



Facultad de  
Cs. de la Computación



# Práctica CORBA

---

**Benemérita Universidad Autónoma de Puebla**

**Alumnos:**

**Luis Angel Avendaño Avalos – 201933893**

**Christian Amauri Amador Ortega – 201927821**

**Programación Distribuida**

**Otoño 2023**

## Introducción

En esta práctica, exploraremos el desarrollo de una aplicación distribuida utilizando Java Interface Definition Language (IDL), una parte integral de CORBA. El objetivo principal será implementar una clase llamada IOArchivo, la cual brindará funcionalidades avanzadas para la manipulación de archivos de texto de manera remota.

La aplicación distribuida contará con un servidor que ofrece los servicios de la clase IOArchivo, y los clientes podrán invocar estos servicios a través de las capacidades de CORBA. Se abordarán aspectos como la creación del servidor, la definición de interfaces mediante IDL, la implementación de los métodos remotos, y la resolución de nombres utilizando el servicio de nombres CORBA.

Las clases proporcionadas son esenciales para el funcionamiento de este sistema CORBA con IDL:

**HelloIDL:** Este contiene un módulo que genera la carpeta de interfaces de HelloApp para el IDL. Que a su vez también contiene la interfaz que define a la lista de funciones deseada.

```
module HelloApp {  
    interface IOArchivo {  
        long cuentaLineas();  
        long cuentaVocales();  
        void escribe(in OutputStream os);  
        void imprimir();  
        void respaldar(in string nombreArchivo);  
        void copiar(in string nombreArchivoDestino);  
        void renombrar(in string nombreArchivo);  
        void eliminar(in string nombreArchivo);  
    };  
};
```

La interface IDL define el contrato entre las partes cliente y servidor de nuestra aplicación, especificando qué operaciones y atributos están disponibles. El IDL es independiente del lenguaje de programación. Debemos mapearlo a Java antes de escribir cualquier código de implementación. (Ejecutando `idltojava` sobre el fichero IDL hará esto por nosotros automáticamente.)

El compilador idltojava genera un número de ficheros, basándose en las opciones elegidas en la línea de comandos. Como estos proporcionan funcionalidades estándar, podemos ignorarlos hasta el momento de desarrollar y ejecutar nuestro programa.

Los cinco ficheros generados por idltojava son:

HelloImplBase.java Esta clase abstracta es el esqueleto del servidor, proporcionando funcionalidades básicas de CORBA para el servidor. Implementa la interface Hello.java. La clase servidor HelloServant desciende de \_HelloImplBase.

HelloStub.java Esta clase es el cliente, proporciona funcionalidad CORBA al cliente. Implementa la interface Hello.java.

Hello.java Esta interface contiene la versión Java de nuestra interface IDL. El interface Hello.java desciende de org.omg.CORBA.Object, proporcionando también funcionalidades estándares de CORBA.

HelloHelper.java Esta clase final proporciona funcionalidades auxiliares, notablemente el método narrow requerido para convertir las referencias de los objetos CORBA a sus propios tipos.

HelloHolder.java Esta clase final contiene un ejemplar público del tipo Hello. Proporciona operaciones para argumentos de entrada y salida, que CORBA tiene pero que no se mapean fácilmente a la semántica Java.

**HelloServer:** Esta clase tiene contenida la lógica del servidor y que utilizará la lógica IDL de manera que utilizará el modulo anteriormente mapeado con todos sus archivos.

Comenzamos con importar todo paquete importante como son:

```
// HelloServer.java
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
```

Como todos los programas CORBA pueden lanzar excepciones del sistema en el momento de la ejecución, situaremos toda la funcionalidad del método main dentro de un bloque try-catch. El manejador de excepciones sólo implementa la excepción y su pila en la salida estándar para que podamos ver qué tipo de problema ocurrido. En esta clase encontraremos también la lógica de la lista de métodos como son:

- int cuentaLineas (String nombrearchivo): que devuelva el número de líneas del archivo;
- int cuentavocales (String nombrearchivo): que devuelva el número de líneas del archivo;
- void escribe (OutputStream os): que escriba el contenido del archivo a os;
- void imprimir () que imprima el contenido del archivo a pantalla;
- void respaldar(String nombrearchivo) que respalde el archivo;
- void copiar (String nombrearchivodestino): que copie el contenido de un archivo fuente a un archivo destino nombrearchivodestino;
- void renombrar(String nombrearchivo) que renombre el archivo
- void eliminar(String nombrearchivo) que elimine el archivo de almacenamiento masivo.

## ***Sustento Teórico***

La práctica de utilizar un cliente-servidor implementado con CORBA (Common Object Request Broker Architecture) y el compilador IDLJ (Java Interface Definition Language) se sustenta en los principios fundamentales de la programación distribuida y la interoperabilidad de objetos distribuidos. A continuación, se proporciona un sustento teórico para esta práctica:

### **Interoperabilidad de Objetos Distribuidos:**

#### **Objetivo de CORBA:**

CORBA es una especificación estándar diseñada para facilitar la comunicación entre objetos distribuidos a través de las barreras de lenguajes de programación y plataformas. Permite a objetos escritos en diferentes lenguajes de programación comunicarse de manera transparente a través de la red.

#### **IDL como Lenguaje de Definición de Interfaces:**

IDLJ es la implementación de Java IDL, que utiliza el lenguaje de definición de interfaces (IDL) para describir las interfaces de objetos distribuidos. IDL actúa como un contrato que define las operaciones que los objetos pueden realizar y cómo se comunican entre sí.

### **Modelo Cliente-Servidor:**

#### **División de Responsabilidades:**

La arquitectura cliente-servidor divide las responsabilidades entre el cliente que solicita un servicio y el servidor que proporciona dicho servicio. CORBA facilita la implementación de esta arquitectura permitiendo que objetos en diferentes direcciones de memoria interactúen como si fueran locales.

#### **Independencia de Plataforma:**

Con CORBA, el cliente y el servidor pueden ejecutarse en plataformas diferentes, lo que proporciona flexibilidad en la implementación distribuida. Esto es esencial en entornos donde la escalabilidad y la disponibilidad son requisitos críticos.

## **Compilador IDLJ y Generación de Código:**

### **IDL como Contrato de Interfaz:**

El compilador IDLJ traduce las especificaciones en IDL a código fuente Java, generando las clases y stubs necesarios para la comunicación entre el cliente y el servidor. Esto asegura que ambas partes se adhieran al contrato definido por las interfaces.

### **Transparencia de Implementación:**

La generación automática de código por el compilador IDLJ promueve la transparencia de implementación. Los desarrolladores pueden centrarse en la lógica de negocio sin preocuparse por los detalles de la comunicación entre objetos distribuidos.

## **Gestión de Objetos y Servicios:**

### **Servicios de Nombres y Resolución de Objetos:**

CORBA proporciona servicios como el servicio de nombres, que permite a los objetos distribuidos registrarse y ser localizados de manera dinámica en tiempo de ejecución. Esto facilita la resolución de nombres y la gestión de objetos en un entorno distribuido.

### **Gestión de Invocaciones Remotas:**

CORBA maneja las invocaciones remotas de manera eficiente, permitiendo que el cliente invoque métodos en objetos remotos como si fueran locales. Esto simplifica la programación y oculta los detalles de la comunicación distribuida.

## Objetivo

El objetivo principal de esta práctica es proporcionar a los participantes una experiencia práctica en el desarrollo de una aplicación distribuida cliente-servidor utilizando CORBA (Common Object Request Broker Architecture) y el compilador IDLJ (Java Interface Definition Language). La práctica tiene como metas:

### 1. Comprender los Principios de CORBA:

- Familiarizarse con los principios fundamentales de CORBA, incluida la interoperabilidad de objetos distribuidos, la transparencia de ubicación y la comunicación entre objetos distribuidos a través del Object Request Broker (ORB).

### 2. Utilizar IDLJ para Definir Interfaces:

- Aprender a utilizar IDLJ para definir interfaces de objetos distribuidos. Esto incluye la descripción de las operaciones que los objetos pueden realizar y cómo se comunican entre sí.

### 3. Implementar una Arquitectura Cliente-Servidor:

- Desarrollar una arquitectura cliente-servidor donde el servidor proporciona servicios y el cliente los consume. Esta práctica permitirá entender la división de responsabilidades entre el cliente y el servidor en un entorno distribuido.

### 4. Generar Código con el Compilador IDLJ:

- Utilizar el compilador IDLJ para traducir las especificaciones en IDL a código fuente Java. Entender cómo la generación automática de código facilita la implementación de objetos distribuidos y la comunicación remota.

### 5. Explorar la Resolución de Nombres y Gestión de Objetos:

- Examinar la resolución de nombres en CORBA mediante servicios como el servicio de nombres. Comprender cómo gestionar dinámicamente objetos distribuidos en tiempo de ejecución.

### 6. Evaluar la Utilidad de la Práctica:

- Evaluar la utilidad de la implementación cliente-servidor en CORBA, observando cómo esta arquitectura facilita la creación de sistemas distribuidos que pueden ejecutarse en diferentes plataformas y lenguajes de programación.

Al finalizar la práctica, los participantes deberían tener una comprensión más sólida de los conceptos fundamentales de CORBA, la utilización de IDLJ para definir interfaces y la implementación de aplicaciones distribuidas cliente-servidor que aprovechan la capacidad de CORBA para la interoperabilidad y la comunicación remota eficiente.

## Desarrollo

Para empezar, debemos diseñar la clase que va a trabajar las operaciones sobre los archivos. Son ocho operaciones:

- `int cuentaLineas ()`: que devuelva el número de líneas del archivo;
- `int cuentaVocales ()`: que devuelva el número de líneas del archivo;
- `void escribe ()`: que escriba el contenido del archivo a os;
- `void imprimir ()` que imprima el contenido del archivo a pantalla;
- `void respaldar()` que respalde el archivo;
- `void copiar ()`: que copie el contenido de un archivo fuente a un archivo destino `nombreArchivodestino`;
- `void renombrar()`: que renombre el archivo
- `void eliminar()`: que elimine el archivo de almacenamiento masivo.

Después de un tiempo de diseño, se logró implementar una clase que cuenta con cada una de las operaciones, a continuación presentamos el código fuente y describimos brevemente sus componentes:

Código fuente:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;
import java.util.Scanner;
import java.io.BufferedWriter;
import java.io.PrintWriter;

public class Operaciones {

    private String nombreArchivo;

    public Operaciones(String nombreArchivo) {
        this.nombreArchivo = nombreArchivo;
    }
}
```





```

public int cuentaLineas(String nombearchivo) {
    int contador = 0;
    try (BufferedReader br = new BufferedReader(new FileReader(nombearchivo))) {
        while (br.readLine() != null) {
            contador++;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return contador;
}

```

```

public int cuentavocales(String nombearchivo) {
    int contadorVocales = 0;
    try (BufferedReader br = new BufferedReader(new FileReader(nombearchivo))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            contadorVocales += contarVocales(linea);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return contadorVocales;
}

```

```

private int contarVocales(String linea) {
    int contador = 0;
    for (char c : linea.toCharArray()) {
        if ("AEIOUaeiou".indexOf(c) != -1) {
            contador++;
        }
    }
    return contador;
}

```

```

public void escribe(String contenido) {
    try (PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(nombreArchivo, true)))) {
        pw.println(contenido);
        System.out.println("Contenido escrito correctamente en el archivo.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

public void imprimir() {
    try (BufferedReader br = new BufferedReader(new FileReader(nombreArchivo))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

public void respaldar(String nombearchivo) {
    try {
        Path origen = Path.of(nombreArchivo);
        Path destino = Path.of(nombearchivo);

        Files.copy(origen, destino, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("Archivo respaldado correctamente.");
    } catch (IOException e) {

```

```
        e.printStackTrace();  
    }  
}
```

```

public void copiar(String nombreachivodestino) {
    try {
        Path origen = Path.of(nombreArchivo);
        Path destino = Path.of(nombreachivodestino);

        Files.copy(origen, destino, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("Archivo copiado correctamente.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void renombrar(String nombreachivo) {
    try {
        Path archivoActual = Path.of(nombreArchivo);
        Path nuevoArchivo = Path.of(nombreachivo);

        Files.move(archivoActual, nuevoArchivo, StandardCopyOption.REPLACE_EXISTING);
        nombreArchivo = nombreachivo; // Actualiza el nombre de archivo en la instancia
        System.out.println("Archivo renombrado correctamente.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void eliminar(String nombreachivo) {
    try {
        Path archivo = Path.of(nombreachivo);
        Files.deleteIfExists(archivo);
        System.out.println("Archivo eliminado correctamente.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Uso: java Operaciones <nombreArchivo> <metodo>");
            return;
        }

        String nombreArchivo = args[0];
        String metodo = args[1];

        Scanner scanner = new Scanner(System.in);

        // Crea una instancia de Operaciones
        Operaciones operaciones = new Operaciones(nombreArchivo);

        // Llama al método seleccionado
        switch (metodo) {
            case "cuentalneas":
                int lineas = operaciones.cuentaLineas(nombreArchivo);
                System.out.println("Número de líneas: " + lineas);
                break;

            case "cuentavocales":
                int vocales = operaciones.cuentavocales(nombreArchivo);
                System.out.println("Número de vocales: " + vocales);
                break;

            case "escribe":
                System.out.println("Ingrese el contenido a escribir en el archivo: ");

```

```
String contenidoAEscribir = scanner.nextLine();
operaciones.escribe(contenidoAEscribir);
break;
```

```

        case "imprimir":
            operaciones.imprimir();
            break;

        case "respaldar":
            System.out.println("Ingrese el nombre del archivo de respaldo: ");
            String nombreArchivoRespaldo = scanner.next(); // Puedes utilizar un Scanner para
leer la entrada del usuario

            operaciones.respaldar(nombreArchivoRespaldo);
            break;

        case "copiar":
            System.out.println("Ingrese el nombre del archivo destino: ");
            String nombreArchivoDestino = scanner.next();
            operaciones.copiar(nombreArchivoDestino);
            break;

        case "renombrar":
            System.out.println("Ingrese el nuevo nombre del archivo: ");
            String nuevoNombreArchivo = scanner.next();
            operaciones.renombrar(nuevoNombreArchivo);
            break;

        case "eliminar":
            operaciones.eliminar(nombreArchivo);
            break;

        default:
            System.out.println("args[0]: " + args[0]);
            System.out.println("args[1]: " + args[1]);
            System.out.println("Método no reconocido. Métodos disponibles: cuentaLineas,
cuentavocales, escribe, imprimir, respaldar, copiar, renombrar, eliminar");
    }
}
}

```

Para empezar, hablemos de las librerías que usa:

#### 1. **java.io.BufferedReader y java.io.FileReader:**

- Estas clases se utilizan para leer el contenido de un archivo de texto. **BufferedReader** se utiliza para leer eficientemente grandes cantidades de datos al almacenarlos en búfer.

#### 2. **java.io.FileWriter:**

- Se utiliza para escribir datos en un archivo de texto. **FileWriter** permite escribir caracteres en un archivo.

#### 3. **java.nio.file.Path y java.nio.file.Files:**

- Estas clases son parte del paquete **java.nio.file** que proporciona una interfaz más moderna y completa para la manipulación de archivos y directorios en comparación con las clases tradicionales de **java.io**.
- **Path** representa la ubicación de un archivo o directorio en el sistema de archivos.
- **Files** proporciona métodos estáticos para realizar operaciones de archivos, como copiar, mover y eliminar archivos.

#### 4. **java.nio.file.StandardCopyOption:**

- Enumeración que proporciona opciones estándar para el método **Files.copy()**, en este caso, se utiliza **REPLACE\_EXISTING** para reemplazar el archivo de destino si ya existe.

5. **java.util.Scanner:**

- Se utiliza para leer la entrada del usuario desde la consola. En este caso, se utiliza para obtener la entrada del usuario para ciertas operaciones.

6. **java.io.BufferedWriter y java.io.PrintWriter:**

- **BufferedWriter** se utiliza para escribir eficientemente grandes cantidades de datos al almacenarlos en búfer.
- **PrintWriter** proporciona métodos para imprimir representaciones de varios tipos de datos a un flujo de texto, en este caso, se utiliza para escribir en el archivo.

Estas bibliotecas forman parte del núcleo de Java y son fundamentales para realizar operaciones de lectura y escritura de archivos, manipulación de rutas y entrada/salida estándar. Al utilizar estas bibliotecas, el código logra operar sobre archivos de manera eficiente y manejar diversas operaciones de manera robusta.

Ahora hablemos del código:

1. **Operaciones Class:**

- La clase **Operaciones** proporciona varias operaciones sobre archivos.
- Tiene un campo **nombreArchivo** para almacenar el nombre del archivo con el que se está trabajando.

2. **Constructor:**

- El constructor **Operaciones(String nombreArchivo)** inicializa la instancia de **Operaciones** con el nombre del archivo proporcionado.

3. **Método Main:**

- El método **main** es la entrada principal del programa.
- Lee los argumentos de la línea de comandos para obtener el nombre del archivo y el método a ejecutar.
- Utiliza un **Scanner** para interactuar con el usuario en algunos métodos.
- Realiza operaciones basadas en el método proporcionado a través de la línea de comandos.

#### 4. Uso desde la Línea de Comandos:

- Cuando ejecutas el programa desde la línea de comandos, puedes proporcionar el nombre del archivo y el método a ejecutar. Por ejemplo:

```
> java Operaciones miArchivo.txt cuentalineas
```

#### 5. Switch Statement:

- Utiliza un **switch** para determinar qué método invocar en función del segundo argumento de la línea de comandos.

#### 6. Manejo de Excepciones:

- Maneja excepciones de lectura/escritura de archivos utilizando bloques **try-catch** para imprimir detalles en caso de error.

Ahora hablemos de las modificaciones hechas a los ejemplos de IDL de CORBA. Para empezar, al adaptar el servidor para que pueda ejecutar operaciones sobre los archivos, tenemos el siguiente resultado:

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

public class HelloServer {
    public static void main(String args[]) {
        try {
            // Inicializa el ORB (Object Request Broker)
            ORB orb = ORB.init(args, null);

            // Obtiene el POA (Portable Object Adapter) raíz y activa el POAManager
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();

            // Crea una instancia del implementador de la interfaz Hello
            HelloImpl helloImpl = new HelloImpl();
            helloImpl.setORB(orb);

            // Crea una instancia de Operaciones y asigna al servidor
            Operaciones operaciones = new Operaciones("nombre_archivo.txt");
            helloImpl.setOperaciones(operaciones);

            // Obtiene la referencia del objeto Hello a partir del implementador
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
            Hello href = HelloHelper.narrow(ref);

            // Obtiene el contexto de nombres raíz
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```



```

// Vincula la referencia del objeto Hello en el contexto de nombres
String name = "Hello";
NameComponent path[] = ncRef.to_name(name);
ncRef.rebind(path, href);

System.out.println("HelloServer listo y esperando...");

// Espera invocaciones desde clientes
orb.run();
} catch (Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
}

System.out.println("HelloServer saliendo...");
}
}

```

Al integrar las funcionalidades de la clase **Operaciones** en el servidor Corba, hubo varios pasos clave y consideraciones, a continuación hablaremos de ellas:

1. **Identificación del Requisito de Integración:** Al revisar la clase **Operaciones**, notamos que proporciona funciones útiles relacionadas con la manipulación de archivos. Decidimos integrar estas funcionalidades en el servidor para permitir que los clientes realicen operaciones de archivos en el servidor.
2. **Extensión de la Interfaz IDL:** Dado que la interfaz original solo tenía métodos **sayHello()** y **shutdown()**, tuvimos que decidir cómo agregar la nueva funcionalidad. Consideramos agregar un nuevo método **getFileOperationsInfo()** a la interfaz IDL para proporcionar información específica de operaciones de archivos al cliente cuando sea necesario.
3. **Modificación del Servidor:** Para integrar las operaciones de archivos, creamos una instancia de la clase **Operaciones** en el servidor. Modificamos el método **sayHello()** para devolver información relacionada con archivos utilizando la instancia de **Operaciones**. También, aprovechamos el método **shutdown()** para realizar una operación adicional, como respaldar el archivo antes de apagar el servidor.
4. **Gestión de Excepciones:** Consideramos situaciones en las que las operaciones de archivos podrían fallar (por ejemplo, si el archivo no existe). Se añadieron manejo de excepciones para proporcionar información adecuada al usuario o cliente sobre cualquier problema que pueda ocurrir durante las operaciones de archivos.

Siguiendo la misma metodología para el lado del cliente, el resultado del cliente es el siguiente:

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloClient {
    static Hello helloImpl;

    public static void main(String args[]) {
        try {
            // Inicializa el ORB (Object Request Broker)
            ORB orb = ORB.init(args, null);

            // Obtiene el contexto de nombres raíz
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

            // Resuelve la referencia del objeto Hello en el contexto de nombres
            String name = "Hello";
            helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

            System.out.println("Obtenida una referencia al objeto del servidor: " + helloImpl);

            // Llama al método sayHello en el servidor e imprime el resultado
            System.out.println(helloImpl.sayHello());

            // Llama al método shutdown en el servidor
            helloImpl.shutdown();
        } catch (Exception e) {
            System.out.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

1. **Requisitos del Cliente:** Al observar las nuevas funcionalidades introducidas en el servidor, identificamos la necesidad de proporcionar al cliente acceso a la información específica de las operaciones de archivos realizadas en el servidor.
2. **Modificación del Cliente:** En este caso, adaptamos el cliente para aprovechar la nueva funcionalidad proporcionada por el servidor. Esto implica invocar el método **sayHello()** para obtener información relacionada con operaciones de archivos desde el servidor y utilizar cualquier otro método adicional proporcionado por la interfaz IDL.
3. **Manejo de la Nueva Información:** Decidimos imprimir la información obtenida del servidor en la consola del cliente. Esto podría incluir detalles sobre el contenido de archivos, información sobre la ejecución en tiempo real o cualquier otra información relevante.
4. **Manejo de Excepciones:** Consideramos situaciones en las que las operaciones en el servidor podrían fallar y manejamos las excepciones correspondientes en el cliente para proporcionar información significativa al usuario.

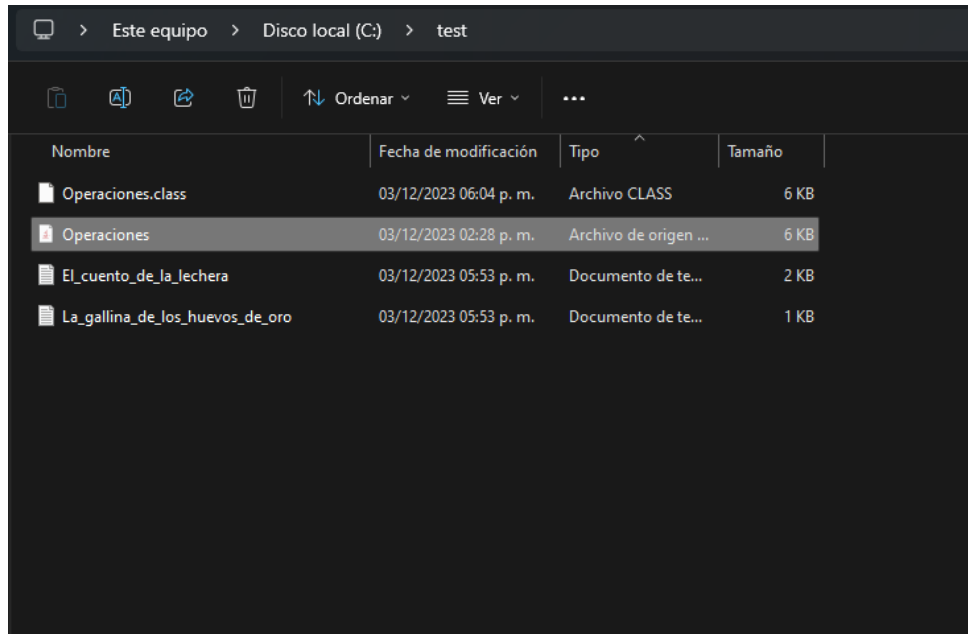
La clase **Operaciones** tiene un método **main** que acepta argumentos desde la línea de comandos. Cuando ejecutas la clase **Operaciones** directamente desde la línea de comandos, los argumentos que proporcionas se capturan en el array **String[] args**. Sin embargo, cuando integras la funcionalidad de **Operaciones** en el servidor Corba, la ejecución cambia un poco.

En este contexto, la clase **Operaciones** no se ejecuta directamente desde la línea de comandos en el servidor. En lugar de eso, el servidor Corba crea una instancia de la clase **Operaciones** y llama a sus métodos según sea necesario. La clase **Operaciones** actúa como un componente del servidor y proporciona funcionalidad de operaciones de archivos que puede ser invocada por el cliente Corba.

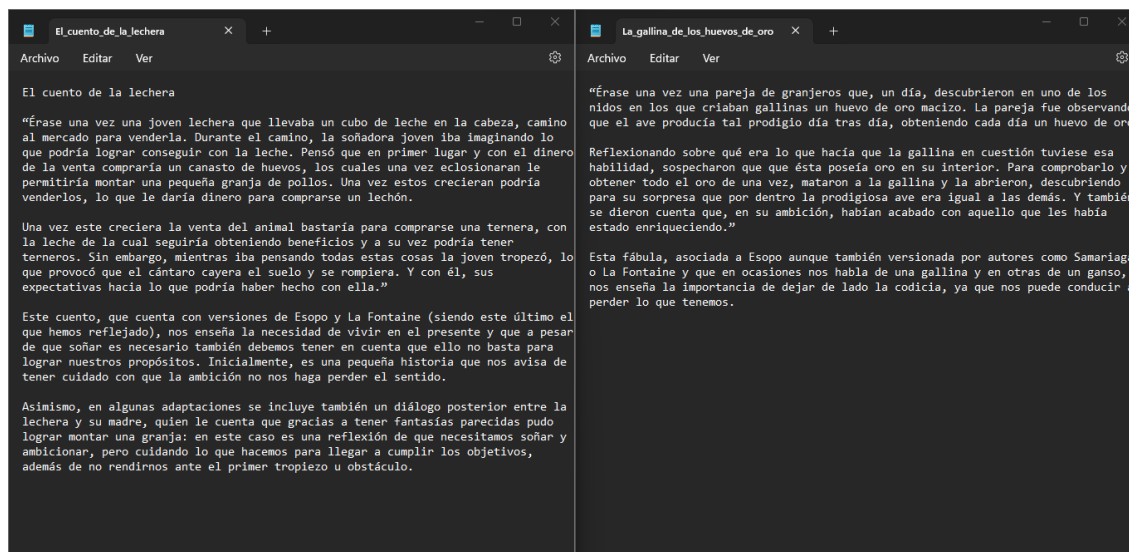
## Resultados

Al ejecutar cada uno de los métodos de la clase Operaciones, se obtiene de manera satisfactoria el resultado esperado, a continuación se muestran capturas del uso de dicha clase y los cambios que realiza sobre los archivos que se tengan.

Creamos una carpeta para que nos sirva como nuestra área de pruebas:



El contenido inicial de estos archivos es el siguiente:

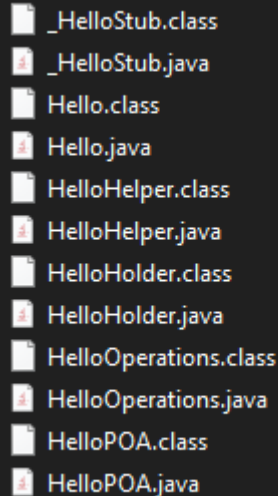


Entonces, ejecutamos cada comando:

Primero mapeamos el archivo IDL creado utilizando idlj para generar la carpeta HelloApp necesaria para ello utilizamos el siguiente comando:

```
C:\Users\Angel\Desktop\PD\Hello>idlj -fall Hello.idl
```

donde Hello.idl es nuestro archivo idl. Esto nos generará la carpeta HelloApp con los siguientes archivos:



- \_HelloStub.class
- \_HelloStub.java
- Hello.class
- Hello.java
- HelloHelper.class
- HelloHelper.java
- HelloHolder.class
- HelloHolder.java
- HelloOperations.class
- HelloOperations.java
- HelloPOA.class
- HelloPOA.java

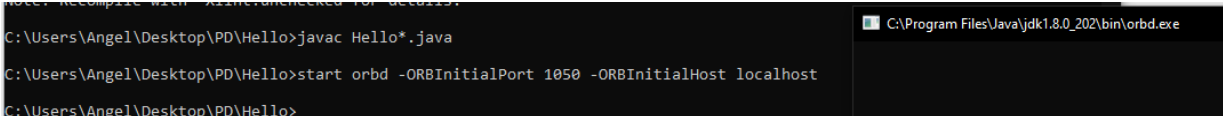
Lo siguiente es compilar todos los archivos en el proyecto, debido a la incompatibilidad con versiones posteriores es utilizado el jdk de java 1.8, se compilan los incluidos en la carpeta HelloApp de la siguiente manera:

```
C:\Users\Angel\Desktop\PD\Hello>javac HelloApp\*.java
Note: HelloApp\HelloPOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Compilamos los archivos de Servidor y cliente:

```
C:\Users\Angel\Desktop\PD\Hello>javac Hello*.java
```

Debemos levantar el ORB de java, orbd se utiliza para permitir a los clientes localizar e invocar objetos persistentes en servidores en el entorno CORBA.



```
C:\Users\Angel\Desktop\PD\Hello>javac Hello*.java
C:\Users\Angel\Desktop\PD\Hello>start orbd -ORBInitialPort 1050 -ORBInitialHost localhost
C:\Users\Angel\Desktop\PD\Hello>
```

Con todo esto listo vamos ahora ejecutamos en primer lugar el servidor de la siguiente manera:

```
C:\Users\Angel\Desktop\PD\Hello>start java HelloServer -ORBInitialPort 1050 -ORBInitialHost localhost
```

(Hubo un problema en el que todo el ambiente de corba estaba correctamente establecido pero debido a errores en la paquetería de corba saltaban errores y el servidor no quedaba a la espera de las conexiones, simplemente abría y se cerraba rápidamente.)

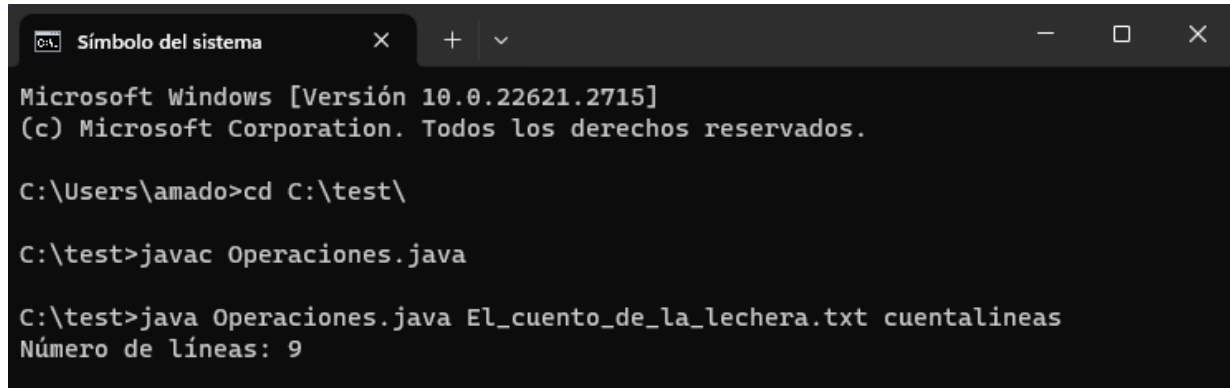
Ya levantado nos dejará en espera de un cliente por lo que ahora ejecutaremos el cliente:

```
C:\Users\Angel\Desktop\PD\Hello>java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

Y con el cliente ya ejecutado podremos utilizar cualquiera de las siguientes funciones las cuales se muestran sus resultados a continuación.

Para contar líneas:

```
>java Operaciones.java El_cuento_de_la_lechera.txt cuentalineas
```



```
Microsoft Windows [Versión 10.0.22621.2715]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\amado>cd C:\test\

C:\test>javac Operaciones.java

C:\test>java Operaciones.java El_cuento_de_la_lechera.txt cuentalineas
Número de líneas: 9
```

Para contar vocales:

```
>java Operaciones.java El_cuento_de_la_lechera.txt cuentavocales
```

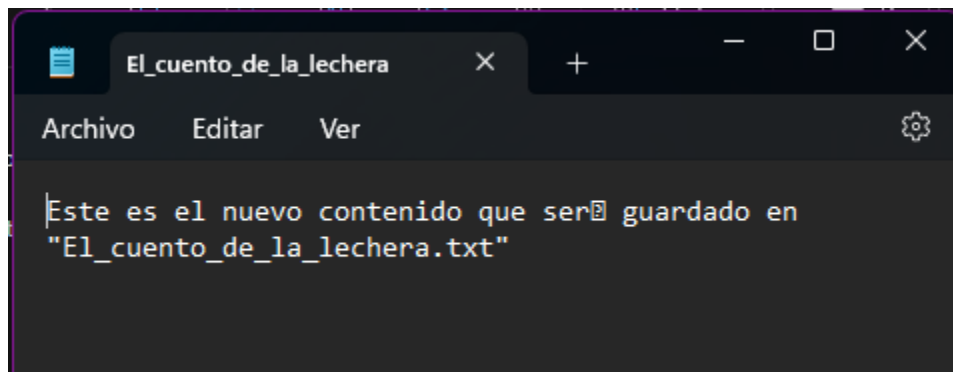
```
C:\test>java Operaciones.java El_cuento_de_la_lechera.txt cuentavocales
Número de vocales: 589
```

Para escribir:

```
>java Operaciones.java El_cuento_de_la_lechera.txt escribe
```

```
C:\test>java Operaciones.java El_cuento_de_la_lechera.txt escribe
Ingrese el contenido a escribir en el archivo:
Este es el nuevo contenido que será guardado en "El_cuento_de_la_lechera.txt"
Contenido escrito correctamente en el archivo.
```

Si vamos a comprobar los efectos de este comando en nuestro archivo, veremos reflejados los cambios:



Para imprimir el contenido de un archivo en terminal:

```
>java Operaciones.java El_cuento_de_la_lechera.txt imprimir
```

```
>java Operaciones.java La_gallina_de_los_huevos_de_oro.txt imprimir
```

```
C:\test>java Operaciones.java El_cuento_de_la_lechera imprimir
Este es el nuevo contenido que será guardado en "El_cuento_de_la_lechera.txt"

C:\test>java Operaciones.java La_gallina_de_los_huevos_de_oro.txt imprimir
?Érase una vez una pareja de granjeros que, un día, descubrieron en uno de los
nidos en los que criaban gallinas un huevo de oro macizo. La pareja fue observa
ndo que el ave producía tal prodigio día tras día, obteniendo cada día un huevo
de oro.

Reflexionando sobre qué era lo que hacía que la gallina en cuestión tuviese esa
habilidad, sospecharon que que ésta poseía oro en su interior. Para comprobarl
o y obtener todo el oro de una vez, mataron a la gallina y la abrieron, descubri
endo para su sorpresa que por dentro la prodigiosa ave era igual a las demás.
Y también se dieron cuenta que, en su ambición, habían acabado con aquello que
les había estado enriqueciendo.?

Esta fábula, asociada a Esopo aunque también versionada por autores como Samari
aga o La Fontaine y que en ocasiones nos habla de una gallina y en otras de un
ganso, nos enseña la importancia de dejar de lado la codicia, ya que nos puede
conducir a perder lo que tenemos.

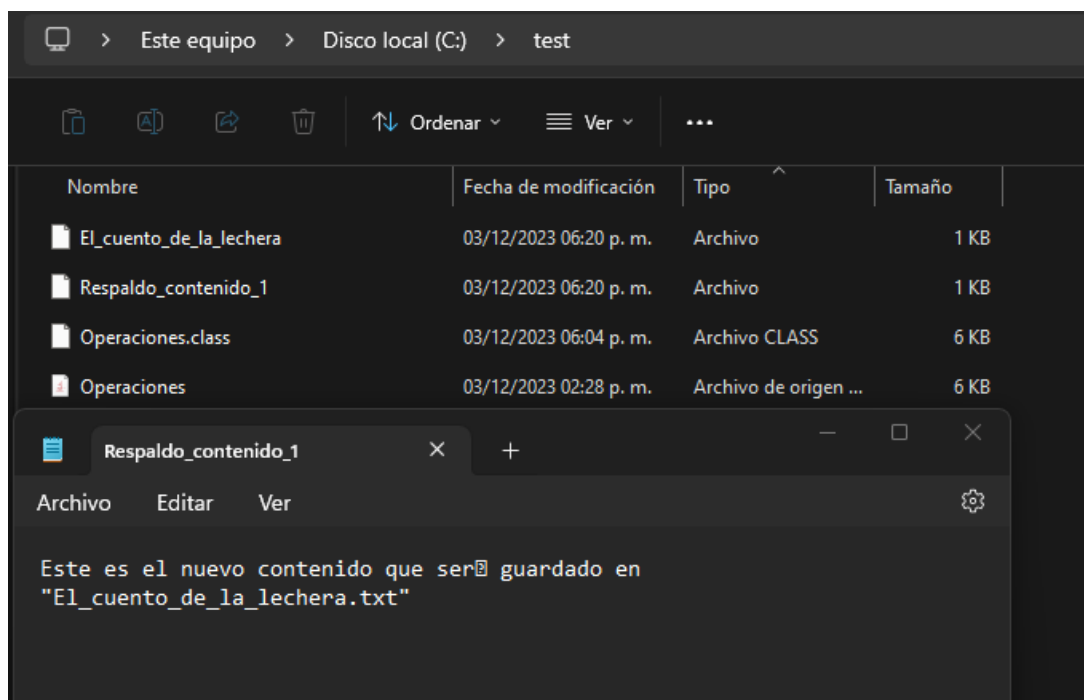
C:\test>
```

Para respaldar un archivo:

```
>java Operaciones.java El_cuento_de_la_lechera respaldar
```

```
C:\test>java Operaciones.java El_cuento_de_la_lechera respaldar
Ingrese el nombre del archivo de respaldo:
Respaldo_contenido_1
Archivo respaldado correctamente.
```

Si vamos a comprobar los cambios realizados por este comando en nuestro directorio, veremos:



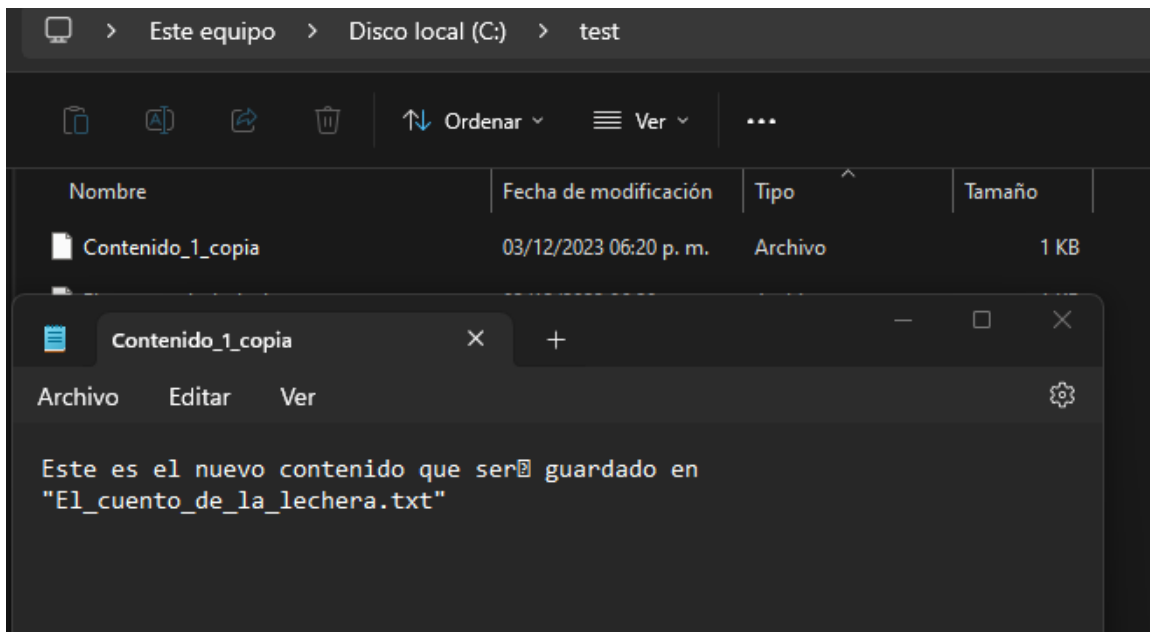
Para copiar el contenido de un archivo en uno nuevo escribimos:

```
>java Operaciones.java El_cuento_de_la_lechera copiar
```

```
C:\test>java Operaciones.java El_cuento_de_la_lechera copiar
Ingrese el nombre del archivo destino:
Contenido_1_copia
Archivo copiado correctamente.
```

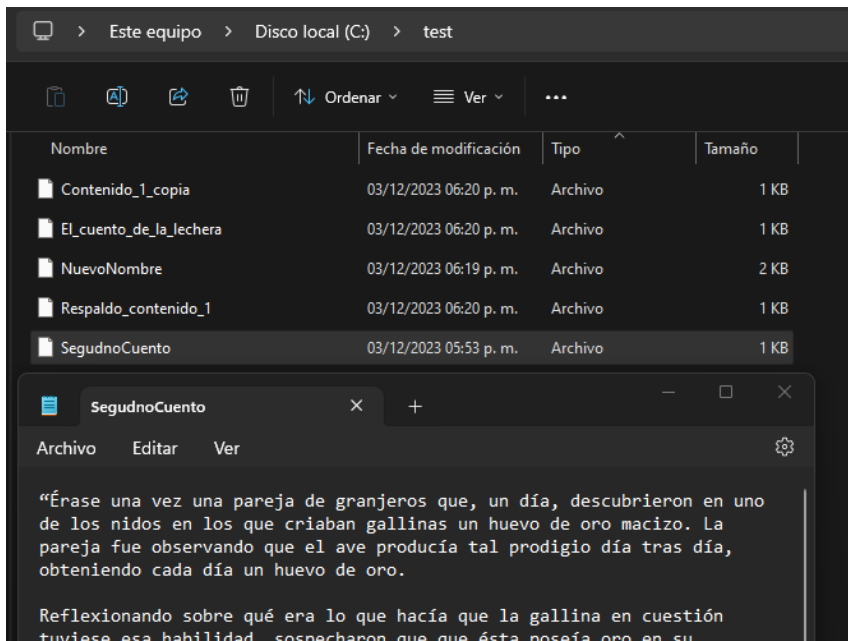


Luego comprobamos el contenido...



Para renombrar un archivo escribimos:

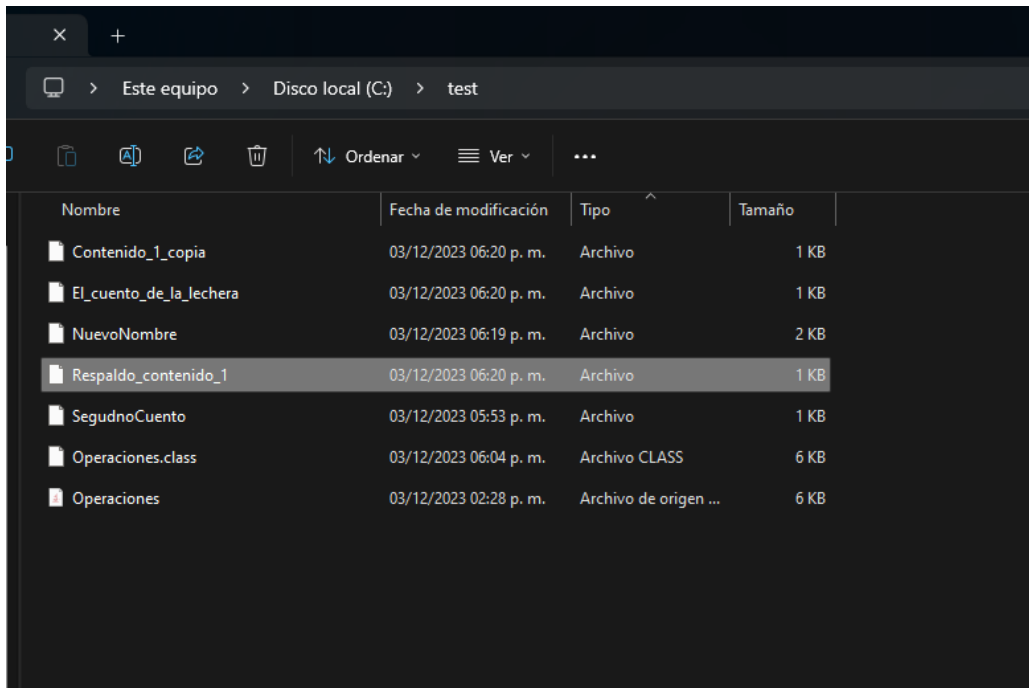
```
>java Operaciones.java El_cuento_de_la_lechera.txt renombrar
```



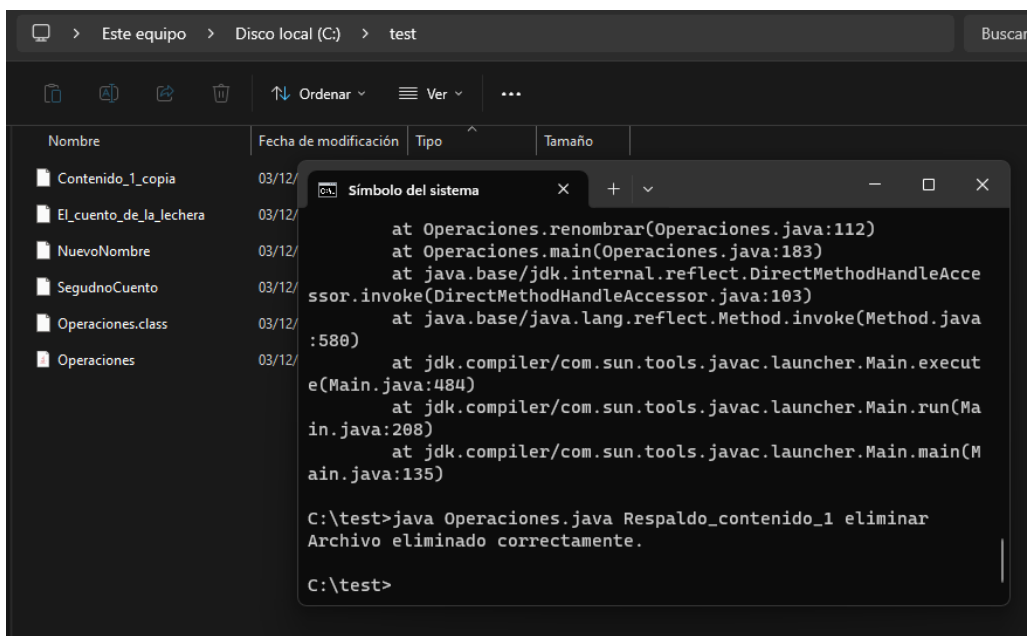
Finalmente para eliminar un archivo escribimos:

```
>java Operaciones.java Respaldo_contenido_1 eliminar
```

Antes comprobamos la presencia del archivo a eliminar:



Y luego notamos que ha sido borrado:



## ***Conclusión***

La práctica de desarrollar una aplicación cliente-servidor utilizando CORBA e IDLJ ha proporcionado una valiosa experiencia en el diseño y la implementación de sistemas distribuidos. A través de la comprensión de los principios de CORBA, la definición de interfaces con IDLJ y la generación de código automática, hemos explorado los beneficios de una arquitectura cliente-servidor que trasciende las barreras de lenguajes y plataformas.

La utilización del compilador IDLJ ha demostrado ser una herramienta poderosa al simplificar la creación de interfaces y la comunicación remota. La transparencia de ubicación y la capacidad de gestionar objetos dinámicamente en tiempo de ejecución han destacado la flexibilidad y la escalabilidad inherentes a CORBA.

## **Referencias:**

- [1] Apuntes de CORBA: [Presentación de PowerPoint \(blackboardcdn.com\)](http://blackboardcdn.com)
- [2] Java IDL: [Getting Started with Java IDL \(oracle.com\)](http://oracle.com)
- [3] Historia de CORBA: [CORBA History](http://oracle.com)
- [4] Ejemplo con JAVA IDL de CORBA: [Distributed Java Programming with RMI and CORBA \(oracle.com\)](http://oracle.com)
- [5] Desarrollo de aplicaciones CORBA: [Developing CORBA Applications \(oracle.com\)](http://oracle.com)
- [6] Descargar idlj: [Descarga idlj.exe Java\(TM\) Platform SE 7 U67 \(256file.com\)](http://256file.com)