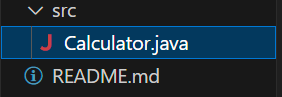
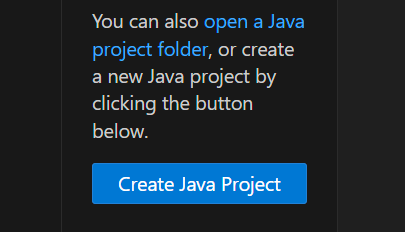
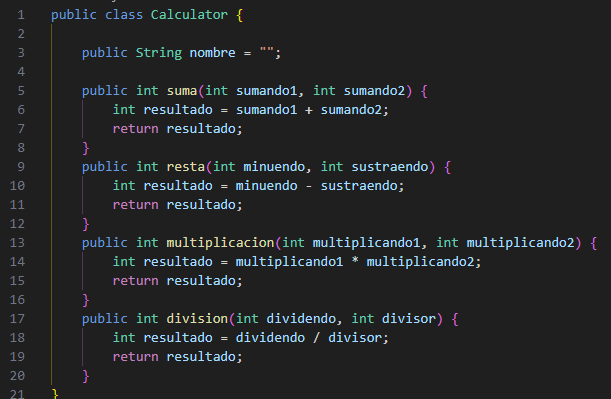
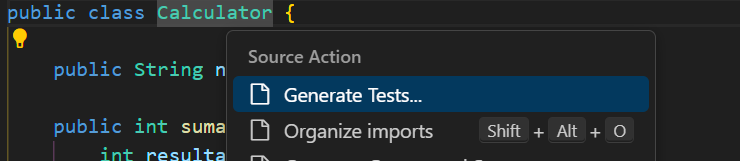
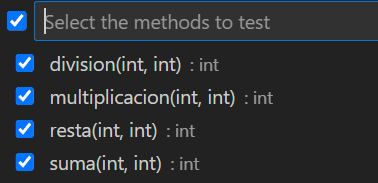
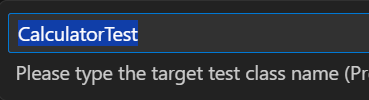
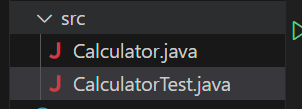
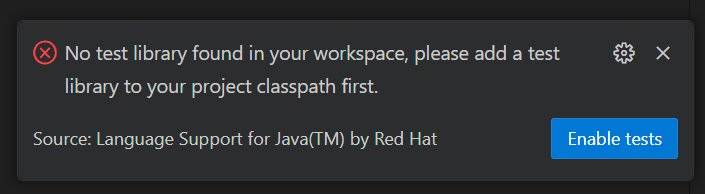
Tarea: Depuración y pruebas

# Ejercicio 1: Calculadora Simple

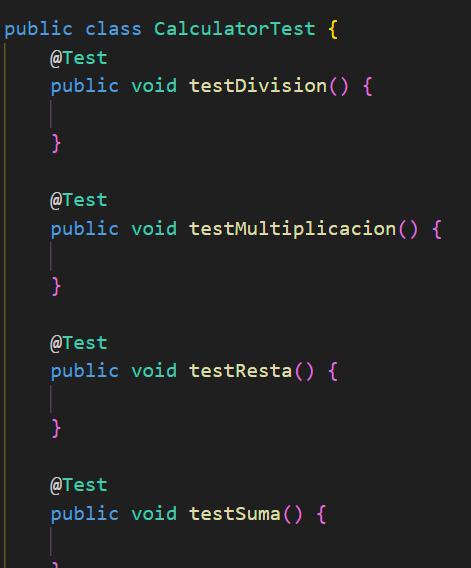
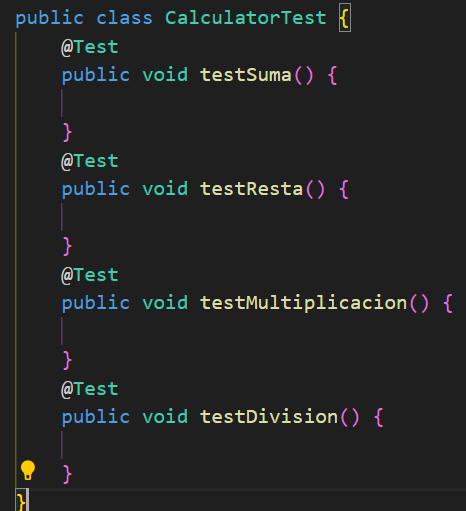
Lo primero que se debe hacer es crear un proyecto de java nuevo, en él se crea un archivo .java que será donde se escribirá el código del programa de la calculadora, con todos los métodos requeridos, así como nombre de atributos diferentes para los datos de los diferentes métodos para poder diferenciar los tests bien y de manera clara y concisa posteriormente.



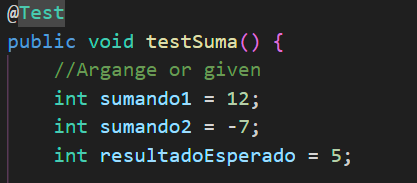
Una vez el programa de la calculadora con todos sus métodos esté completado se debe habilitar y generar los test, lo que dará lugar a un nuevo archivo .java, que se podrá nombrar, en el que se almacenarán los tests para los diferentes métodos previamente creados. 



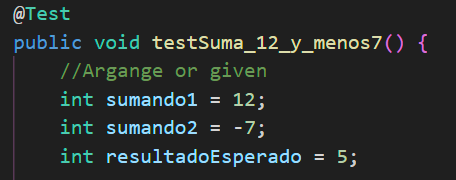
Dentro de este archivo deberá antes de nada ordenarse los diferentes tests ya automáticamente creados por Visual Studio según su orden real en el archivo de la calculadora, puesto que por defecto viene con otro orden y por tanto es menos intuitivo.

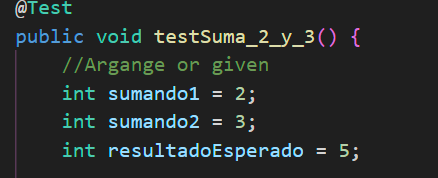


Una vez hecho eso, se procederá a elaborar el código para cada uno de los tests, el cual en este ejercicio concreto será muy similar para todos los métodos, ya que se trata simplemente de operaciones básicas. Para ello lo primero será dividir el proceso del test en 3 etapas: Arrange/given, la sección donde se establecen los datos que se usarán para realizar el test así como resultantes esperados en base a ellos o datos similares, además de inicializar la calculadora.



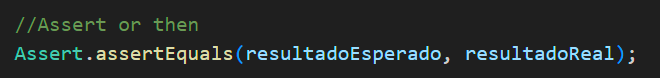
Puesto que con este tipo de test solo se está comprobando una suma particular entre 2 valores establecidos, y no el funcionamiento de la calculadora en base a otros medios, sería preciso añadir en el nombre además una especificación del cálculo que se está haciendo, puesto que en caso de hacerse más tests de suma con otros valores diferentes, lo cual sería útil de hacer para comprobar que el método funciona correctamente y no ha sido coincidencia, no habría forma de diferenciar ambos tests.



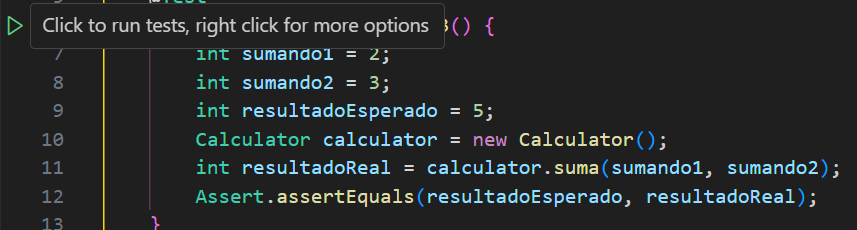
**(En base a esto deberían hacerse al menos 2 tests diferentes para cada método)**

2º Etapa: Act/when, la parte del código donde se realiza el cálculo/comprobación del código original, debido a haber inicializado la calculadora previamente, ahora es posible utilizar el método suma del código original, usando como datos los números establecidos arriba, dando lugar a el resultado que el código original da como respuesta, obviando el resultado esperado.



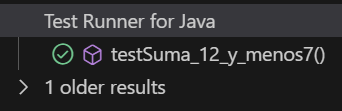
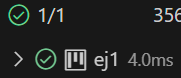
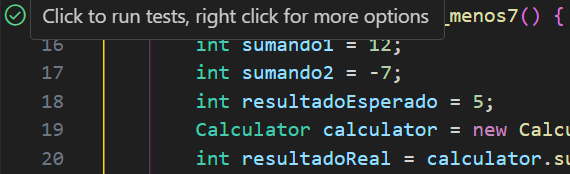
3º Etapa: Assert/then, donde se realiza la comprobación entre el resultado esperado, el que debería dar el programa si estuviese bien estructurado, y el resultado real que este ofrece. 

Para está última etapa se utiliza un assert, es decir, métodos que trae la librería de JUnit que permiten comprobar si el resultado es correcto, debe ser importado. 

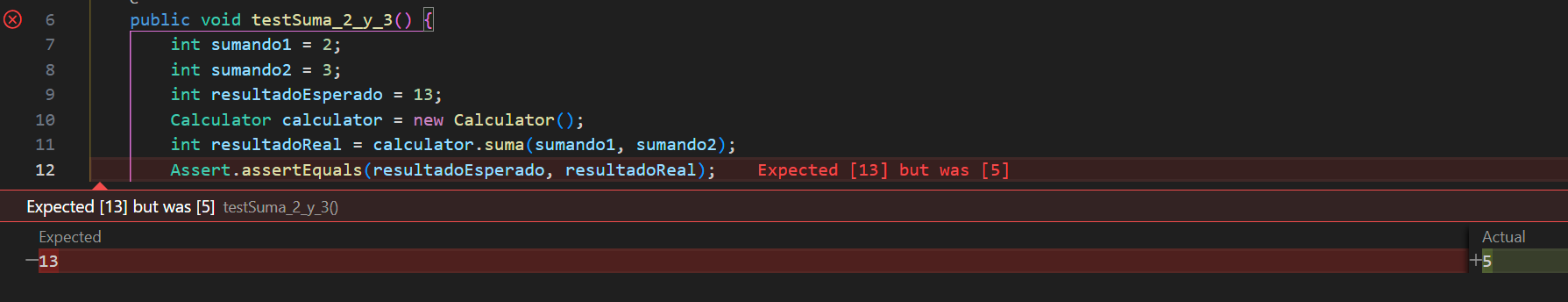
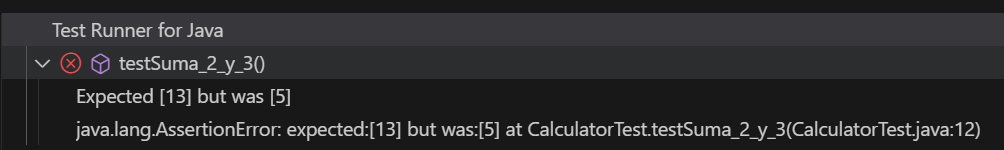
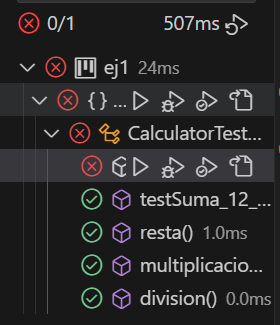
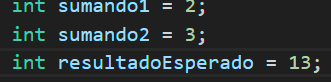
Una vez realizada la comprobación del código con el test, pudiendo hacerlo individualmente test a test, o globalmente todos a la vez, Visual Studio dará el resultado, en caso correcto, no ocurrirá nada y simplemente aparecerá un tick verde al lado del test indicando que es correcto, en caso de fallo, saldrá un aviso que mostrará que la comprobación a fallado, señalando cual es el resultado esperado en comparación con el que el programa a dado. 

**Test Global:**

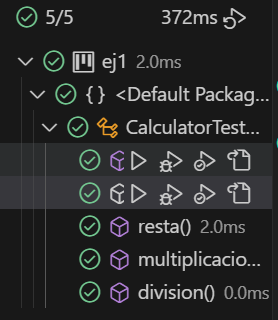
**Test correcto:**



**Test fallido:**

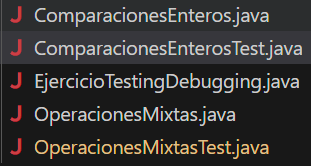


Si tras todo esto finalmente se comprueba mediante cada uno de los test que el código funciona en su completitud bien, habrá finalizado la fase de pruebas y el código estará listo.

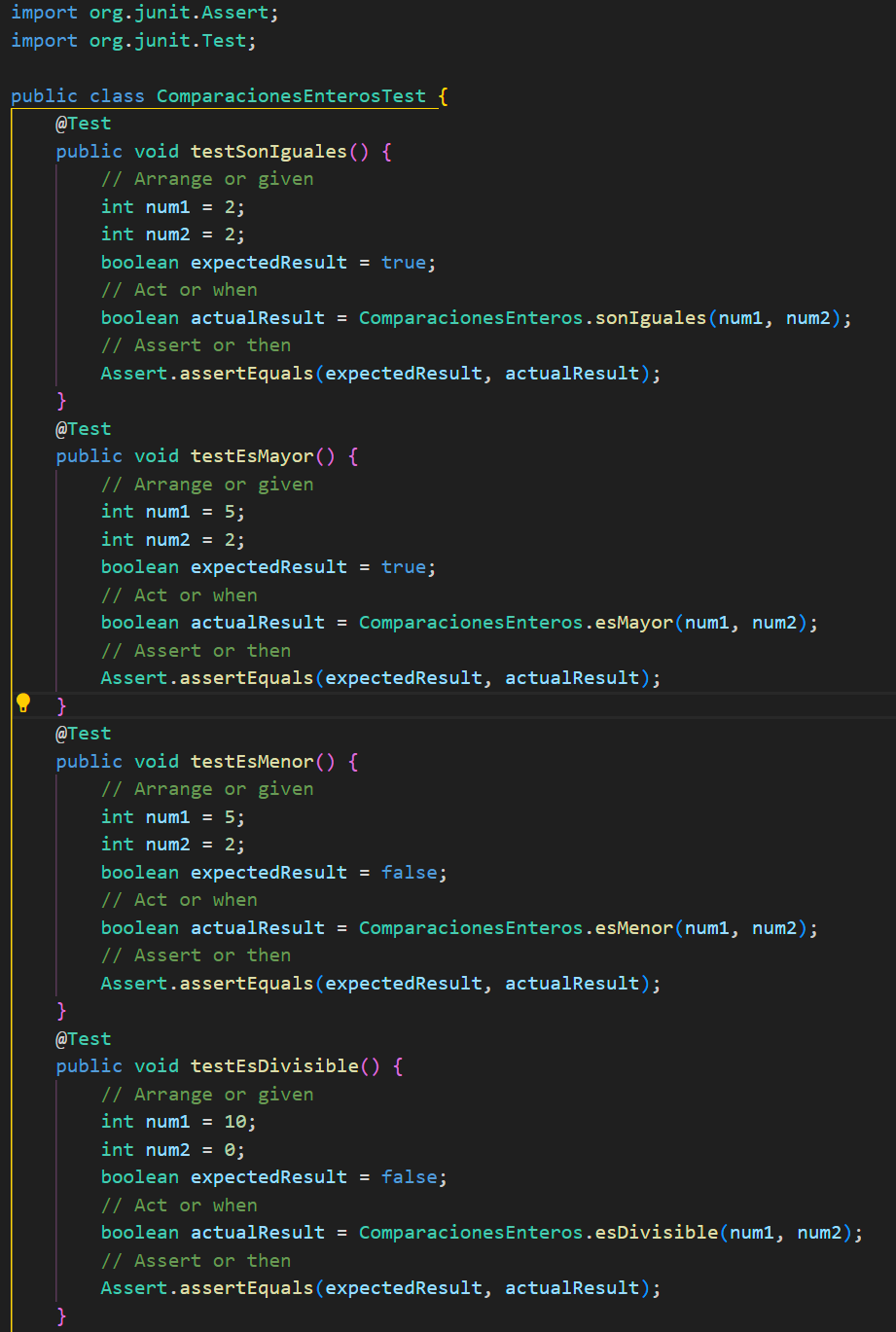


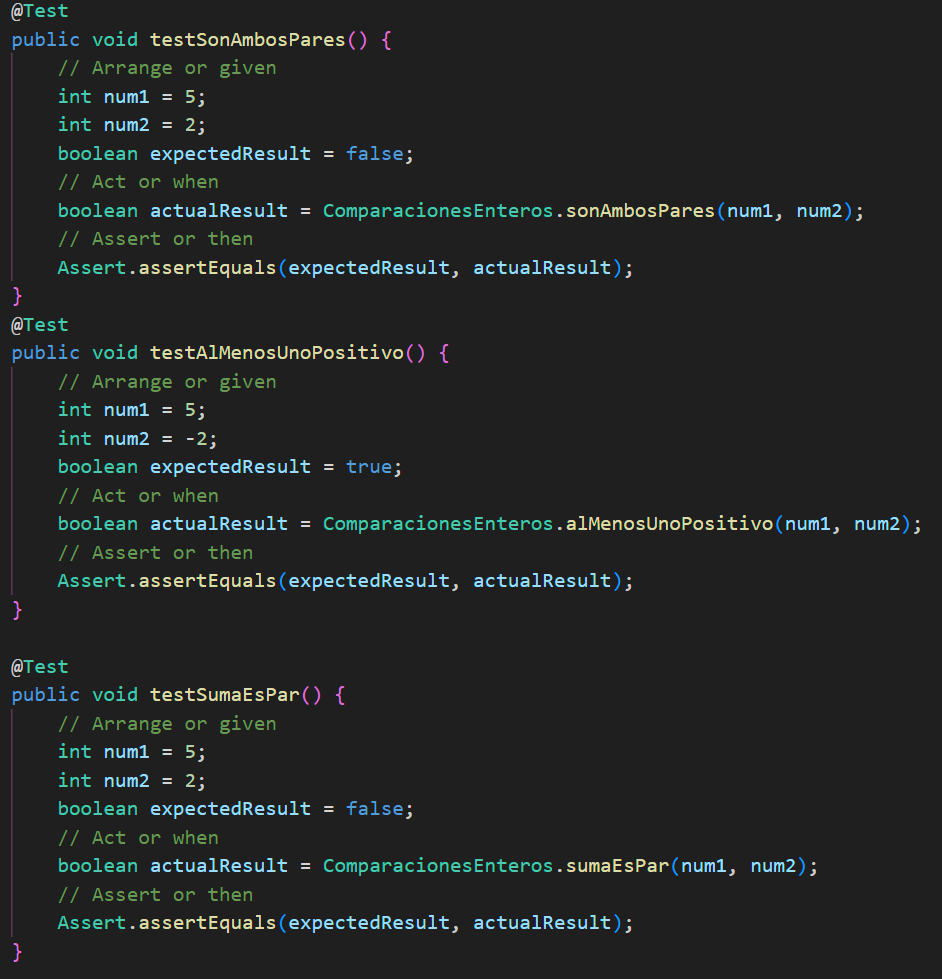
# Ejercicio 2: Testing y Debugging

Lo primero antes de nada es revisar los 3 archivos del código inicial, una vez hecho eso, se crean los 2 archivos con tests unitarios para ComparacionEnteros y para OperacionesMixtas, de la misma forma que en el ejercicio anterior, habilitando los tests, creando la carpeta…

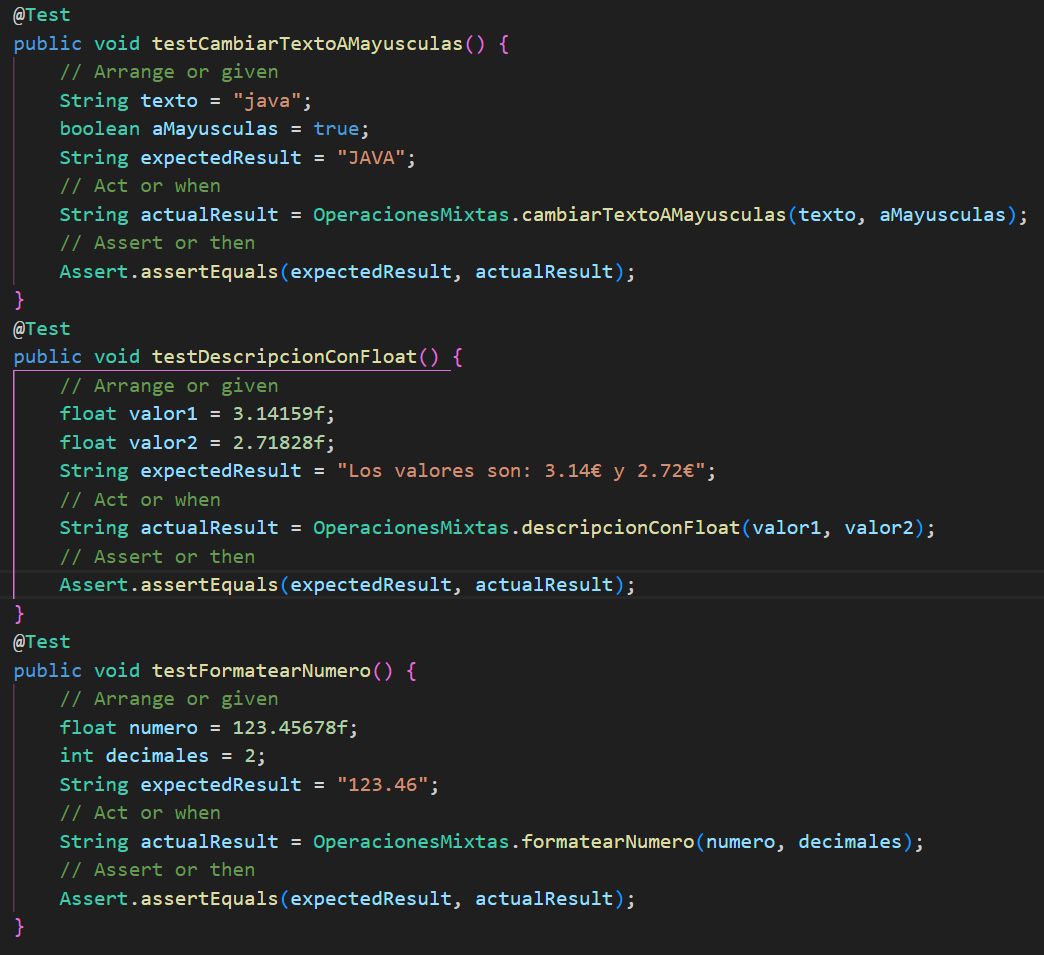


Una vez generados ambos archivos, se diseñan los diferentes tests para cada uno de los métodos de ellos, de nuevo igual que en el ejercicio anterior, es decir, asignando valores de prueba así como el resultado real que el código debería ofrecer en base a ellos y comparándolo con el resultado real que el código genera en base a lo que hay escrito en él, para cada test, o por lo menos los más complejos, sería preciso realizar varios tests con diferentes valores cada uno. Al igual que en el anterior, en este caso solo se hará en base a unos únicos valores para agilizar el ejercicio.

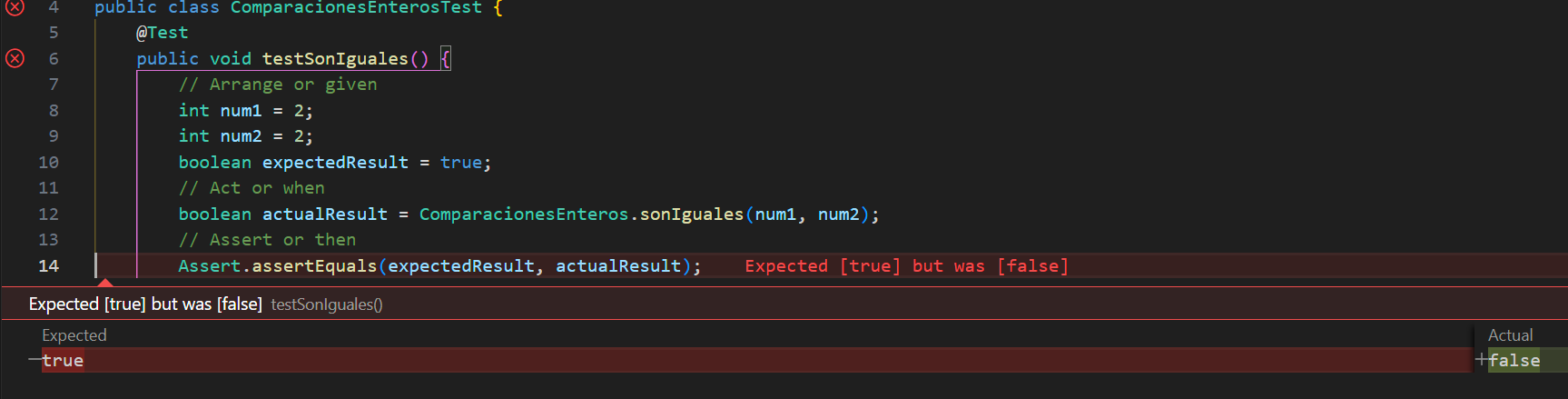


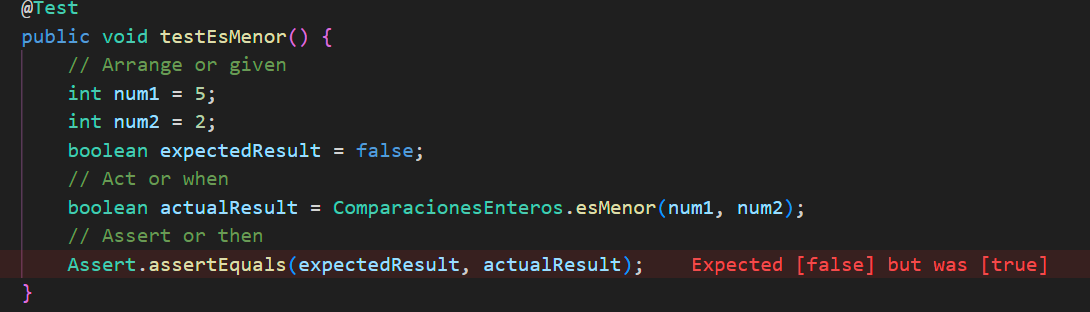


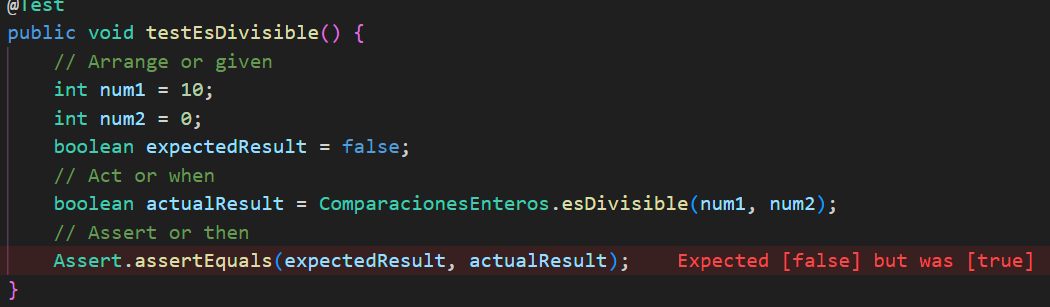


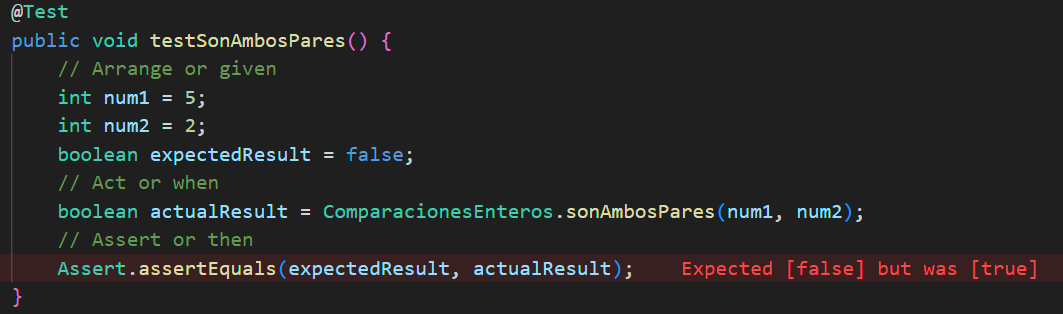


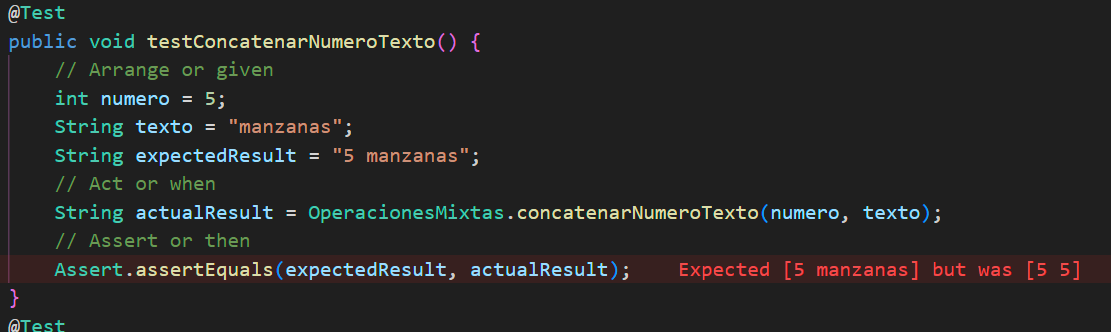
Una vez que ya estén todos los diferentes test de cada uno de los métodos diseñados para cada uno de sus métodos, se ejecutan dichos test, pudiendo ver que tests son superados, significando esto que el resultado que se esperaba es el mismo que el que finalmente se genera, y también que tests no son superados.

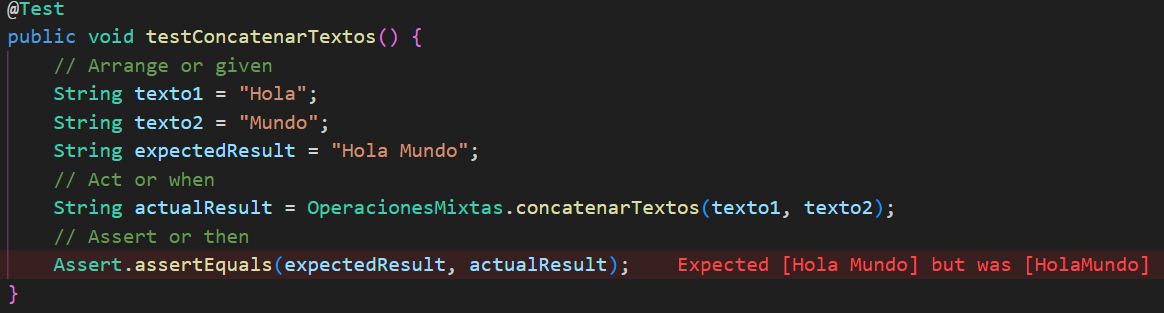


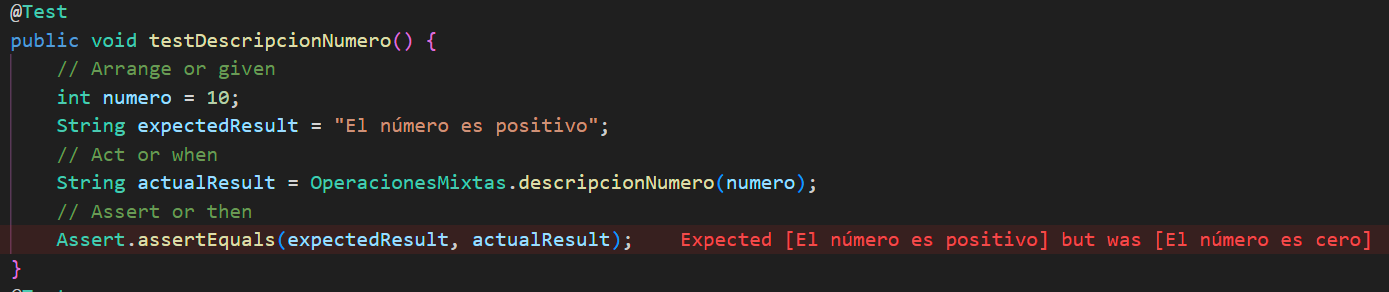


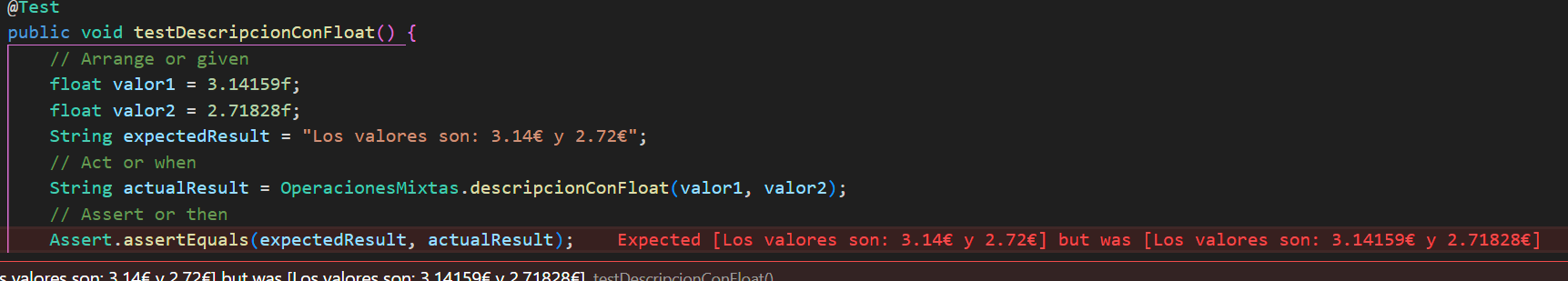




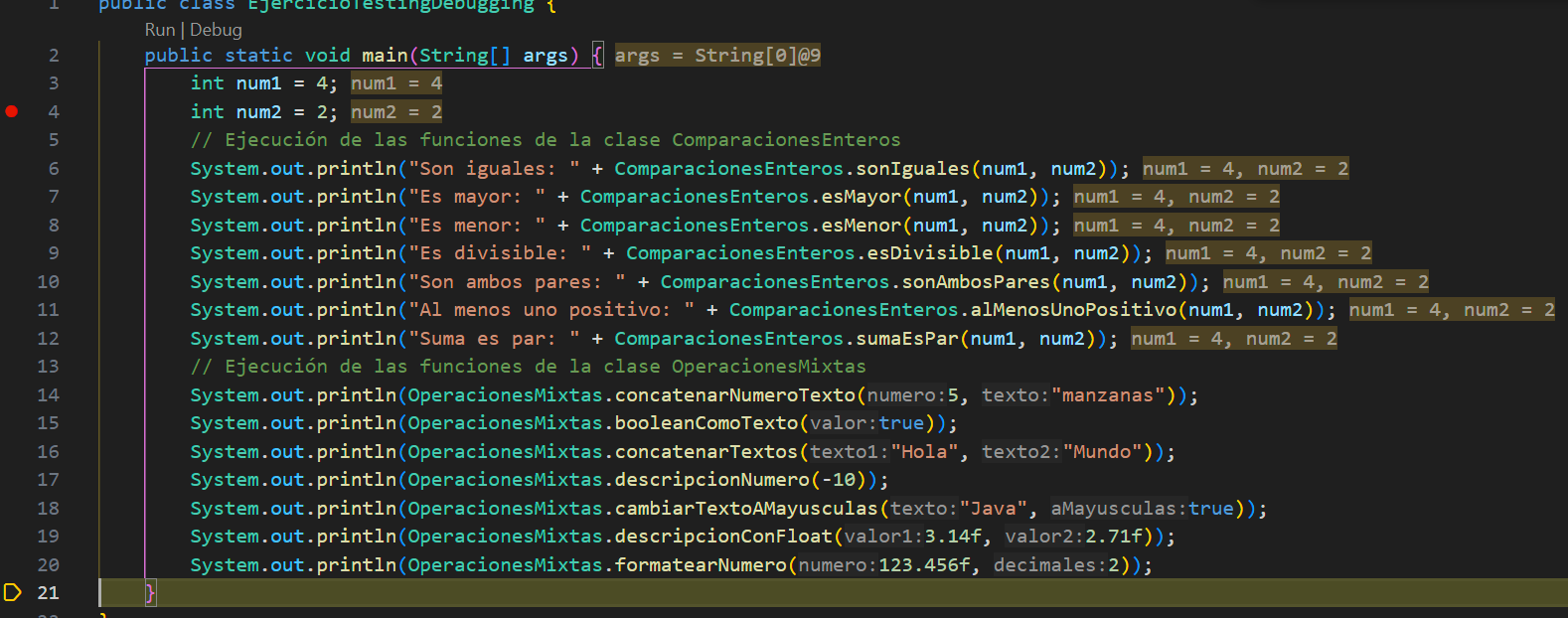


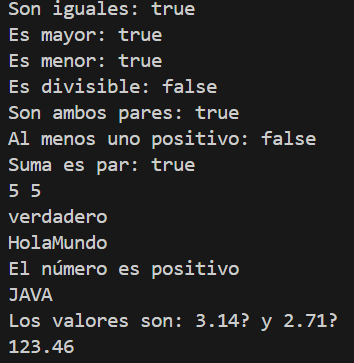






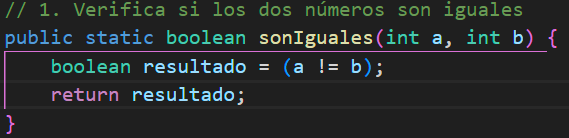
Tras haber identificado los tests fallidos, para poder visualizar fácilmente por qué el código no da el resultado deseado, una forma de hacerlo sería hacer debugging en un archivo en donde los diferentes métodos sean llamados, en este caso EjercicioTestDebugging. Ahí añadiendo un breakpoint y haciendo debug, podrá verse, en base a unos datos preestablecidos, por qué el código no funciona de la forma deseada paso a paso, viendo a que datos cada método llama, los resultados que ofrece y el texto que escribe en consola.



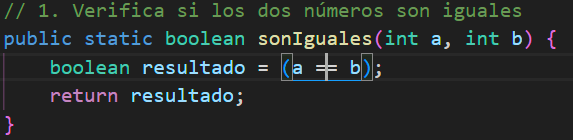


Además de hacer el debugging en base a unos solos datos, también podría intentarse usar otros valores diferentes y volver a hacer los tests, las veces que sean necesarias, comprobando así que el resultado de cada método no ha sido pura coincidencia y verdaderamente no funciona como debería. Una vez hecho el debugging, se habrá visto más claro cómo es que los diferentes métodos fallan y ya será viable dirigirse al código a corregirlo.

Diriéndonos primero al archivo ComporacionesEnteros, el primer método estaría mal puesto que en vez de ver si ambos números son iguales y devolver un true o false, comparaba si eran desiguales y en base a eso ofrecía el resultado.

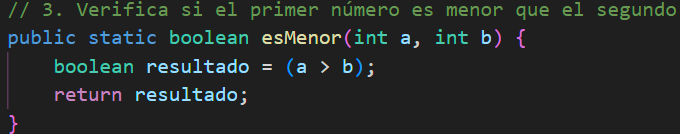


**Corregido:**

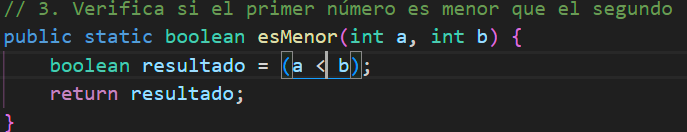


El segundo método estaría correcto.

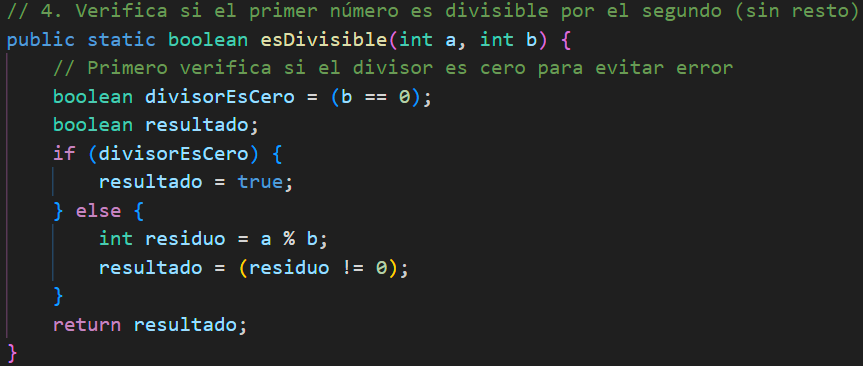
El tercer método estaría mal, ya que en lugar de verificar si el primer número es menor que el segundo, comprobaba si era mayor, al igual que en el anterior.



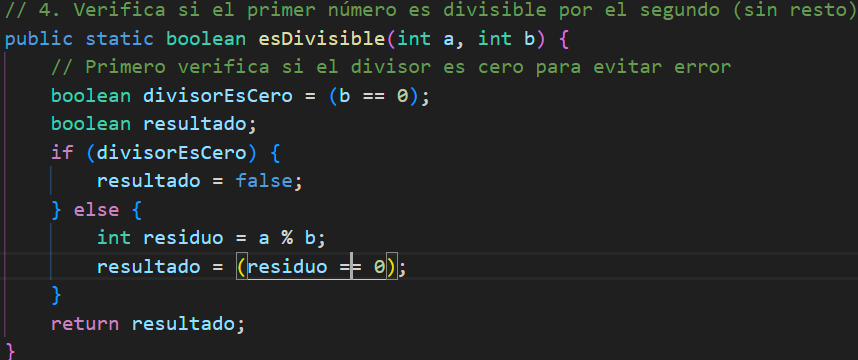
**Corregido:**



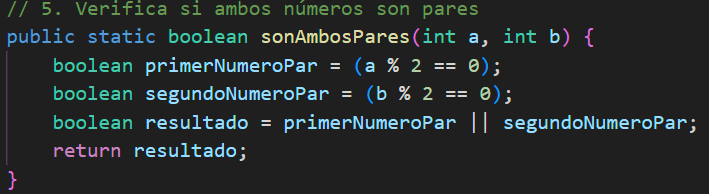
El cuarto método estaría mal, ya que devolvía true si el divisor era 0, ignorando que no se puede dividir por cero, y evaluaba incorrectamente el resto.



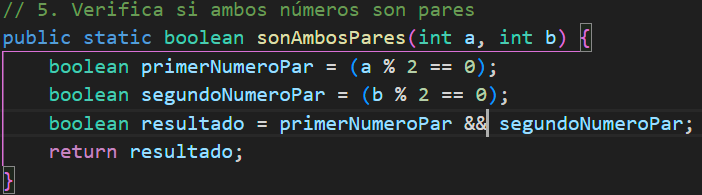
**Corregido:**



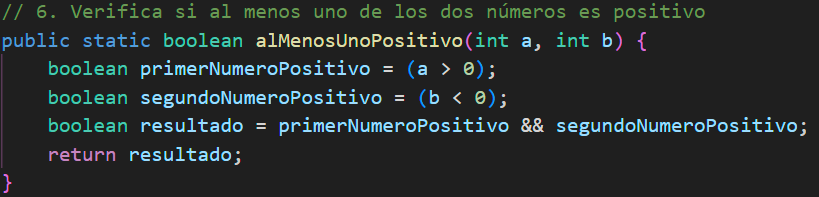
El quinto método estaría mal, ya que utilizaba || en lugar de &&, devolviendo true si cualquiera de los dos números era par, en lugar de verificar que ambos lo fueran.



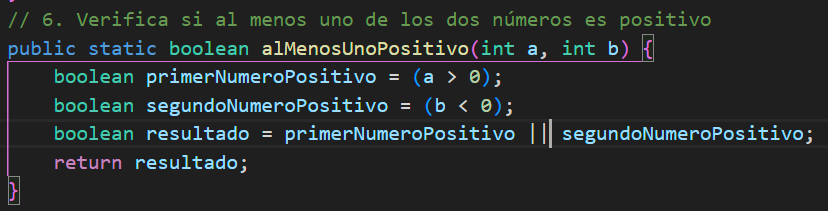
**Corregido:**



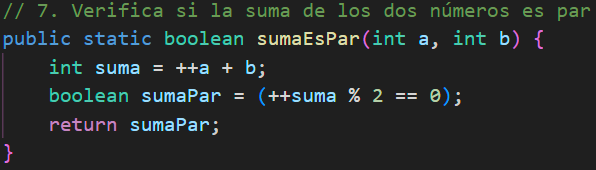
El sexto método estaría mal, ya que utilizaba && en lugar de ||, devolviendo true solo si ambos números eran positivos en vez de si al menos uno lo era, además calcula mal si el segundo número es positivo.

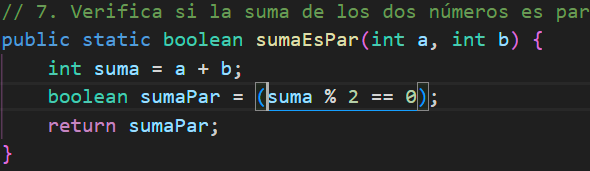


**Corregido:**

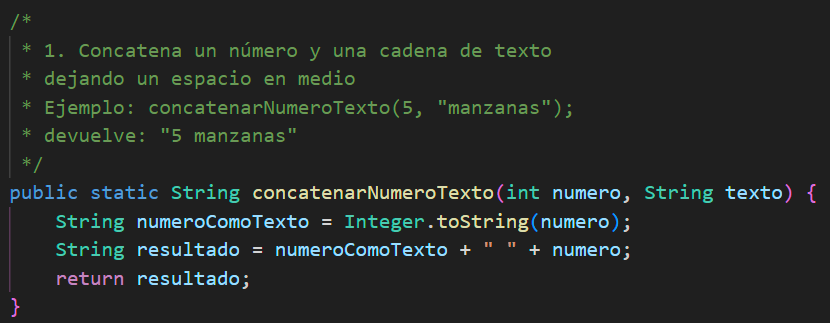


El septimo método estaría mal, ya que incrementaba los valores originales (++a y ++suma), alterando los datos de entrada y evaluando erróneamente si la suma era par.

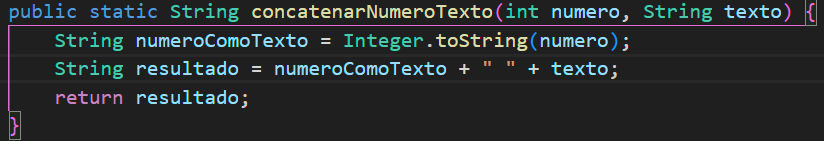


**Corregido:**

En el archivo OperacionesMixtas, el primer método estaría mal, ya que duplicaba el número en lugar de concatenarlo correctamente con el texto.



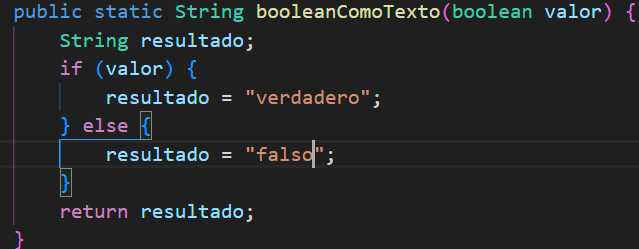
**Corregido:**



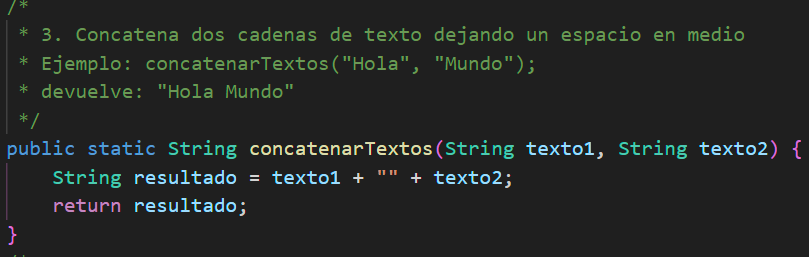
El segundo método estaría mal, ya que siempre devolvía "verdadero", ignorando el valor booleano recibido.



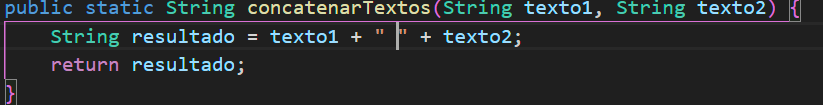
**Corregido:**



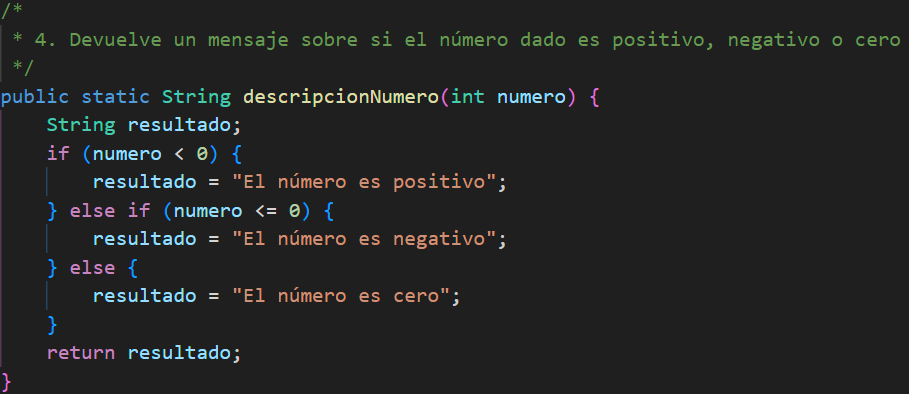
El tercer método estaría mal, ya que no añadía un espacio entre los textos concatenados.



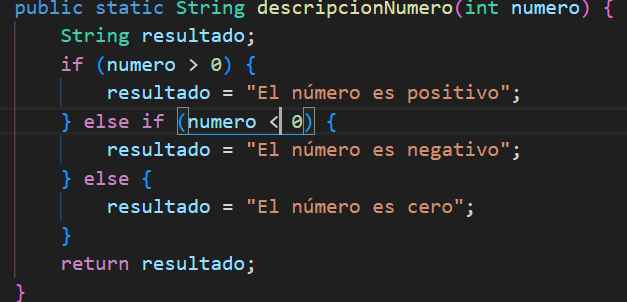
**Corregido:**



El cuarto método estaría mal, ya que los mensajes estaban invertidos y la lógica para determinar si el número era cero era incorrecta.

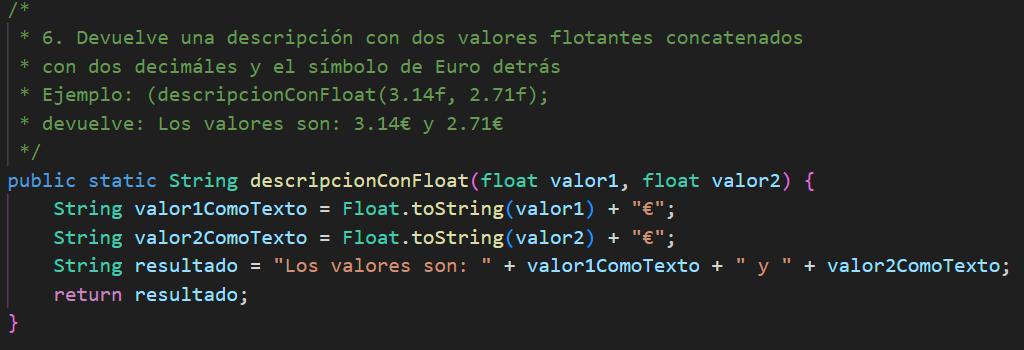


**Corregido:**

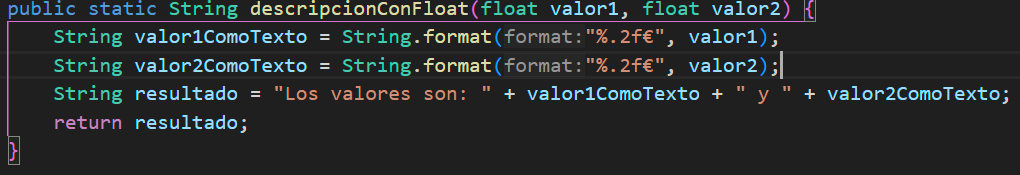


El quinto método estaría correcto, ya que transforma adecuadamente el texto según el valor booleano recibido.

El sexto método estaría mal, ya que no limitaba los valores flotantes a dos decimales como lo indicaba la descripción.



**Corregido:**



El septimo método estaría correcto, ya que formatea adecuadamente el número flotante según los decimales indicados.

Una vez corregido el comportamiento de todos y cada uno de los métodos se podrían volver a hacer los tests unitarios y comprobar si ahora está bien el código, en caso de no estarlo se debería repetir el proceso, haciendo debugging y corrigiendo el código hasta que finalmente el resultado sea el esperado, entonces el programa estará plenamente funcional en base a lo previsto.





