Tema 6

6.1.-Tratamiento de errores.

- Lanzamiento de excepciones (throw).
- Tratamiento de excepciones. Try – Catch – finally
- Jerarquía de excepciones.
- Ventajas del tratamiento de excepciones.

6.2.-Estructuras de datos externas.

- Definición de archivo.
- Organización de archivos.
- Entrada y salida en Java.
- Serialización.

6.1.- Tratamiento de errores.

Cuando un programa Java viola las restricciones semánticas del lenguaje (**se produce un error**), la máquina virtual Java comunica este hecho al programa mediante una **excepción**.

Una excepción, puede ser provocada por diferentes tipos de errores, desde un desbordamiento de memoria o un disco duro estropeado hasta un disquete protegido contra escritura, un intento de dividir por cero o intentar acceder a un vector fuera de sus límites.

Cuando esto ocurre, la máquina virtual Java crea un objeto de la **clase Exception** o Error y se notifica el hecho al sistema de ejecución. Se dice que se ha lanzado una excepción ("*Throwing Exception*").

Un método es capaz de tratar una excepción ("*Catch Exception*") si ha previsto el error que se ha producido y prevé también las operaciones a realizar para "recuperar" el programa de ese estado de error.

En el momento en que es lanzada una excepción, la máquina virtual Java recorre la pila de llamadas de métodos, en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual Java muestra un mensaje de error y el programa termina.

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción **throw**, lo que facilita la devolución de un "código de error" al método que invocó el método que causó el error.

```
class Excepcion {
public static void main(String arg[ ]) {
int i=5, j=0;
int k=i/j; // División por cero
  }
}
```
ejem_1

Como primer encuentro con las excepciones, ejecuta el programa: ejem**_1**

Produce la siguiente salida al ser ejecutado:

java.lang.ArithmeticException: / by zero at Excepcion.main (Excepcion.java: 4)

Lo que ha ocurrido es que la máquina virtual Java ha detectado una condición de error y ha creado un objeto de la clase java.lang.ArithmeticException. Como el método donde se ha producido la excepción **no es capaz de tratarla,** se trata por la máquina virtual Java, que muestra el mensaje de error anterior y finaliza la ejecución del programa.

• Lanzamiento de excepciones (throw).

Un método también es capaz de lanzar excepciones. Por ejemplo:

```
class LanzaExcepcion {
 public static void main(String args[ ] ) throws
                                ArithmeticException {
   int i=1, j=2;
    if (i/j< 1)    throw new ArithmeticException();
         else     System.out.println(i/j);
   }
}
```

Este programa genera un error si el dividendo es menor que el divisor:

Genera el siguiente mensaje:
java.lang.ArithmeticException at LanzaExcepcion.main
(LanzaExcepcion.java:5)

En primer lugar, es necesario declarar todas las posibles excepciones que se pueden generar en el método, utilizando la cláusula **throws** de la declaración de métodos.

Para lanzar la excepción es necesario crear un objeto de tipo Exception o alguna de sus subclases (por ejemplo: ArithmeticException) .

En Java, se pueden tratar las excepciones previstas por el programador utilizando los manejadores de excepciones, que se estructuran en tres bloques:

- El bloque **try.**
- El bloque **catch.**
- El bloque **finally**

Un manejador de excepciones es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar.

El bloque try.

Lo primero que hay que hacer para que un método sea capaz de tratar una excepción generada por la máquina virtual Java o por el propio programa mediante una instrucción throw es encerrar las instrucciones susceptibles de generarla en un bloque try.

```
try {
Bloque_De_Intrucciones
 }
```

Cualquier excepción que se produzca dentro del bloque try será analizada por el bloque o bloques **catch**. En el momento en que se produzca la excepción, se abandona el bloque try y las instrucciones que sigan al punto donde se produjo la excepción no se ejecutaran**.**

Cada bloque try debe tener asociado por lo menos un bloque catch.

El bloque catch.

```
try {
        BloqueDeIntrucciones
    }catch(TipoExcepción  nombreVariable)  {
                BloqueCatch_1
         }
      catch (TipoExcepción  nombreVariable) {
                BloqueCatch
         }
```

Por cada bloque **try** pueden declararse uno o varios bloques **catch,** cada uno de ellos capaz de tratar un tipo u otro de excepción.

Para declarar el tipo de excepción que es capaz de tratar un bloque **catch**, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.

Cuando se intenta dividir por cero, la máquina virtual Java genera un objeto de la clase **ArithmeticException.**

```
class ExcepcionTratada {
public static void main(String argumentos[]) {
 int i=5, j=0;
   try {
      int k=i/j;
      System.out.println("Esto no se va a ejecutar.");
     } catch (ArithmeticException   ex) {
            System.out.println("Ha intentado dividir por cero");
            }
System.out.println("Fin del programa");
}
  }
```

Al producirse la excepción dentro de un bloque try, la ejecución del programa se pasa al primer bloque catch. Si la clase de la excepción se corresponde con la clase o alguna subclase de la clase declarada en el bloque catch, se ejecuta el bloque de instrucciones catch y a continuación se pasa el control del programa a la primera instrucción a partir de los bloques try-catch.

```
Salida:
Ha intentado dividir por cero
Fin del programa
```

También se podría haber utilizado en la declaración del bloque catch, una superclase de la clase ArithmeticException. Por ejemplo: **catch (RuntimeException ex) o catch (Exception ex).**

Sin embargo, es mejor utilizar excepciones lo más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error y si "se meten todas las condiciones en el mismo saco", seguramente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

El bloque finally. El bloque finally se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque try y en los bloques catch.

```
try {
  BloqueDeIntrucciones
    } catch (TipoExcepción nombreVariable) {
              BloqueCatch
         } catch (TipoExcepción nombreVariable) {
                    BloqueCatch
               } …
               } catch (TipoExcepción nombreVariable) {
                          BloqueCatch
                     }
finally {
     BloqueFinally
  }
```

- **Jerarquía de excepciones.**

Las excepciones son objetos pertenecientes a la clase Throwable o alguna de sus subclases. Dependiendo del lugar donde se produzcan existen dos tipos de excepciones:

1) Las excepciones **síncronas** no son lanzadas en un punto arbitrario del programa sino que, en cierta forma, son previsibles en determinados puntos del programa como resultado de evaluar ciertas expresiones o la invocación de determinadas instrucciones o métodos.

2) Las excepciones **asíncronas** pueden producirse en cualquier parte del programa y no son tan "previsibles". Pueden producirse excepciones asíncronas debido a dos razones:

   - La invocación del método stop() de la clase Thread que se está ejecutando.
   - Un error interno en la máquina virtual Java.

Dependiendo de si el compilador comprueba o no que se declare un manejador para tratar las excepciones, se pueden dividir en:

1) Las excepciones **comprobables** son repasadas por el compilador Java durante el proceso de compilación, de forma que si no existe un manejador que las trate, generará un mensaje de error.

2) Las excepciones **no comprobables** son la clase *RuntimeException* y sus subclases junto con la clase Error y sus subclases.

También pueden definirse por el programador subclases de las excepciones anteriores. Las más interesantes desde el punto de vista del programador son las subclases de la superclase **Exception** ya que éstas pueden ser comprobadas por el compilador.

- **Ventajas del tratamiento de excepciones.**

  Las ventajas del tratamiento de excepciones son varias:

  1. Separación del código "útil" del tratamiento de errores.
  2. Propagación de errores a través de la pila de métodos.
  3. Agrupación y diferenciación de errores.
  4. Claridad del código y obligación del tratamiento de errores.

1. Separación del código "útil" del tratamiento de errores.

Queremos realizar un bloque de instrucciones que realiza un procesamiento secuencial de un fichero:

Si se quieren tener en cuenta los posibles errores, debería hacerse algo parecido a esto:

```
{
abrir_fichero("prueba");
Mientras ! Fin_Fichero("prueba") {
  auxiliar = leer_registro("prueba");
  procesar(auxiliar);
  }
cerrar_fichero("prueba");
}
```

Por último faltaría tratar cada uno de los posibles errores, por ejemplo mediante una construcción del tipo switch. Observa que las instrucciones "útiles" son las que se encuentran en **negrita**.

En Java, el código sería algo parecido a esto:

En este caso se ha incluido la estructura equivalente a la que falta en el caso anterior (switch).

```
{
coderror = abrir_fichero("prueba");
if (coderror ==0) {
 Mientras( ! Fin_Fichero("prueba") y ( coderror==0 )) {

  auxiliar = leer_registro("prueba");
 if (auxiliar != ERROR) {
        coderror=procesar(auxiliar);
           if (coderror != 0)    coderr = -3;
    }
 else    coderror = -2;

 }
 if(cerrar_fichero("prueba")!=0)   coderror=-4;
  else    coderror = -1;
```

En Java, el código útil (en negrita) se encuentra agrupado. De esta forma se consigue una mayor claridad en el código que realmente interesa.

```
try {
   abrir_fichero("prueba");
   Mientras ! Fin_Fichero("prueba") {
      auxiliar = leer_registro("prueba");
      procesar(auxiliar);
      }
   cerrar_fichero("prueba");
   } catch (ExceptionAbrirFichero) {
            Hacer algo;
      } catch (ExceptionLeerRegistro) {
            Hacer algo;
        } catch (ExceptionProcesar) {
            Hacer algo;
          }catch (ExceptionCerrarFichero) {
              Hacer algo;
            }
```

2. Propagación de errores a través de la pila de métodos.

Supongamos que existen cuatro métodos y que el primero de ellos es el interesado en procesar una condición de error que se produce en el cuarto:

```
método1 ( ) {
int error;
error = método2 ( );
if ( error != 0 ) Procesar Error;
   else  Hacer algo;
}
```

```
int método2 ( ) {
int error;
error = método3 ( );
if ( error != 0 )return error;
     else  Hacer algo;
}
```

```
int método3 ( ) {
int error;
error = método4 ( );
if ( error != 0 ) return error;
else  Hacer algo;
}
```

```
int método4 ( ) {
int error;
error = Proceso que puede generar
un error ( );
if ( error != 0 )  return error;
     else  Hacer algo;
  }
```

Todos y cada uno de los métodos deben tener en cuenta el posible error y adaptar su código para tratarlo, así como devolver el código de error al método que lo invocó.

En Java:

Cada uno de los métodos únicamente declara mediante la cláusula throws el error /errores que puede generar.

Éstos se propagan automáticamente a través de la pila de llamadas sucesivas de métodos. El método interesado en tratar el error es el único que debe encargarse de "tratarlo". Los demás métodos únicamente deben "ser conscientes" del error que puede producirse al llamar a otro método.

```
try {
    abrir_fichero("prueba");
    Mientras ! Fin_Fichero("prueba") {
        auxiliar = leer_registro("prueba");
        procesar(auxiliar);
        }
    cerrar_fichero("prueba");
    } catch (ExceptionAbrirFichero) {
                Hacer algo;
        } catch (ExceptionLeerRegistro) {
                    Hacer algo;
            } catch (ExceptionProcesar) {
                        Hacer algo;
                }catch (ExceptionCerrarFichero) {
                            Hacer algo;
                    }
```

Si además el error se declara como una subclase de Exception, el compilador se encarga de mostrar un mensaje de error en caso de que el método no declare la cláusula **throws** adecuada (o trate la excepción), lo cual facilita la tarea del programador.

3. Agrupación y diferenciación de errores.

Si se ha declarado una jerarquía de excepciones como la siguiente:

```
class ExceptionGrupo1 extends Exception { }
class ExceptionGrupo2 extends Exception { }
class Exception1_1 extends ExceptionGrupo1 { }
class Exception1_2 extends ExceptionGrupo1 { }
class Exception1_3 extends ExceptionGrupo1 { }
class Exception2_1 extends ExceptionGrupo2 { }
class Exception2_2 extends ExceptionGrupo2 { }
class Exception2_3 extends ExceptionGrupo2 { }
```

Las excepciones pueden ser tratadas particularmente:

En este caso se detectarían las excepciones Exception1_1, Exception1_2 y Exception1_3 además de las excepciones de la clase ExceptionGrupo1.

También podrían detectarse todos los tipos de excepciones mediante: catch(Exception exc).

```
try {
    abrir_fichero("prueba");
    Mientras ! Fin_Fichero("prueba") {
        auxiliar = leer_registro("prueba");
        procesar(auxiliar);
        }
    cerrar_fichero("prueba");
    } catch (ExceptionAbrirFichero) {
            Hacer algo;
        } catch (ExceptionLeerRegistro) {
```

4. **Claridad del código y obligación del tratamiento de errores.** Como se ha visto en los puntos anteriores, el código resulta más claro ya que:

- Se separa el código "útil" del de tratamiento de errores.
- Se acerca el código de tratamiento de errores al método que realmente es el interesado en tratarlos.
- Cada tipo de error se trata en un bloque catch diferenciado.

Además, se obliga al programador a tratar los errores que puede generar un método, suponiendo que las excepciones declaradas correspondientes a los distintos tipos de error sean subclases de la clase Exception, que es comprobable por el compilador.

Quien programa un método declara las excepciones que puede generar mediante la cláusula throws**.** Si otro programador (o el mismo) desea utilizar ese método deberá utilizar una estructura try-catch para tratarlo o en caso contrario lanzarlo mediante la cláusula throws para que sea el método que invoque al mismo, quien lo trate. En cualquier caso, el programador que desea utilizar un método es consciente, porque el compilador se lo señala, de los tipos de errores (excepciones) que puede generar.

```
Try {
                    …
    } catch (Exception1_2 ex){
                    …
    }
O pueden tratarse como grupos:
    try {
                …
    } catch (ExceptionGrupo1) {
                    …
    }
```

6.2.-Estructuras de datos externa.

- Conceptos básicos

Archivo. Un archivo es un conjunto de datos estructurados en una colección de entidades elementales básicas denominadas registros que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas campos

Clave (indicativo). Una clave o indicativo es un campo de datos que identifica unívocamente al registro y lo diferencia de otros registros. Esta clave debe ser diferente para cada registro. Claves típicas son nombres o números de identificación.

Registro físico o bloque.Un registro físico o bloque es la cantidad de información que pueden transferirse en una operación de entrada/salida entre la memoria central y los dispositivos periféricos o viceversa.

Un bloque puede contener uno o más registros lógicos.

Factor de bloqueo. Otra característica que es importante en relación con los archivos es el concepto de factor de bloqueo o blocaje. El número de registros lógicos que puede contener un registro físico se denomina factor de bloqueo. Un registro lógico puede ocupar menos de un registro físico, exactamente un registro físico o más de un registro físico.

- **Organización de archivos**

Según las características del soporte empleado y el modo en que se han organizado los registros, se consideran dos tipos de acceso a los registros de un archivo:

El acceso secuencial implica el acceso a un archivo según el orden de almacenamiento de sus registros.

El acceso directo implica el acceso a un registro determinado, sin que ello implique la consulta de los registros anteriores. Este tipo de acceso sólo es posible en soportes direccionables..

En general, se consideran tres organizaciones fundamentales: Secuencial , Directa o aleatoria y Secuencial indexada.

Organización secuencial. Es una secuencia de registros almacenados consecutivamente sobre el soporte externo, de tal modo que para acceder a un registro determinado es obligatorio pasar por todos los registros que le preceden.

Todos los tipos de dispositivos de memoria auxiliar soportan la organización secuencial. Los archivos organizados secuencialmente contienen un registro particular, el último, que contiene una marca de fin de archivo, detectable mediante la función de EOF FDA/ FDF .

Organización directa. Un archivo está organizado en modo directo cuando el orden físico no se corresponde con el orden lógico. Los datos se sitúan en el archivo y se accede a ellos directamente mediante su posición, es decir, el lugar relativo que ocupan.

Esta organización tiene la ventaja de que se pueden leer y escribir registros en cualquier orden y posición, y el acceso a la información que contienen es más rápido. La organización directa tiene el inconveniente de que necesita programar la relación existente entre el contenido de un registro y la posición que ocupa. El acceso a los registros en modo directo implica la posible existencia de huecos libres dentro del soporte y, por consecuencia, pueden existir huecos libres entre registros.

La correspondencia entre clave y dirección debe poder ser programada y la determinación de la relación entre el registro y su posición física se obtiene mediante una fórmula. Las condiciones para que un archivo sea de organización directa son:

- Almacenado en un soporte direccionable.
- Los registros deben contener un campo específico denominado clave que identifica cada registro de modo único, es decir, dos registros distintos no pueden tener un mismo valor de clave.
- El programador creará una relación perfectamente definida entre la clave indicativa de cada registro y su posición física dentro del dispositivo de almacenamiento.

Organización secuencial indexada. La organización de un archivo de registros clasificados de acuerdo con los valores de un campo clave, se conoce como archivo indexado.

Para facilitar la búsqueda se utiliza una tabla de índices, que a su vez estará grabada en un segundo archivo, llamado archivo de índices.

El archivo de índices consta de pares (clave, dirección física del registro que tiene esa clave). La tabla se mantiene ordenada respecto al campo clave con lo que la búsqueda es más eficiente al aplicar el algoritmo de búsqueda binaria.

- **Operaciones con archivos.** Tras decidir el tipo de organización del archivo y los métodos de acceso que se van a aplicar para su manipulación, definimos las posibles operaciones a realizar con los archivos y sus registros.

1.- Crear un objeto File, **descriptor** del archivo.

2.- Crear un objeto FileOutputStream que representa el **flujo de salida** asociado al descriptor que se ha creado.

3.- Crear un objeto PrintStream, cuyo constructor recibe el nombre del flujo de salida creado.

4.- Operar sobre el objeto PrintStream mediante alguno de los métodos proporcionados por esta clase.

Entrada y Salida en JAVA

En Java existe una clase para representar la funcionalidad de un fichero denominada **File.** El constructor crea una estructura de datos asociada al nombre del fichero que recibe como argumento. A esta estructura se le denomina *descriptor del fichero.*

Ejem.

```
File df1 = new File("fichero1");
File df2;
df2=new File("Fichero2") ;
```

Cuando se crea un objeto de esta manera, no se crea el fichero real al que se referencia, sino que simplemente se establece el **descriptor** asociado.

Si el fichero real existiera, después de ejecutar el constructor, es posible borrar utilizando el método **delete( )** o renombrar dicho fichero utilizando el método **renameTo(File)** .

Ejem. Suponemos que el fichero1 ya existía, pero fichero2 no.

```
df1.renameTo(fichero2) ;          // df1 es el descriptor de un fichero de nombre "fichero2 "
```

- **Salida de datos a un fichero.**

Java proporciona una clase predefinida que representa un **flujo de salida** que se dirige hacia un fichero, es la clase **FileOutputStream**. Su constructor recibe como argumento un objeto de tipo File.

```
File df = new File ("datos.dat");
FileOutputStream flujofich = new FileOutputStream(df);
// Ahora el fichero datos.dat existe, está vacío y abierto para escribir.
```

El flujo **flujofich** no tiene estructura ni organización alguna, es simplemente una secuencia de bytes.

La clase **FileOutputStream** es realmente una extensión de una clase genérica de Java que modela los flujos de salida en bytes, denominada *OutputStream*. Otra clase que también hereda de esta es la **FilterOutputStream**, que nos proporciona un conjunto de métodos para estructurar y organizar la información de un flujo. Esta clase la podemos considerar como un "filtro" que nos permite ver la información de forma estructurada.

Sin embargo, los métodos de esta clase permiten poco más que escribir arrays de bytes.

Finalmente la clase **FilterOutputStream** es heredada por otra que sí proporciona un conjunto de métodos que permiten estructurar la información en secuencias de caracteres imprimibles, es la clase **PrintStream.**

El constructor de **PrintStream** recibe como argumento el nombre de un flujo de salida, que será el nombre del flujo asociado al fichero.

Completamos el ejemplo anterior:

File **df** = new File ("datos.dat");

FileOutputStream **flujofich** = new FileOutputStream(**df**);

PrintStream **salida** = new PrintStream (**flujofich**) ;

salida.print ("Esto se escribe en el archivo datos.dat");

Este proceso necesario para poder declarar y escribir en un archivo nuevo se puede resumir, creando directamente todos los objetos, y obteniendo sólo la referencia al objeto PrintStream.

PrintStream salida = new PrintStream (new FileOutputStream (new File ("datos.dat")));

El objeto **salida** representa el flujo estructurado para ser imprimible.

salida.print ("Esto se escribe en el archivo datos.dat");

Salida estándar. Un caso especial del objeto PrintStream es el System.out. **System** es una clase predefinida en Java que posee entre otros, dos atributos públicos (static) denominados **in** y **out** que representan los flujos de entrada y salida estándar por defecto.

- Entrada desde fichero.

Igual que en la escritura, la entrada desde un fichero exige en Java realizar los siguientes pasos:

1. Crear un objeto **File**, descriptor del archivo.
2. Crear un objeto **FileInputStream** que representa el flujo de entrada asociado al descriptor que se ha creado.
3. Construir un objeto **InputStreamReader**, cuyo constructor recibe el nombre del flujo de entrada creado.
4. Operar sobre el objeto **BufferedReader** mediante alguno de los métodos proporcionados por esta clase.

Ejemplo: Genera un fichero de texto

```
import java.io.*;
public class GeneraFicheroTexto  {
 public static void main(String args[])  {
 /* BufferedReader teclado= new
BufferedReader( new InputStreamReader(System.in));
*/
Scanner teclado = new Scanner(System.in);
String nombre;
```

```
import java.io.*;
public class LeeFicheroTexto  {
 public static void main(String args[])  {
 /* BufferedReader teclado= new
BufferedReader( new
InputStreamReader(System.in));   */
Scanner teclado = new Scanner(System.in);
String texto='';
 try{
```

```
 try{
FileWriter flS = new FileWriter("Agenda.txt");
  BufferedWriter fS=new BufferedWriter ( flS );
   do{
     System.out.println("Introduce un nombre");
     nombre = teclado.nextLine();
    if(nombre.length() > 0) {
         System.out.println("Telefono");
          String telefono = teclado.next();
          fS.write(nombre+","+telefono);
          fS.newLine();            }
   }  while(nombre.length() > 0);
  fS.close();
  }  catch(IOException e){
       System.out.println("Error en el fichero"); }
 }
} // cierra clase
```

```
  FileReader flE = new FileReader("Agenda.txt");
  BufferedReader fE=new BufferedReader ( flE );
  while(texto != null){
    texto = fE.readLine();
   if(texto != null){
    int posi=texto.indexOf(",");
    String nombre=texto.substring(0,posi);
    String telefono=texto.substring(posi+1);
    System.out.print("Nombre: "+nombre);
    System.out.println("Telefono: "+telefono);   }
 }
 fE.close();
  }catch(IOException e){
   System.out.println("Error en el fichero");     }
 }
} // cierra la clase
```

```
import java.io.*;
import java.util.*;

public class GeneraFicheroTexto   {
 public static void main(String args[])   {
    Scanner teclado = new Scanner(System.in);
     String nombre , telefono;
 try{
   // definimos el archivo (flujo de salida)
  FileWriter flS = new FileWriter("Agenda.txt", true);
  BufferedWriter fS=new BufferedWriter ( flS );
 System.out.println("Introduce un nombre");
   nombre = teclado.next();
   while(!nombre.equalsIgnoreCase("F")){
      System.out.println("Telefono");
      telefono = teclado.next();
      fS.write(nombre+","+telefono);
      fS.newLine();
      System.out.println("Introduce un nombre");
      nombre = teclado.next();
    }
   fS.close();
   }  catch(IOException e){
     System.out.println("Error en el fichero");
    }
 }
} // cierra clase
```

```
import java.io.*;
import java.util.*;

class LeeFicheroTexto   {
 public static void main(String args[])   {
   Scanner teclado = new Scanner(System.in);
   String texto="";
   try{
    FileReader flE = new FileReader("Agenda.txt");
    BufferedReader fE=new BufferedReader ( flE );
   texto = fE.readLine();
   while(texto != null){
    int posi=texto.indexOf(",");
    String nombre=texto.substring(0,posi);
    String telefono=texto.substring(posi+1);
    System.out.print("Nombre: "+nombre+"\t");
    System.out.println("Telefono: "+telefono);
    texto = fE.readLine();
    }
    fE.close();
   }catch(IOException e){
     System.out.println("Error en el fichero");
   }
 }
} // cierra la clase
```

Ficheros.

Antes de empezar a ver cómo utilizar las clases que crean los *Streams* de entrada o salida sobre ficheros es necesario conocer algunas clases que permiten acceder a información referente a los mismos y que no intervienen en los flujos de datos hacia o desde ellos.

Para ello existen las clases **File y FileDescriptor:**

La clase File.

La clase **File** sirve para representar ficheros o directorios en el sistema de ficheros de la plataforma específica. Mediante esta clase pueden abstraerse las particularidades de cada sistema de ficheros y proporcionar los métodos necesarios para obtener información sobre los mismos.

public class java.io.File implements Serializable {

// Atributos

```
public static final String pathSeparator;
// separador de paths
```

// Constructores

```
public File(String pathYNombre);
public File(String path, String nombre);
public File(File path, String nombre);
```

// Métodos

```
public boolean canRead();            //true si se puede leer
public boolean canWrite();           //true si se puede escribir
public boolean delete();             //borrar el fichero
public boolean exists();             // true si existe el fichero.
public String getName();             //nombre del fich. o direct.
public String getParent();           //el path del direct. padre
public String getPath();             // el path del fichero
public int hashCode();               // devuelve un código hash para el fichero
public boolean isDirectory();        // true si es un directorio y no un fichero
public boolean isFile();             // true si es un fichero
public long length();                // El tamaño en bytes del fich.
public String[] list();              // Devuelve los ficheros de un directorio
public boolean mkdir();              // Crea el directorio indicado
```

```
import java.io.*;
class DatosDeFile {
   public static void main(String arg[]) {
  File f = new File("sub      \\prueba.txt");
  System.out.println("pathSeparator: "+File.pathSeparator);
  System.out.println("pathSeparatorChar:”+ File.pathSeparatorChar);
  System.out.println("separator: "+File.separator);
  System.out.println("separatorChar: "+File.separatorChar);
  try {
     System.out.println("canRead():"+f.canRead());
     System.out.println("canWrite():"+f.canWrite());
     System.out.println("exists():"+f.exists());
     System.out.println("getName():"+f.getName());
     System.out.println("getParent():"+f.getParent());
     System.out.println("getPath():"+f.getPath());
     System.out.println("hashCode():"+f.hashCode());
     System.out.println("isAbsolute():"+f.isAbsolute());
     System.out.println("isDirectory():"+f.isDirectory());
     System.out.println("isFile():"+f.isFile());
     System.out.println("lastModified():"+f.lastModified());
     System.out.println("length():"+f.length());
   } catch (IOException e) {
      System.out.println(e); }
          }
}
```

Salida por pantalla

```
pathSeparator: ;
pathSeparatorChar: ;
separator: \
separatorChar: \
canRead():true
canWrite():true
canRead():true
exists():true
 getName():prueba.txt
getParent():sub
getPath():sub\prueba.txt
hashCode():-2131944982
isAbsolute():false
isDirectory():false
isFile():true
lastModified():868560320000
length():29
```

La clase FileDescriptor.

public* class *java.io.FileDescriptor {

// Atributos

public static final FileDescriptor **in**; // FD de la entrada estándar

public static final FileDescriptor **out**; // FD de la salida estándar

public static final FileDescriptor **err**; // FD de la salida de error

public FileDescriptor(); **// Constructor**

// Métodos

public native void sync() throws SyncFailedException;

// Este método no termina hasta que se vacíen todos los buffers intermedios.

public native boolean valid(); // true si el FD es válido

}

Esta clase no debería utilizarse para instanciar objetos de la misma, sino que se debe utilizar algún método que devuelve un objeto de la clase **FileDescriptor.**

Acceso a ficheros secuenciales.

Para acceder a datos almacenados en ficheros sin demasiadas pretensiones de formateo, y con acceso secuencial, puede utilizarse la clase **FileInputStream** que implementa los métodos de la **clase InputStream** sin mayores funcionalidades.

Para almacenar datos en un fichero, puede crearse un S*tream* de salida mediante la clase más simple que implementa los métodos de la clase **OutputStream:**

Los constructores respectivos son:

FileInputStream	FileOutputStream
FileInputStream(String nombre);	FileOutputStream(String nombre);
FileInputStream(File fichero);	FileOutputStream(File fichero);
FileInputStream(FileDescriptor fd);	FileOutputStream(FileDescriptor fd);
	FileOutputStream(FileDescriptor fd, **boolean añadir**);

Pueden crearse S*treams* de entrada o salida sobre ficheros utilizando como parámetro en el constructor:

- Un **String:** con el nombre del fichero junto con el path.
- Un objeto de la clase **File:** con el fichero a utilizar.
- Un objeto de la clase **FileDescriptor:** con el descriptor de fichero.

Métodos de lectura y escritura de bytes:

FileOutputStream	FileInputStream
public native void write(int b);l	public native int read();
public void write(byte b[])	public int read(byte b[])
public void write(byte b[], int d, int l);	public int read(byte b[], int d, int l);

El puntero del fichero avanza según el número de bytes leídos en cada operación **read().**

En ficheros de entrada, puede utilizarse el método **public native long skip(long n)** para saltar **n** bytes adelante.

Ambas clases permiten obtener el Descriptor de Fichero (FD) asociado mediante el método **public final FileDescriptor getFD(),** y cerrarlo mediante public **native void close().**

El siguiente ejemplo toma un fichero de entrada y convierte las mayúsculas a minúsculas (siempre que esté en código ascii y únicamente los códigos del 65 al 90):

Si se pretende tratar con S*treams* de caracteres en lugar de bytes, pueden utilizarse las nuevas clases equivalentes a FileInputStream y FileOutputStream: FileReader y FileWriter respectivamente, cuyos constructores son:

Fichero de entrada	Fichero de salida
FileReader (String nombre);	FileWriter(String nombre);
FileReader(File fichero);	FileWriter(File fichero);
FileReader(FileDescriptor fd);	FileWriter(FileDescriptor fd);
	FileWriter(FileDescriptor fd, boolean añadir);

Los métodos para leer y escribir caracteres son heredados de las superclases InputStreamReader y OutputStreamWriter y son:

FileReader	FileWriter
public int read();	public void write(char b[]);
public int read(char b[])	public void write(int b);
public int read(char b[], int d, int l);	public void write(String s);
	public void write(char b[], int d, int l);
	public void write(String s, int d, int l);

En este caso, se ha utilizado la clase **BufferedReader** envolviendo a la clase InputStreamReader porque añade el método readLine(), muy conveniente para los propósitos del programa. También se utiliza la **clase PrintWriter** envolviendo a la **clase FileWriter** para utilizar un método de escritura de línea completa como **println().**

El siguiente ejemplo lee líneas de texto escritas desde el teclado y va escribiéndolas en el fichero de salida “prueba.salida” hasta que se encuentra la marca de final de fichero ^Z.

```
import java.io.*;
import java.util.*;
class Fichero3 {
public static void main(String[] args) {
Scanner teclado=new Scanner(System.in);
InputStreamReader inStream=new InputStreamReader(System.in);
File fSalida = new File("prueba.salida");
try {
    String s;
    FileWriter fwSalida = new FileWriter(fSalida);
    PrintWriter w = new PrintWriter(fwSalida);
    while ((s = teclado.readLine())!=null ) {
```

```
                    w.println(s);
            }
            fwSalida.close();
      inStream.close();
   } catch (FileNotFoundException e) {
       System.err.println("Fichero no encontrado");
}
    catch (IOException e) {
          System.err.println("Error de E/S");   }
  }
}
```

La clase RandomAccessFile.

public class java.io.RandomAccessFile implements DataOutput, DataInput {

// Constructores

public RandomAccessFile(File fichero, String modo) throws IOException;

public RandomAccessFile(String nomF, String modo) throws IOException

// El modo del fichero puede ser "r" (lectura) o "rw" (lectura/escritura).

// Métodos

public native void **close()** throws IOException; // Cierra el fichero y recursos

public final FileDescriptor **getFD()** throws IOException; // Devuelve el FD

public native long **getFilePointer()** throws IOException; // Devuelve la posición del puntero del fichero (distancia en bytes desde el principio)

public native void **seek**(long posición) throws IOException; // posiciona el puntero del fichero directamente (en bytes)

public int skipBytes(int n) throws IOException // Salta n bytes

public native void write(int b) throws IOException; // Escribe un byte

public void write(byte b[]) throws IOException;

public void write(byte b[], int desplaza, int longitud) throws IOException;

}

Esta clase sirve para crear un objeto de acceso a un fichero de acceso directo a partir de un objeto de la **clase File** o, directamente, a partir de un **String** que contenga el nombre del fichero. Los constructores aceptan un segundo parámetro que indica el modo de apertura, pudiendo ser uno de los dos siguientes valores:

"r" Apertura en modo lectura.

"rw" Apertura en modo lectura/escritura.

```
fichero.seek(Integer.parseInt(posición));
punteroF=fichero.getFilePointer();                    // obtener el puntero del fichero
pantalla.println("Valor del byte = "+fichero.readByte());// leer y mostrar el byte en la posición
pantalla.println("Nuevo valor del byte:");
b=teclado.readLine();
/* posicionar el puntero del fichero en el lugar guardado anteriormente*/
fichero.seek(punteroF);
fichero.writeByte(Integer.parseInt(b));          // escribir el nuevo valor del byte
} catch(IOException e) {
     System.err.println("No existe ese byte.");
}
 catch(NumberFormatException e) {
       System.err.println("error num.");
 }
pantalla.println("byte a examinar:");
}
   teclado.close();
   pantalla.close();
   fichero.close();
} catch (IOException e) {
       System.err.println("Error de E/S");  }
}
}
```

Esta clase tiene dos métodos para mover el puntero del fichero:

seek(long posición) posicionamiento absoluto en el byte indicado por el parámetro.

skipBytes(int n) posicionamiento relativo saltando n bytes.

getFilePointer() devuelve la posición actual (puntero del fichero).

Acceso a ficheros aleatorios.

Los ficheros de acceso aleatorio o directo permiten situar el puntero del fichero en una posición determinada sin tener que recorrer previamente el contenido del fichero como ocurre con los ficheros de acceso secuencial vistos anteriormente.

Para trabajar con ficheros aleatorios, únicamente es necesaria la clase **RandomAccessFile,** que proporciona directamente (no heredados) todos los métodos para leer y escribir datos del fichero (los indicados en las interfaces DataInput y DatoOutput).

El siguiente programa solicita que le sea proporcionado el nombre de un fichero. Después pregunta el byte a modificar, muestra su valor y solicita el nuevo valor. Este proceso se repite hasta pulsar ^Z (final de fichero).

A la hora de modificar un byte en el fichero de acceso directo, hay que tener en cuenta que primero se lee el byte mediante **readByte().** Este método, además de realizar la lectura, avanza el

puntero de fichero un byte adelante, por lo que antes de realizar la escritura con **writeByte(),** habrá que retrocederlo 1 byte.

System.in	InputStreamReader	**RandomAccessFile**	readByte()
InputStream	BufferedReader	**Datos** readLine()	writeByte()
OutputStream	PrintWriter	**System.out**	

También puede realizarse este proceso de la forma en que se ha hecho en el ejemplo; guardando la posición del puntero del fichero mediante **getFilePointer()** antes de realizar la lectura y restaurándolo mediante **seek()** antes de escribir.

- **Serialización de objetos.**

De todos es conocida la importancia de poder salvar en memoria secundaria y permanente determinados estados o valores de una aplicación en un momento determinado para ser restablecidos en algún momento posterior.

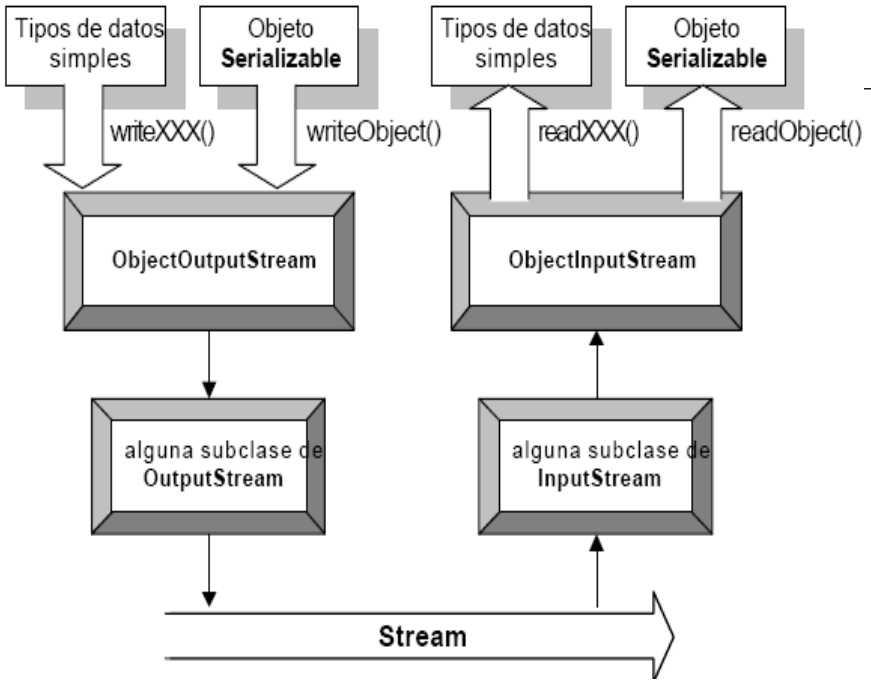
Si un programador desea guardar permanentemente el estado de un objeto puede utilizar las clases vistas hasta el momento para ir almacenando todos los valores de los atributos como valores char, int, byte, etc. Pero esto puede ser muy molesto y complicado, ya que un objeto puede tener atributos que sean otros objetos, que a su vez tengan atributos que hagan referencia a otras clases y así sucesivamente.

Pero existe una forma más cómoda de enviar objetos a través de un *Stream* como una secuencia de bytes para ser almacenados en disco, y también para reconstruir objetos a partir de S*treams* de entrada de bytes. Esto puede conseguirse mediante la "**serialización**" de objetos.

La **serialización** consiste en la transformación de un objeto Java en una secuencia de bytes para ser enviados a un Stream. Mediante este mecanismo pueden almacenarse objetos en ficheros o incluso en bases de datos como BLOBs (*Binary Large OBject*), o se pueden enviar a través de *sockets,* en aplicaciones cliente/servidor de un equipo a otro, por ejemplo. Pero todo esto no es nuevo, ya podía hacerse antes de la serialización de objetos, lo que ocurre es que con este instrumento se consigue de una forma mucho más cómoda y sencilla.

Para enviar y recibir objetos serializados a través de un *Stream* se utilizan las clases java.io.ObjectOutputStream para la salida y java.io.ObjectInputStream para la entrada.

La clase **ObjectOutputStream** necesita como parámetro en su constructor un objeto de la clase OutputStream (alguna de sus subclases, ya que OutputStream es abstracta):



Esquema de funcionamiento de la serialización.

public ObjectOutputStream(OutputStream out) throws IOException;

Si, por ejemplo, se utiliza como parámetro un objeto de la clase FileOutputStream, los objetos serializados a través de este *Stream* serán dirigidos a un fichero.

La clase ObjectInputStream necesita como parámetro en su constructor un objeto de la clase InputStream (alguna de sus subclases, ya que IutputStream es abstracta):

public ObjectInputStream(InputStream in) **throws IOException,StreamCorruptedException;**

Si, por ejemplo, se utiliza como parámetro un objeto de la clase **FileInputStream,** los objetos se deserializan después de obtenerlos a través de este *Stream* que, a su vez, obtiene los bytes de un fichero.

Objetos serializables.

Sólo los objetos de clases que implementen la interface **java.io.Serializable** o **java.io.Externalizable** o aquellos que pertenezcan a subclases de clases serializables pueden ser serializados.

En este capítulo únicamente van a estudiarse los objetos serializables mediante la interface Serializable, que es más sencilla porque automatiza el proceso. Los objetos que implementan la interface Externalizable deben encargarse de crear el formato en el que se almacenarán los bytes.

La interface **Serializable** no posee ningún método. Sólo sirve para "marcar" las clases que pueden ser serializadas. Cuando un objeto es serializado, también lo son todos los objetos alcanzables

```
public class Persona implements java.io.Serializable  {
/* Esta clase debe ser Serializable para poder ser escrita en un stream de objetos */
   private String nombre;
   private int edad;
public Persona(String s, int i) {
      nombre=s;
      edad=i;  }
public String toString() {
     return nombre+":"+edad;
   }
}
```

desde éste (los atributos que son objetos), ignorándose todos los atributos static , transient y los no serializables.

No se almacenan los valores de los atributos static porque éstos pertenecen a la clase (no al objeto), y son compartidos por todos los objetos implementados a partir de ésta. Tampoco se almacenan los atributos transient porque esta palabra reservada se utiliza precisamente para eso, para no formar parte de las características permanentes de los objetos.

Como puede comprobarse en el ejemplo anterior, en el fichero Persona.java, se ha declarado la clase Persona de forma que implemente la interface Serializable para poder ser serializada.

Escritura.

Para serializar objetos (también vectores y Strings) y escribirlos a través del stream de salida se llama al método writeObject() del objeto ObjectOutputStream creado:

public final void writeObject(Object objeto)throws IOException**;**

También pueden escribirse, además de objetos, valores de tipos de datos simples mediante cualquiera de los métodos de la interface DataOutput, que implementa la clase ObjectOutput Stream:

write(int b); writeChars(String x);	writeBoolean(boolean x); writeInt(int x);
write(byte b[]); writeDouble(double x);	writeByte(int x); writeLong(long x);
write(byte b[], int desplaza, int longitud);	writeBytes(String x); writeShort(int x);
writeFloat(float x);	writeChar(int x); writeUTF(String x);

Todos estos métodos generan excepciones de la clase IOException.

En realidad, la clase ObjectOutputStream no implementa la interface DataOutput, sino la interface ObjectOutput, que es una subinterface de DataOutput, por lo que hereda sus métodos.

Veamos un ejemplo en el que se crea un *stream* de salida, en el cual se escribirán:

un objeto de la clase Persona (visto en el punto anterior), un objeto de la clase fecha (Date) y un entero:

En el caso de los objetos persona y fecha se utiliza el método writeObject(), y pueden ser serializados porque tanto la clase Persona como la clase Date implementan la interface Serializable; y en el caso del literal entero 13, se utiliza el método writeInt().

Lectura.

Para deserializar objetos después de leerlos a través del stream de entrada se llama al método readObject() del objeto ObjectInputStream creado:

public final Object readObject() throws OptionalDataException,
ClassNotFoundException, IOException;

También pueden leerse, además de objetos, valores de tipos de datos simples mediante cualquiera de los métodos de la interface DataInput, que implementa la clase ObjectInputStream:

Boolean readBoolean(); int readInt();	void readFully(byte b[]); int readUnsignedShort();
byte readByte(); String readLine();	String readUTF();
char readChar(); long readLong();	void readFully(byte b[], int desplaza,int longitud);
double readDouble(); short readShort();	int skipBytes(int n)
float readFloat(); int readUnsignedByte();	

Todos estos métodos generan excepciones de la clase IOException y EOFException. Para leer objetos y valores de tipos de datos simples de un stream, evidentemente, deben ser recuperados en el mismo orden en que se introducen. Si se introducen objetos de las clases C1, C2, C3, C2 , … deben ser recuperados en ese mismo orden C1, C2, C3, C2, …

El siguiente ejemplo lee y muestra los objetos y el entero guardado anteriormente en el fichero prueba.dat:

En realidad, la clase ObjectInputStream no implementa la interface DataInput, sino la interface ObjectInput, que es una subinterface de DataInput, por lo que hereda sus métodos.

En los casos de lectura de objetos, es necesario realizar una conversión de tipo referencial, ya que el método readObject() devuelve un objeto de la clase Object.

Para ello hay que anteponer el nombre de la clase entre paréntesis a la expresión a convertir. La salida del programa según los datos almacenados en el ejemplo del punto anterior sería la siguiente:

Victor:30
Tue Jul 15 18:10:00 GMT+01:00 1997

Personalización en la serialización.

Una clase puede controlar por sí misma cómo será serializada, alterando qué datos serán almacenados o leídos, definiendo los métodos writeObject() y readObject() con, exactamente, las siguientes signaturas:

private void readObject(java.io.ObjectInputStream in)throws IOException, ClassNotFoundException;

private void writeObject(java.io.ObjectOutputStream out) throws IOException

Estos métodos declaran los parámetros ObjectInputStream y ObjectOutputStream, por lo que puede accederse a sus métodos. Entre ellos se encuentran:

public final void defaultWriteObject() throws IOException;

Este método escribe la información del objeto por defecto: todos sus atributos que no sean static o transient o no sean serializables.

Puede utilizarse para realizar la escritura por defecto y, después, agregar al *stream* información adicional sobre el objeto.

public final void defaultReadObject() throws IOException, ClassNotFoundException, NotActiveException;

Este método lee la información del objeto por defecto: todos sus atributos que no sean static o transient o no sean serializables. Puede utilizarse para realizar la lectura por defecto y, después, leer del *stream* información adicional sobre el objeto.

Véase el siguiente ejemplo que modifica la clase Persona de forma que serialice, además de los atributos propios de Persona2 (nombre y edad), el momento(Date) en que se escribe:

Los programas para la escritura y para la lectura no necesitan de ninguna modificación especial:

```
import java.io.*;
class SerialEscribe2 {
public static void main(String arg[]) {
try {
   FileOutputStream f=new FileOutputStream("prueba.dat");
   ObjectOutputStream ost;
   ost = new  ObjectOutputStream(f );
   Persona2 persona=new Persona2("Victor",30);
   ost.writeObject(persona);
   ost.flush(); // vaciar el buffer
   ost.close();
 } catch (IOException e) {
                System.err.println(e);  }
  }
}
```

```
   import java.io.*;
```

```
  class SerialLee2 {
  public static void main(String arg[]) {
  try {
   FileInputStream f=newFileInputStream("prueba.dat");
  ObjectInputStream fi = new ObjectInputStream(f);
    Persona2 persona=(Persona2) fi.readObject();
   System.out.println(persona.ver());
   f.close();
    } catch (IOException e) {
              System.err.println(e);    }
     catch (ClassNotFoundException e) {
            System.err.println(e) ;  }
   }
}
```