Utilizamos el contenido aprendido en la asignatura de programación, para crear varias clases simples que ejecuten las operaciones Resta, Multiplicación yDivisión, sobre dos números enteros como inputs. Estas clases posteriormente, serán puestas a prueba con otras clases y JUnit.

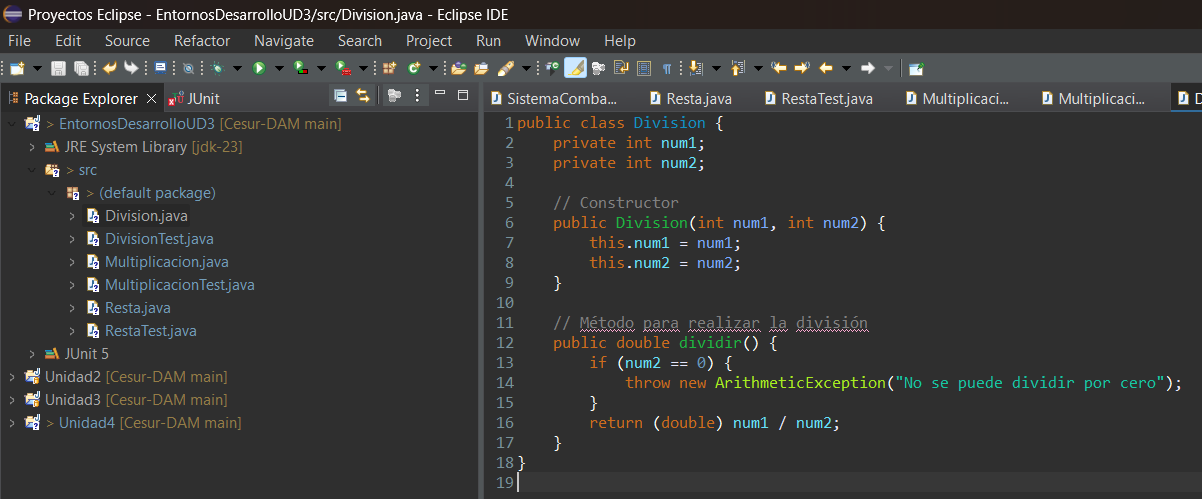
**2.1 Clase Resta:**



**2.2 Clase Multiplicación:**



**2.3 Clase División:**

****

Una vez que tenemos nuestras clases y JUnit listo, el siguiente paso es crear las clases de prueba. Esto nos ayuda a confirmar que nuestro código funciona como esperamos. Aquí te explico cómo hacerlo:

**1. Crear una clase de prueba**

Primero, vamos a crear una nueva clase solo para las pruebas. Esta clase puede estar en el mismo paquete que el código principal o en un paquete distinto solo para pruebas. Es recomendable añadir el sufijo Test al nombre (por ejemplo, MiClaseTest), aunque no es obligatorio. En esta clase, escribiremos los métodos que probarán nuestras clases principales.

**2. Escribir los métodos de prueba**

Cada método debe tener la anotación @Test, que indica a JUnit que es una prueba. También, cada prueba debe ser independiente. Esto significa que cada método debe configurar su propio entorno y comprobar solo una parte del código. Así, si algo falla, sabremos exactamente qué parte tiene problemas.

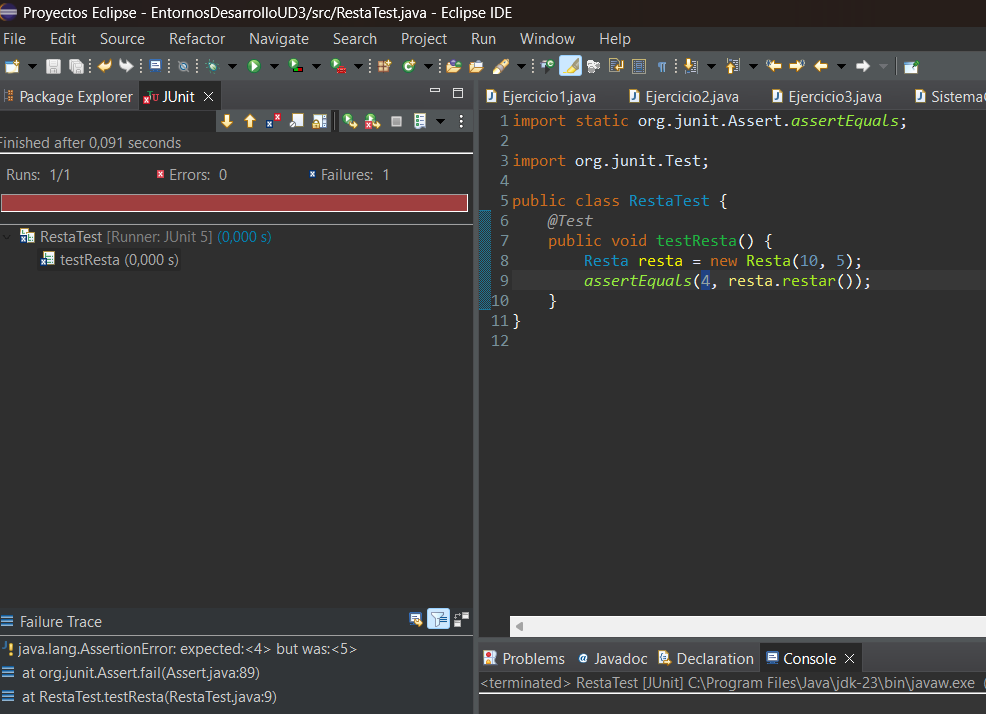
**3. Configurar las pruebas con JUnit**

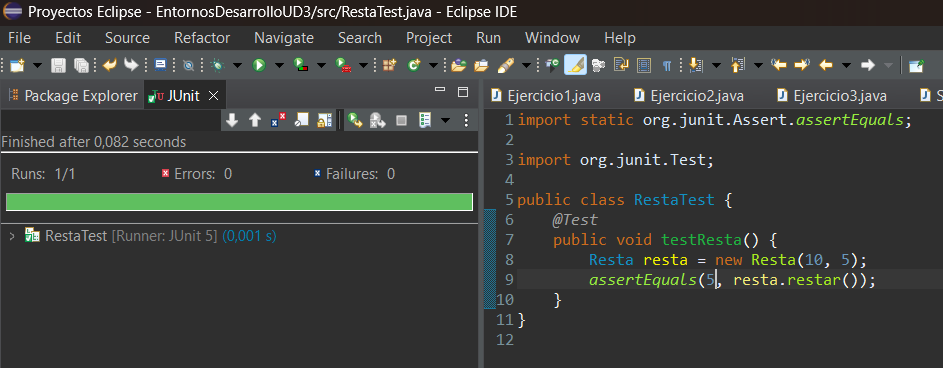
Para ver si nuestros métodos funcionan bien, podemos crear instancias de las clases a probar dentro de los métodos de prueba. Luego, llamamos a sus métodos y comparamos los resultados con lo que esperamos. Usamos las aserciones que JUnit nos ofrece.

# 03

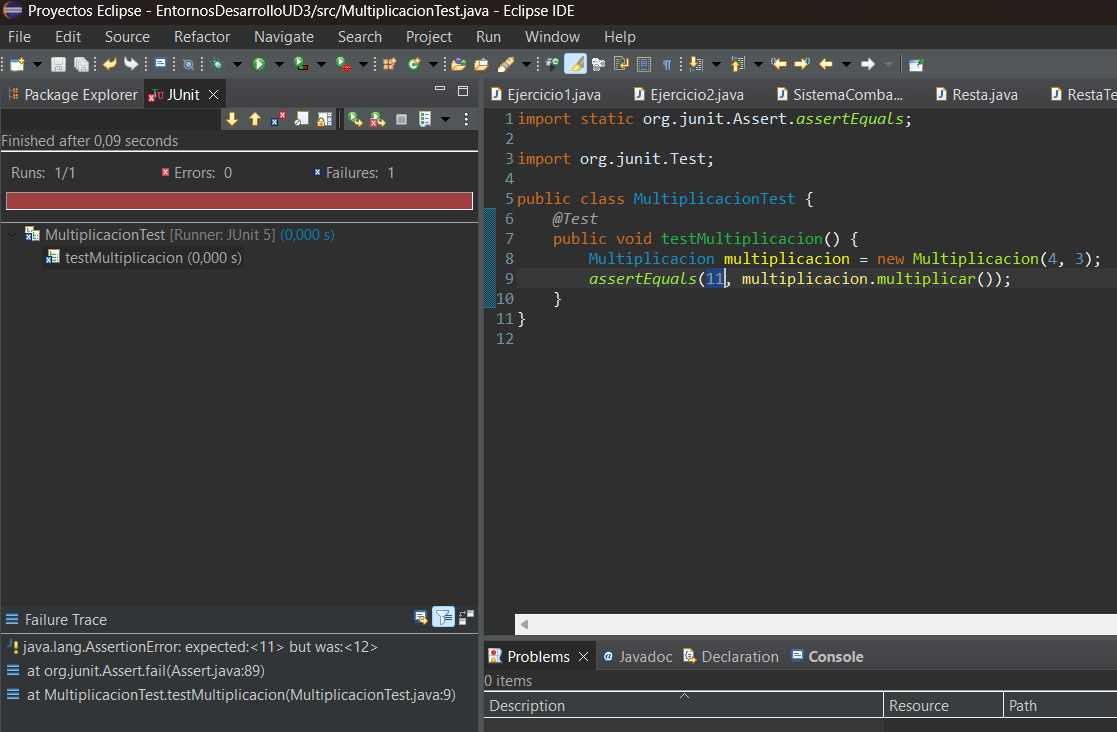
Implementación de las pruebas unitarias con JUnit

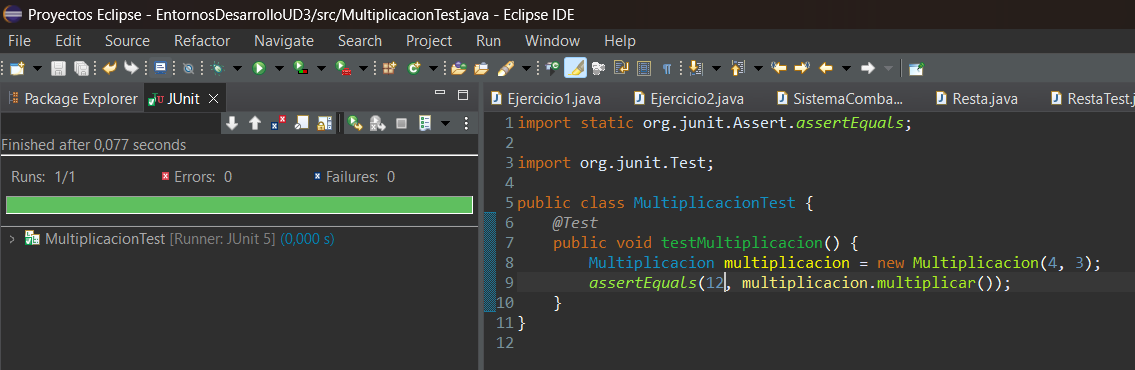
**3.1 Pruebas para la clase Resta:**



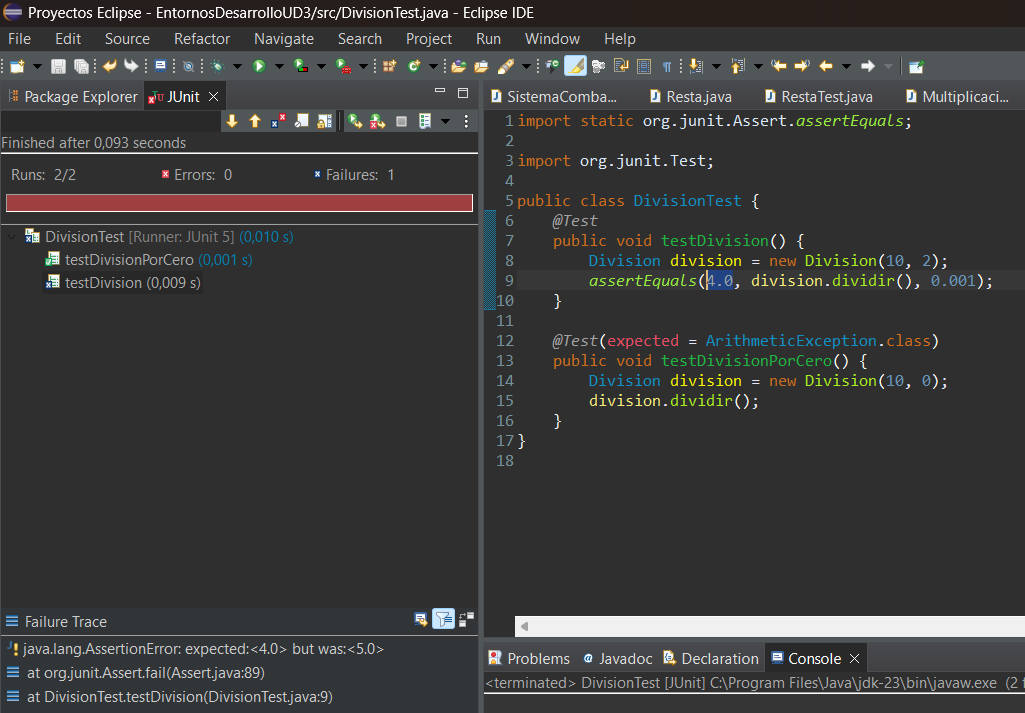


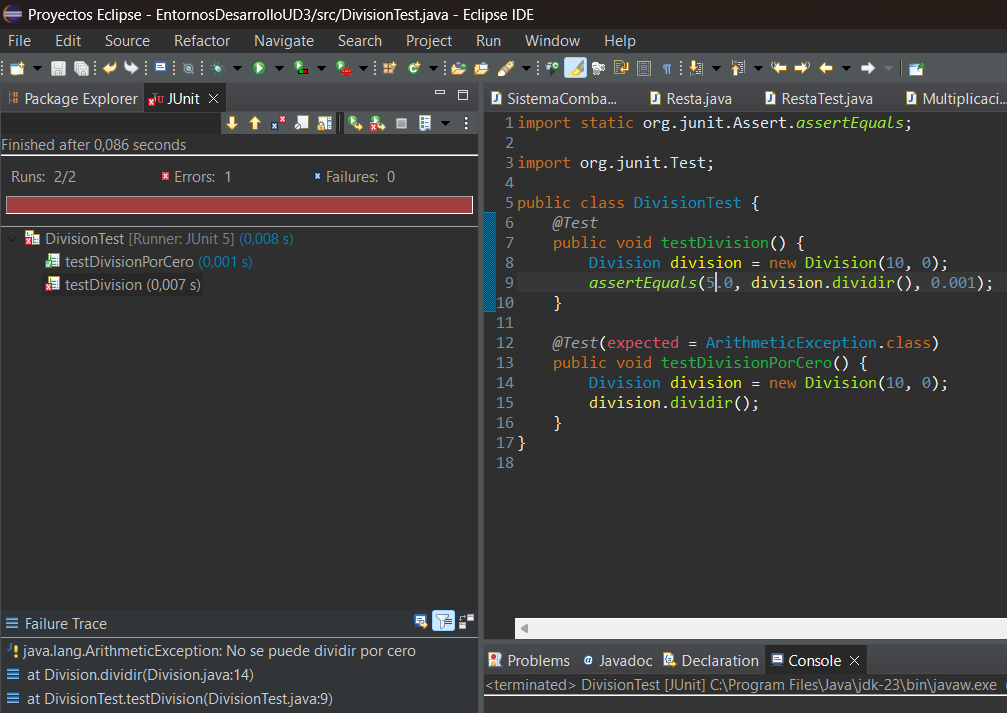
**3.2 Pruebas para la clase Multiplicación:**

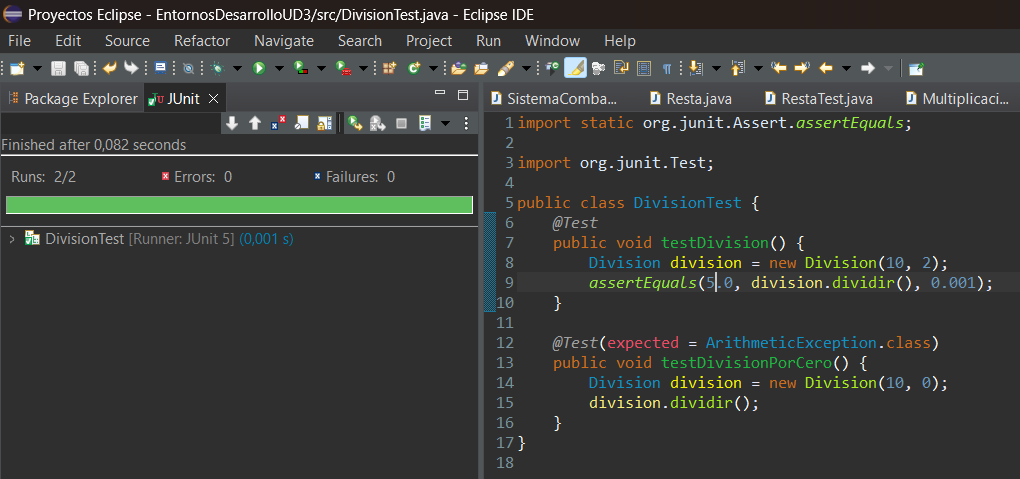




**3.3 Pruebas para la clase División:**







# 04

Documentación del Plan de Pruebas

Después de que se hicieron las clases y sus métodos, hicimos pruebas para verificar que cada operación matemática funciona bien. Aquí abajo están los casos de prueba y sus resultados:

Resta: Verificamos que la resta funcione con los números 10 y 5. El resultado fue 5 , como esperábamos. Mostramos que falla con el resultado 4.

Multiplicación: Para la multiplicación, usamos los números 4 y 3. El resultado fue 12 , que era lo que previmos. Mostramos que falla con el resultado 11.

División: Para la división, usamos 10 y 2. El resultado fue 5.0 , lo que muestra que funcionó bien. Mostramos que falla con el resultado 4.

División por cero: Por último, verificamos un caso especial: la división por cero. Cuando dividimos 10 entre 0, el sistema lanzó una excepción, como esperábamos.

En conclusión, todos los casos de prueba funcionaron como se previó, lo que indica que las clases y métodos realizados están bien diseñados y operan correctamente.

# 05

# Conclusiones

Trabajar con pruebas unitarias ha sido una experiencia útil, porque permiten encontrar errores desde el inicio del desarrollo. Esto es importante porque si encontramos problemas pronto, gastamos menos tiempo y esfuerzo en arreglarlos después. En este caso práctico, al usar pruebas con JUnit en Eclipse, hemos verificado que los métodos matemáticos funcionen bien, incluso en casos como la división por cero, que podrían no ser detectados sin esta validación.

Además, he notado que la automatización de las pruebas hace el proceso mucho más simple. Antes, tendría que ejecutar manualmente cada caso para ver si todo funcionaba, pero con JUnit, eso se hace automáticamente. Esto no solo ahorra tiempo, sino que también reduce el riesgo de olvidar algún detalle importante.

Usar herramientas como JUnit es esencial para asegurar que nuestro software sea fuerte y confiable. Mejora la calidad del código y reduce el tiempo que dedicamos a arreglar errores, minimizando los costos de mantenimiento posteriores. Para proyectos futuros, podría ser útil ampliar las pruebas e incluir casos extremos o escenarios complejos, asegurándonos de que el código sea resistente ante cualquier situación.

En conclusión, las pruebas unitarias son una buena inversión en tiempo y calidad. Aunque exigen un poco más de esfuerzo al inicio, los beneficios a largo plazo son claros.

# 07

# REFERENCIAS

<https://junit.org/junit5/>

<https://www.youtube.com/watch?v=wkXL3emg-NU>

<https://www.baeldung.com/junit-5>

<https://docs.oracle.com/en/java/>

<https://www.w3schools.com/java/java_ref_reference.asp>