5. Cadenas y Entrada/Salida

- Manejo de cadenas
  - `java.lang.String`
  - `java.lang.StringBuffer`
  - `java.util.StringTokenizer`
- Control de errores
- Entrada/Salida (`java.io`)
  - Streams de datos (`DataInputStream/DataOutputStream`)
  - Streams de caracteres (`Reader/Writer`)
  - Entrada y salida estándar (`System.in/System.out`)
  - Manejo de ficheros: `File, RandomAccessFile`

Cadenas de texto

- **`String`** es solo-lectura mientras que **`StringBuffer`** es mutable
- Métodos básicos de la clase `String`:

```
public String()                  //constructor de un obj de valor “”
public String(String value)      //constructor de obj copia de value
public int length()              //longitud
public char charAt(int index)    //0 ≤ index ≤ length-1
```

(si se accede a cualquier otra posición lanza **IndexOutOfBoundsException**)

Ejemplo:

```
static int numCaracteresEntre (String str, char car){
       int primeraPos = str.indexOf(car);
       if (primeraPos<0) return -1; //no existe el car en str
       int ultimaPos = str.lastIndexOf(car);
       return ultimaPos - primeraPos -1;
}
```

Clase `String`

- **Comparación:**
  - `boolean` **`equals`**: compara contenido de los strings.
  - `boolean` **`equalsIgnoreCase`**: no distingue entre mayúsculas y minúsculas.
  - `int` **`compareTo`**: útil para ordenar Strings.
- **Modificación:**
  - `String` **`replace`** `(char oldChar, char newChar)`
  - `String` **`toLowerCase`**`()`
  - `String` **`toUpperCase`**`()`
  - `String` **`trim`**`()`: quitar espacios en blanco
- **Otros:**
  - `boolean` **`startsWith`** `(String prefix)`
  - `boolean` **`endWith`** `(String suffix)`
  - `String` **`substring`** `(int beginIndex)`
  - `String` **`substring`**`(int beginIndex, int endIndex)`

Ejemplo: `String`

Ejemplo2:

```
public static String subCadena (String str, char ini, char fin){
       int posInicial = str.indexOf(ini);
       int posFinal = str.lastIndexOf(fin);

       if (posInicial==-1) return null;   //no existe ini en str
       else if (posFinal == -1)
                     return str.substring(posInicial);
       else
              return str.substring(posInicial, posFinal+1);
}
```

Clase `StringBuffer`

- `StringBuffer` permite modificar el `String` en lugar de crear uno nuevo en cada paso intermedio.
- Métodos similares e igual nombre que `String` pero son independientes.
- Constructores:
  - **`StringBuffer()`**
  - **`StringBuffer(String)`**
- Métodos principales:
  - **`setCharAt`**`(int, char)`
  - `StringBuffer` **`insert`** `(int offset, valor)`
  - `StringBuffer` **`append`** `(valor)` donde `valor` puede ser:
    - `int, char, boolean, float, double, long`
    - `Object`
    - `String`
    - `char [ ] str`

Ejemplo: `StringBuffer`

Ejemplo3:

```
public static String escribirRaiz (int i){
        StringBuffer buf = new StringBuffer();

        buf.append("sqrt(").append(i).append(')');
        buf.append(" = ").append(Math.sqrt(i));
        return buf.toString();
}
```

int

char

String

double

Ejemplo4:

```
public static StringBuffer insertaFecha (StringBuffer buf)){
       String hoy = new java.util.Date().toString();

       buf.ensureCapacity(buf.length()+hoy.length()+2);
       buf.insert(0,hoy).insert(hoy.length(), ":");
       return buf;
}
```

Ejemplo `String` y `StringBuffer`

- **Ejercicio:** Escribir un método que tome como entrada un string que representa un número y devuelva el mismo numero divido por puntos cada tres dígitos. Esto es, para la entrada "12345" la salida sería "12.345"

```
public static String ponerPuntos(String numero){
        int size = numero.length();
        int numptos= (size -1)/3;


        if (numptos==0) return numero;
        else {
            StringBuffer buf = new StringBuffer(numero);
            for (int i=1;i<=numptos;i++)
                    buf.insert((size-3*i),'.');

            return buf.toString();
        }
    }
```


Clase `StringTokenizer`

- Utilizado para dividir una cadena según unos delimitadores
- Constructores habituales:
  - **`StringTokenizer(String cadena)`**
    Utiliza los delimitadores por defecto: " \t\n\r\f"
  - **`StringTokenizer(String cadena, String delimitadores)`**
- Métodos para iterar:
  - **`boolean hasMoreTokens()`**
  - **`String nextToken()`**

Ejemplo StringTokenizer

```
StringTokenizer st = new StringTokenizer("esto es una
 prueba");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
```

Imprime la siguiente salida:

```
esto
es
una
prueba
```


Ejemplo StringTokenizer

```
public class Logica{
 public static String separaAND (String proposicion){
    StringTokenizer strTokenizer;
     strTokenizer = new StringTokenizer(proposicion, "^");
    StringBuffer buffer = new StringBuffer();

    String token = null; boolean inicio=true;
    while (strTokenizer.hasMoreTokens()){
      token = strTokenizer.nextToken();
      if (!inicio) buffer.append("^");
      buffer.append("(").append(token.trim()).append(")");
      inicio = false;
    }
    return buffer.toString();
  }
}
```

Control de errores

- Cuando se produce un error en un método se genera una **excepción**.
- Estructura:

```
try{
  //bloque que puede provocar un error
}catch (tipoExcepcion identificador){
  /* acciones a realizar si se produce
     el error
  */
}finally{
  // acciones a realizar haya o no error
}
```


Entrada/Salida. Paquete `java.io`

- Define la entrada/salida en términos de ***streams***
- Un *stream* es una secuencia ordenada de datos
- Tienen:
  - una *fuente* = streams de entrada o
  - un *destino* = streams de salida
- El paquete `java.io` tiene dos partes principales:
  - Stream de caracteres (caracteres Unicode de 16 bits)
  - Stream de bytes (8 bits)

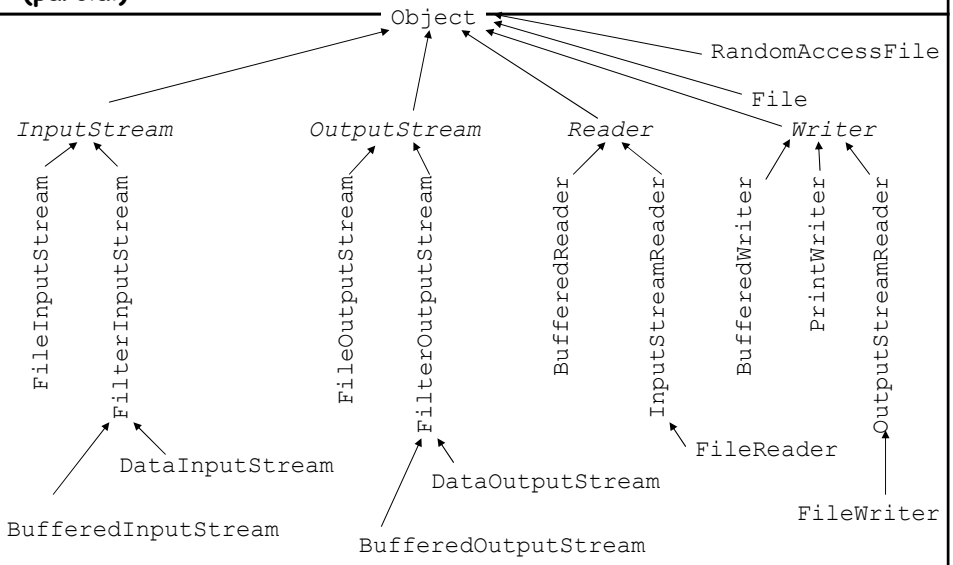
Entrada/Salida. Paquete `java.io`

- E/S puede estar **basada**:
  - **En texto**: *streams* de caracteres legibles
    Ejemplo: el código fuente de un programa
  - **En datos**: *streams* de datos binarios
    Ejemplo: patrón de bits de una imagen
- Los ***streams* de caracteres** se utilizan en la E/S basada en texto.
  - Se denominan **lectores** (*reader*) y **escritores** (*writer*)
- Los ***streams* de bytes** se utilizan en la E/S basada en datos.
  - Se denominan **streams de entrada** y **streams de salida**

Clases principales de `java.io`

- **Clases de flujo de entrada:**
  - Se utilizan para leer datos de una fuente de entrada (archivo, cadena o memoria)
  - Flujo de bytes: **`InputStream`**, **`BufferedInputStream`**, **`DataInputStream`**, **`FileInputStream`**
  - Flujo de caracteres: **`Reader`**, **`BufferReader`**, **`FileReader`**
- **Clases de flujo de salida:**
  - Son las homólogas a las clases de flujo de entrada y se utilizan para enviar flujos de datos a dispositivos de salida
  - Flujo de bytes: **`OutputStream`**, **`PrintStream`**, **`BufferedOutputStream`**, **`DataOutputStream`** y **`FileOutputStream`**
  - Flujo de caracteres : **`Writer`**, **`PrintWriter`**, **`FileWriter`**
- **Clases de archivo:**
  - **`File`** y **`RandomAccesFile`** (mayor control sobre los archivos)

Jerarquía de clases de `java.io`
(parcial)



`InputStream`

Método	Descripción
`read()`	Lee el siguiente byte del flujo de entrada y lo **devuelve** como **un entero**. Cuando alcanza el final del flujo de datos, devuelve -1. **EOF**
`read(byte b[])`	Lee múltiples bytes y los almacena en la matriz b. Devuelve el número de bytes leídos o -1 cuando se alcanza el final del flujo de datos.
`read(byte b[], int off, int long)`	Lee hasta len bytes de datos del flujo de entrada, empezando desde la posición indicada por el desplazamiento off, y los almacena en una matriz.
`available()`	Devuelve el número de bytes que se pueden leer de un flujo de entrada sin que se produzca un bloqueo por causa de una llamada a otro método que utiliza el mismo flujo de entrada.
`skip(long n)`	Omite la lectura de n bytes de datos de un flujo de entrada y los descarta.
`close()`	Cierra un flujo de entrada y libera los recursos del sistema utilizados por el flujo de datos.

OutputStream

Método	Descripción
`write(int b)`	Escribe b en un flujo de datos de salida.
`write(byte b[])`	Escribe la matriz b en un flujo de datos de salida.
`write(byte b[], int off, int long)`	Escribe len bytes de la matriz de bytes en el flujo de datos de salida, empezando en la posición dada por el desplazamiento off
`flush()`	Vacía el flujo de datos y fuerza la salida de cualquier dato almacenado en el búfer.
`close()`	Cierra el flujo de datos de salida y libera cualquier recurso del sistema asociado con él.

Streams sobre Ficheros

- **`FileInputStream`**: muy similar a la clase `InputStream`, sólo que está diseñada para leer archivos.
  - `FileInputStream(String name)`
  - `FileInputStream(File name)`
- **`FileOutputStream`**: muy similar a la clase `OutputStream`, sólo que está diseñada para escribir en archivos.
  - `FileOutputStream(String name)`
  - `FileOutputStream(String name, boolean append)`
    - Si `append==true` queremos añadir al final del fichero
  - `FileOutputStream(File name)`

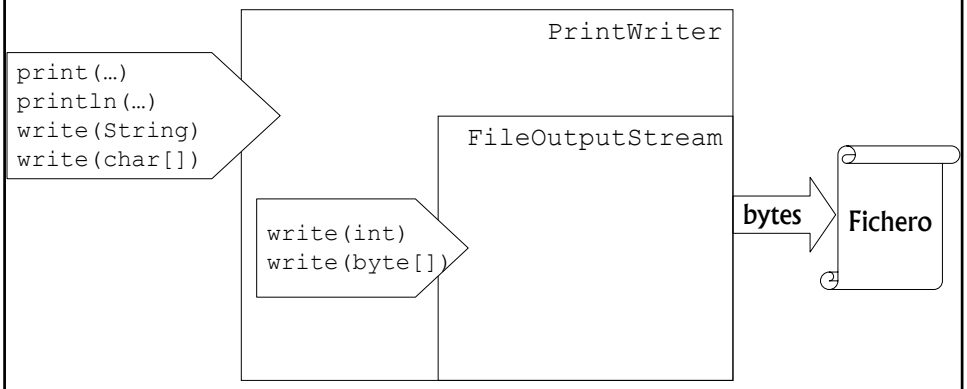
Ejemplo: streams sobre ficheros

```
import java.io.*;
public class Copia {
  public static void main(String args[]) {
    FileInputStream origen = null;
    FileOutputStream destino = null;
    try {
      origen = new FileInputStream(args[0]);
      destino = new FileOutputStream(args[1],true); //añadir
      int i = origen.read();
      while (i != -1) { // mientras not EOF
        destino.write(i);
        i = origen.read();
      }
     origen.close(); destino.close();
    }catch (IOException e) {
       System.out.println("Error de ficheros");
     }
   }
}
```


Reader y Writer

- Dar soporte Unicode en todas las operaciones de E/S.
- En ocasiones hay que combinar streams de caracteres y de bytes:
  - **`InputStreamReader`**: convierte un `InputStream` en un `Reader`
  - **`OutputStreamWriter`**: convierte un `OutputStream` en un `Writer`
- Casi todas las clases de la jerarquía de streams de bytes tienen su correspondiente clase `Reader` o `Writer` con interfaces casi idénticas.
- **`BufferedReader`** y **`BufferedWriter`**: almacenamiento temporal en un *buffer*, para no actuar directamente sobre el stream.
- Igual que los streams de bytes se deben cerrar explícitamente para liberar sus recursos asociados (`close`).

Escribir en fichero



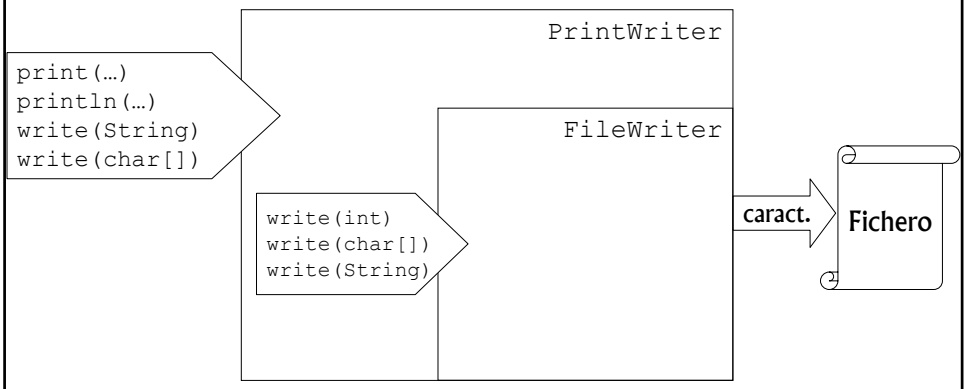

```
FileOutPutStream fos = new FileOutputStream("fichero.txt");
PrintWriter pr = new PrintWriter(fos);
…
pr.println("Escribimos texto");
```


Ejemplo: Escribir en un fichero

```
public class EscribirFichero {
  public static void main(String[] args) {
    try{
      FileOutputStream fos = new FileOutputStream("salida.txt");
      PrintWriter pw = new PrintWriter(fos);
      pw.println("Imprimimos una cadena y un entero " + 5);
      pw.flush();
      pw.close();
      fos.close();
    }catch (FileNotFoundException e){ }
     catch (IOException e2){ }
  }
}
```

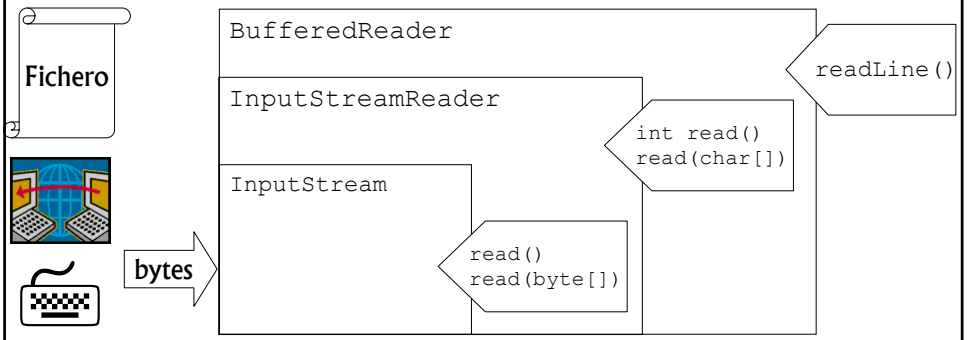
NOTA: El flujo de salida se convierte en un **`PrintWriter`** para hacerlo legible como un archivo de texto normal.

Escribir en fichero (2)




```
FileWriter fw = new FileWriter("fichero.txt");
PrintWriter pr = new PrintWriter(fw);
…
pr.println("Escribimos texto");
```


Lectura de líneas



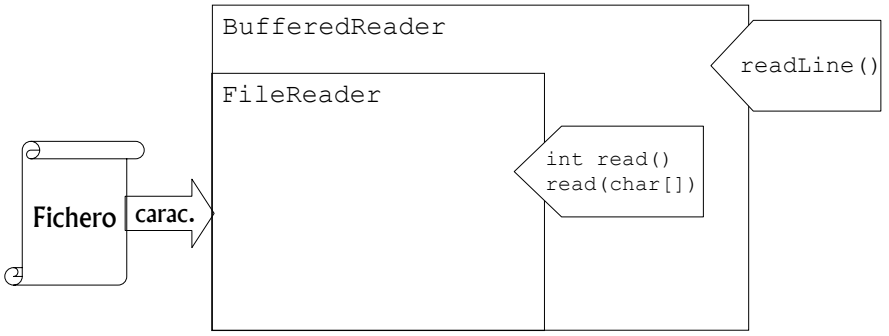

```
FileInputStream fis = new FileInputStream("fichero.txt");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader buffer = new BufferedReader(isr);
…
String linea = buffer.readLine();
```

Ejemplo: Lector de fichero

```
public class LectorFichero {
  public static void main(String args[]) {
       byte [] buffer = new byte[80];
       try {
          FileInputStream fichero = new
                        FileInputStream("Leeme.txt");
          int i = fichero.read(buffer);
          String s = new String(buffer);
          System.out.println(s);
        }catch(FileNotFoundException e) {
        }catch(IOException e) {
        }
  }
}
```

NOTA: Podemos cambiar FileInputStream por FileReader en cuyo caso el buffer sería un char[].

Leer de fichero



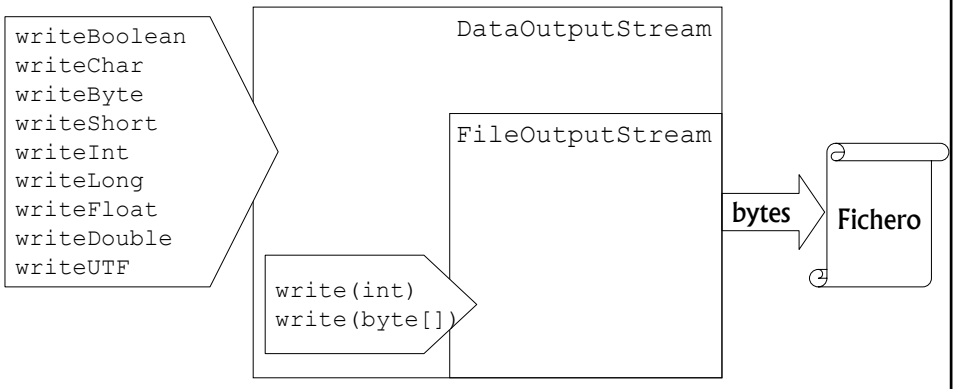

```
FileReader fr = new FileReader("fichero.txt");
BufferedReader buffer = new BufferedReader(fr);
…
String linea = buffer.readLine();
```

Clases DataInputStream y DataOutputStream

- Permiten transmitir tipos primitivos por un stream.

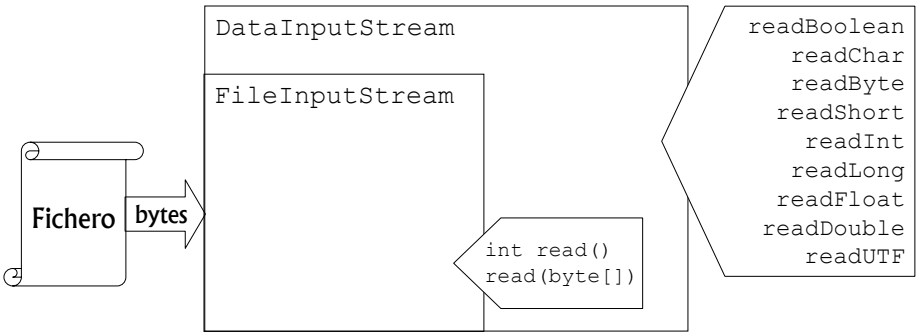
Lectura	Escritura	Tipo
readBoolean	writeBoolean	boolean
readChar	writeChar	char
readByte	writeByte	byte
readShort	writeShort	short
readInt	writeInt	int
readLong	writeLong	long
readFloat	writeFloat	float
readDouble	writeDouble	double
readUTF	writeUTF	String

Escritura en modo datos




```
FileOutputStream fos = new FileOutputStream("salida.dat");
DataOutputStream dos = new DataOutputStream(fos);
dos.writeInt(5);
```

Lectura en modo datos




```
FileInputStream fis = new FileInputStream("salida.dat");
DataInputStream dis = new DataInputStream(fis);
int entero = dis.readInt();
```

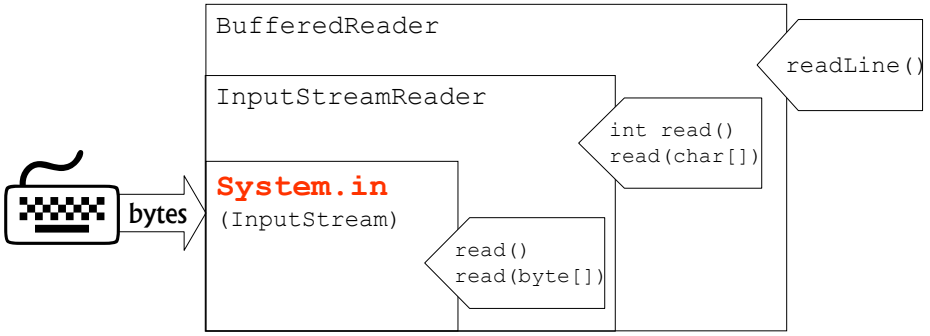

Entrada/Salida estándar

- Stream de salida estándar: **`System.out`**
  - Objeto `PrintStream` (es un tipo de `OutputStream`)
  - Métodos de escritura **`print(valor)`** y **`println(valor)`** para los siguientes tipos:

`char`	`int`	`float`	`Object`	`boolean`
`char[]`	`long`	`double`	`String`	

- Stream de entrada estándar: **`System.in`**
  - Objeto `InputStream`
- Salida de error "estándar": **`System.err`**
  - Objeto `PrintStream`
  - Mostrar mensajes de error o cualquier otra información que requiera la atención inmediata del usuario

Ejemplo lectura del teclado




```
BufferedReader teclado = new BufferedReader(new
                                InputStreamReader(System.in));
String entrada = teclado.readLine();
```


Clase `File`

- Representa realmente una vía de acceso, no necesariamente un archivo.
- Independiente de la plataforma: **`File.separator`**
- Constructores:
  - `File (String viaAcceso)`
  - `File (String directorio, String fichero)`
  - `File (File directorio, String fichero)`
- Métodos para crear y borrar archivos o directorios, cambiar el nombre de un archivo, leer el nombre del directorio, consultar si un nombre representa un fichero o directorio, listar el contenido de un directorio (**`String [] list()`**), ...

Ejemplo: `File`

```
class DatosArchivo{
  public static void main (String [] args){
      File f = new File(args[0]);
      System.out.println(
            "Ruta absoluta: " + f.getAbsolutePath()+
            "\n Puede leer: " + f.canRead()+
            "\n Puede escribir: " + f.canWrite()+
            "\n Nombre del fichero: " + f.getName()+
            "\n Padre del fichero: " + f.getParent()+
            "\n Ruta del fichero: " + f.getPath()+
            "\n Longitud: " + f.length()+
            "\n Ultima modificación: " + f.lastModified());
      if (f.isFile())
            System.out.println("Es un archivo");
      else if (f.isDirectory())
            System.out.println("Es un directorio");
  }
}
```


Ficheros de acceso directo. `RandomAccessFile`

- Tiene todas las propiedades de las clases `DataInputStream` y `DataOutputStream`
  - lectura y escritura como tipos de datos primitivos
- Constructores:
  - `RandomAccessFile(File fichero, String modo)`
  - `RandomAccessFile(String fichero, String modo)`

    donde modo = "**r**" (lectura) o "**rw**" (lectura-escritura)
- Métodos:
  - `void` **`seek`**`(long posicion)`: establece posición actual del puntero.
  - `long` **`getFilePointer`**`()`: devuelve la posición actual (en bytes).
  - `int` **`skipBytes`**`(int desplazamiento)`: mueve el puntero el nº de bytes del parámetro
  - `long` **`length`**`()`: longitud del fichero en bytes

Ejemplo: Leemos por el teclado el nombre de un fichero y el valor real que vamos a almacenar en el mismo.

```
class Ficheros{
   public static void main(String[] args) {
   try{
     BufferedReader  teclado = new BufferedReader(
                          new InputStreamReader(System.in));
     RandomAccessFile salida = new RandomAccessFile(
                          teclado.readLine(), "rw");
     salida.seek(salida.length()); //vamos al final del fichero

     System.out.print("Introduzca un valor real: ");
     double valor = Double.parseDouble(teclado.readLine());
     salida.writeDouble(valor);
     salida.close();
   }catch (IOException e){
       System.out.println("Error de E/S");
   }
   }
}
```