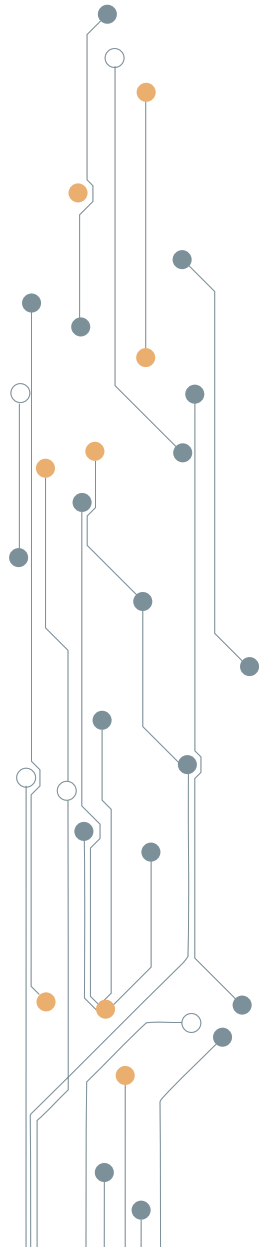
Entrada y salida en Java

Telefónica
EDUCACIÓN DIGITAL

Índice



1. La jerarquía de excepciones

El paquete java.io contiene casi todas las clases que se necesitan para llevar a cabo la entrada y salida en Java, en forma de "stream", flujos de datos de un origen de entrada a un destino de salida. Estos flujos soporta varios tipos de datos, tales como los primitivos y Object.

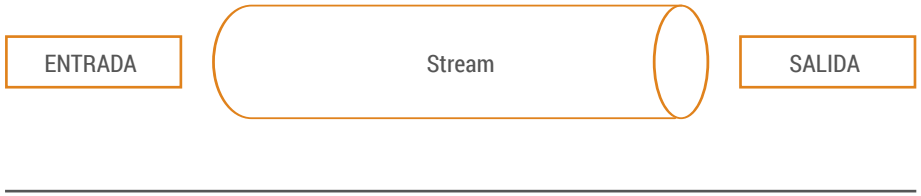


IMAGEN 10.1: FLUJO DE ENTRADA SALIDA

En la jerarquía de clases de E/S de Java en la parte superior se encuentran las dos clases abstractas siguientes:

- **InputStream:** utilizado para leer datos de un origen.

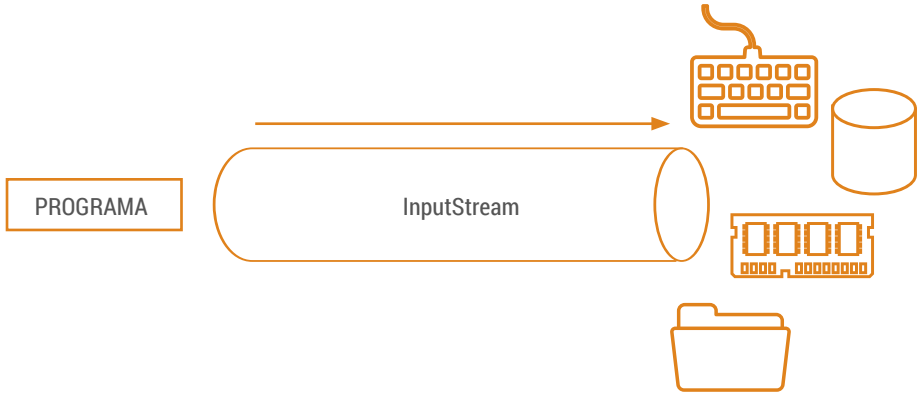
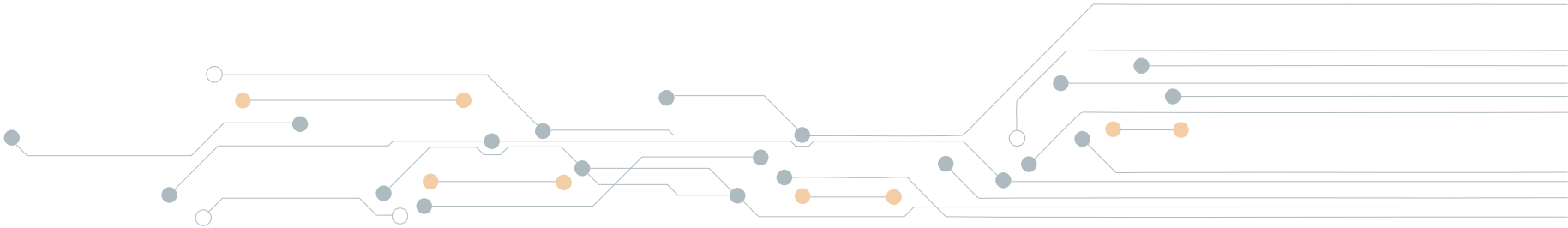


IMAGEN 10.2: INPUTSTREAM



- **OutputStream:** utilizado para escribir datos en un destino.

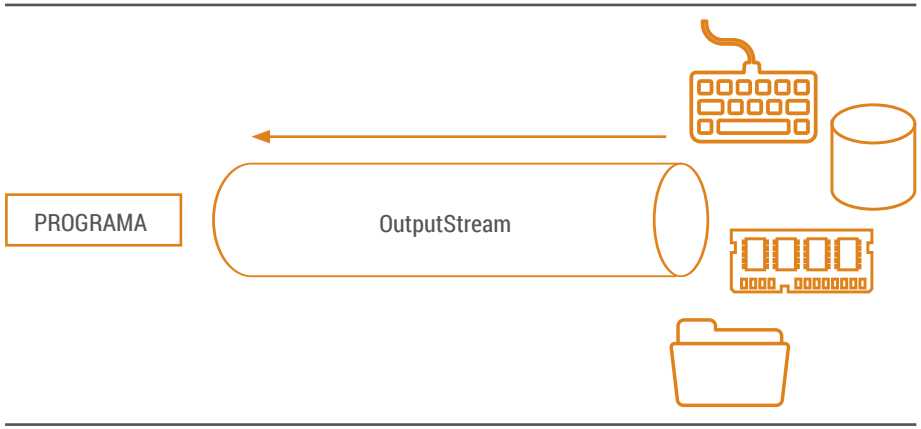


IMAGEN 10.3: OUTPUTSTREAM

A partir de estas clases se encuentran subclases concretas con diferentes funcionalidades especializadas en leer o escribir en distintos dispositivos (archivos, memoria, teclado, pantalla, etc.).

2. Flujos de bytes: leer y escribir

Los flujos de bytes se utilizan para realizar las operaciones de entrada y salida. Aunque hay muchas clases relacionadas con estos flujos, unas de las más utilizados son: **FileInputStream** y **FileOutputStream**.

En el ejemplo siguiente se muestra ejemplo que hace uso de estas dos clases para copiar un archivo de entrada en un archivo de salida:

```
package com.juan.entradasalida;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class StreamBytes {
    public static void main(String args[]) throws
IOException {
          FileInputStream entrada = null;
          FileOutputStream salida = null;
          try {
                entrada = new FileInputStream("c:/
ficheros/input.txt");
                salida = new FileOutputStream("c:/
ficheros/output.txt");
                int i;
                while ((i = entrada.read()) != -1) {
                      System.out.print((char)i);
                      salida.write(i);
                }
          }catch (IOException ioex){
                System.out.println(ioex.getMessage());
          }finally {
                if (entrada != null)    entrada.
close();
                if (salida != null)           salida.
close();
          }
    }
}
```

Los métodos definidos en **OutputStream** e **InputStream**, que heredan las clases de la jerarquía son los que se muestran en las tablas siguientes:

Método	Funcionalidad
void write(int i)	Escribe un sólo byte.
void write(byte buffer[])	Escribe un array de bytes en la salida.
void write(byte buffer[], int desplazamiento, int numbytes)	Escribe un número de bytes (numbytes), empezando en la posición (desplazamiento) del array.
void close()	Cierra el flujo de salida. Siguientes intentos de escritura generan IOException.
void flush()	Vacía el buffer de salida. Manda a su destino lo guardado en el buffer.

Tabla 10.1: Métodos definidos en OutputStream.

Método	Funcionalidad
int read()	Retorna una representación en tipo int del byte disponible en la entrada. Devuelve -1 si se ha llegado al final del flujo.
int read(byte buffer[])	Igual que el anterior pero intenta leer buffer. length bytes.
int read(byte buffer[], int desplazamiento, int numbytes)	Intenta leer numbytes bytes desde la posición desplazamiento del array buffer. Devuelve -1 si se ha llegado al final del flujo.
void close()	Cierra el flujo de entrada. Siguientes intentos de lectura generan IOException.
int available()	Retorna número de bytes disponibles en la entrada.
void mark(int numbytes)	Añade una marca en el punto actual del flujo de entrada que es válida hasta que lean un número de bytes.
long skip(long numerobytes)	Ignora el número de bytes del flujo de entrada indicados en el parámetro que recibe.
boolean markSupported()	Devuelve true si el flujo de entrada soporta mark y reset.
void reset()	Restablece el puntero de entrada a la marca definida previamente con mark.
long skip(long numerobytes)	Ignora el número de bytes del flujo de entrada indicados en el parámetro que recibe.

Tabla 10.2: Métodos definidos en InputStream.

3. Flujos de caracteres: leer y escribir

Los flujos de byte se utilizan para realizar la entrada y salida de bytes de 8 bits, y los flujos de caracteres se utilizan para realizar la entrada y salida de 16-bit Unicode.

Hay muchas clases relacionadas con los flujos de caracteres, pero dos de las clases más utilizadas son: **FileReader** y **FileWriter**.

Internamente **FileReader** utiliza **FileInputStream** y **FileWriter** utiliza **FileOutputStream** pero la principal diferencia es que **FileReader** lee dos bytes cada vez y **FileWriter** escribe dos bytes en cada operación.

El ejemplo anterior se puede cambiar fácilmente para que utilice las clases de **FileInputStream** y **FileOutputStream**, siendo la lógica del programa la misma.

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

FileReader entrada = null;
    FileWriter salida = null;
    try {
          entrada = new FileReader(“c:/ficheros/input.
txt”);
          salida = new FileWriter(“c:/ficheros/output.
txt”);
          int i;
          while ((i = entrada.read()) != -1) {
                System.out.print((char)i);
                salida.write(i);
          }
```

4. Clases de flujos de bytes

La jerarquía de clases de lectura y escritura de flujos de bytes parte de las clases abstractas **InputStream** y **OutputStream**:

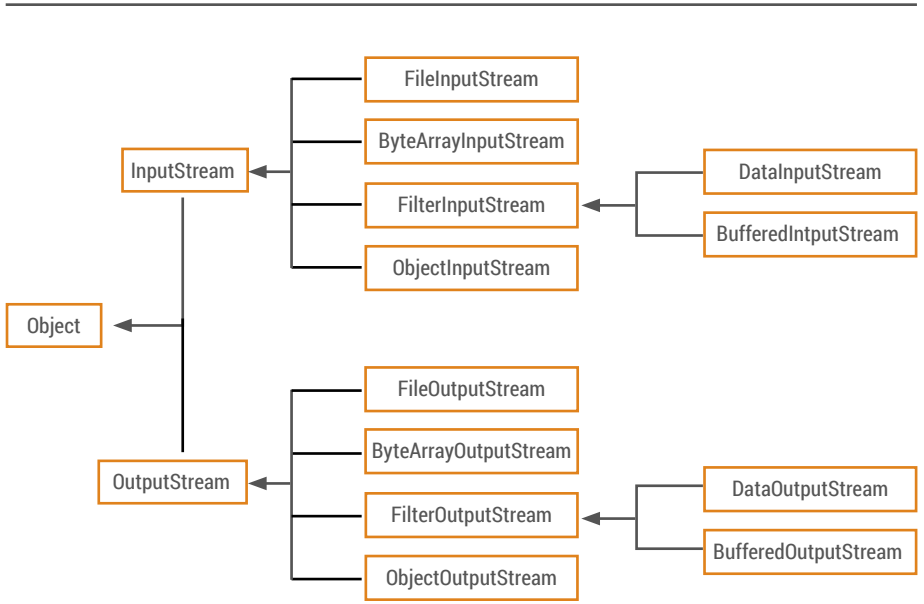


IMAGEN 10.4: JERARQUÍA DE CLASES DE FLUJO DE BYTES

5. Clases de flujos de caracteres

La jerarquía de clases de lectura y escritura de flujos de caracteres parte de las clases abstractas **Reader** y **Writer**:

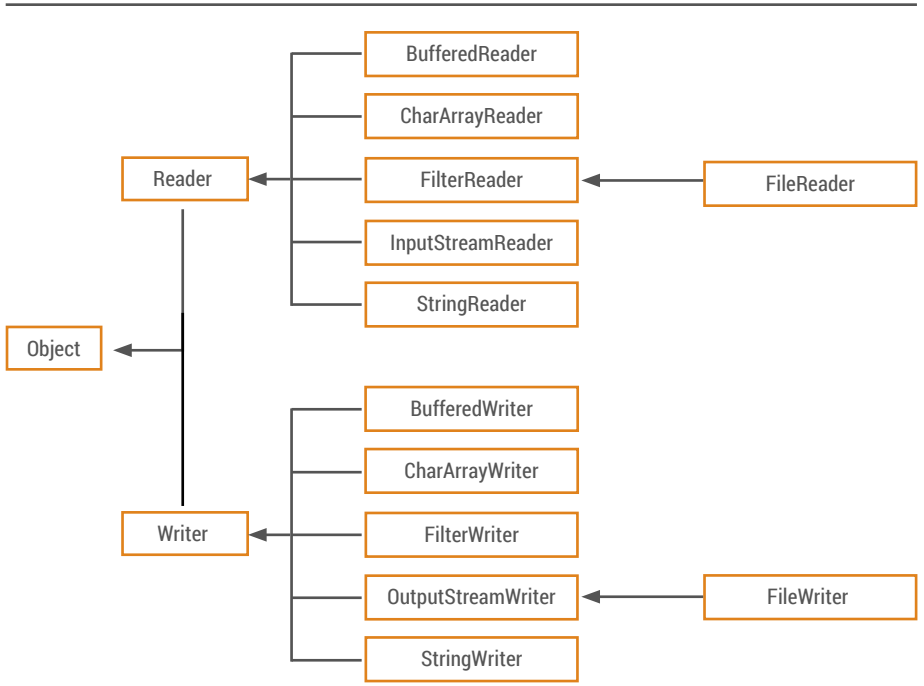


IMAGEN 10.5: JERARQUÍA DE CLASES DE FLUJO DE CARACTERES.

6. Leer y escribir datos binarios

Para leer y escribir otros tipos de datos distintos a los bytes de caracteres, como son int, double o short, se utilizan las clases **DataInputStream** y **DataOutputStream** que permiten leer y escribir valores binarios de tipos primitivos del lenguaje.

DataInputStream implementa la interfaz **DataInput**, en la tabla siguiente se muestran algunos de sus métodos más utilizados:

Métodos (lanzan IOException)
boolean readBoolean()
byte readByte()
char readChar()
short readShort()
int readInt()
long readLong()
float readFloat()
double readDouble()

Tabla 10.3:Métodos de entrada de DataInput.

DataOutputStream implementa la interfaz **DataOutput**, en la tabla siguiente se muestran algunos de sus métodos más utilizados:

Métodos (lanzan IOException)
void writeBoolean(boolean valor)
void writeByte(byte valor)
void writeChar(char valor)
void writeShort(short valor)
void writeInt(int valor)
void writeLong(long valor)
void writeFloat(float valor)
void writeDouble(double valor)

Tabla 10.4:Métodos de salida de DataOutput.

En el código siguiente se muestra un ejemplo en el que se escriben unos valores de tipos primitivos en un fichero, después se leen esos alores del fichero y se muestran en pantalla.

```
package com.juan.entradasalida;

import java.util.Random;
import java.io.*;

public class EscribirBinario {
    public static void main(String args[])throws
IOException {
          Random aleatorio = new Random();
          boolean b = aleatorio.nextBoolean();
          int n = aleatorio.nextInt();
          long l = aleatorio.nextInt()*10L;
          float f = aleatorio.nextFloat();

          //escribir datos binarios
          DataOutputStream escribir=null;
          try {
                escribir = new DataOutputStream(new
FileOutputStream(
                           "C:/ficheros/outputDatos.
txt"));
                escribir.writeBoolean(b);
                escribir.writeInt(n);
                escribir.writeLong(l);
                escribir.writeFloat(f);
          } catch (IOException ioex) {
                System.out.println(ioex.getMessage());
          } finally {
                if (escribir != null) {
                      escribir.close();
                }
          }
```

En este código se utilizan los constructores de **DataOutputStream** y **DataInputStream**, que reciben como parámetro **OutputStream** o **InputStream** respectivamente. Como quiera que se va a escribir o leer de un archivo, dicho parámetro es un objeto **FileOutputStream** o **FileInputStream** en cada uno de los dos casos.

```
//leer datos binarios
          DataInputStream leer=null;
          try {
          leer = new DataInputStream(new
FileInputStream(
                    “C:/ficheros/outputDatos.txt”));
          System.out.println(“Booleano: “+leer.
readBoolean());
          System.out.println(“Entero: “+leer.
readInt());
          System.out.println(“Long: “+leer.
readLong());
          System.out.println(“Float: “+leer.
readFloat());
          } catch (IOException ioex) {
          System.out.println(ioex.getMessage());
          } finally {
                    if (leer != null) {
                              leer.close();
                    }
          }
     }
}
```

7. Leer y escribir en archivos de acceso aleatorio

En java para trabajar con archivos en forma de acceso aleatorio hay que utilizar la clase **RandomAccessFile**. Esta clase encapsula una archivo de acceso aleatorio.

No deriva ni de **InputStream** ni de **OutputStream**, sino de Object, pero si implementa las interfaces **DataInput** y **DataOutput**, que definen las operaciones básicas de entrada y salida.

Sus constructores son:

RandomAccessFile(File file, String modo) //File clase que encapsula un archivo
RandomAccessFile(String nombre_archivo, String modo)
Ambos lanzan la excepción FileNotFoundException

El String del segundo parámetro de ambos constructores define la forma de acceso al archivo, siendo sus valores permitidos los que se indican en la tabla siguiente

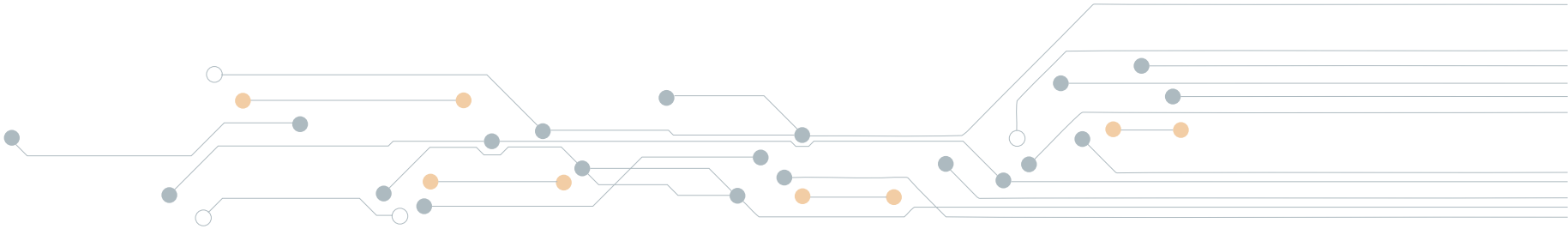
Valor	Significado
r	Abre para lectura solamente. La invocación de cualquiera de los métodos de escritura del objeto creado lanza la excepción IOException.
rw	Abre para lectura y escritura. Si el archivo no existe se intenta crear.
rws	Abre para lectura y escritura, como con "rw", todos los datos se actualizan inmediatamente en el dispositivo de almacenamiento de forma segura, aunque puede que de forma lenta.
rwd	Abrir para leer y escribir, al igual que con "rw", igual que el anterior pero reduce el número de operaciones de entrada y salida, sólo requiere la actualización del fichero en las operaciones de escritura

Tabla 10.5: Modos de acceso con RandomAccessFile.

Para acceder de forma aleatoria a los datos contenidos en el fichero, la clase **RandomAccessFile** dispone de varios métodos, entre ellos:

Método	Funcionalidad
long getFilePointer()	Devuelve la posición actual del puntero del fichero. Indica la posición (en bytes) donde se va a leer o escribir.
long length()	Devuelve la longitud del fichero en bytes.
void seek(long pos)	Coloca el puntero del fichero en una posición pos determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 indica el principio del fichero. La posición length() indica el final del fichero.

Tabla 10.6: Métodos para trabajar con puntero a posición en ficheros con RandomAccessFile



Además de los métodos implementados de las interfaces **DataInput** y **DataOutput**, dispone de métodos de lectura/escritura:

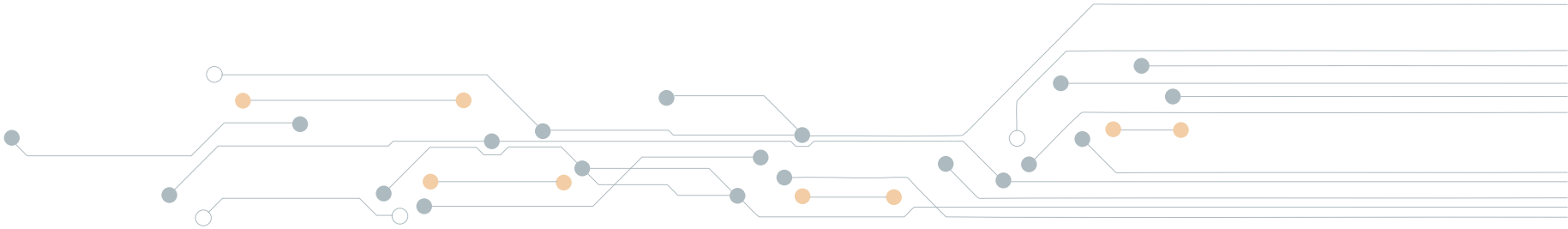
Método	Funcionalidad
int read()	Devuelve el byte leído en la posición marcada por el puntero. Devuelve -1 si alcanza el final del fichero. Se debe utilizar este método para leer los caracteres de un fichero de texto.
String readLine()	Devuelve la cadena de caracteres que se lee, desde la posición marcada por el puntero, hasta el siguiente salto de línea que se encuentre.
void write(int b)	Escribe en el fichero el byte indicado por parámetro. Se debe utilizar este método para escribir caracteres en un fichero de texto.
void writeBytes(String s)	Escribe en el fichero la cadena de caracteres indicada por parámetro.

Tabla 10.7: Métodos para leer y escribir en ficheros con RandomAccessFile

En el ejemplo siguiente se introduce un número entero por teclado y se añade al final de un fichero binario que contiene números enteros. El programa utiliza una función para mostrar el contenido del fichero.

```
package com.juan.entradasalida;
import java.io.*;
import java.util.Scanner;
public class FicheroRandom {
    public static void main(String[] args) {
          Scanner sc = new Scanner(System.in);
          RandomAccessFile fichero = null;
          int numero;
          try {
                fichero = new RandomAccessFile
                      (“c:/ficheros/numeros.dat”,
“rw”);
                mostrarFichero(fichero);
                System.out.print(“Introduce un número
entero: “);
                numero = sc.nextInt();
                fichero.seek(fichero.length());
                fichero.writeInt(numero);
                mostrarFichero(fichero);
          } catch (IOException ex) {
                System.out.println(ex.getMessage());
          } catch (InputMismatchException ex) {
                System.out.println(“No se ha
introducido un numero”);
          } finally {
                try {
                      if (fichero != null) {
                            fichero.close();
```

```
                    }
                    sc.close();
                } catch (IOException e) {
                    System.out.println(e.
getMessage());
                }
            }
        }

public static void mostrarFichero(RandomAccessFile
fichero) {
            int n;
            try {
                fichero.seek(0);
                System.out.println("Inicio de
fichero");
                while (true) {
                    n = fichero.readInt();
                    System.out.println(n);
                }
            } catch (EOFException e) {
                System.out.println("Fin de fichero");
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
            }
        }
}
```



8. Obtener datos de la entrada con Scanner

Como se ha podido ir comprobando, la clase **Scanner** provee métodos para leer valores de entrada de varios tipos. Hasta este momento los valores de entrada se obtenían sólo a través del teclado, sin embargo **Scanner** puede obtenerlos de otras fuentes como pueden ser datos almacenados en un archivo o una cadena.

En la imagen siguiente se muestra como se puede obtener (abrir) un objeto Scanner asociada a otras fuentes.

```
Scanner teclado = new Scanner(System.in);
Scanner scanner =new Scanner("3 4 78,5 HOLA 21");
Scanner fichero = new Scanner(new File(carpeta,
archivo));
```

CREACIÓN

IMAGEN 10.6: CREACIÓN DE OBJETOS SCANNER.

Cualquier objeto Scanner cuya fuente no sea System.in debe ser cerrado, como sucede con cualquier otro flujo de datos estudiado.

```
scanner.close()
```

Todos los scanner. Excepto Sytem.in

CIERRE

IMAGEN 10.6: CERRAR OBJETO SCANNER.

Los objetos Scanner se pueden configurar en tres características:

- Delimitadores de los datos, con la función **useDilimiter(String plantilla)**.
- Localización, con la función **useLocale(Locale localizacion)**.
- Base del sistema de representación numérica, con la función **useRadix(int radix)**.

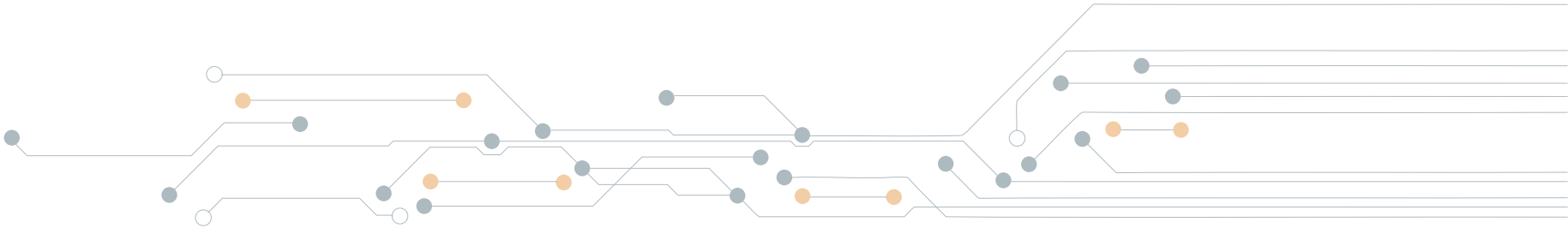


IMAGEN 10.7: CONFIGURACIÓN DE SCANNER.

La función **reset()**, establece la configuración por defecto de Scanner, con espacios y "\n" como delimitadores, la localización por omisión ("ES") y la base numérica decimal.

El uso de los objetos Scanner, una vez configurados, si ello es necesario, se hace con dos tipos de funcionalidades:

- De lectura, son las funciones ya utilizadas

  nextXXX(), next(): ignoran delimitadores a la izquierda y finalizan la lectura cuando encuentran un delimitador.

  nextLine():Lee hasta encontrar "\n", lo lee, lo quita y no lo almacena.

- De comprobación, devuelven un valor true si en el buffer hay una entrada del tipo de la comprobación, son las funciones **hashNextXXX()** y **hashNext()**.

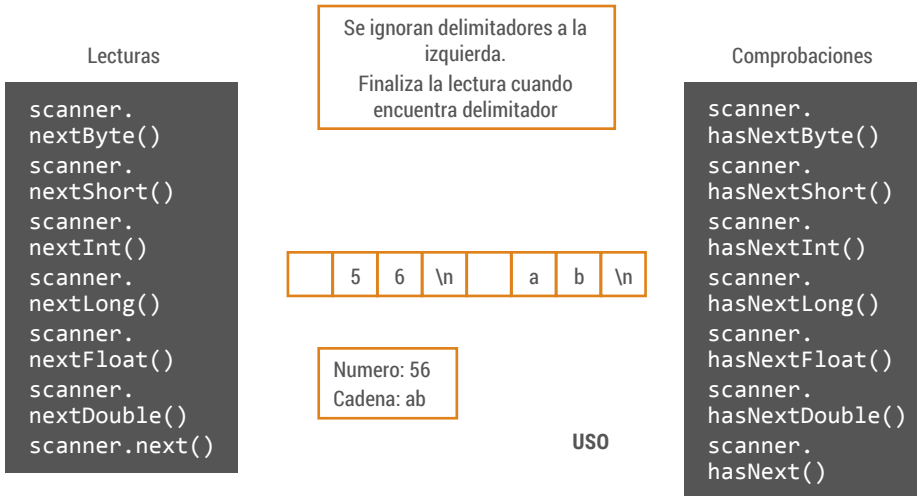


IMAGEN 10.8: USO DE OBJETOS SCANNER.

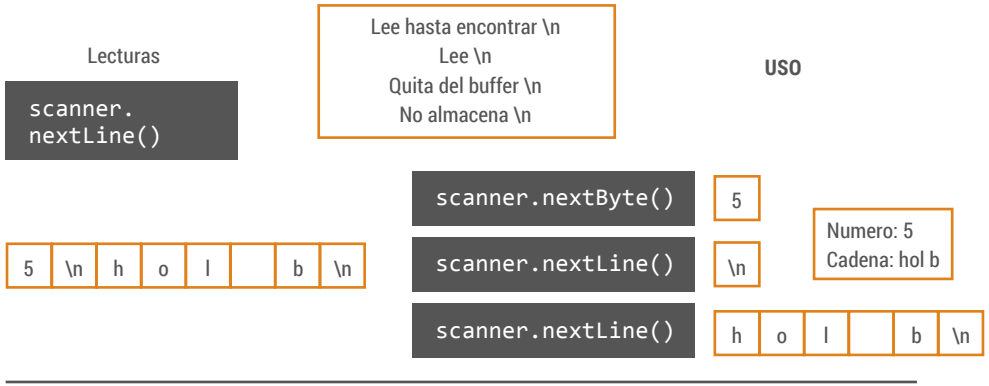


IMAGEN 10.9: USO DE OBJETOS NEXTLINE.

En el código siguiente se prueba el uso de delimitadores, localización, base numérica y lectura de una fuente de datos String.

```
package com.juan.entradasalida;
import java.util.Locale;
import java.util.Scanner;
import java.util.regex.MatchResult;
public class ScannerPruebas {
    public static void main(String[] args) {
          Scanner teclado = new Scanner(System.in);
          String string = new String("3 2 4 AF Pepe
8");
          Scanner scString = new Scanner(string);
          while (scString.hasNext()) {
                String cadena = scString.next();
                System.out.print(cadena + "-");
          }
scString.close();
          System.out.println("");
          System.out.println("Cadena: " + string);
          Scanner scBase = new Scanner(string);
          scBase.useRadix(16);
          System.out.println("La base numérica es: "
+ scBase.radix());
          while (scBase.hasNextInt()) {
                int n = scBase.nextInt();
                System.out.print(n + " ");
          }
          scBase.close();
          System.out.println("");

          String stringDelimiter = "aa /     /// bb/
cc";
          System.out.println("Cadena: " +
stringDelimiter);
          Scanner scDelimiter = new
Scanner(stringDelimiter);
          // scDelimiter.useDelimiter("\\s*\\+\\s*");
          scDelimiter.useDelimiter("/");
          System.out.println("El delimitador usado
es: "
                      + scDelimiter.delimiter());
          while (scDelimiter.hasNext()) {
                String s = scDelimiter.next();
                System.out.print(s + "-");
          }
          scDelimiter.close();
          System.out.println("");
```

```
        String stringLocale = "1.234,6 0,888
1_344_899,098";
        System.out.println("Cadena: " +
stringLocale);
        Scanner scLocale = new
Scanner(stringLocale);
        scDelimiter.useLocale(new Locale("ES"));
        while (scLocale.hasNextDouble()) {
              double d = scLocale.nextDouble();

              System.out.print(d + " ");
        }
        scLocale.close();
        System.out.println("");
        teclado.useDelimiter("=");
        System.out.print("teclea cadenas con
separador"
                          + teclado.delimiter()
                    + " : ");
        while (teclado.hasNext()) {
              String s = teclado.next();
              System.out.print(s + " ");
              if (s.equals("FIN")) {
                    teclado.reset();
                    break;
              }
        }
        System.out.println("");
        // teclado.useDelimiter("\\
p{javaWhitespace}+")
        // .useLocale(Locale.getDefault(Locale.
Category.FORMAT))
        // .useRadix(10);
```

```
        System.out.println(

        "Reset reinicia delimitador, localizacion y
base numerica");
        System.out.println("Delimitador es: " +
teclado.delimiter());
        System.out.println("La localizacion es: "
                    + teclado.locale().
getCountry());
        System.out.println("La base numerica es: "
+ teclado.radix());
    }
}
```

En el ejemplo siguiente se utiliza Scanner con un archivo como fuente de datos.

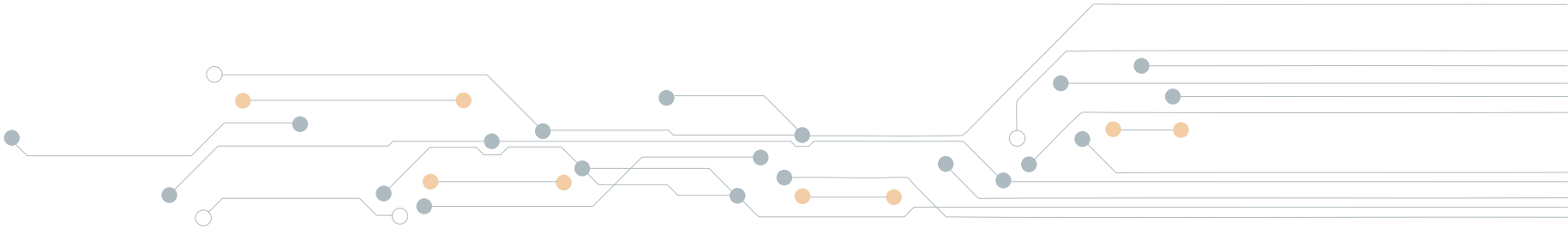
```
package com.juan.entradasalida;
import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.Scanner;
public class ScannerLeerFichero {
    public static void main(String[] args) throws IOException{
      String carpeta=”C:\\ficheros\\”;
      String archivo=”leerfichero.txt”;
      //Path path =
      //FileSystems.getDefault().getPath(carpeta, archivo);
     //Scanner leerfichero = new Scanner(path);
        File fichero = new File(carpeta+archivo);

        Scanner leerfichero = new Scanner(fichero);
        leerfichero.useDelimiter(“=”);
        while (leerfichero.hasNext()) {
            String cadena = leerfichero.next();
            System.out.print(cadena + “ “);
        }
        leerfichero.close();
    }
}
```

9. try-with-resources

Hasta este momento los recursos como los archivos, se han gestionado dentro de bloques **try-catch-finally**, de forma que dentro del bloque try se utilizaban los recursos y en el bloque **finally** se cerraban todos los recursos abiertos. Si se olvidan cerrar los recursos se pueden producir problemas de memoria y rendimiento.

```
try     (       TipoRecurso recurso01 = new TipoRecurso(parametros);
        TipoRecurso recurso02 = new TipoRecurso(parametros);
        OtroTipoRecurso recurso03 = new OtroTipoRecurso(parametros)
  )
{
        //sentencias que utilizan los recursos
}
```

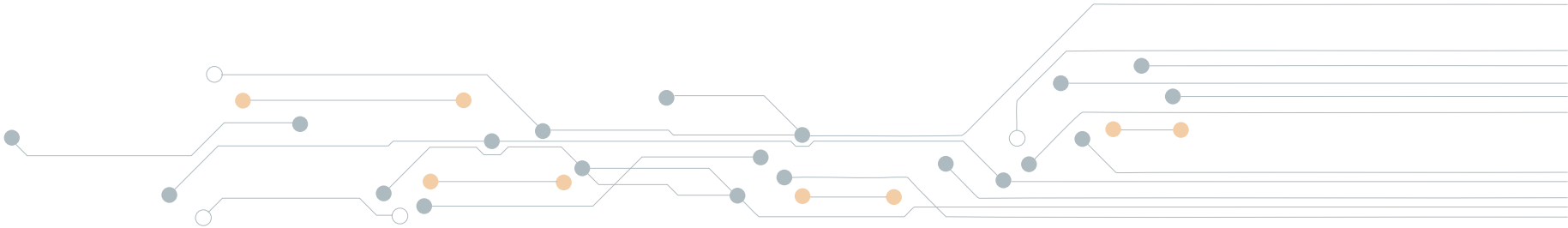
A partir de la versión 7 de Java se añade la funcionalidad **try-with-resources**, que permite indicar a la derecha de try, entre paréntesis y separados por ";" la creación de los recursos a utilizar. Así de esta manera no se necesita incluir en bloque **finally** las llamadas a las funciones que cierran los recursos.



En el ejemplo siguiente, se muestra la utilización de **try-with-resources** en una función que utiliza dos ficheros, copiando el contenido de uno en otro.

```
package com.juan.entradasalida;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
public class TryWithReaources {
    public static void main(String []args){
          File origen = new File("C:/ficheros/origen.
txt");
          File destino = new File("C:/ficheros/destino.
txt");
          copiarFicheros(origen, destino);
    }
```

```
private static void copiarFicheros(
                    File origen, File destino) {
        try (InputStream forigen = new
FileInputStream(origen);
             OutputStream fdestino =
                 new FileOutputStream(destino)){
             byte[] buf = new byte[9000];
             int i;
             while ((i = forigen.read(buf)) != -1) {
                 fdestino.write(buf, 0, i);
             }
        }
        catch (Exception e) {
             e.printStackTrace();
        }
    }
}
```



Telefónica

EDUCACIÓN DIGITAL