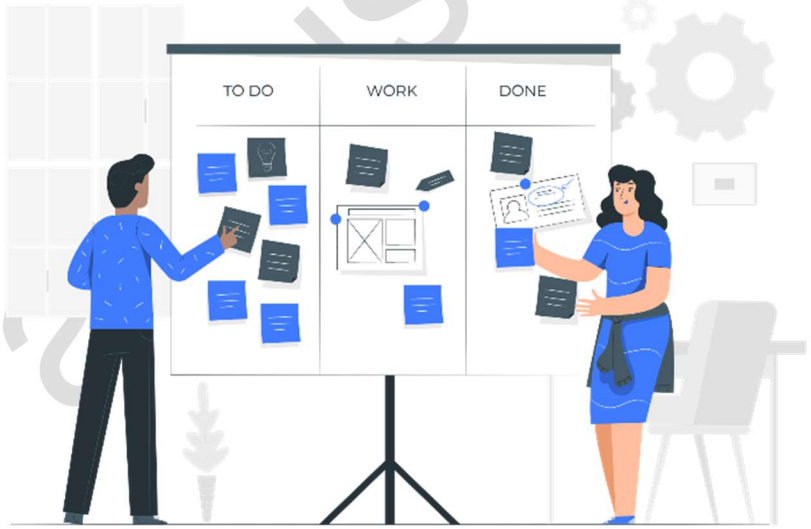
TEMA 11: Lectura Y Escritura De Información

ÍNDICE



1. INTRODUCCIÓN

De manera habitual, en la codificación de un programa existe la intención de que dicho programa pueda interactuar con los usuarios del mismo modo; es decir, que el usuario pueda pedirle que realice tareas y pueda suministrarle los datos requeridos para realizar una función. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos para proporcionar una respuesta a lo solicitado.

Además, en muchas ocasiones interesa que el programa guarde los datos que se le han introducido, de forma que si el programa se interrumpe, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más habitual de conseguirlo es mediante la utilización de ficheros que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como ***Entrada/Salida (E/S).***

Existen dos tipos de E/S; la E/S **estándar** que se realiza con el terminal del usuario y la E/S **a través de fichero**, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar de la API de Java denominado java.io que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

2. FLUJOS

Ha pasado ya muchas décadas desde nuestro primer mensaje “Hola Mundo”, y a su vez, hemos utilizado el intercambio de información de forma inconsciente, sin saber que por detrás hay una forma de transmisión llamada **“Flujos de Comunicación”.** Cuando hablamos de un flujo, o *steam,* mencionamos un conjunto de datos de una fuente o un destino.

Uno de los aspectos más importantes es el intercambio de información entre el programa y el usuario. En este tema, analizaremos la gran variedad de flujos existentes.

2.1. FLUJOS PREDETERMINADOS

En esta sección, únicamente se tratará la entrada/salida que se comunica con el usuario a través de la pantalla o de la ventana del terminal.

El acceso a la entrada y salida estándar es controlado por tres objetos que se crean automáticamente al iniciar la aplicación: **System.in**, **System.out** y **System.err.**

2.1.1. SYSTEM.OUT

Este objeto implementa la **salida estándar**. Los métodos que nos proporciona para controlar la salida son:

- **print(a):** Imprime a en la salida, donde a puede ser cualquier tipo básico Java ya que Java hace su conversión automática a cadena.
- **println(a):** Es idéntico a print(a) salvo que con println() se imprime un salto de línea al final de la impresión de *a*.

EJEMPLO: ENTRADA Y SALIDA DE println(a)

```
System.out.println("Hola, mundo!");
```

```
Hola, mundo!
```

2.1.2. SYSTEM.ERR

Utilizado para imprimir mensajes de error y excepciones en la consola o en el registro de errores. La salida de error estándar es un flujo de salida de bytes que está asociado con el proceso de la JVM en el que se está ejecutando la aplicación.

El flujo de salida de error estándar se utiliza para imprimir mensajes de error y excepciones que indican un problema grave que impide que la aplicación funcione correctamente. En general, se utiliza para imprimir mensajes que requieren la atención inmediata del usuario o del desarrollador, como errores de sistema, excepciones no controladas y problemas de seguridad.

Las funciones son similares a las proporcionadas por System.out.

EJEMPLO: ENTRADA Y SALIDA DE System.err

```
public class EjemploSystemErr {
    public static void main(String[] args) {
        int dividendo = 10;
        int divisor = 0;

        try {
            int resultado = dividendo / divisor;
            System.out.println("El resultado es: " + resultado);
        } catch (ArithmeticException e) {
            System.err.println("Error: no se puede dividir por cero.");
        }
    }
}
```

```
Error: no se puede dividir por cero.
```

En este ejemplo, se intenta dividir la variable dividendo por la variable divisor, que tiene el valor cero. Como resultado, se produce una excepción de tipo *ArithmeticException.* Para manejar esta excepción, se utiliza un bloque try-catch. Dentro del bloque try, se realiza la división y se imprime el resultado utilizando ***System.out.println().*** Dentro del bloque catch, se captura la excepción y se imprime un mensaje de error utilizando ***System.err.println().***

2.1.3. SYSTEM.IN

Este objeto implementa la entrada estándar (normalmente, a través del teclado). Los métodos que nos proporciona para controlar la entrada son:

- **read():** Devuelve el carácter que se ha introducido por el teclado leyéndolo del buffer de entrada y eliminándolo, para leer el siguiente carácter en la siguiente lectura. Si no se ha introducido ningún caracter por el teclado, devuelve el valor -1.
- **skip(n):** Permite saltar un número determinado de bytes dentro del flujo de entrada.

EJEMPLO DE System.in read()

```java
import java.io.*;

public class ReadExample {
   public static void main(String[] args) throws IOException {
      // crear un flujo de entrada
      InputStream input = System.in;

      // leer un byte de entrada
      int ch = input.read();

      // imprimir el byte leído
      System.out.println("Byte leído: " + ch);
   }
}
```

En este ejemplo, se crea un objeto de flujo de entrada *InputStream* utilizando *System.in.* Luego, se lee un byte de entrada utilizando el método *read().* Finalmente, se imprime el byte leído utilizando *System.out.println().* Es importante tener en cuenta que el método *read()* puede bloquear el hilo de ejecución hasta que se haya leído un byte de entrada, por lo que es posible que se desee utilizar un subproceso separado para la lectura de entrada si se desea mantener la capacidad de respuesta de la interfaz de usuario.

2.2. TIPOS DE FLUJO

La clasificación de los flujos es variada: según la **dirección** del flujo, según la **forma** de flujo, y según el **acceso** a ese flujo.

2.2.1. SEGÚN DIRECCIÓN

La dirección de los flujos puede ser:

a. Flujos de entrada **b.** Flujos de salida. **c.** Flujos de entrada/salida.

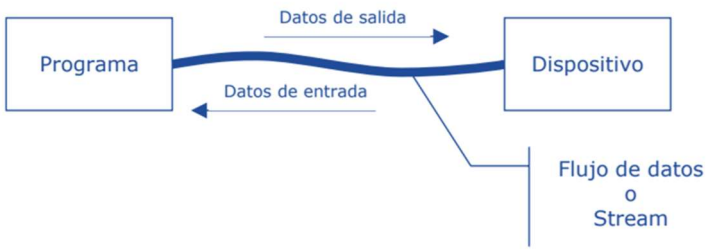


Diagrama obtenido de: *http://di002.edv.uniovi.es/~alberto_mfa/computadores2001/apuntes/Tema%2011.pdf*

2.2.2. SEGÚN FORMA

Existen dos formas de flujos definidos en Java: unos que trabajan con ***bytes*,** y otros que trabajan con **caracteres.** Así mismo, existen clases conversoras que permiten obtener un flujo de *bytes* a partir de uno de caracteres, y viceversa, tanto para lectura como para escritura.

2.2.3. SEGÚN EL TIPO DE ACCESO

2.2.3.1. ACCESO SECUENCIAL

FileReader (String path)	▪ Int read() ▪ Long skip(n) ▪ Void close()

EJEMPLO DE FileReader (String path)

```
Hola mundo!
Este es un archivo de ejemplo.
```

Supongamos que tenemos un archivo de texto llamado *datos.txt* en la siguiente ruta: */home/usuario/documentos/datos.txt.*

```
import java.io.FileReader;
import java.io.IOException;

public class EjemploFileReader {
    public static void main(String[] args) {
        String rutaArchivo = "/home/usuario/documentos/datos.txt";

        try (FileReader fr = new FileReader(rutaArchivo)) {
            char[] buffer = new char[1024];
            int charsRead = fr.read(buffer);
            while (charsRead != -1) {
                System.out.print(new String(buffer, 0, charsRead));
                charsRead = fr.read(buffer);
            }
        } catch (IOException e) {
            System.err.println("Error al leer el archivo: " + e.getMessage()
        }
    }
}
```

En este ejemplo, se crea un objeto *FileReader* utilizando la ruta del archivo como argumento del constructor. Luego, se utiliza un ciclo *while* para leer los caracteres del archivo en bloques de tamaño 1024. Dentro del ciclo, se convierten los caracteres leídos a una cadena de texto utilizando el constructor *String(char[] chars, int offset, int length)* y se imprime la cadena en la consola utilizando *System.out.print().*

Cuando se alcanza el final del archivo, el método ***read()*** devuelve -1 y el ciclo se detiene. Si se produce un error al leer el archivo, se captura la excepción y se imprime un mensaje de error utilizando ***System.err.println().***

FileReader (File file)	***Ejemplo creación:*** File file = new File("c:\temp\test.txt"); FileReader input = new FileReader(file);

FileInputStream(String path)	Fichero de entrada de texto. Representa ficheros de texto de sólo lectura a los que se accede de forma secuencial. ▪ Int read() ▪ Int available() ▪ Void close

EJEMPLO: FileInputStream(String path)

```java
import java.io.*;

public class EjemploFileInputStream {
    public static void main(String[] args) {
        try {
            // Abrir un archivo para lectura
            FileInputStream fis = new FileInputStream("ejemplo.txt");

            // Leer datos del archivo
            int byteLeido;
            while ((byteLeido = fis.read()) != -1) {
                // Procesar el byte leído
                System.out.print((char) byteLeido);
            }

            // Cerrar el archivo
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En el ejemplo anterior, creamos un objeto FileInputStream llamado fis y abrimos un archivo llamado "ejemplo.txt" para lectura. Luego, leemos datos del archivo en forma de bytes en un bucle while, y los procesamos como caracteres utilizando la función (char) byteLeido. Finalmente, cerramos el archivo utilizando el método close() para liberar los recursos.

DataInputStream	Para leer datos de entrada usando la clase *DataInputStream*, se puede crear una instancia de la clase y utilizar sus métodos para leer diferentes tipos de datos primitivos, como *int*, *double* y *boolean*.

EJEMPLO: DataInputStream

```java
import java.io.*;

public class EjemploDataInputStream {
    public static void main(String[] args) {
        try {
            // Crear un flujo de entrada de archivo
            FileInputStream fis = new FileInputStream("datos.bin");
            DataInputStream dis = new DataInputStream(fis);

            // Leer datos primitivos del flujo de entrada
            int entero = dis.readInt();
            float flotante = dis.readFloat();
            double doble = dis.readDouble();
            boolean booleano = dis.readBoolean();
            String cadena = dis.readUTF();

            // Imprimir los datos leídos
            System.out.println("Entero: " + entero);
            System.out.println("Flotante: " + flotante);
            System.out.println("Doble: " + doble);
            System.out.println("Booleano: " + booleano);
            System.out.println("Cadena: " + cadena);

            // Cerrar el flujo de entrada
            dis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, se crea un flujo de entrada de archivo utilizando la clase *FileInputStream* para leer datos binarios de un archivo llamado *"datos.bin".* Luego, se utiliza la clase DataInputStream para leer datos primitivos, como enteros, flotantes, dobles, booleanos y cadenas, del flujo de entrada. Los datos leídos se imprimen en la consola. Finalmente, se cierra el flujo de entrada utilizando el método *close*() de la clase DataInputStream.

FileInputStream

FileInputStream(File file)

BufferedReader newBufferedReader (Path path [, Charset cs])

BufferedReader newBufferedReader (Path path [, Charset cs][, OpenOption opcion1 …, OpenOption opcionN])

EJEMPLO

```java
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class BufferedReaderEjemplo {
    public static void main(String[] args) {
        String rutaArchivo = "ruta/del/archivo.txt"; // Ruta del archivo de

        try {
            Path path = Paths.get(rutaArchivo);

            // Crear el BufferedReader
            BufferedReader bufferedReader = Files.newBufferedReader(path);

            // Leer caracteres del archivo y mostrarlos en la consola
            int caracter; // Variable para almacenar cada caracter leído
            while ((caracter = bufferedReader.read()) != -1) {
                System.out.print((char) caracter);
            }

            bufferedReader.close(); // Cerrar el BufferedReader
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, se crea un objeto ***Path*** a partir de la ruta del archivo de texto. Luego, se llama a **Files.newBufferedReader** con el objeto *Path* para crear un ***BufferedReader*** que se puede utilizar para leer caracteres del archivo de texto. El método *read() de BufferedReader* se utiliza para leer caracteres del archivo uno por uno. El valor de retorno de *read()* es el código Unicode del caracter leído, y se puede convertir a un char utilizando (*char*). El bucle *while* se ejecuta hasta que se llegue al final del archivo, que se indica por el valor -1 devuelto por *read().* Finalmente, se llama al método *close()* en *BufferedReader* para cerrar el flujo después de leer los caracteres del archivo.

InputStream newInputStream (Path path [, OpenOption opcion1 …, OpenOption opcionN])

FileWriter(String path [, boolean append])	Métodos: ▪ Void write(int c) ▪ Void write(String cadena) ▪ Void write(char [] array) ▪ Void flush() ▪ Void close()

EJEMPLO FileWriter(String path [, boolean append])

```java
import java.io.*;

public class FileWriterEjemplo {
    public static void main(String[] args) {
        String rutaArchivo = "ruta/del/archivo.txt"; // Ruta del archivo de

        try {
            // Crear un objeto FileWriter con la ruta del archivo
            FileWriter fileWriter = new FileWriter(rutaArchivo);

            // Escribir caracteres en el archivo
            fileWriter.write("Hola, mundo!"); // Escribir una cadena de cara
            fileWriter.write('\n'); // Escribir un salto de línea
            fileWriter.write("¡Hola desde Java!"); // Escribir otra cadena

            fileWriter.close(); // Cerrar el FileWriter
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, se crea un objeto *FileWriter* con la ruta del archivo de texto como argumento. Luego, se utiliza el método *write() de FileWriter* para escribir caracteres en el archivo. Puedes escribir cadenas de caracteres, caracteres individuales y cualquier otro tipo de dato que se pueda convertir a una representación de texto. Finalmente, se llama al método *close() en FileWriter* para cerrar el flujo después de escribir los caracteres en el archivo.

FileWriter(File path [, boolean append])	
FileOutputStream(String path [, boolean append])	Fichero de salida de texto. Representa ficheros de texto para escritura a los que se accede de forma secuencial. Métodos: ▪ Void write(int b) ▪ Void write(byte[] a) ▪ Void flush() ▪ Void close()

EJEMPLO FileOutputStream(String path [, boolean append])

```java
import java.io.*;

public class FileOutputStreamSimpleEjemplo {
    public static void main(String[] args) {
        String rutaArchivo = "ruta/del/archivo.bin"; // Ruta del archivo bina

        try {
            // Crear un objeto FileOutputStream con la ruta del archivo
            FileOutputStream fileOutputStream = new FileOutputStream(rutaArch

            // Escribir un byte en el archivo
            int dato = 65; // Representa el valor ASCII de la letra 'A'
            fileOutputStream.write(dato); // Escribir el byte en el archivo

            fileOutputStream.close(); // Cerrar el FileOutputStream
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Se crea un objeto *FileOutputStream* con la ruta del archivo binario como argumento. Luego, se utiliza el método *write() de FileOutputStream* para escribir un byte en el archivo. En este caso, se escribe el **valor ASCII** de la letra ***'A' (65)*** en el archivo. Finalmente, se llama al método *close() en FileOutputStream* para cerrar el flujo después de escribir el byte en el archivo.

FileOutputStream(File path [, boolean append])

- MÉTODOS GENERALES

– **OutputStream newOutputStream (Path path [, OpenOption opcion1 …, OpenOption opcionN])**

EJEMPLO

```java
import java.io.*;

public class OutputStreamEjemplo {
    public static void main(String[] args) {
        String rutaArchivo = "ruta/del/archivo.bin"; // Ruta del archivo bin

        try {
            // Crear un nuevo flujo de salida (OutputStream) para el archivo
            OutputStream outputStream = new FileOutputStream(rutaArchivo);

            // Escribir datos en el flujo de salida
            byte[] bytes = {72, 111, 108, 97}; // Bytes a escribir en el arc
            outputStream.write(bytes); // Escribir los bytes en el flujo de

            outputStream.close(); // Cerrar el flujo de salida
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En el ejemplo superior, se crea un nuevo flujo de salida ***(OutputStream)*** utilizando la clase ***FileOutputStream,*** que se utiliza para escribir datos en un archivo en forma de bytes. Luego, se define un arreglo de bytes que contiene los valores *ASCII* correspondientes a la cadena "Hola". A continuación, se utiliza el método *write() de OutputStream* para escribir los bytes en el flujo de salida. Finalmente, se llama al método *close()* en *OutputStream* para cerrar el flujo después de escribir los bytes en el archivo. Es importante cerrar el flujo para asegurarse de que los datos escritos se guarden correctamente en el archivo.

– **StandartOpenOption**	– Append – Create – Create_new – Dsync – Read	– Sparse – Sync – Truncate_existing – Write
– **DataOutputStream**	Para escribir datos primitivos y otros tipos de datos en un archivo binario o en una secuencia de bytes.	

EJEMPLO

```java
public class DataOutputStreamEjemplo7 {
    public static void main(String[] args) throws IOException {
        OutputStream outputStream = new FileOutputStream("datos.bin");
        DataOutputStream dataOutputStream = new DataOutputStream(outputStream);

        String nombre = "Juan";
        int edad = 30;
        double salario = 2500.50;

        // Escribir datos en el flujo de salida
        dataOutputStream.writeUTF(nombre);
        dataOutputStream.writeInt(edad);
        dataOutputStream.writeDouble(salario);

        // Cerrar el flujo de salida de datos
        dataOutputStream.close();

        System.out.println("Datos escritos en el archivo datos.bin exitosamente.");
    }
}
```

En el ejemplo anteriormente expuesto, se utiliza *DataOutputStream* para escribir una cadena de caracteres en formato *UTF-8 (writeUTF()), un entero (writeInt())* y un número de punto flotante *(writeDouble())* en el flujo de salida de datos *(outputStream).* Luego, se cierra el flujo de salida de datos llamando al método *close()* en *DataOutputStream* para asegurarse de que los datos escritos se guarden correctamente en el archivo "*datos.bin*". Finalmente, se muestra un mensaje de éxito en la consola mediante ***System.out.println().***

- **ObjectInputStream**

Clase en Java que proporciona funcionalidad para *deserializar* objetos desde un flujo de entrada. Se utiliza para convertir objetos serializados (datos binarios) en objetos Java que se pueden usar en un programa Java.

EJEMPLO

```java
import java.io.*;

no usages
public class EjemploObjectInputStream {
    public static void main(String[] args) {
        try {
            // Crear un flujo de entrada desde un archivo
            FileInputStream fileInputStream = new FileInputStream("objeto_serializado.ser");

            // Crear un objeto ObjectInputStream para leer desde el flujo de entrada
            ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);

            // Leer el objeto serializado
            Object objeto = objectInputStream.readObject();

            // Realizar las operaciones necesarias con el objeto deserializado
            // ...

            // Cerrar el flujo de entrada
            objectInputStream.close();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, se crea un *ObjectInputStream* a partir de un *FileInputStream* que representa un archivo que contiene un objeto serializado. Luego, se utiliza el método *readObject()* para leer el objeto serializado del flujo de entrada y se realiza cualquier operación necesaria con el objeto deserializado. Finalmente, se cierra el flujo de entrada con el método *close()*. Es importante manejar las excepciones IOException y ClassNotFoundException que pueden ocurrir durante la deserialización.

- **ObjectOutputStream**

Clase en Java que se utiliza para escribir objetos en forma de flujo de bytes en un flujo de salida. Es parte del paquete *java.io* y se utiliza para serializar objetos en Java, lo que permite que los objetos sean convertidos en una secuencia de bytes para ser almacenados o transmitidos a través de una red.

EJEMPLO

```java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.io.IOException;
// Clase que será serializada
2 usages
class MiObjeto implements Serializable {
    no usages
    private static final long serialVersionUID = 1L; // Identificador de versión para la serialización
    1 usage
    private String nombre;
    1 usage
    private int edad;

    1 usage
    public MiObjeto(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Getters y setters (omitidos para simplificar el ejemplo)
}
no usages
public class EjemploObjectOutputStream {
    public static void main(String[] args) {
        // Crear un objeto para serializar
        MiObjeto objeto = new MiObjeto( nombre: "Juan",  edad: 30);

        // Crear un flujo de salida
        try (FileOutputStream archivoSalida = new FileOutputStream("objeto_serializado.bin")) {
            // Crear un ObjectOutputStream
            ObjectOutputStream objetoSalida = new ObjectOutputStream(archivoSalida);

            // Escribir el objeto en el flujo de salida
            objetoSalida.writeObject(objeto);

            // Cerrar el flujo de salida
            objetoSalida.close();

            System.out.println("El objeto ha sido serializado y guardado en objeto_serializado.bin");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

En este ejemplo, se crea un objeto de la clase *MiObjeto*, que implementa la interfaz Serializable, para indicar que puede ser serializado. Luego, se crea un flujo de salida utilizando *FileOutputStream* para escribir en un archivo llamado "objeto_serializado.bin". A continuación, se crea un *ObjectOutputStream* para escribir el objeto en el flujo de salida utilizando el método *writeObject*(). Finalmente, se cierra el flujo de salida con el método *close()*.

2.2.3.2. ACCESO DIRECTO

En el paquete *java.io* existe un paquete de clases especial, nombrado como **RandomAccessFile,** utilizado para el acceso aleatorio de archivos. Dicho de otro modo, permite utilizar los archivos en modo lectura y escritura simultáneamente, o acceder a datos de forma aleatoria indicando la posición en la que se quiere operar.

1. **Constructores** de la clase *RandomAcessFile*

New RandomAccessFile(String path, String modo)
New RandomAccessFile(File path, String modo)

EJEMPLO: New Randomaccessfile(String path, String modo)

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {
    public static void main(String[] args) {
        String filePath = "archivo.txt";
        String mode = "rw";
        try (RandomAccessFile file = new RandomAccessFile(filePath, mode)) {
            String contenido = "Este es el contenido del archivo.";
            file.write(contenido.getBytes());
            System.out.println("Se escribió el contenido en el archivo.");
        } catch (IOException e) {
            System.err.format("Error al escribir en el archivo: %s%n", e);
        }
    }
}
```

Siguiendo el ejemplo, se crea una cadena de texto llamada *filePath* que representa la ruta del archivo que se desea escribir y se crea otra cadena de texto llamada *mode* que especifica el modo de apertura del archivo como "**rw**" (lectura y escritura). El constructor **RandomAccessFile** se utiliza para crear un objeto **RandomAccessFile** que se utilizará para escribir en el archivo, que se inicializa con el *path* y el *mode*. Dentro del bloque *try-with-resources*, se utiliza el método *write* del objeto RandomAccessFile para escribir el contenido en el archivo. El método *getBytes* se utiliza para obtener un arreglo de bytes que representa el contenido en el conjunto de caracteres predeterminado. Si se produce una excepción al escribir en el archivo, se captura y se muestra un mensaje de error en la consola.

Fichero de entrada o salida binario con acceso aleatorio: es la base para crear los objetos de tipo fichero de acceso aleatorio, permitiendo el acceso a cualquier registro del archivo de forma aleatoria. Es decir, se pueden leer o escribir datos en cualquier posición del archivo sin tener que recorrer los registros previos en secuencia.

Los archivos de acceso aleatorio son muy útiles para trabajar con grandes cantidades de datos que se almacenan de forma estructurada, como por ejemplo, archivos de bases de datos. En lugar de leer o escribir todo el archivo cada vez que se necesita acceder a un registro, se puede acceder directamente al registro deseado.

Modos de acceso	– r	– rw	– rwd	– rws

2. **Métodos** de acceso directo

▪ **Void close()**	▪ **Int skipBytes(int n)**	▪ **Void write(int b)**
▪ **Int readItn()**	▪ **Void writeDouble(double v)**	▪ **Int read()**
▪ **String readUTF()**	▪ **Void writeFloat(float v)**	▪ **Long lenght()**
▪ **Void seek(long pos)**		

3. LA CLASE FILE

En Java, la clase File se utiliza para trabajar con archivos y directorios en el sistema de archivos. Esta clase proporciona métodos para crear, leer, escribir, eliminar y renombrar archivos y directorios.

Para trabajar con un archivo, primero se crea una instancia de la clase File. Esto se puede hacer utilizando una ruta relativa o absoluta:

```
File archivo = new File("ruta/al/archivo.txt"); // ruta relativa
File archivo = new File("C:/Users/usuario/archivo.txt"); // ruta absoluta
```

Una vez que se tiene una instancia de la clase File, se pueden utilizar los métodos de la clase para realizar operaciones en el archivo o directorio. Algunos de los métodos más comunes son:

- **exists():** devuelve true si el archivo o directorio existe, y false en caso contrario.
- **createNewFile():** crea un nuevo archivo en el sistema de archivos.
- **delete():** elimina el archivo o directorio.
- **getName():** devuelve el nombre del archivo o directorio.
- **isDirectory():** devuelve true si el objeto File representa un directorio, y false si representa un archivo.
- **listFiles():** devuelve un arreglo de objetos File que representan los archivos y subdirectorios de un directorio.
- **Y otros como: isFile(),renameTo(),canRead(),canWrite(),getPath(),getAbsolutePath(), getParent(),mkdir(),list().**

Por ejemplo, para crear un nuevo archivo y escribir texto en él, se podría hacer lo siguiente:

```
File archivo = new File("ruta/al/archivo.txt");
try {
    if (archivo.createNewFile()) {
        FileWriter writer = new FileWriter(archivo);
        writer.write("Este es el contenido del archivo");
        writer.close();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

En este ejemplo, se crea un nuevo archivo en la ruta especificada y se escribe el texto "Este es el contenido del archivo" en él. Si el archivo ya existía, no se crea uno nuevo.

EJEMPLO 1

```
import java.io.File;
import java.io.IOException;

no usages
public class CrearArchivo {
    public static void main(String[] args) {
        // Nombre del archivo a crear
        String nombreArchivo = "nuevo_archivo.txt";

        // Crear un objeto de la clase File para representar el archivo
        File archivo = new File(nombreArchivo);

        try {
            // Intentar crear el archivo con el método createNewFile()
            if (archivo.createNewFile()) {
                System.out.println("El archivo " + nombreArchivo + " ha sido creado exitosamente.");
            } else {
                System.out.println("El archivo " + nombreArchivo + " ya existe.");
            }
        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al crear el archivo.");
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, primero se crea un objeto de la clase File con el nombre del archivo que se desea crear. Luego, se llama al método createNewFile() en este objeto para crear el archivo en el sistema de archivos. Si el archivo se crea correctamente, el método devuelve true y se muestra un mensaje de éxito en la consola. Si el archivo ya existe, el método devuelve false y se muestra un mensaje de archivo existente. Si ocurre un error al crear el archivo, se muestra un mensaje de error y se imprime la pila de excepción.