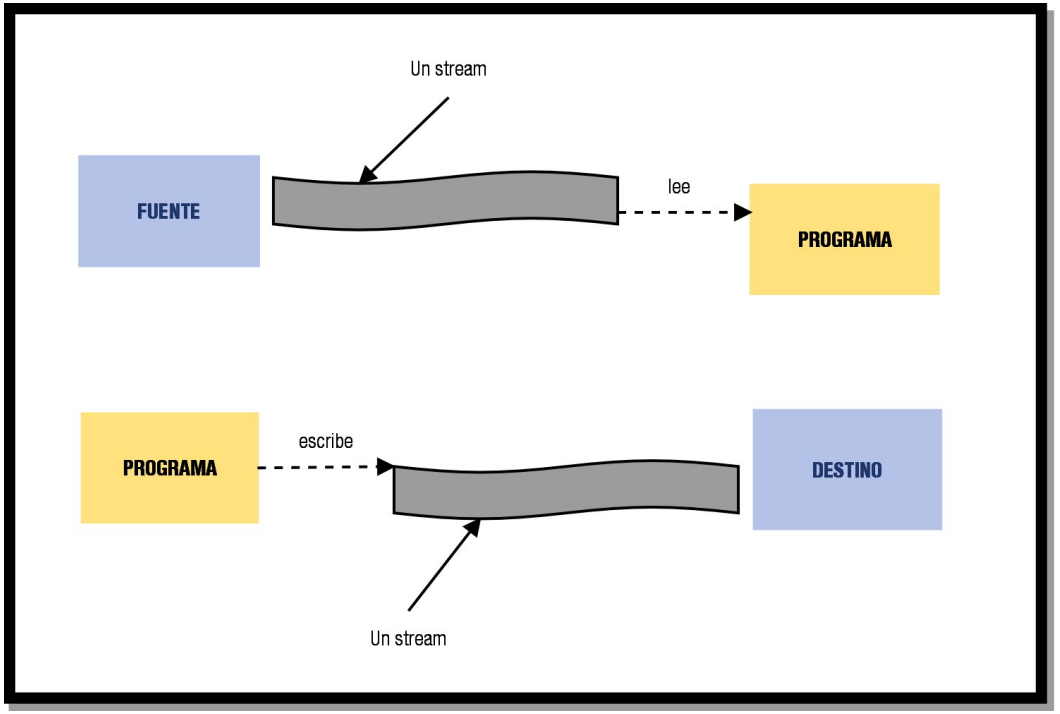
PROGRAMACIÓN

# Unidad 8. Almacenando datos




## MANEJO DE EXCEPCIONES Y FICHEROS

José L. Berenguel

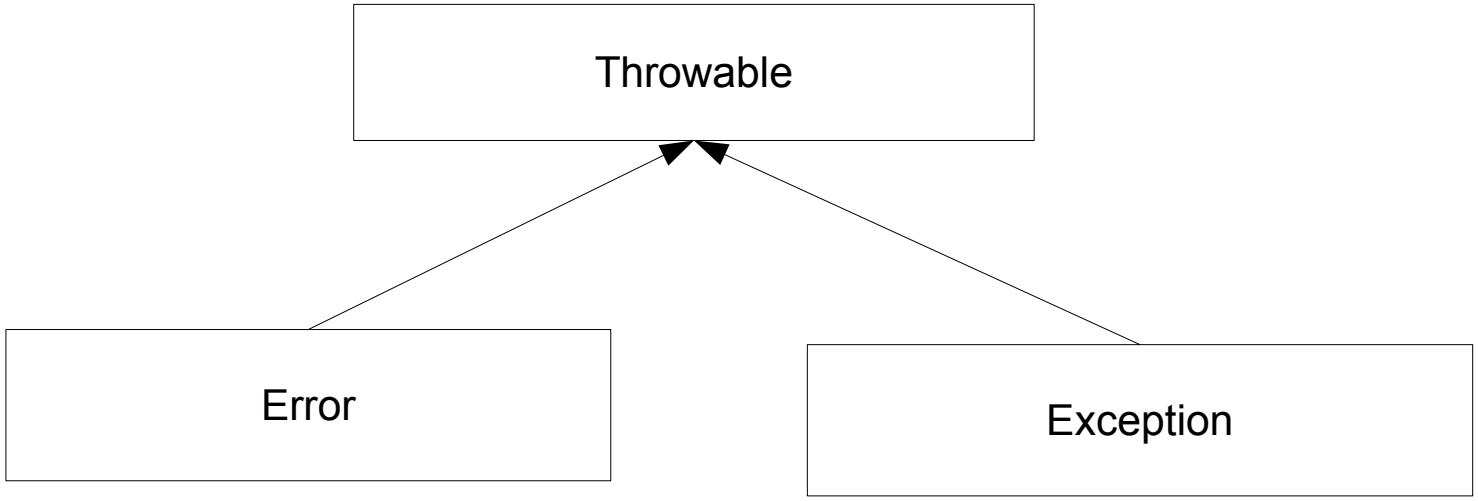
# Tabla de Contenidos

# Introducción

- En esta presentación hablaremos de **excepciones** y **ficheros**.
- Los **ficheros** se utilizan para **almacenar datos de manera persistente** ya que los datos en memoria se pierden al apagar la computadora.
- Paquete ***java.io***, contiene las clases relacionadas para operaciones de entrada y salida (E/S).
  - E/S estándar: teclado y pantalla.
  - E/S fichero.
- Los ficheros se implementan como ***flujos*** de entrada y/o salida. Dependiendo de los datos que transmiten, pueden ser de **carácter** (también llamados texto) o **binarios**.

# Introducción

- **Excepción**: situación anómala en la ejecución de un programa que impide que se siga ejecutando el flujo normal del programa, por lo que el control pasa a otro ámbito capaz de manejar esa situación.
- Jerarquía de clases de excepciones en Java:



# Excepciones

- **Error**: los objetos de este tipo indican que ha sucedido un fallo irrecuperable y es imposible seguir la ejecución del programa por lo que la JVM se encargará de finalizarlo.
- **Exception**: los objetos de este tipo indican una situación anómala que puede corregirse para continuar con la ejecución o que se debe a errores de una mala programación.
  - *NullPointerException*, *IndexOutOfBoundsException*, *ClassCastException*, *IllegalArgumentException*, *InputMismatchException*, *FileNotFoundException*…
  - Podemos **definir nuestras propias excepciones**.

# Excepciones

- **Tipos de excepciones** desde el punto de vista de su tratamiento:
  - **Marcadas**: captura obligatoria en un bloque **try-catch** o relanzarlas a un nivel superior.
  - **No marcadas**: su captura no es obligatoria. Este tipo de excepciones son subclases de ***RuntimeException***.
- Las **excepciones no marcadas** se producen como fruto de errores de programación.

```
int [] numeros = {1,2,3,4,5,6,7,8,9,10}

//Se producirá la excepción IndexOutOfBoundsException
for(int i=0;i<=10;i++){
    System.out.printl(numeros[i]);
}
```

# Excepciones

- Esquema de captura de excepciones con la estructura ***try-catch***.

```
try{
    //bloque de instrucciones donde se pueden producir excepciones

} catch(Excepcion1 arg) {
    //tratamiento de la excepción de tipo Excepcion1

} catch(Excepcion2 arg) {
    //tratamiento de la excepción de tipo Excepcion2

} catch(...) {
    //tantos bloques catch como sean necesarios

} finally {
    //instrucciones que siempre se ejecutarán
}
```

# Excepciones

- Podemos **capturar más de una excepción** usando el operador lógico | .

```
try{
    //bloque de instrucciones donde se pueden producir excepciones

} catch(Excepcion1 | Excepcion2 | ... e) {
    //tratamiento de varios tipos de excepciones
}
```

# Captura de excepciones

- Sección **try**: encierra el conjunto de instrucciones que pueden generar las excepciones.
- Sección **catch**: encierra el código que manejará la excepción, si se produce.
  - Puede haber un bloque *catch* para cada tipo de excepción que se produzca. Solo se ejecutará uno de ellos si la excepción se produce.
  - No puede haber un bloque *try* sin bloque *catch*.
- Sección **finally**: sentencias que se ejecutarán siempre, se produzca o no la excepción.
- El **uso de llaves** en los bloques es **obligatorio**.

# Normas en el manejo de excepciones

- La excepción es tratada por el primer bloque **catch** cuyo parámetro coincida con el tipo de la excepción lanzada (**cuidado** con las propiedades polimórficas de la herencia).
- Cuando se maneja la excepción, la ejecución continúa por la siguiente instrucción tras el último bloque **catch** (o **finally**).
- Si una excepción no se captura por ningún bloque **catch** se relanza hacia el método anterior en la llamada.
  - Si es una excepción marcada, debe indicarse en la cabecera del método con ***throws***.
- Si ningún método maneja la excepción será la JVM la encargada de hacerlo.
- El bloque **finally** se escribe después del último bloque **catch**.

# Normas en el manejo de excepciones

- Los bloques ***catch*** se deben ordenar de modo que las subclases estén antes que una superclase genérica.
- En el siguiente ejemplo, la segundo bloque ***catch*** nunca se ejecutará.

```
try{

  c = a/b;

} catch(Exception e){
    System.out.println("Estoy en el primer catch");
} catch(ArithmeticException e){
    System.out.println("Estoy en el segundo catch");
}
```

# Normas en el manejo de excepciones

- Si en un bloque **catch** se genera una nueva excepción, esta se propagaría hacia los métodos llamantes.
- Si se transfiere el control desde un bloque **try** a otra sección de código con **break**, **continue**, o **return**, primero se ejecutará el bloque **finally**, si lo hubiere.

```
try{
  System.out.println("Código que genera una excepción");
  throw new Exception();

} catch(Exception e){
    return ...;
} finally {
    System.out.println("Se ejecuta antes que el return");
}
```

# Crear una excepción propia

- Se debe crear una clase que herede de ***Exception*** (o ***RuntimeException***, para excepciones no marcadas).
- ***throw***: lanza la excepción en ese lugar del código:
  - *throw new MiExcepcion();*
- ***throws***: se coloca en la cabecera de un método, indicando que ese método genera esa excepción. Su uso es obligatorio en el caso de excepciones marcadas que no se capturan por un *try-catch*.
  - *public void miMetodo() throws MiExcepcion {...}*

# Crear una excepción propia

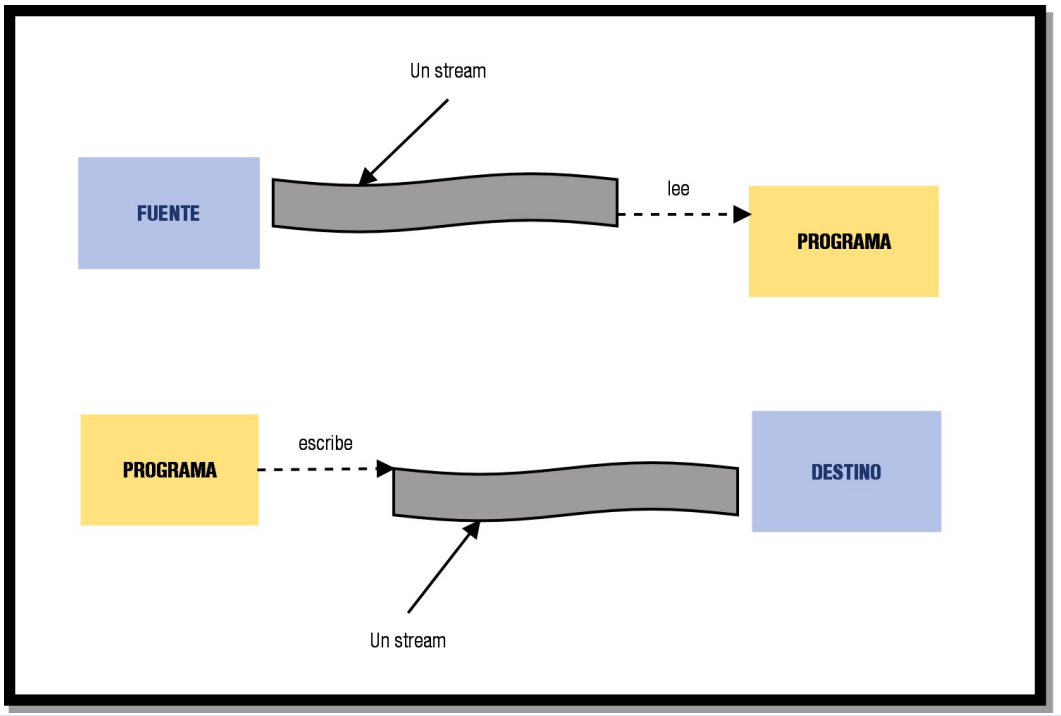
```
public class MiExcepcion extends Exception {

    public MiExcepcion() {
        super("Error: Excepción generada. ");
    }
}
```

```
public void miMetodo() throws MiExcepcion, OtraExcepcion {
    ...
    if(..){
        throw new MiExcepcion();
    }
    if(..){
        throw new OtraExcepcion();
    }
}
```

# Ficheros y flujos (*stream*)

- El manejo de ficheros en Java se realiza a través de un concepto más amplio: un **flujo** (***stream***).
- Este concepto nació durante el desarrollo del lenguaje de programación C, ligado al desarrollo del S.O. Unix.
- En este entorno, **todo dispositivo de entrada y salida es considerado un flujo** y su tratamiento es idéntico.



# Ficheros y flujos (*stream*)

- **Superclases abstractas**:
  - ***InputStream***: flujo de entrada.
  - ***OutputStream***: flujo de salida.
- La JVM provee varios objetos **static** dentro de la clase *System* para lectura y escritura de información:
  - ***System.err***: objeto ***PrintStream***, subclase de ***FilterOutputStream*** y esta de ***OutputStream***.
  - ***System.out***: objeto ***PrintStream***.
  - ***System.in***: objeto de tipo ***InputStream***.
- Dos **modos de escritura**: **texto** o **binario**.
  - Clases ***Stream***: orientadas a bytes (binario).
  - Clases ***Reader***/***Writer***: orientadas a caracteres (texto).

# Ficheros y flujos (*stream*)

- Jerarquía de clases del API Java orientadas a byte (binario).

| Jerarquía de clases *InputStream* | Jerarquía de clases *OutputStream* |
|---|---|
| InputStream<br>**FileInputStream**<br>PipedInputStream<br>ByteArrayInputStream<br>StringBufferInputStream<br>SequenceInputStream<br>FilterInputStream<br>**DataInputStream**<br>LineNumberInputStream<br>**BufferedInputStream**<br>PushbackInputStream<br>**ObjectInputStream** | OutputStream<br>**FileOutputStream**<br>PipedOutputStream<br>ByteArrayOutputStream<br>FilterOutputStream<br>**DataOutputStream**<br>PrintStream<br>**BufferedOutputStream**<br>PushbackOutputStream<br>**ObjectOutputStream** |

# Ficheros y flujos (*stream*)

- Jerarquía de clases del API Java orientadas a carácter (texto).

| Jerarquía de clases *Reader* | Jerarquía de clases *Writer* |
|---|---|
| Reader<br>CharArrayReader<br>PipedReader<br>StringReader<br>**BufferedReader**<br>LineNumberReader<br>InputStreamReader<br>**FileReader**<br>FilterReader<br>PushbackReader | Writer<br>CharArrayWriter<br>PipedWriter<br>StringWriter<br>**BufferedWriter**<br>OutputStreamWriter<br>**FileWriter**<br>FilterWriter<br>**PrintWriter** |

# Ficheros y flujos (*stream*)

- Algunas clases del paquete ***java.io*** relativas a flujos:
  - ***BufferedInputStream***. Permite leer los datos a través de un flujo con un buffer intermedio.
  - ***BufferedOutputStream***. Permite escribir en un flujo a través de un buffer intermedio.
  - ***FileInputStream***. Permite leer bytes de un fichero.
  - ***FileOutputStream***. Permite escribir bytes en un fichero.
  - ***StreamTokenizer***. Recibe un flujo de entrada, lo analiza (parse) y divide en trozos (tokens), permitiendo leer uno en cada momento (ver código de ejemplo 3.1).
  - ***StringReader***. Flujo de caracteres cuya fuente es una cadena de caracteres o *String*.
  - ***StringWriter***. Flujo de caracteres cuya salida es un buffer de cadena de caracteres que puede utilizarse para construir un *String*.
- Hay clases que se montan sobre otros flujos para modificar su comportamiento. Por ejemplo, podemos añadir un ***BufferedInputStream*** a un ***FileInputStream*** para mejorar la eficiencia en el acceso.

# La clase *File*

- ***File***: define una ruta hacia un fichero o directorio y permite consultar todo tipo de información del mismo.

```
    File file1 = new File("fichero.txt");
    System.out.println("Archivo: " + file1.getName() + (file1.isFile()
? " es un archivo" : " es una plantilla"));
    System.out.println("Archivo: " + file1.getName() +
(file1.isDirectory()?" es un directorio" : " no es un directorio"));
    System.out.println("Tamaño: " + file1.length());
    System.out.println("Vía de acceso: " + file1.getPath());
    System.out.println("Vía de acceso absoluta:\n\t" +
file1.getAbsolutePath());
    if (file1.exists()){
        boolean borrado=file1.delete();
        if (borrado)
            System.out.println("El fichero ha sido borrado
correctamente");
        else
            System.out.println("No se pudo borrar el fichero. ");
    }else
        System.out.println("El fichero no existe, y por tanto no pudo
ser borrado.");
```

# La clase *File*

- La construcción del objeto ***File*** no implica que exista el fichero o el directorio.
  - *boolean createNewFile();*
  - *boolean mkdir();*
- Si un objeto ***File*** hace referencia a un fichero que no existe y no es creado de forma explícita invocando a *createNewFile()*,se crea cuando se construya un objeto ***Writer*** u ***OutputStream***.

```
File f = new File(“datos”);
f.mkdir();
File fichero = new File(f,”info.txt”);
fichero.createNewFile();
```

# La clase *File*

- A través del objeto ***File*** se pueden examinar los atributos del archivo, cambiar su nombre, borrarlo, etc.
  - Renombrar el archivo con el método ***renameTo()***. El objeto *File* dejará de referirse al archivo ya que el *String* con el nombre del archivo no cambia en el objeto *File*.
  - Borrar el archivo con el método ***delete()***. También con ***deleteOnExit()*** que se borra cuando finaliza la ejecución.
  - Crear un nuevo fichero con un nombre único. El método estático ***createTempFile()*** crea un archivo temporal y devuelve un objeto File.
  - Establecer la fecha y la hora de modificación del archivo con ***setLastModified()***.
  - Crear un directorio con ***mkdir()***. También existe ***mkdirs()*** que crea los directorios superiores si no existen.
  - Listar el contenido de un directorio con los métodos ***list()*** y ***listFiles***. El primero devuelve un array de *String* con los nombres de los archivos y el segundo un array de objetos *File*.
  - Listar los nombres de archivo de la raíz del sistema con el método estático ***listRoots()***.

# Interfaz *FilenameFilter*

- Se emplea para establecer filtros relativos al nombre de los ficheros.
- Se debe implementar el método:
  - *boolean accept(File dir, String name)*

```
class Filtro implements FilenameFilter {
  String filtroAUsar;

  Filtro(String filtro) {
    filtroAUsar = filtro;
  }
  public boolean accept(File dir, String name) {
    File fichero=new File(name);
    String nombreFichero = fichero.getName();
    boolean debeIncluirse =(fichero.isFile() &&
(nombreFichero.indexOf(filtroAUsar) != -1));
    return  debeIncluirse;
  }
}
```

# La ruta de los ficheros

- Hemos de tener en cuenta que las rutas de ficheros en Windows y Linux tienen **separadores** diferentes.
  - Windows: “c:\datos.txt”.
  - Linux: “/home/usuario/datos.txt”.
- Se debe usar ***File.separator*** para adecuar las rutas a los ficheros sobre los que se trabaja en la aplicación.
- Adicionalmente, para ficheros Windows, las rutas se deben escribir con \\ para escapar la barra dentro del *String*.

# Lectura de ficheros de texto

- Dos clases: ***FileReader*** y ***BufferedReader***
- Construcción de un objeto ***FileReader***:
  - *FileReader(String path);*
  - *FileReader(File fichero);*
- La lectura se hace a través del método ***read()***. Los caracteres **se devuelven como tipo *byte*, debiendo ser convertidos a *String***.
- No se recomienda trabajar directamente con esta clase. Mejor usar *BufferedReader*.

# Lectura de ficheros de texto

- Construcción de un objeto ***BufferedReader***:
  - *BufferedReader(Reader entrada);*
- El método ***readLine()*** devuelve líneas del fichero.

```
File f = new File("datos");
if(f.exist()){
    BufferedReader bf = new BufferedReader(new FileReader(f));
    String cad;
    while((cad=bf.readLine())!=null){
        System.out.println(cad);
    }
} else {
    System.out.println("El fichero no existe");
```

- **EJERCICIO**: Crea el fichero ***datos.txt*** con un editor de textos. Haz un programa que cargue todos los datos del fichero en un String. Añade el código ***try-catch-finally*** necesario.

# Flujos de entrada y clase Scanner

- Hemos usado la clase *Scanner* para leer datos introducidos por teclado.
- La clase Scanner también se puede emplear para leer datos a partir de un fichero de texto.
- Constructores de *Scanner*:
  - *Scanner(File source);*
  - *Scanner(Readable source); //BufferedReader*
  - *Scanner(InputStream source);*
  - *Scanner(Path source);*
- **EJERCICIO**: Crea el fichero ***numeros.txt*** con distintos números por línea. Haz un programa que sume todos los números. Debes utilizar ***Scanner*** para leer los datos del fichero.

# Escritura en ficheros de texto

- Dos clases: ***FileWriter*** y ***PrintWriter***.
- A menos que sea necesario que los datos lleguen al flujo inmediatamente, es conveniente utilizar un ***BufferedWriter***.
- Creación de un objeto ***FileWriter***:
  - *FileWriter(String path, boolean append);*
  - *FileWriter(File fichero, boolean append);*
- Objeto ***PrintWriter***: la escritura sobre un fichero se realiza de la misma forma que en pantalla (métodos *print*, *println*, *printf*…).

# Escritura en ficheros de texto

- El objeto ***PrintWriter*** dispone de los métodos ***print***, ***println*** o ***printf*** para escribir en el fichero.
- Es muy similar a escribir en *System.out*, la salida estándar que es un objeto de tipo ***PrintStream***.
- Se recomienda montar el flujo de escritura sobre un buffer, en este caso ***BufferedWriter***.

```
String [] nombres = {“paco”, “roxana”, “joaquín”, “fran”}
PrintWriter salida = new PrintWriter(“datos.txt”);

//BufferedWriter buffer = new BufferedWriter(new FileWriter(“datos.txt”));
//PrintWriter salida = new PrintWriter(buffer);

for(String s: nombres){
    salida.println(s);
}
salida.flush();
salida.close();
```

# Lectura y escritura de datos primitivos

- Dos clases: ***DataInputStream*** y ***DataOutputStream***.
- La clase ***DataOutputStream*** proporciona métodos para escribir en cada uno de los tipos primitivos: ***writeXxx()***

```
int [] notas = {5, 6, 3, 10};
DataOutputStream ds = new DataOutputStream(new
FileOutputStream("datos.txt", false));
for(int i = 0;i<notas.length; i++){
    ds.writeInt(notas[i]);
}
ds.flush();
ds.close();
```

```
DataInputStream ds = new DataInputStream(new
FileInputStream("datos.txt));
try{
    while(true) System.out.println(ds.readInt());
} catch(EOFException e){}
```

# Introducción a ficheros binarios

- En ficheros binarios, los datos están formados por bytes. Se emplea para representar todos aquellos datos que no son texto: imágenes, vídeo, audio, archivos comprimidos, ejecutables, etc.
  - Dos clases: ***FileInputStream*** y ***FileOutputStream***.
- Los objetos en memoria de un POO pueden ser convertidos en secuencias de bytes para ser enviados por la red o para ser almacenados en disco. Este proceso se conoce como **serialización/deserialización**.
  - Dos clases: ***ObjectInputStream*** y ***ObjectOutputStream***.

# Escritura en ficheros binarios

- Dos clases: ***FileOutputStream*** y ***BufferedOutputStream***
- Construcción del objeto ***FileOutputStream***:
  - *FileOutputStream(File fichero, boolean append);*
  - *FileOutputStream(String path, boolean append);*
- La información convertida en bytes se escribe a través del método ***write()***.

```
byte [] data = “Cadena de texto a guardar”.getBytes();
BufferedOutputStream bf = new BufferedOutputStream(new
FileOutputStream(“datos.dat”));
bf.write(data);
bf.flush();
bf.close();
```

# Lectura en ficheros binarios

- Dos clases: ***FileInputStream*** y ***BufferedInputStream***.
  - Se debe conocer cuál es el formato de la información leída para realizar la conversión.
  - Ejemplo de código que lee el fichero binario guardado en la diapositiva anterior.

```
BufferedInputStream bf = new BufferedInputStream(new
FileInputStream("datos.dat"));
byte [] byteArray = new byte[50];
int tam;
while((tam=bf.read(byteArray))!=-1){
    System.out.println(new String(byteArray,0,tam));
}
bf.close();
```

# Acceso aleatorio

- Los ejemplos anteriores muestran lecturas de ficheros completos, de principio a fin (acceso secuencial).
- La clase ***RandomAccessFile*** proporciona acceso aleatorio.
  - *RandomAccessFile (String ruta, String modo)*
  - Modos: “r” (lectura) y “rw” (lectura-escritura).
- Implementa los interfaces ***DataInput*** y ***DataOutput***.
- Dispone de los métodos ***seek*** y ***skipBytes*** para posicionarse en el lugar del fichero desde donde se desea comenzar a leer.

```
RandomAccessFile in = new RandomAccessFile(“input.dat”, “r”);
```

# Serialización de objetos

- Para que un objeto pueda ser escrito en un flujo, la clase a la que pertenece debe **implementar el interfaz *java.io.Serializable***.
- La interfaz ***Serializable*** no contiene ningún método.
- Al serializar un objeto también se serializan todos los objetos que lo contengan siempre que sean también serializables.
- Los tipos primitivos son serializables, al igual que los arrays u otras clases del API como las colecciones.
- Las variables ***static*** y los atributos ***transient*** no se serializan.

```
public class Persona implements Serializable {
    //atributos y métodos de la clase.
}
```

# Serialización de objetos

- Dos clases: ***ObjectInputStream*** y ***ObjectOutputStream***.
- Lectura:
  - Tras el método ***readObject*** hay que realizar un casting explícito al tipo de objeto que se está leyendo. El método puede lanzar ***ClassNotFoundException***.
  - Los datos miembro no serializables (aquellos heredados de una clase no serializable) serán inicializados utilizando el constructor por defecto de su clase.
  - La deserialización consiste en la reconstrucción del objeto a partir de la información recibida.
- Se recomienda que toda clase serializable contenga el atributo ***serialVersionUID***:
  - *private static final long serialVersionUID=valor*.

# Escritura de objetos: serialización

- Algunos métodos en ***ObjectOutputStream***:
  - *void writeBoolean(Boolean n);*
  - *void writeChar(int n);*
  - *void writeInt(int n);*
  - *void writeObject(Object o);*
- Es posible **serializar un array o una colección** de objetos directamente, en lugar de serializar cada objeto uno a uno.

```
ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream("datos.obj"));
os.writeObject(new Persona());
os.flush();
os.close();
```

- **EJERCICIO**: Guarda en el fichero ***numeros.dat*** los valores de un array de 10 enteros. Crea dos versiones del programa: la primera escribirá los valores de uno en uno y la segunda guardará el array completo.

# Lectura de objetos: deserialización

- Algunos métodos en ***ObjectInputStream***:
  - *boolean readBoolean();*
  - *char readChar();*
  - *int readInt();*
  - *Object readObject();*
- Es posible **deserializar un array o una colección** de objetos directamente, en lugar de hacerlo uno a uno.
- Si leemos de un archivo serializado más objetos de los que realmente hay almacenados se lanzará la excepción ***EOFException***.

```
ObjectInputStream is = new ObjectInputStream(new
FileInputStream("datos.obj"));
Persona p = (Persona) is.readObject();
is.close();
```

- **EJERCICIO**: Escribe dos programas que lean los datos serializados en el ejercicio anterior.

# Unidad 8. Almacenando datos

FIN