

**ALUMNO: Carlos Barranco Tena y Pedro Zuñeda Diego**

Asignatura: Programación de Sistemas Distribuidos

Curso: 2023/2024  
Semestre: 2º

Fecha: 01-03-2024

## **PRÁCTICA 2: Calculadora RMI**

Para poder realizar y ejecutar este programa necesitaremos un editor como IntelliJ, VS Code o Eclipse y el JDK de Java, que lo puedes descargar de <http://www.java.com/es/> o en los propios editores. Versión de java recomendada java 17.0.9 2023-10-17 LTS. También lo puedes ejecutar en Replit.

Puedes compilar los ficheros con `javac NombreFichero` y ejecutar directamente el Servidor y luego el cliente con `java NombreFichero`. Recuerda lanzar el servidor antes.

Para entregar la práctica hay que subir por un lado este doc en pdf con evidencias y capturas del código y ejecución. Por otro lado 1 zip: Practica2.zip. También puedes subirlo a Github.

### **1. Vamos a hacer una calculadora sobre una base de comunicación RMI. Pero tenemos un problema. Hay partes del código que no están bien, ¿Puedes corregirlas? Spoiler ChatGPT no lo sabe corregir. (5 punto)**

Para este ejercicio había que corregir los errores de los 3 archivos que había de Interfaz.java, Servidor.java y Cliente.java:

- Corrección Interfaz.java:  
La Interfaz tiene un método sumar que solo acepta un parámetro, pero en el cliente y el servidor se intenta usar con dos parámetros. Además, venía sin hacer los métodos de multiplicar y dividir.
- Corrección Servidor.java:  
En el servidor, la implementación de la Interfaz debe coincidir con los métodos definidos en la interfaz. Además, el método multiplicar estaba incorrectamente implementado (sumaba en lugar de multiplicar) y no estaba implementado el método de restar
- Corrección Cliente.java:  
En el cliente, hay un error en el caso de la multiplicación, donde se pasa numero1 dos veces en lugar de numero1 y numero2. Además, se debe agregar la implementación del método restar.

### **2. ¿Te atreves a añadir algún método más? Que haga una raíz cuadrada, o modificar alguno de los que hay para que se puedan añadir más números en cada operación. (2 puntos)**

Para este ejercicio nosotros hemos hecho el método de la raíz cuadrada en el cual utilizamos el `Math.sqrt` que es una función de la clase `Math` para poder realizar la raíz cuadrada.



```
2 usages
@Override
public float raizCuadrada(float numero) throws RemoteException {
    if (numero < 0) {
        return Float.NaN; // Si el número es negativo, devolver NaN
    }
    return (float) Math.sqrt(numero); // Si no, devolver la raíz cuadrada
}
}, port: 0);
```

3. ¿Añadimos tests a este código? ¿Conoces JUnit? Las pruebas unitarias conforman casos de prueba para verificar el funcionamiento correcto de la aplicación. ¿Conoces TDD? (2 puntos)

En este ejercicio creamos una nueva clase llamada “InterfazTexto” utilizando JUnit para poder realizar las pruebas unitarias, el @BeforeEach se ejecuta antes de cada prueba, utilizandose para preparar el entorno de prueba, por ejemplo, inicializar los campos en la clase de prueba, configurar el entorno, etc y el @Test identifica un método como método de prueba.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class InterfazTexto {

    1 usage
    private Servidor servidor;
    6 usages
    private Interfaz interfaz;

    @BeforeEach
    void setUp() throws RemoteException, AlreadyBoundException, NotBoundException {
        // Inicializar el servidor y obtener la interfaz remota
        servidor = new Servidor();
        Registry registry = LocateRegistry.getRegistry(host: "localhost", port: 1200);
        interfaz = (Interfaz) registry.lookup(name: "Calculadora");
    }
}
```

Despues se hacen los test de los distintos metodos de Interfaz de la calculadora.

```

@Test
void testSumar() throws RemoteException {
    assertEquals( expected: 5.0f, interfaz.sumar(2.0f, 3.0f));
}

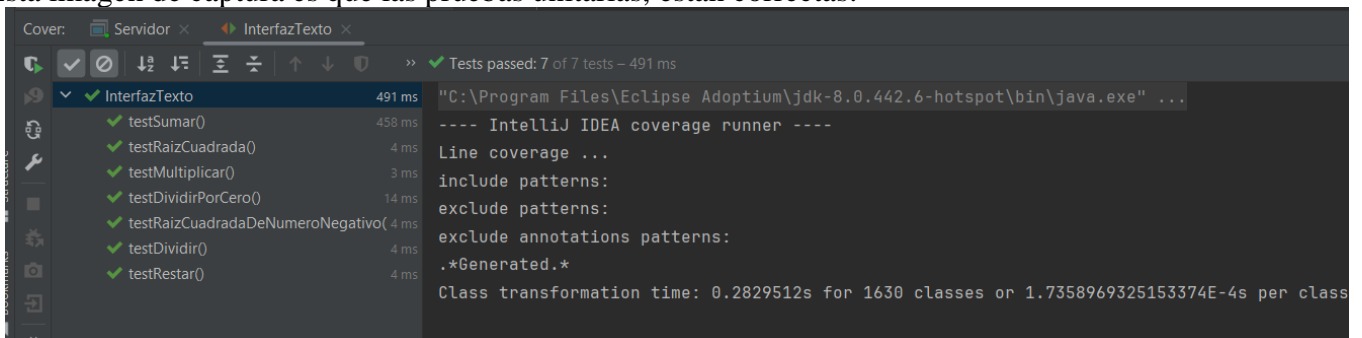
@Test
void testRestar() throws RemoteException {
    assertEquals( expected: 1.0f, interfaz.restar(3.0f, 2.0f));
}

@Test
void testMultiplicar() throws RemoteException {
    assertEquals( expected: 6.0f, interfaz.multiplicar(2.0f, 3.0f));
}

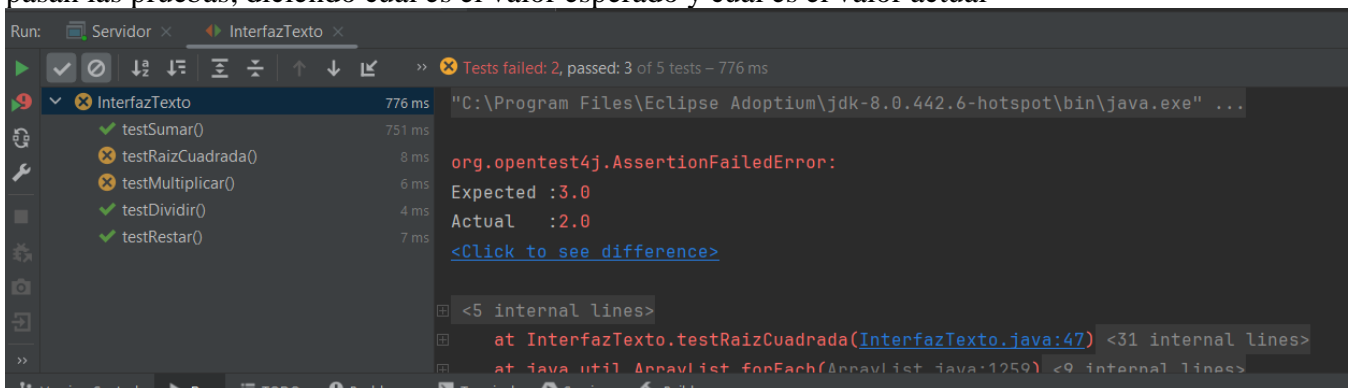
@Test
void testDividir() throws RemoteException {
    assertEquals( expected: 2.0f, interfaz.dividir(6.0f, 3.0f));
}

```

Esta imagen de captura es que las pruebas unitarias, estan correctas.




A continuacion, una imagen de que las pruebas realizadas son algunas que han dado error y por tanto no pasan las pruebas, diciendo cual es el valor esperado y cual es el valor actual





#### 4. ¿Qué porcentaje de cobertura de código tienen tus tests? (1 punto)

Al final se requieren los porcentajes de código utilizando la cobertura de código, en este caso el porcentaje de InterfazTexto al realizar las pruebas unitarias tiene un 100% en este ejemplo.



Element	Class, %	Method, %	Line, %
all	50% (2/4)	56% (9/16)	18% (13/69)
Cliente	0% (0/1)	0% (0/1)	0% (0/43)
Interfaz	100% (0/0)	100% (0/0)	100% (0/0)
InterfazTexto	100% (1/1)	100% (9/9)	100% (12/12)
Servidor	50% (1/2)	0% (0/6)	7% (1/14)