UT8 Manejando errores: excepciones. Entrada/Salida en Java.

Sabemos que los errores lógicos que se producen en un programa son más difíciles de detectar que los errores sintácticos puesto que el compilador no nos avisa. Estos errores se producen por diferentes causas:la solución a un problema se ha resuelto incorrectamente, pedimos a un objeto que haga una operación que no puede. Aunque los tests nos ayudan a identificar y eliminar este tipo de error un programa puede seguir fallando debido a circunstancias que escapan al control del programador (por ejemplo, intento de escribir en un disco lleno).
En esta unidad de trabajo aprenderemos a anticipar y responder a situaciones de error que ocurren durante la ejecución de un programa.

Una de las situaciones en la que los errores pueden ocurrir fácilmente es en las operaciones que requieren acceso a ficheros en disco. Estudiaremos cómo trabajar con ficheros de texto, cómo efectuar este tipo de operaciones de entrada/salida.

8.1.- Programación defensiva.

Cuando se diseña una clase cuyos objetos actuarán como objetos servidor (toda su actividad está dirigida por las peticiones de sus clientes, por ejemplo, el caso de la clase `AgendaNotas`) se pueden adoptar dos posibles visiones:

- asumir que los objetos cliente saben lo que tienen que hacer y solicitar únicamente servicios de la clase bien definidos
- asumir que la clase servidora operará en un entorno hostil y tomará las precauciones necesarias en caso de mal uso de la clase por parte de los clientes.

La práctica habitual está entre las dos posiciones.

Las cuestiones que se plantean son:

- ¿cuánto se debe verificar las peticiones de los clientes?
- ¿cómo debe informar una clase servidora de los errores a sus clientes?
- ¿cómo puede un cliente anticipar posibles fallos de las peticiones que hace a un servidor?
- ¿cómo deberá un cliente gestionar los posibles fallos ante las peticiones que hace?

8.1.1.- Verificación de argumentos.

Un objeto servidor es más vulnerable si el constructor y/o métodos tienen parámetros. Es fundamental que el servidor sepa si puede confiar en los valores de los argumentos o si debe verificar su validez antes de utilizarlos. Verificar los argumentos es una medida defensiva.

```
public int dividir(int x, int y)
{
        if (y != 0)
                return x / y;
        return -1;
}
```

```
public int dividir(int x, int y)
{
        if (y == 0)
            throw
                new ArithmeticException("Error,división entre 0");
        return x / y;
}
```

```
public void borrarNota(int numNota)
{
        if  (numNota >= 0 && numNota < numeroNotas())
                notas.remove(numNota);
}
```

8.1.2.- Informar acerca de los errores.

Es buena práctica que el objeto servidor haga algún esfuerzo para indicar que un problema ha ocurrido (además de preguntar por la validez de los argumentos).

Si en el ejemplo anterior hacemos: `dividir(14, 0);` no se produce un error porque el objeto servidor lo impide pero no hay ninguna notificación acerca de lo que ha ocurrido.

a) se puede informar al usuario enviando un mensaje de error:

- con sentencias `println()` (ya lo hemos hecho)
- mostrando una ventana gráfica de error (lo hemos hecho también)

La desventaja de hacer esto es que asumimos, por un lado, que es un usuario humano el que utiliza la aplicación y, por otro lado, aunque vea el mensaje no puede hacer nada.

b) Otra posibilidad es notificar al objeto cliente de que algo ha ido mal. El servidor puede en este caso:

- devolver un valor que indique que ha habido un fallo (puede ser un valor negativo o *null*). Esta solución no siempre es posible.

- cuando a través del tipo de retorno no es posible avisar del error hay que recurrir a otra alternativa, lanzar una excepción (***throwing an exception***). ***Lanzar una excepción*** es el modo más efectivo de que un objeto servidor indique que no puede llevar a cabo una petición del cliente. Las excepciones además es imposible ignorarlas y son independientes de los valores de retorno.

8.2.- Excepciones. Lanzar una excepción.

Una excepción se lanza para indicar que ha ocurrido un error en el programa. Una excepción es un objeto que contiene detalles acerca de dicho error.

```
/**
*  @throws IllegalArgumentException si  y = 0
*/
public int dividir(int x, int y)
{
        if (y == 0)
                throw new IllegalArgumentException(...);
        return x / y;
}
```

Para lanzar una excepción:

a) se crea un objeto excepción (objeto de tipo `IllegalArgumentException`, `NullPointerException`, ...)

b) se lanza la excepción a través de ***throw***

`throw new tipo_excepción("……");`

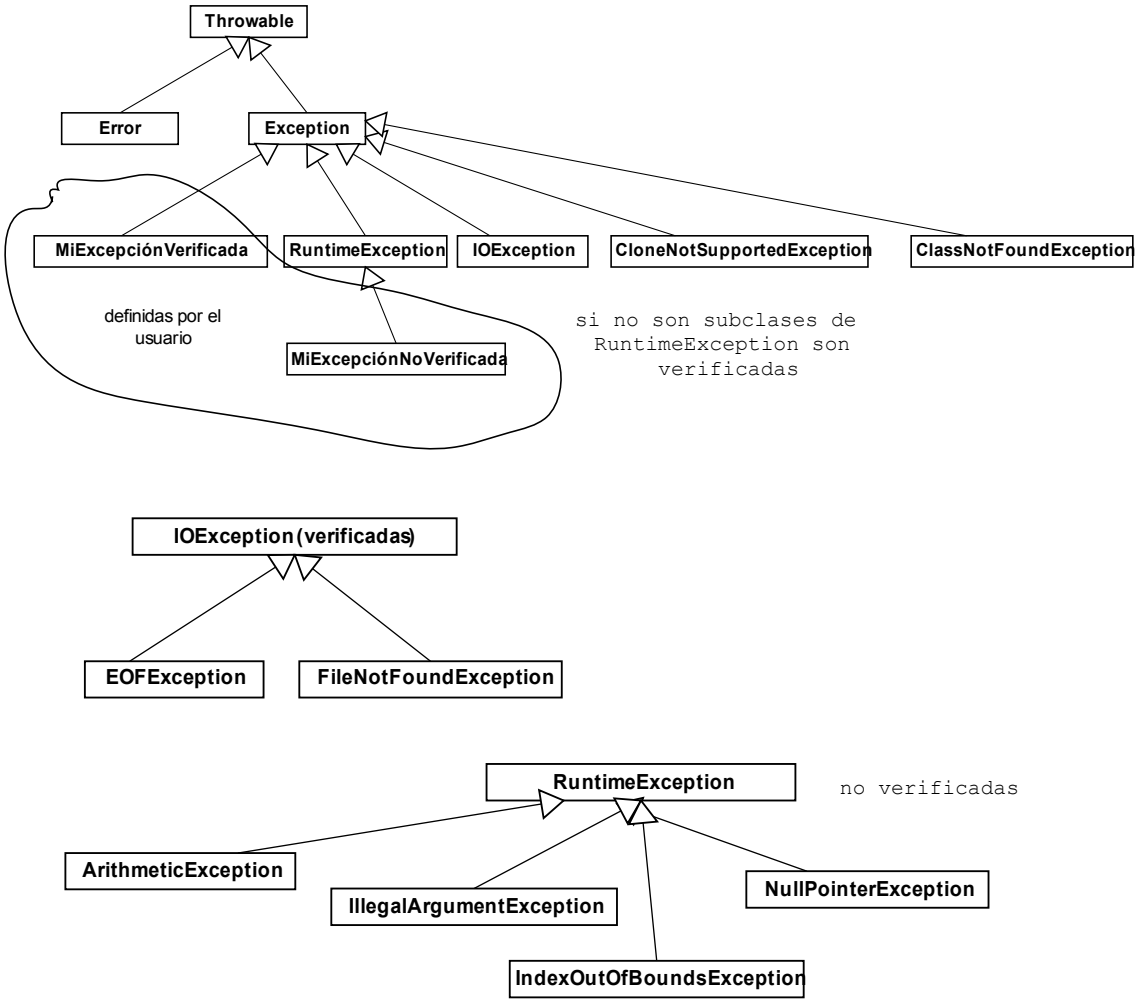
Cuando se lanza una excepción podemos pasar un mensaje descriptivo al constructor de la excepción. Este mensaje está disponible después al receptor de la excepción a través del accesor `getMessage()` del objeto excepción.

La etiqueta javadoc *@throws* permite documentar la excepción.

8.3.- Jerarquía de excepciones.

Un objeto excepción es siempre una instancia de una clase que forma parte de una jerarquía de excepciones. Podemos crear nuevos tipos de excepciones como subclases de esta jerarquía.

Las clases excepción están en el paquete *java.lang*. Todas heredan de Throwable. Cuando creemos nuevas excepciones extenderemos la clase Exception.



8.3.1.- Excepciones verificadas y no verificadas.

Java divide las excepciones en 2 categorías:

a) ***Excepciones no verificadas*** – son todas las subclases de la clase RuntimeException. Se lanzan cuando se dan situaciones de error evitables que conducen a errores lógicos de programación (un índice incorrecto, por ej. - IllegalArgumentException, NumberFormatException, ...).

b) ***Excepciones verificadas*** – las que derivan de Exception. Se lanzan cuando se produce un error fuera del control del programador (operaciones de E/S – escribir datos en un disco y que éste esté lleno, por ej.) y para aquellos en los que una recuperación es posible.

El uso de un tipo u otro es diferente: las excepciones no verificadas pueden ignorarse, las verificadas no. Si un objeto cliente llama a un método que puede lanzar una excepción verificada ha de gestionar la excepción obligatoriamente (no puede ignorarse, o se capturan o se propagan).

Las subclases de Error normalmente están reservadas para errores de ejecución de sistema (errores en la JVM, por ejemplo).

8.4.- Tratamiento de excepciones.

Los métodos que llaman a métodos que pueden lanzar excepciones tienen que:

a) capturar la excepción y tratarla o

b) volverla a lanzar, es decir, propagarla

Cuando se lanza una excepción hay que considerar dos efectos:

a) *el efecto en el método que lanza la excepción* – la ejecución del programa termina inmediatamente, no se devuelve ningún valor y el control del programa no vuelve al punto del cliente a no ser que el cliente en el que se llamó al método "*capture*" la excepción.

b) *el efecto en el método llamante* – dependerá de si se ha escrito o no código para capturar (*catch*) la excepción.

8.4.1.- Excepciones no verificadas.

En el caso de las excepciones no verificadas (las que hemos utilizado hasta ahora) son las más sencillas de utilizar desde el punto de vista del programador ya que el compilador no verifica nada especial en el método que la lanza o en el sitio en el que se invoca al método que produce una excepción.

Simplemente se puede lanzar con la claúsula ***throw***.

Una excepción no verificada muy usual es *IllegalArgumentException* utilizada por un constructor o método para indicar que los valores de los argumentos son inapropiados.

```
public boolean  buscarProducto(String nombre)
{
        if (nombre == null)
                throw new NullIlPointerException("Nombre nulo");
        nombre = nombre.toUpperCase();
         .....
}
```

Otro uso habitual de las excepciones no verificadas es para prevenir que los objetos sean creados con valores que los dejen en un estado incorrecto o inconsistente.

```
public class  Cuenta
{
        private String nombre;
        private int importe;

        public Cuenta(String nombre,  int importe)
        {
                if (nombre == null) || nombre.equals(""))
                        throw new IllegalStateException("Nombre inválido");
                if (importe < 0)
                        throw new IllegalStateException("Importe inválido");
                this.nombre = nombre;
                this.importe = importe;
        }
}
```

8.4.2.- Excepciones verificadas.

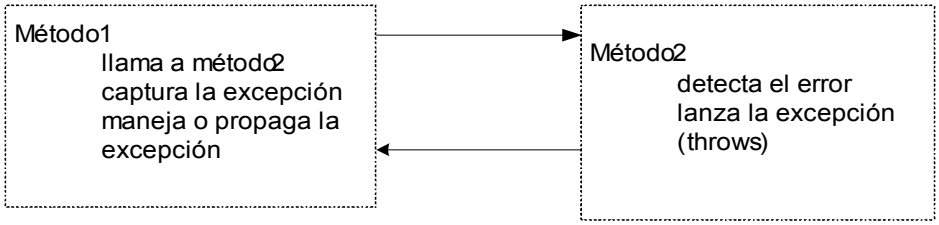
El lanzamiento de excepciones (con la claúsula *throw*) es idéntico para verificadas y no verificadas pero su manejo (la captura) es obligatorio en las verificadas.

Las excepciones verificadas recordemos son aquellas que derivan de Exception pero no de RuntimeException.

El compilador, en estas excepciones, hace comprobaciones tanto en el método que lanza la excepción verificada como en el método llamante.

El primer requerimiento del compilador para una excepción de este tipo lo hace en el método que lanza la excepción, es obligatorio que el método "avise" de que va a lanzar la excepción con la claúsula ***throws*** en su cabecera:

```
public  void guardarEnFichero(String nombre) throws IOException

public  Set<Estudiante> leerEstudiante(String fichero) throws IOException
```



(En las excepciones no verificadas también puede ponerse *throws* en la cabecera del método que lanza la excepción pero no es obligatorio).

El segundo requerimiento del compilador es que el método que llama a un método que lanza una excepción verificada debe proporcionar código para manejar la excepción o sino propagarla. Esto significa escribir un gestor de excepciones a través de la sentencia ***try – catch***.

Si se propaga, el método llamante lo indica incluyendo la cláusula *throws* en su cabecera. (Propagar una excepción significa declararla con *throws* en la cabecera del método).

8.4.2.1.- La sentencia try .... catch. El bloque finally.

Escribir una sentencia *try ... catch* es escribir un ***gestor de excepción*** *(exception handler)*.

```
try
{
        //código a proteger
}
catch (Exception e)
{
         //código de manejo de la  //excepción
}
```

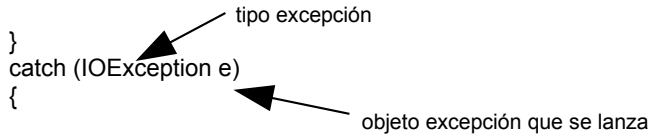
```
try
{
          agenda.guardarEnFichero(nombre);

}
catch (IOException e)
{
          System.out.println("Imposible operación en
                              fichero");
}
```

Una excepción interrumpe el flujo normal de ejecución del programa. En el bloque try { ... } se sitúan las sentencias que pueden causar una excepción. Si alguna de estas sentencias genera una excepción en ese momento la ejecución continúa en ese momento en el bloque catch { ... }. Si no ocurre ninguna excepción durante la ejecución de las sentencias protegidas el bloque catch { ... } se ignora.

En nuestro ejemplo, si al intentar ejecutar el método guardarEnFichero() se genera una excepción el control de ejecución se transfiere al bloque catch { ... } y se ejecutará la sentencia println().

El bloque catch() es el que captura la excepción y en él se indica el tipo de excepción que captura y la variable que referencia al objeto excepción que se crea.

```
try
{

}
catch (IOException e)
{

}
```



8.4.2.2.- Lanzar y capturar múltiples excepciones.

A veces un método puede lanzar más de un tipo de excepción indicando diferentes problemas que pueden surgir en su ejecución.

```
public void procesar() throws EOFException, FileNotFoundException
```

Si son verificadas hay que indicar todas en la cabecera del método que las puede lanzar con *throws* y separadas por , .

El gestor de excepciones debe proporcionar código para manejar todas las posibles excepciones. Una sentencia *try* ... puede contener múltiples bloques *catch* .....

```
try
{
        .....
        obj.procesar();
        .....
}

catch (EOFException e)
{

}
catch (FileNotFoundException e)

{

}
```

Los bloques catch() cuando aparecen en una misma sentencia try ... se analizan en el orden en que se han escrito hasta encontrar uno que concuerde con la excepción lanzada. En el ejemplo si se lanza EOFException el control se transfiere al primer bloque catch(). Una vez completado este bloque la ejecución del programa continúa detrás del último catch(). Si la excepción lanzada es FileNotFoundException el bloque catch() ejecutado es el 2º.

Cuando hay varios bloques catch() importa el orden en que se escriben en una sentencia try. Un bloque catch() para un tipo concreto de excepción no puede seguir a un bloque catch() para una excepción de su supertipo.

Se puede utilizar el polimorfismo para evitar escribir múltiples bloques catch() (aunque es más recomendable tomar acciones específicas para cada tipo de excepción).

```
try
{
        ....................
        obj.procesar();
        ....................
}

catch (Exception e)
{

}
```

Si se produce la excepción en el bloque catch() únicamente se comprueba que el objeto *e* sea una instancia del tipo indicado, en este caso Exception. Puesto que todas las excepciones son del tipo (o subtipos) de Exception, este bloque podría capturar cualquier excepción generada (verificada y no verificada).

Ejemplo

```
try
{
        ....................
        Persona p = bd.buscar(nombrePersona);
        ....................
}

catch (Exception e)
{

}
catch (RuntimeException e)
{

}
```

Incorrecto porque Exception es un supertipo de RuntimeException. Hay que comenzar siempre los bloques catch() por las excepciones más específicas.

8.4.2.3.- Propagando una excepción.

Hasta ahora se ha sugerido que una vez lanzada una excepción sea manejada lo más pronto posible. Si un método procesar() lanza una excepción lo mejor sería capturarla y gestionarla en el método que llama a procesar(). Incluso el propio método procesar() puede lanzar y capturar la excepción.

Sin embargo Java permite que una excepción sea propagada desde el método receptor de la excepción (en principio el que debería manejarla) hacia atrás, hacia el método que lo llamó e incluso más atrás.

Un método propaga una excepción no incluyendo un gestor (try ... catch()) que proteja las sentencias que pueden ocasionar la excepción. Para las excepciones verificadas Java obliga a que el método que las propaga incluya una cláusula *throws* en su cabecera, incluso si él mismo lanza la excepción. Si la excepción es no verificada la cláusula *throws* es opcional (nosotros la omitiremos). La cadena de propagación puede llegar incluso hasta el método main(). La propagación es habitual cuando el método llamante no sabe cómo tratar la excepción, o no puede, o no lo necesita y deja que se haga en llamadas de más alto nivel.

8.4.2.4.- El bloque *finally*.

Es un bloque opcional que puede incluir la sentencia try ... catch. Proporciona código que ha de ser ejecutado tanto si se lanza la excepción como si no. Si se lanza la excepción se ejecuta el bloque catch y luego finally. Si no se lanza se ignora el bloque catch y se ejecuta finally. El tipo de sentencias que se incluyen permiten, por ejemplo, cerrar ficheros, liberar recursos, ...

```
try
{
        .....

}

catch (Exception e)


}
finally
{
        //acciones a ejecutar siempre, se lance o no la excepción
}
System.out.println("Bloque después de finally);
```

Si se lanza una excepción que no concuerda con ninguna del *catch* se ejecuta el bloque *finally* pero no lo que esté detrás de él, el programa se interrumpe

8.4.2.5.- Mejoras introducidas por Java 7 en el manejo de excepciones.

Es posible capturar múltiples excepciones en un gestor con una única cláusula catch.

```
try
{
   file = new File("readme.txt");
   process(file);
}
catch (FileNotFoundException | UnsupportedEncodingException ex)
{
   ...
}
```

8.5.- Definiendo nuevas clases excepción.

Si las excepciones que nos da Java no nos son apropiadas podemos definir nuestras propias clases excepción utilizando la herencia.

Las nuevas clases para excepciones verificadas pueden ser definidas como subclases de cualquier excepción verificada, por ejemplo, Exception.

Las nuevas excepciones no verificadas pueden extender de RuntimeException. Las nuevas clases incluyen constructores, atributos, ...y nos permiten introducir medidas de diagnóstico más precisas que las que nos dan las clases existentes, por ejemplo, un método toString() más descriptivo en la nueva clase excepción.

```
public class FormatoExcepcion extends IOException
{
        private String valorErroneo;
        public  FormatoExcepcion(String valorErroneo)
        {
                this. valorErroneo  =  valorErroneo;
        }
        public  String toString()
        {
                return "Se ha introducido un valor incorrecto " +
                        valorErroneo;
        }
}
```

8.6.- Aserciones.

Una ***aserción*** es una sentencia que afirma acerca de un hecho que debería ser *true* en un punto normal de ejecución de un programa.

Las aserciones se utilizan para hacer asunciones explícitas en momentos puntuales del desarrollo de un programa y así poder detectar errores más fácilmente. Permiten detectar inconsistencias en el proceso de desarrollo y test de un proyecto, no se usan al distribuir el código. Tampoco son alternativas a las excepciones.

Java incluye una sentencia ***assert*** de dos formas posibles:

assert expresión_booleana
assert expresión_booleana: expresión

Expresa que algo debe ser cierto en ese punto. Si no es así la aserción es falsa y se lanza AssertionError que es una subclase de Error (no se proporciona manejador). Si la aserción es cierta no tiene efecto.

Ejemplo

```
public class Coche
{
        private  static final double MAX_LITROS = 50.00;
        private int kmos;
        ..........................

        public  int getKmos()
        {
                assert kmos > 0: “Kilómetros negativos”;
                return kmos;
        }

        public  void rellenarDeposito()
        {
                assert litros < MAX_LITROS;
                litros =  MAX_LITROS;
        }
}
```

8.6.1.- Aserciones y BlueJ.

El soporte que BlueJ proporciona para el tests de proyectos (que está basado en el framework de `JUnit`) está basado en el uso de aserciones. Los métodos de `JUnit`, como assertEquals(), se construyen alrededor de aserciones.

Ejer.8.1.

Realiza la hoja de cuestiones proporcionada acerca de las excepciones.

8.7.- I/O en Java.

Todos los programas necesitan efectuar operaciones de E/S (I/O – Input/Output) para comunicarse con el mundo exterior.

Los mecanismos de salida de datos pueden ser:

- los resultados de métodos (en el caso de BlueJ)
- envío de texto a la consola, impresora
- envío de datos a un fichero de disco
- envío de datos a otros programas, a través de la red, ...
- salida gráfica (GUI)

Los mecanismos de entrada de datos pueden ser:

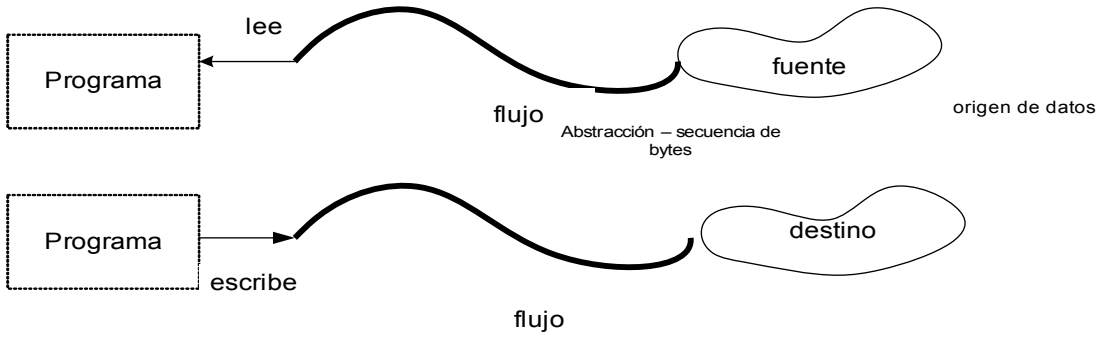
- parámetros en métodos (BueJ)
- parámetros en línea de comandos
- lectura de texto desde teclado
- lectura de datos de un fichero de disco
- obtención de datos desde otros programas, a través de la red, ...
- entrada a través de una GUI (click en un botón, selección de un menú, ...)

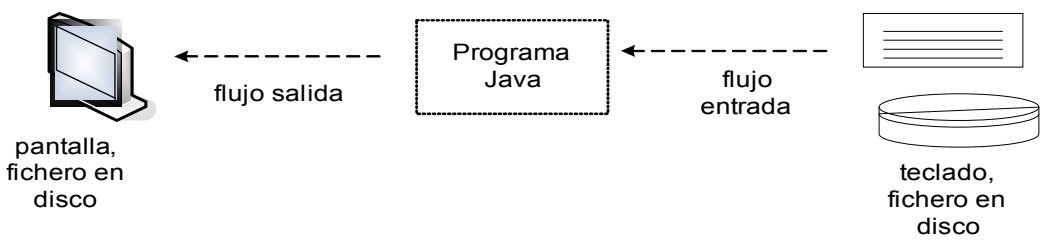
En Java toda la I/O se realiza a través de un conjunto de clases que están en el paquete *java.io*.

8.7.1.- El concepto de flujo (*stream*) de I/O.

Los datos que llegan a un programa pueden venir de varias fuentes. Los datos que produce un programa pueden enviarse a varios destinos. La conexión entre un programa y una fuente de datos o un destino es lo que se denomina flujo (*stream*).

Un flujo de entrada *(input stream*) maneja datos hacia un programa. Un flujo de salida (*output stream*) maneja datos que salen de un programa.





En escritura:

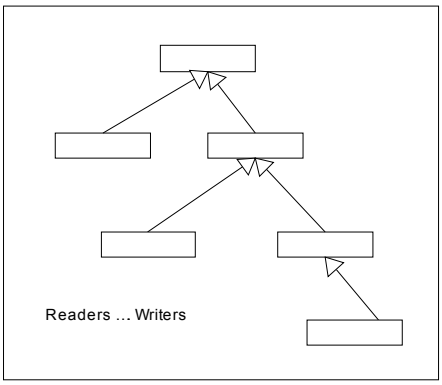
```
abrir el flujo (puede asociarse a un fichero)
while (haya información)
{
        escribir información en el flujo
}
cerrar el flujo
```

En lectura:

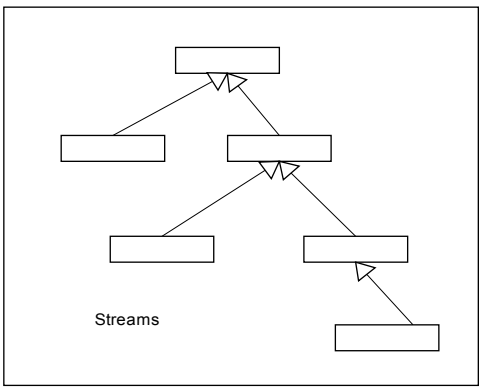
```
abrir el flujo (puede asociarse a un fichero)
while (haya información)
{
        leer  información  del flujo
}
cerrar el flujo
```

8.7.2.- Flujos de bytes y flujos de caracteres.

- Los flujos de caracteres (*character stream*) están optimizados para manejar datos carácter. La fuente y el destino habitualmente es en estos flujos un fichero de texto, un fichero que contiene bytes que representan caracteres.
- Los flujos de bytes (*byte stream*) manejan entrada/salida de propósito general, la lectura/escritura de datos binarios (nºs, imágenes, sonido, ficheros ejecutables, ...).



Flujos carácter — Lectura/ Escritura de texto



Flujos bytes - Streams — Lectura / Escritura de datos binarios

Un flujo de bytes (*stream*):

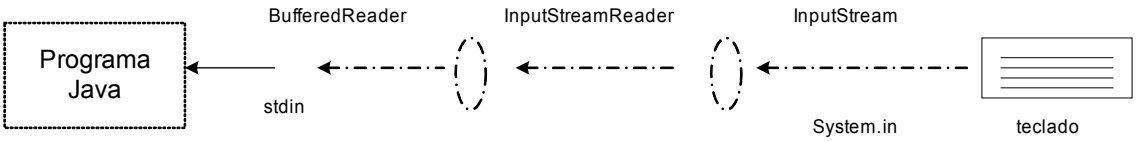
- permite leer/escribir bytes
- puede usarse con cualquier flujo de datos
- se utiliza en serialización (escritura de objetos en disco)

Un flujo de caracteres (*reader/writer*):

- permite leer / escribir caracteres
- se usa en entrada / salida de texto
- aparecieron más tarde

8.7.3.- Flujos de proceso (filtros, clases envolventes, decoradores).

Un flujo de proceso opera sobre una serie de datos proporcionados por otro flujo. A menudo un flujo de proceso actúa como un *buffer* que almacena datos que vienen de otro flujo. Proporcionan funcionalidad adicional a los flujos existentes. Por ejemplo, el teclado envía datos al flujo InputStreamReader que a su vez se conecta al flujo BufferedReader. *System.in* es un objeto que Java crea automáticamente cuando un programa empieza a ejecutarse.

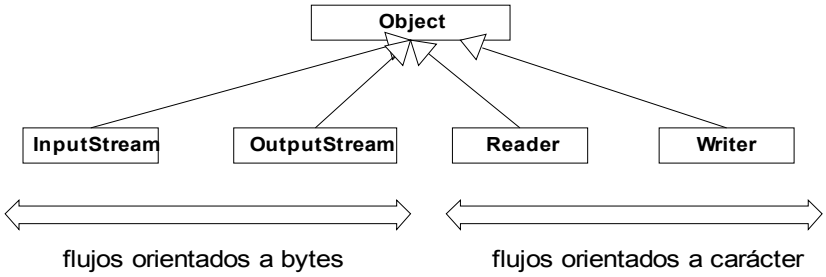



```
BufferedReader stdin  = new BufferedReader (new InputStreamReader (System.in));
```

Los datos se transforman a lo largo del camino. Las filas de bytes del teclado se agrupan en objetos String que el programa lee a través de readLine()).

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
```

8.8.- Jerarquía de clases de I/O. El paquete *java.io*.



Las cuatro son clases abstractas que actúan como clases bases para flujos más especializados.

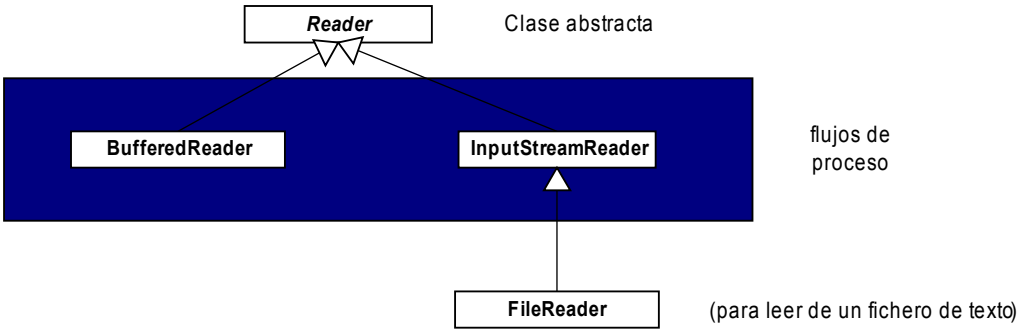
8.8.1.- Readers / Writers.

Reader y Writer son clases abstractas. Representan flujos de caracteres.

Los flujos de caracteres están optimizados para manejar caracteres. Transforman el formato interno de los datos utilizados por los programas Java en el formato externo utilizado por los ficheros de texto. Dentro de un programa Java los caracteres se representan mediante 16 bits. En un fichero en disco, los caracteres se representan en un formato denominado UTF (*Universal Text Format)* que utiliza 1, 2 o 3 bytes. Habitualmente un fichero de texto UTF 8 es idéntico a un fichero ASCII.

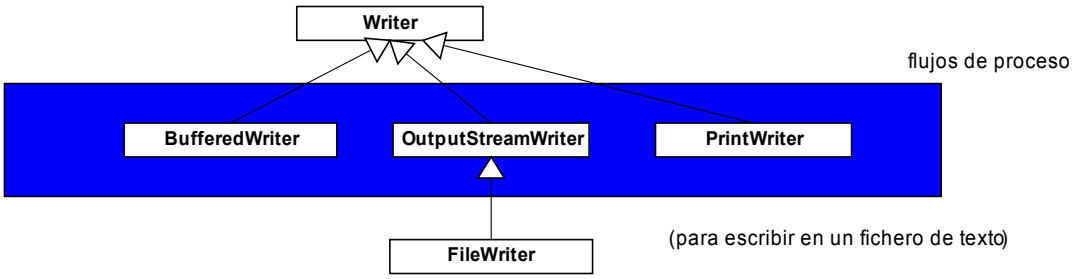
8.8.1.1.- Readers.

Reader es una clase abstracta de la que derivan todos los flujos de entrada orientados a carácter. Envía datos de 16 bits a un programa aunque la fuente tenga formato diferente (por ejemplo, un fichero de texto en formato UTF8).



8.8.1.2.- Writers.

Writer es una clase abstracta de la que derivan todos los flujos de salida orientados a carácter. Estos flujos reciben datos de 16 bits de un programa y los envían a su destino como puede ser un fichero de texto en formato UTF8.



8.8.2.- Clase FileWriter.

Esta clase se utiliza para escribir caracteres en un fichero de texto en disco.

```
public void crearFichero(String nombre)   // si no capturo con try .. catch pondría aquí throws IOException
{
        try
        {
            FileWriter f = new FileWriter(nombre);
            f.write("Creando una linea de texto\r\n");
            f.write("otra linea mas\n");
            f.close();
        }
        catch (IOException e)
        {
            System.out.println("Error al crear " + nombre);
        }

}
```

\r\n salto de línea en Windows

El ejemplo anterior crea un fichero de texto ASCII con dos líneas de caracteres, cada carácter ocupa un byte en el fichero.

Dos posibles constructores de `FileWriter` son:

```
FileWriter (String nombre)
FileWriter (String nombre, boolean append)
```

(consulta la ayuda)

El constructor crea un nuevo fichero en el disco con el nombre indicado. Si existe ya un fichero con ese nombre será reemplazado con el nuevo contenido. Si se utiliza el 2º constructor y *append* es *true* se abre un fichero que ya existe sin destruir su contenido. Si el fichero no existe se crea.

El método write() escribe una serie de caracteres en el fichero.

```
public void write(String str) throws IOException
```

El método close() cierra el fichero. Si un fichero no se cierra pueden perderse sus datos.

La mayoría de los métodos de IO lanzan una IOException si ocurre un error que hay que tratar en el bloque try ... catch.

8.8.3.- Clase BufferedWriter (clase envolvente).

La E/S a disco es más eficiente si se utiliza un *buffer*. El flujo BufferedWriter se utiliza en escritura para aumentar la velocidad de salida y reducir las escrituras.

Incluye un método newLine() para escribir un salto de línea.

```
public BufferedWriter(Writer salida)
```

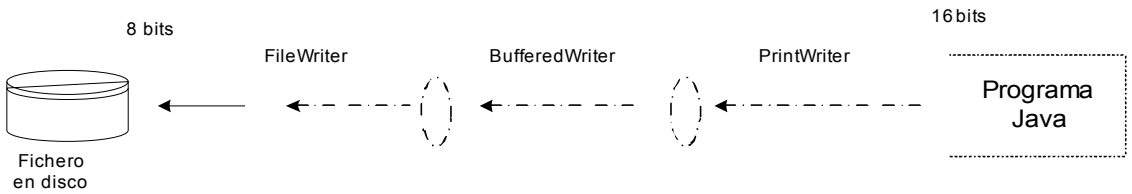
Ej.

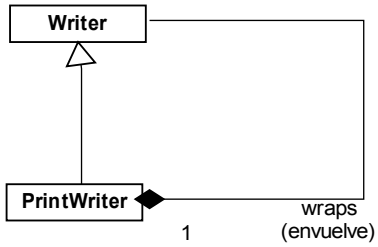
```
BufferedWriter salida = new BufferedWriter(new FileWriter("fichero.txt"));
```

8.8.4.- La clase PrintWriter (clase envolvente).

Es una subclase de Writer que proporciona métodos convenientes para visualizar los diferentes tipos de datos Java en un formato legible. Lo habitual es conectar un flujo PrintWriter a un flujo BufferedWriter que a su vez se conecta a un FileWriter. (Para salida de texto formateada).

Podemos escribir con print(), println() y printf() tal como hacíamos hasta ahora en el terminal de texto.



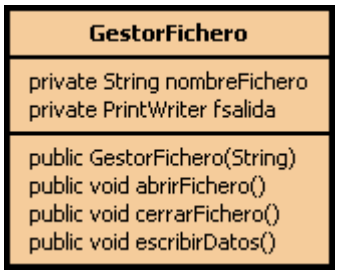


Una ventaja adicional de PrintWriter es que ninguno de sus métodos lanza excepciones.

Ejemplo

```
public void ejemploPrintWriter(String nombre)
{
        PrintWriter fsalida = null;
        try
        {
                fsalida = new PrintWriter(new BufferedWriter(new FileWriter(nombre)));
                fsalida.println("Tabla del 4");
                for (int i = 1; i <= 10; i++)
                        fsalida.println(i + "*" + 4 + "\t" + i * 4);
                 fsalida.close();

        }
        catch (IOException e)
        {

              System.out.println("Error al crear " + nombre);
        }
    }
```

Ejer.8.2.



Crea la siguiente clase en la que el constructor recibe como parámetro el nombre de un fichero .

El método abrirFichero() crea el fichero de ese nombre . Propaga la posible excepción que se puede producir (la capturaremos después en el main()).

cerrarFichero() cierra el fichero. Ten en cuenta que se puede invocar al método sin llamar previamente al método abrirFichero() con lo que se generará una excepción no verificada (haz la prueba y captura la excepción que se produce)

escribirDatos() escribe información (lo que quieras) en el fichero utilizando el método print() y println(). Escribe valores enteros, double, String, ... Consulta la ayuda de la clase PrintWriter. (Ten en cuenta que se puede invocar al método sin llamar previamente al método abrirFichero())

Añade un nuevo método public void abrirFicheroAñadiendo() que abra el fichero pero sin destruirlo, permitiendo añadir información en él.

Incluye una clase UsoGestorFichero que incluya un método main(). Prueba en esta clase el gestor. Captura las posible excepciones que se pueden producir si no se pasa como parámetro al main() el nombre de un fichero.

Ejer. 8.3.

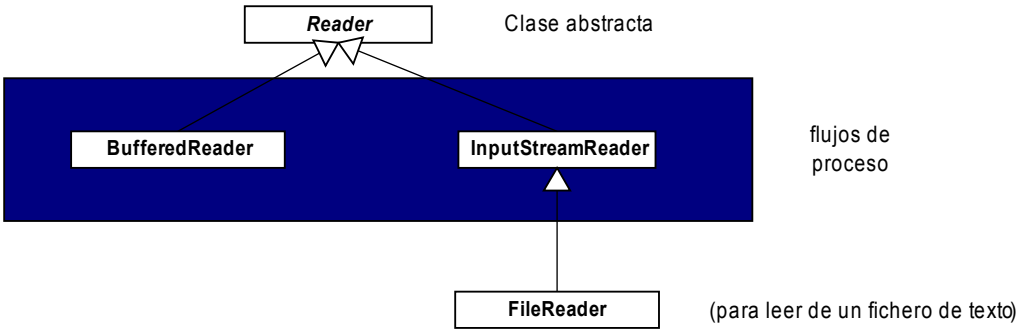
Crea una clase ListaNumeros con un atributo lista que es un array de enteros. El constructor recibe un parámetro que indica el tamaño del array. El constructor puede lanzar una excepción propia ArgumentoIncorrectoExcepcion. Exta excepción indica que el argumento pasado es un valor incorrecto. Si el argumento es correcto llama al método privado inicializar() que inicializa el array con valores aleatorios.

Añade a la clase el método public void guardarEnFichero() throws IOException que guarda en el fichero *numeros.txt* el contenido del array en una línea de texto en la que los números se separan por tabuladores. Utiliza PrintWriter.

Observa que el método avisa de que puede lanzar una excepción pero no la capturará. Lo hará la clase DemoLista que probará la clase ListaNumeros.

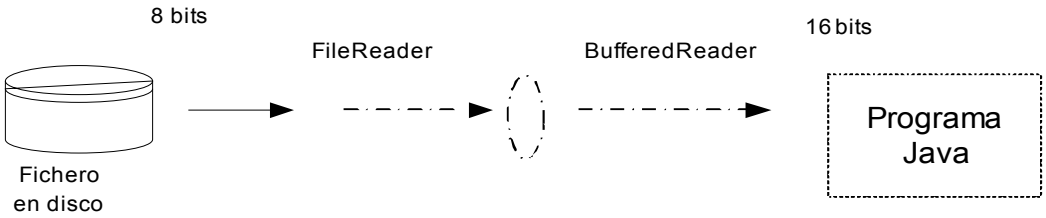
8.8.5.- Lectura de ficheros de texto. Readers.

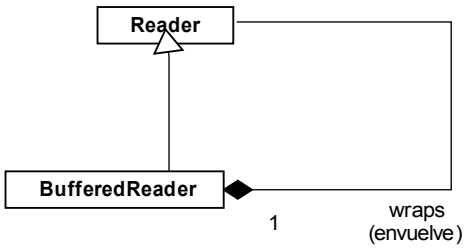
Un flujo de entrada puede ser conectado a un fichero de texto.



8.8.5.1- Clases FileReader y BufferedReader.

La clase FileReader se utiliza para leer caracteres desde un fichero de disco. El fichero de entrada puede ser un fichero ASCII (un byte por carácter en el fichero de texto). El flujo Reader automáticamente traduce los caracteres del fichero al formato interno que utiliza Java en el programa.





Habitualmente se utiliza un *buffer* (una clase envolvente) para mejorar la eficiencia, se usa la clase BufferedReader.

El método readLine() lee una línea de texto del flujo de caracteres y devuelve un String. Si no hay más datos en el fichero devuelve *null*.

```
public void ejemploReader(String nombre)
{
        String linea;
        try
        {
                BufferedReader entrada = new BufferedReader(new FileReader(nombre));
                linea = entrada.readLine();
                while (linea != null)
                {
                        System.out.println(linea);
                        linea = entrada.readLine();
                }
                entrada.close();
        }
        catch (IOException e)
        {
                System.out.println("Error al leer " + nombre);
        }

    }
```

8.8.6.- La clase File.

Proporciona acceso a información acerca de ficheros y directorios. Encapsula (abstrae) las propiedades de un fichero o directorio independientemente del sistema. Muchos constructores de flujos reciben un File.

Esta clase contienen métodos para obtener las propiedades de un fichero y para renombrar y borrar ficheros pero no tiene métodos para leer y escribir en un fichero.

```
public void demoFile(String nombre) throws IOException
{
        File fichero = new File(nombre);
        if (fichero.exists())
        {
            System.out.println("Nombre: " + fichero.getName());
            System.out.println("\nRuta completa: " + fichero.getAbsolutePath());
            System.out.println("\nTamaño: " + fichero.length() + " bytes");
            if (fichero.isFile())
                System.out.println("\nFichero normal");
            else
                if (fichero.isDirectory())
                    mostrarContenidoDirectorio(fichero);
        }
        else
            throw new IOException("El fichero " + nombre +
                " no existe");
 }

 private void  mostrarContenidoDirectorio(File directorio)
 {
        String[] ficheros = directorio.list();
        for (int i = 0; i < ficheros.length; i++)
            System.out.println("\t" + ficheros[i]);
 }
```

Ejer. 8.4.

Añade a la clase GestorFichero un nuevo método public void abrirFicheroLectura() que abra el fichero para leer (define previamente en la clase un nuevo atributo *fentrada* de tipo BufferedReader).

Incluye también el método public ArrayList<String> leerDatos() throws IOException que lee el fichero línea a línea y devuelve una colección con todas las líneas leídas y el método public void cerrarFicheroEntrada()

Prueba los nuevos métodos incluidos en la clase UsoGestorFichero.

8.8.7.- La clase Scanner.

Está en el paquete *java.util*.

Esta clase se ha introducido a partir de Java 5 para hacer más fácil la lectura de los tipos de datos básicos desde una fuente de caracteres. El constructor de la clase especifica la fuente de caracteres desde la que leerá Scanner.

La clase Scanner actúa como una clase envolvente (*wrapper*) para la entrada.

La fuente puede ser un Reader, un InputStream (flujo binario), un String o un File.

Ej.

```
Scanner teclado = new Scanner(System.in);     //System.in es un flujo InputStream

Scanner fichero = new Scanner(new File("texto.txt"));
```

Recordemos que cuando Scanner procesa una entrada trabaja con *tokens* que se separan por delimitadores.

Ejemplos

```
Scanner cs = new Scanner(new
                        File("numeros.txt"));
int suma = 0;
while (sc.hasNextInt())
        suma += sc.nextInt();
```

```
Scanner cs = new  Scanner("12,34,13").useDelimiter(",");

int suma = 0;
while (sc.hasNextInt())
        suma += sc.nextInt();
```

8.8.8.- System.in y System.out.

in y ***out*** que hemos venido utilizando habitualmente son flujos binarios definidos en la clase System.

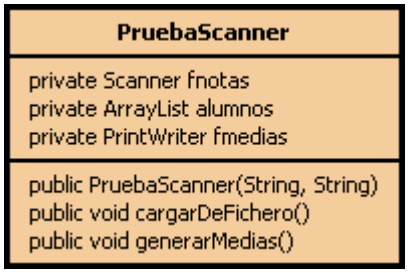
```
public class System
 {
        .....
        public static final InputStream in;
        public static final PrintStream out;
        .....

 }
```

System, por razones históricas, utiliza flujos binarios InputStream y PrintStream para la IO. Es preferible para la IO texto las clases Reader / Writer.

Ejer. 8.5.

Crea las siguientes clases en un proyecto. Antes crea un fichero de texto (con el Bloc de notas, por ejemplo) *"notas.txt"* con una línea por cada alumno indicando nombre@nota1@nota2@nota3.

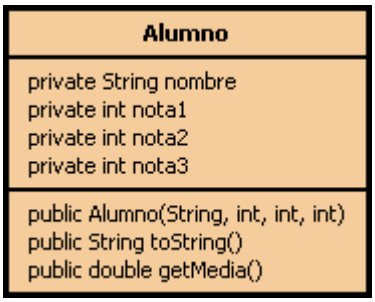
La clase PruebaScanner leerá una a una las líneas de este fichero con ayuda de Scanner y File y creará una colección ArrayList de tipo Alumno (se muestra el diagrama UML de Alumno). A partir de esta colección se generará un fichero de salida con una línea por alumno en la que aparecerá *nombre* y *nota media*.



public PruebaScanner(String nombreEntrada, String nombreSalida) – el constructor crea los dos ficheros iniciales y la colección vacía. Captura las posibles excepciones de IO.

cargarDeFichero() - lee el fichero de entrada (las notas) y procesa cada línea adecuadamente para crear un objeto por alumno y añadirlo a la colección ArrayList. Puedes utilizar split() y crear un array o bien utilizar la clase StringTokenizer.

generarMedias() recorre la colección y crea el fichero de salida, una línea por alumno indicando su nombre y nota media.



8.9.- Serialización.

El proceso de almacenar (hacer persistente) un objeto y sus objetos dependientes en un flujo se denomina **serialización**. En la serialización un objeto se transforma en una secuencia de bytes y puede reconstruirse posteriormente a partir de la misma.

En Java es fácil guardar objetos en un flujo (por ejemplo, en un fichero de texto) sin tener que convertirlos previamente a una representación externa y sin tener que escribir/leer un objeto atributo a atributo.

Imaginemos el siguiente ejemplo, queremos guardar los objetos Empleado de una clase Empresa en un fichero:

```
import java.io.*;

public class Empresa
{
        private  Empleado[] empleados;
        public Empresa()
        {
                empleados = new Empleado[2];
                empleados[0] = new Empleado();
                empleados[1] = new Empleado();
        }

        public void salvar()
        {
                try
                {
                        ObjectOutputStream salida =
                                new  ObjectOutputStream(new  FileOutputStream("personal.txt"));
                        salida.writeObject(empleados);
                        salida.close();
                }
                 catch(IOException e)
                  {
                        e.printStackTrace(System.err);

                  }
        }

        public void leer() throws IOException, ClassNotFoundException
        {
                try
                {
                        ObjectIntputStream entrada =
                                new  ObjectIntputStream(new  FileInputStream("personal.txt"));
                        empleados = ( Empleado[]) entrada.readObject();
                        entrada.close();
                }
                 catch( IOException e)
                 {
                        e.printStackTrace(System.err);

                 }
        }
}
```

Construiremos un ObjectOutputStream asociado a un FileOutputStream.

```
ObjectOutputStream salida =
        new  ObjectOutputStream(new  FileOutputStream("personal.txt"));
```

Escribimos el array y cerramos el flujo.

```
salida.writeObject(empleados);
salida.close();
```

El array y todos los objetos que referencia son salvados en el fichero.

Para leer el proceso es inverso:

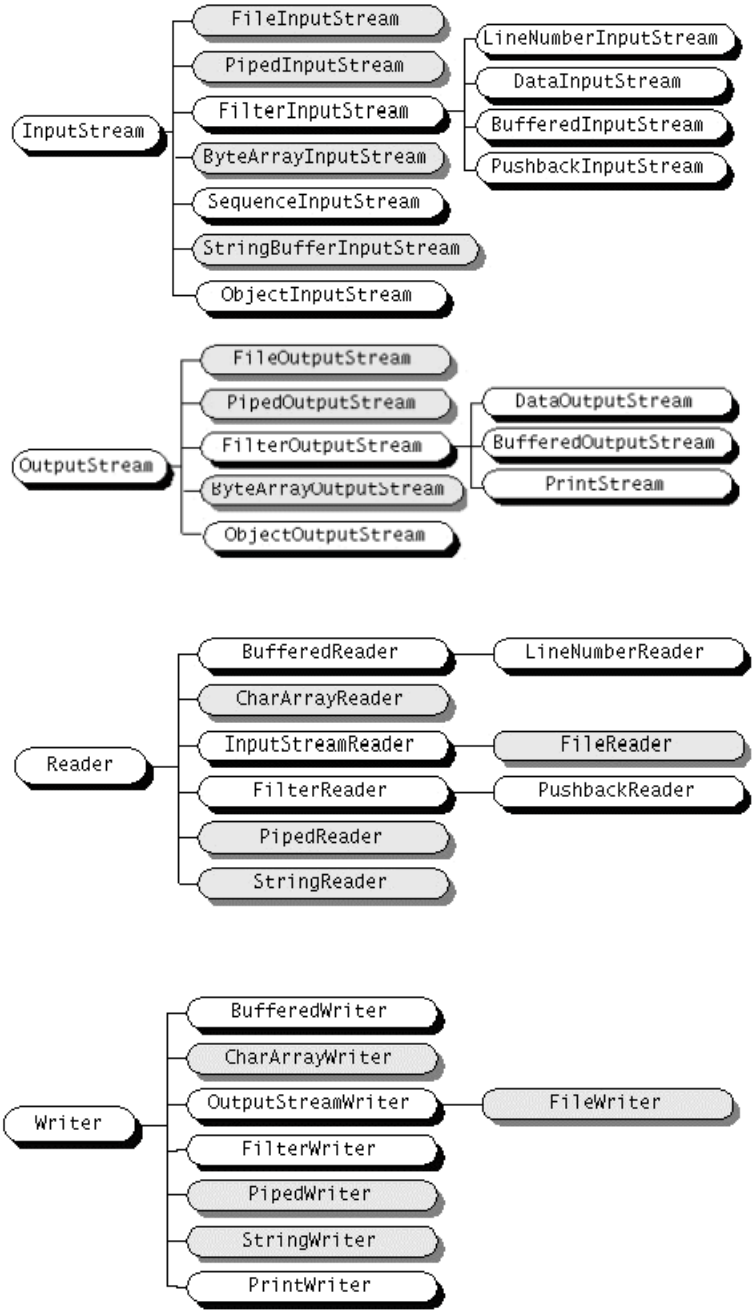
```
ObjectIntputStream entrada =
        new  ObjectIntputStream(new  FileInputStream("personal.txt"));
Empleado[] empleados = ( Empleado[]) entrada.readObject();
entrada.close();
```

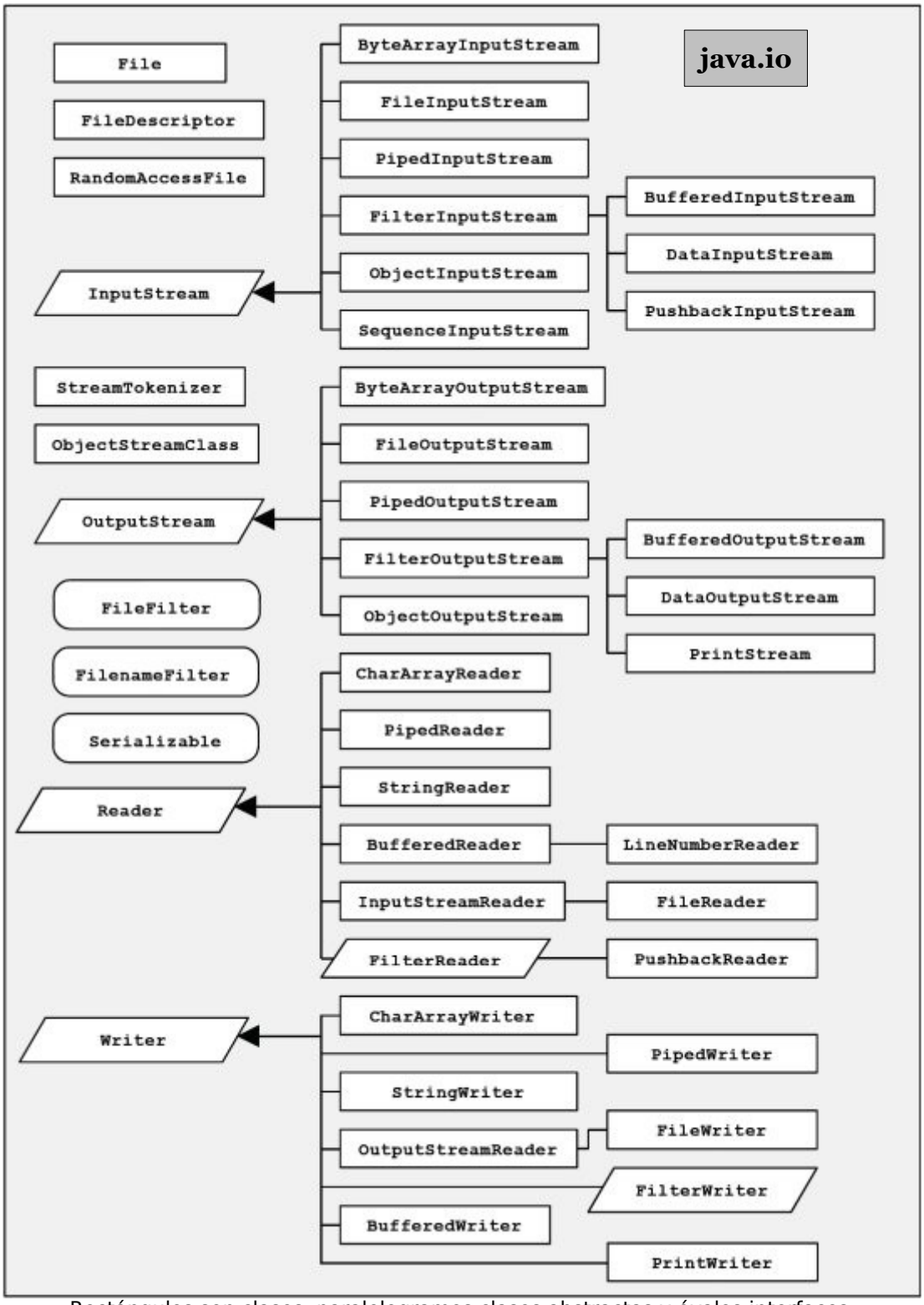
Después de leer el array *empleados* se completa con los objetos Empleado idénticos a como fueron guardados.

El único requerimiento es que la clase Empleado implemente el interfaz Serialiable. Este interface no tiene ningún método (como Cloneable, se les denomina *tagging* interface) por lo que la clase Empleado no ha de proporcionar ningún método adicional.

```
import java.io.Serializable;
public class Empleado implements Serializable
```

java.io


InputStream
FileInputStream
PipedInputStream
FilterInputStream
ByteArrayInputStream
SequenceInputStream
StringBufferInputStream
ObjectInputStream
LineNumberInputStream
DataInputStream
BufferedInputStream
PushbackInputStream
OutputStream
FileOutputStream
PipedOutputStream
FilterOutputStream
ByteArrayOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream
PrintStream
Reader
BufferedReader
CharArrayReader
InputStreamReader
FilterReader
PipedReader
StringReader
LineNumberReader
FileReader
PushbackReader
Writer
BufferedWriter
CharArrayWriter
OutputStreamWriter
FilterWriter
PipedWriter
StringWriter
PrintWriter
FileWriter



Rectángulos son clases, paralelogramos clases abstractas y óvalos interfaces