UT9. LECTURA ESCRITURA DE INFORMACIÓN

Módulo: PROGRAMACIÓN

Curso 2022/2023. 1° DAM

Ruth Lospitao Ruiz

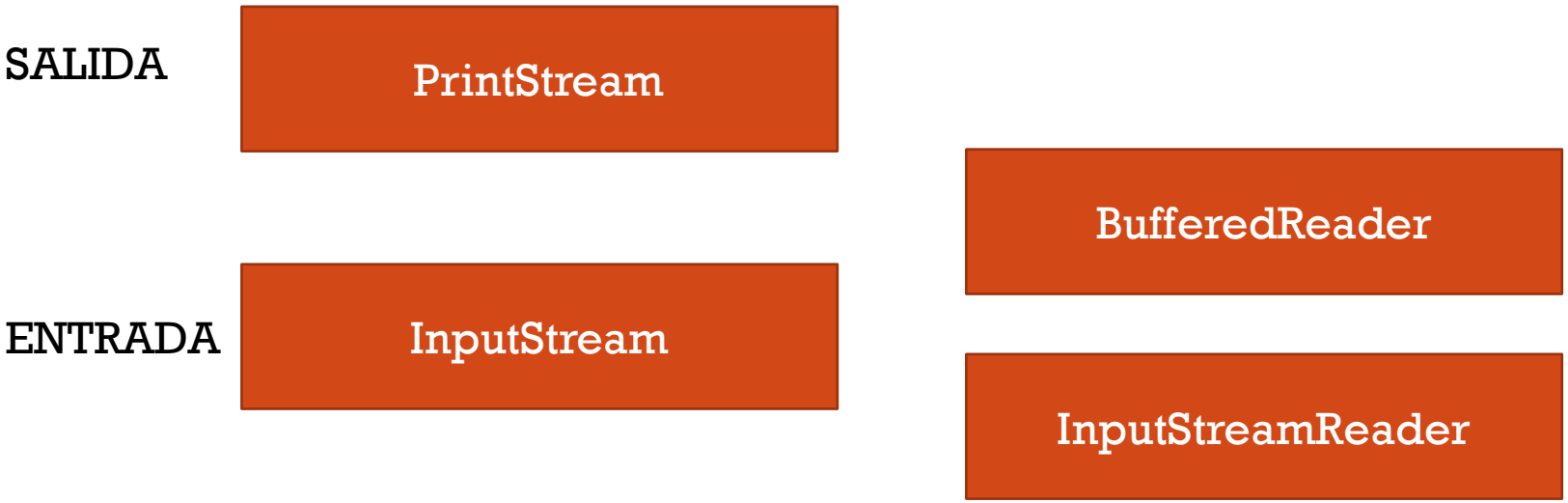
CONTENIDOS

- Entrada/sálida estándar
  - Clases envoltorio
- Flujos (streams)
- Ficheros
- Ficheros de texto
- Serialización de objetos

ENTRADA SALIDA ESTÁNDAR

INTRODUCCIÓN A ENTRADA SALIDA DE DATOS

- ¿Una de las operaciones más habituales que tiene que realizar un programa Java es intercambiar datos con el exterior. Para ello, el paquete java.io de Java SE incluye una serie de clases que permiten gestionar la entrada y salida de datos en un programa, independientemente de los dispositivos utilizados para el envío/recepción de los datos.



SALIDA DE DATOS

- El envío de datos al exterior se gestiona a través de la clase **PrintStream,** utilizándose un objeto de la misma para acceder al dispositivo de salida. Posteriormente, con los métodos proporcionados por esta clase, podemos enviar la información al exterior.
- El proceso de envío de datos a la salida debe realizarse siguiendo dos pasos.

1. Obtención del objeto PrintStream. Se debe crear un objeto PrintStream asociado al dispositivo de salida, la forma de hacerlo dependerá del dispositivo en cuestión. La clase **System** proporciona el atributo estático ***out*** que contiene una referencia al objeto PrintStream asociado a la salida estándar, representada por la consola.

2. **Envío de datos al stream.** La clase PrintStream dispone de los métodos ***print(String cadena)*** y ***println(String cadena)*** para enviar una cadena de caracteres al dispositivo de salida, diferenciándose uno de otro en que el segundo añade un salto de línea al final de la cadena. Esto explica que para enviar un mensaje a la consola se utilice la expresión

SALIDA DE DATOS

- En general, para enviar datos desde un programa Java al exterior habrá que utilizar la expresión:
  - ***objeto printstream.println(dato);***
  - ***objeto printstream.print(dato)***
- Ejemplos:
  System.out.println (“Esto es un mensaje”);

SALIDA CON FORMATO: MÉTODOS FORMAT() PRINTF()

Ambos realizan la misma función y tienen exactamente el mismo formato

- ***format (String formato, Object... datos)***
- ***printf (String formato, Object... datos)***

El argumento *formato* consiste en una cadena de caracteres con las opciones de formato que van a ser aplicadas sobre los datos a imprimir.

Por otro lado, *datos* representa la información que va a ser enviada a la salida y sobre la que se va a aplicar el formato, siendo el número de estos datos variable. La sintaxis *Object ...datos* indica que se trata de un número variable de argumentos, en este caso puede tratarse de cualquier número de objetos Java.

A modo de ejemplo, dadas las siguientes instrucciones:

```
double cuad=Math.PI*Math.PI;
System.out.printf("El cuadrado de %1$.4f es %2$.2f",Math.PI, cuad);
```

La salida producida por pantalla será:
El cuadrado de 3,1416 es 9,87

Consultar y revisar ejemplos de:
https://docs.oracle.com/javase/tutorial/java/data/numberformat.html
https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html
https://docs.oracle.com/javase/tutorial/essential/io/formatting.html

SINTAXIS DE LA CADENA FORMATO

- La cadena de formato puede estar formada por un texto fijo, que será mostrado tal cual, más una serie de especificadores de formato que determinan la forma en que serán formateados los datos.
- En el ejemplo anterior, las expresiones %l$.4f y %2$.2f representan los especificadores de formato para los valores *Pi* y cuadrado de *Pi,* respectivamente.
- Los especificadores de formato para números y cadenas deben ajustarse a la sintaxis:

%[posición_argumento$][indicador][mínimo][.num_decimales] conversión

- ***Posición_argumento***. Representa la posición del argumento sobre el que se va a aplicar el formato. El primer argumento ocupa la posición 1. Su uso es opcional.
- ***Indicador.*** Consiste en un conjunto de caracteres que determina el formato de salida. Su uso es opcional. Entre los caracteres utilizados cabe destacar:
  - "-". El resultado aparecerá alineado a la izquierda.
  - '+'. El resultado incluirá siempre el signo (sólo para argumentos numéricos).
- ***Mínimo***. Representa el número mínimo de caracteres que serán presentados. Su uso también es opcional.
- ***Nmm_decimales***. Número de decimales que serán presentados, por lo que solamente es aplicable con datos de tipo float o double. Obsérvese que este valor debe venir precedido por un punto. Su uso es opcional.
- ***Conversión.*** Consiste en un carácter que indica cómo tiene que ser formateado el argumento. La tabla de la figura 2 contiene algunos de los caracteres de conversión más utilizados.

CARACTERES DE FORMATO DE SALIDA DE DATOS

Carácter	*Función*
's', 'S'	Si el argumento es nuil se formateará como "nuil". En cualquier otro caso se obtendrá argumento.toString()
'c', 'C'	El resultado será un carácter unicode
'd'	El argumento se formateará como un entero en notación decimal
'x', 'X'	El argumento se formateará como un entero en notación hexadecimal
'e', 'E'	El argumento se formateará como un número decimal en notación científica
'f'	El argumento se formateará como un número decimal

FORMATO FECHAS

- Para el formato de fechas, los especificadores de formato deben ajustarse a la sintaxis:
- *[posicion_argumento$][indicador] [mínimo] conversión*
- El significado de los distintos elementos es el indicado anteriormente. En este caso, el elemento *conversión* está formado por una secuencia de dos caracteres. El primer carácter es 't' o 'T', siendo el segundo carácter el que indica cómo tiene que ser formateado el argumento

En siguientes slides se indican los caracteres para formatear fechas

- Por ejemplo, dadas las siguientes instrucciones:

```
Calendar c = Calendar. getlnstance();

System.out.printf("%l$tH:%l$tM:%l$tS---%l$td de %l$tB"+ " de %l$ty", c);
```

- La salida producida por pantalla será la siguiente:

16:27:14---13 de noviembre de 05

CARACTERES PARA FORMATO HORAS

Carácter	***Función***
'H'	Hora del día, formateada *como* un número de dos dígitos comprendido entre 00 y 23
'I'	Hora del día, formateada como un número de dos dígitos comprendido entre 01 y 12
'M'	Minutos de la hora actual, formateados como un número de dos dígitos comprendido entre 00 y 59
'S'	Segundos de la hora actual, formateados como un número de dos dígitos comprendido entre 00 y 59

CARACTERES PARA FORMATO FECHAS

Carácter	*Función*
'B'	Nombre completo del mes
'b'	Nombre abreviado del mes
'A'	Nombre completo del día de la semana
'a'	Nombre abreviado del día de la semana
'y'	Últimos dos dígitos del aflo
'e'	Día del mes formateado como un número comprendido entre 1 y 31

INTRODUCCIÓN A ENTRADA DATOS

- La lectura de datos del exterior se gestiona a través de la clase **InputStream.** Un objeto InputStream está asociado a un dispositivo de entrada, caso de la entrada estándar (el teclado) podemos acceder al mismo a través del atributo estático *in* de la clase System.
- Sin embargo, el método *read()* proporcionado por la clase *InputStream* para la lectura de los datos, no nos ofrece la misma potencia que *print* o *println* para la escritura. La llamada a *read()* devuelve el último carácter introducido a través de dispositivo, esto significa que para leer una cadena completa sería necesario hacerlo carácter a carácter, lo que haría bastante ineficiente el código.
- Por ello, para realizar la lectura de cadenas de caracteres desde el exterior es preferible utilizar otra de las clases del paquete java.io: la clase **BufferedReader.**
- **Otra opción para la entrada de datos es el uso de la clase Scanner**

LECTURA MEDIANTE BUFFERED READER

- La lectura de datos mediante **BufferedReader** requiere seguir los siguientes pasos en el programa:

1. **Crear objeto InputStreamReader.** Este objeto permite convertir los bytes recuperados del stream de entrada en caracteres. Para crear un objeto de esta clase, es necesario indicar el objeto InputStream de entrada, si la entrada es el teclado este objeto lo tenemos referenciado en el atributo estático *in* de la clase System:

```
InputStreamReader rd;
rd=new InputStreamReader(System.in);
```

2. Crear objeto BufferedReader. A partir del objeto anterior se puede construir un BufferedReader que permita realizar la lectura de cadenas:

```
BufferedReader bf;
bf=new BufferedReader(rd);
```

3. Invocar al método readLine(). El método *readLine()* de BufferedReader devuelve todos los caracteres introducidos hasta el salto de línea, si lo utilizamos para leer una cadena de caracteres desde el teclado devolverá los caracteres introducidos desde el principio de la línea hasta la pulsación de la tecla "enter":

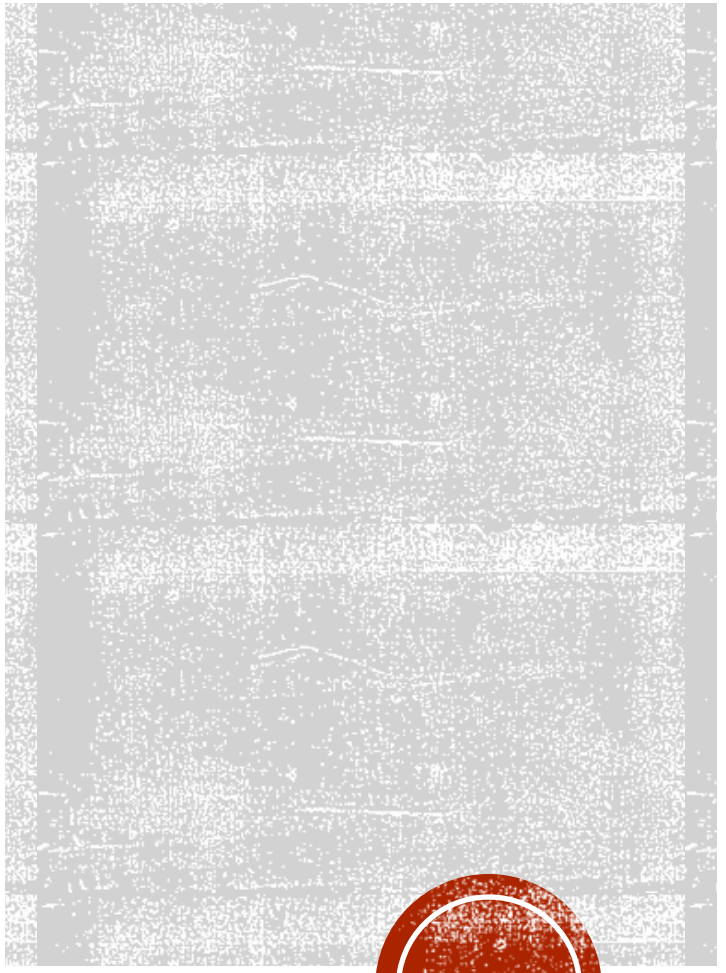
```
String s=bf.readLine();
```

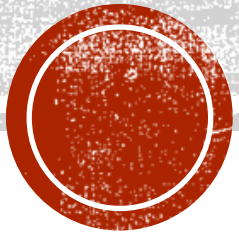
ADVERTENCIA

- Hay que mencionar un punto importante a tener en cuenta cuando se utilizan ciertos métodos de determinadas clases, se trata del hecho de que al invocar a estos métodos el programa puede lanzar una **excepción. Cuando la llamada a un método de un objeto puede lanzar una excepción el programa que utiliza ese método está obligado a capturarla o a relanzarla.**
- Éste es el caso del método *readLine()* de BufferedReader, cuya llamada; puede lanzar la excepción **lOException.** Generalmente se ha opta por relanzar la excepción, incluyendo la expresión ***throws lOException*** en la cabecera del método *main()*.

- Cuando se utiliza *readLine()* para leer datos numéricos, hay que tener en cuenta que el método devuelve los caracteres introducidos como tipo String, por lo que deberemos recurrir a los métodos de las clases de envoltorio (los métodos estáticos *parseXxx(String) se utilizan* para convertir el dato a número y poder operar con él.

CLASES ENVOLTORIO





CONCEPTO

- Para cada uno de los tipos de datos básicos, Java proporciona una clase que lo representa. A estas clases se las conoce como **clases de envoltorio,** y sirven para dos propósitos principales:
- **Encapsular un dato básico en un objeto**, es decir, proporcionar un mecanismo para "envolver" valores primitivos en un Objeto para que los primitivos puedan ser incluidos en actividades reservadas para los objetos, como ser añadido a un vector o devuelto desde un método.
- **Proporcionar un conjunto de funciones útiles para los primitivos**. La mayoría de estas funciones están relacionadas con varias conversiones: convirtiendo primitivos a String y viceversa y convirtiendo primitivos y objetos String en diferentes bases, tales como, binario, octal y hexadecimal.

CLASES ENVOLTORIO

Primitivo	Clase de envoltura	Argumentos del constructor
boolean	Boolean	boolean o String
byte	Byte	byte o String
char	Character	char
double	Double	double o String
float	Float	float, double o String
int	Integer	int o String
long	Long	long o String
short	Short	short o String

CONSTRUCCIÓN

- Todas las clases envoltorio excepto Character provee dos constructores: uno que toma un dato primitivo y otro que toma una representacion String del tipo que esta siendo construido. Por ejemplo:

```
Integer i1 = new Integer(42);

Integer i2 = new Integer("42");

Float f1 = new Float(3.14f);

Float f2 = new Float("3.14f");
```

- La clase Carácter suministra sólo un constructor que toma un char como argumento, ejemplo:

```
Character c1 = new Character('c');
```

MÉTODO VALUEOF

- El método estatico **valueOf()** es suministrado en la mayoría de las clases de envoltorio para darte la capacidad de crear objetos envoltorio. También toma una cadena como representación del tipo de Java como primero argumento. Este método toma un argumento adicional, int radix, que indica la base (por ejemplo, binario, octal, o hexadecimal) del primer argumento suministrado, por ejemplo:

```
Integer i2 = Integer.valueOf("101011", 2); // convierte 101011 a 43 y  asigna el valor  43 al objeto
// Integer i2
```

- o

```
Float f2 = Float.valueOf("3.14f"); // asigna 3.14 al objeto // Float f2
```

MÉTODO XXXVALUE()

- Los métodos **xxxValue().** Se utilizan cuando se necesita convertir el valor de un envoltorio numérico a primitivo. Todos los métodos de esta familia no tienen argumentos. Hay 36 métodos. Cada una de las seis clases envoltorio, tiene seis métodos, así que cualquier número envoltorio puede ser convertido a cualquier tipo primitivo

```
Integer i2 = new Integer(42); // Crea un nuevo objeto envoltorio
byte b = i2.byteValue(); // convierte el valor de i2 a byte
short s = i2.shortValue(); // otro método de xxxValue para números enteros
double d = i2.doubleValue(); // otro más
```

```
Float f2 = new Float(3.14f); // crea un nuevo objeto envoltorio
short s = f2.shortValue(); // convierte el valor de f2's a short
System.out.println(s); // el resultado es 3 (truncado, no redondeado)
```

MÉTODOS PARSEXXX() Y VALUEOF()

- Los seis métodos **parseXxx()** (uno por cada tipo de envoltorio) son muy parecidos a los de **valueOf().** Los dos toman una cadena como argumenta arrojando NumberFormatException (comunmente denominado NFE) si el argumento cadena no esta apropiadamente formateado, y puede convertir objetos de cadena a diferentes bases (radix), cuando el dato primitivo es cualquiera de los cuatro tipos de números enteros. La diferencia entre ambos métodos es:
  - parseXxx() devuelve el primitivo.
  - valueOf() devuelve el recién creado objeto envoltorio del tipo invocado en el método.

```
double d4 = Double.parseDouble("3.14"); // convierte un String a un primitivo
System.out.println("d4 = " + d4); // el resultado es d4 = 3.14
Double d5 = Double.valueOf("3.14"); // crea un objeto Double
System.out.println(d5 instanceof Double); // el resultado es "true"

long L2 = Long.parseLong("101010", 2); // binary String a
System.out.println("L2 = " + L2); // resultado es: L2 = 42
Long L3 = Long.valueOf("101010", 2); // binary String a Long objecto
System.out.println("L3 value = " + L3); // resultado es:  L3 value = 42
```

MÉTODO TOSTRING()

- La clase alfa, o sea, la primera clase de Java, Object, tiene un metodo toString(). Asi que todas las clases de Java tienen también este método. La idea de toString() es permitirte conseguir una representación mas significativa del objeto. Por ejemplo, si tienes una coleccion de varios tipos de objetos, puedes hacer un bucle a través de la colección e imprimir algunas de las representaciones más significativas de cada objeto usando toString(), que está presente en todas las clases. Hablaremos mas de toString() en las colecciones en el capitulo correspondiente, ahora vamos a centrarnos en como funciona toString() en las clases envoltorio, que como ya sabemos, están marcadas como clases finales (ya que son inmutables). **Todas las clases envoltorio devuelven una cadena con el valor del objeto primitivo del objeto,** por ejemplo:

```
Double d = new Double("3.14");
System.out.println("d = "+ d.toString() ); // result is d = 3.14


String d = Double.toString(3.14); // d = "3.14"
```

AUTOBOXING

- El *autoboxing* representa otra de las nuevas características del lenguaje incluidas a partir de la versión Java 5 siendo, probablemente, una de las más prácticas.
- Consiste en la encapsulación automática de un dato básico en un objeto de envoltorio, mediante la utilización del operador de asignación.
- Por ejemplo, según se ha explicado anteriormente, para encapsular un dato entero de tipo *int* en un objeto *Integer,* deberíamos proceder del siguiente modo:

```
int p = 5;
Integer n = new Integer (p);
```

- Utilizando el *autoboxing* la anterior operación puede realizarse de la siguiente forma:

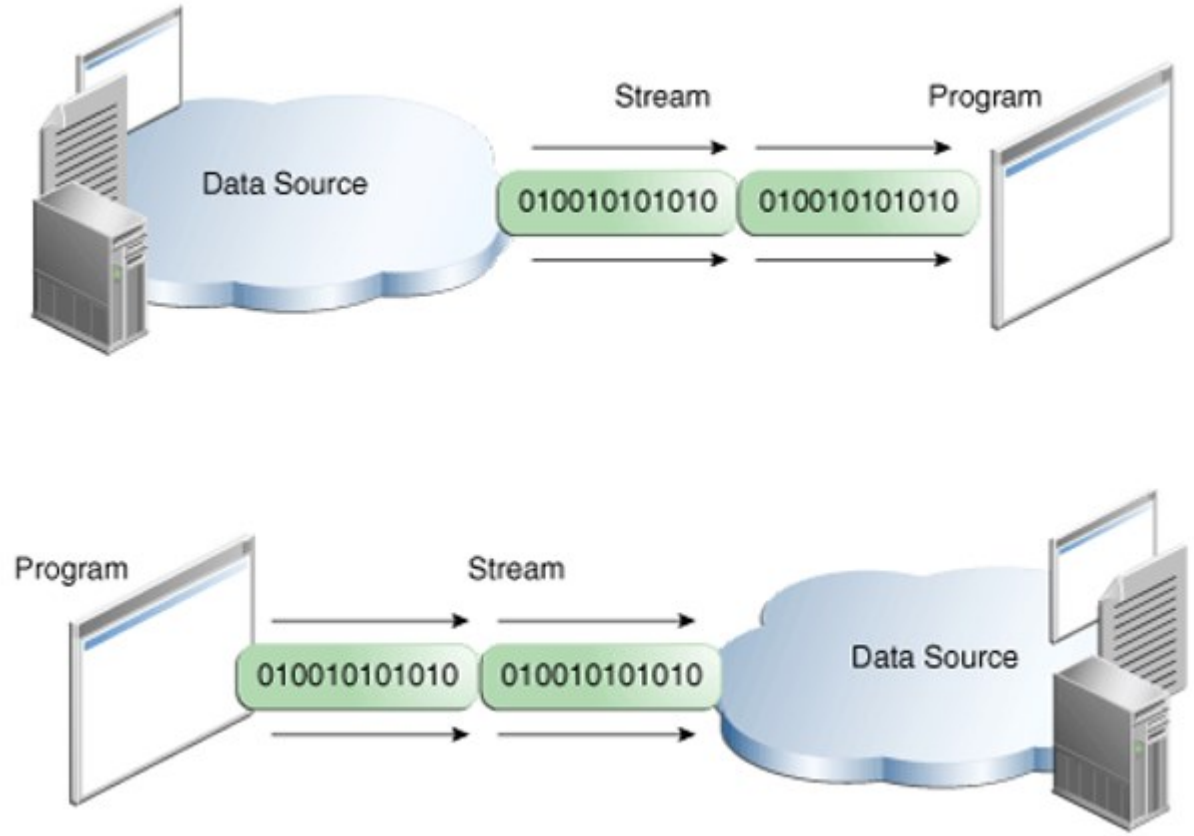
```
int p = 5; Integer n = p;
```

- Es decir, la creación del objeto de envoltorio se produce implícitamente al asignar el dato a la variable objeto.

FLUJOS DE DATOS (STREAMS)

¿QUÉ ES UN STREAM?

- Es una secuencia de bytes, uno tras otro entre un origen y un destino.



CONSIDERACIONES

- Todos los datos fluyen a través del ordenador desde una entrada hacia una salida. Este flujo de datos se denomina también stream. Hay un flujo de entrada (input stream) que manda los datos desde el exterior (normalmente el teclado) del ordenador, y un flujo de salida (output stream) que dirige los datos hacia los dispositivos de salida (la pantalla o un archivo).
- No se almacenan en ninguna parte. Lo que se haga con los bytes es responsabilidad del programador.
- No te puedes mover hacia adelante o hacia atrás en el stream, necesitas almacenarlo en un buffer primero.

TIPOS FLUJOS SEGÚN EL TIPO DATO

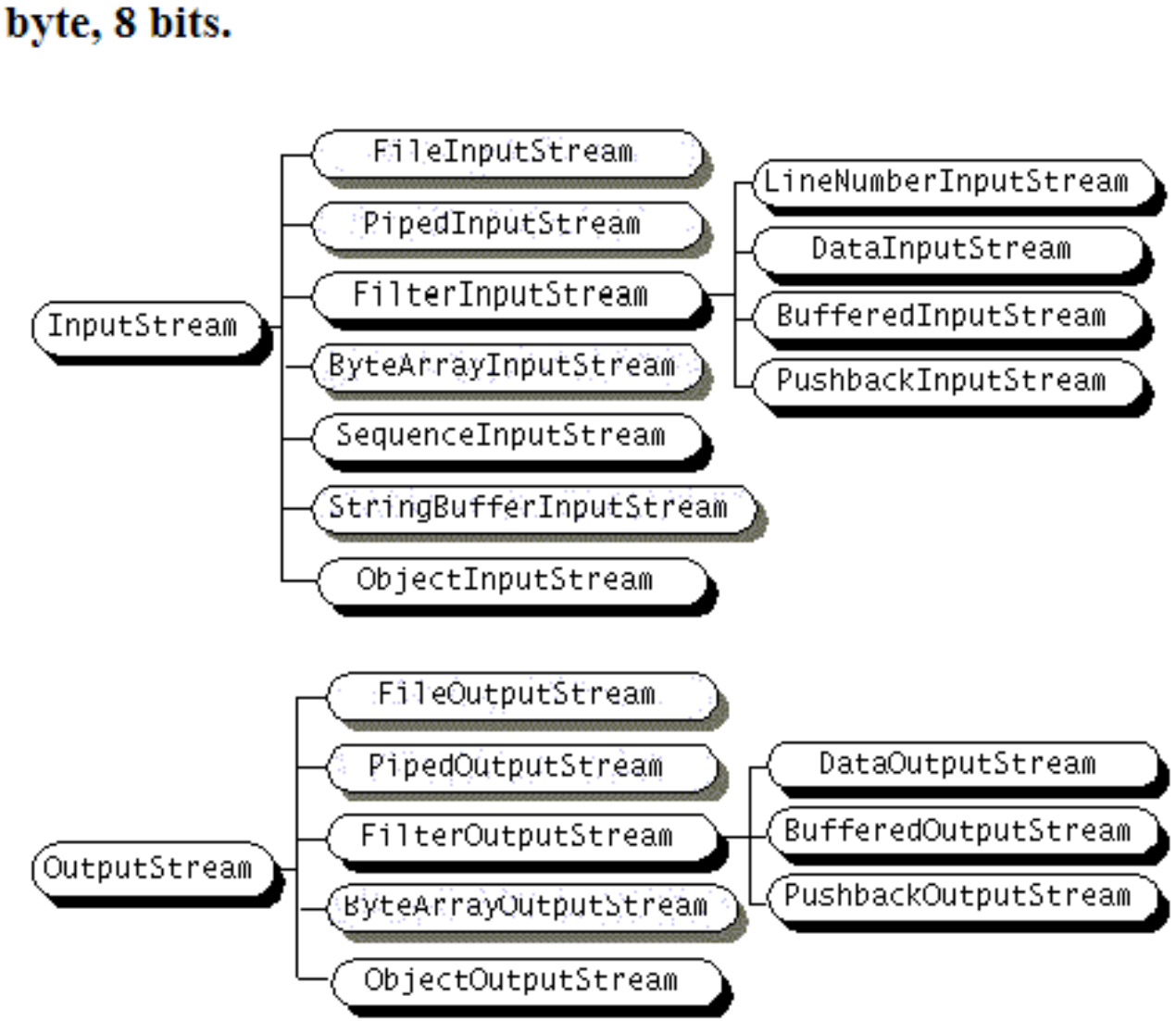
- Existen dos tipos de flujos según el tipo de dato:

Flujos binario byte (8-bits)

Flujos de caracteres char (16 bits)

FLUJOS BINARIOS BYTE (8-BIT)

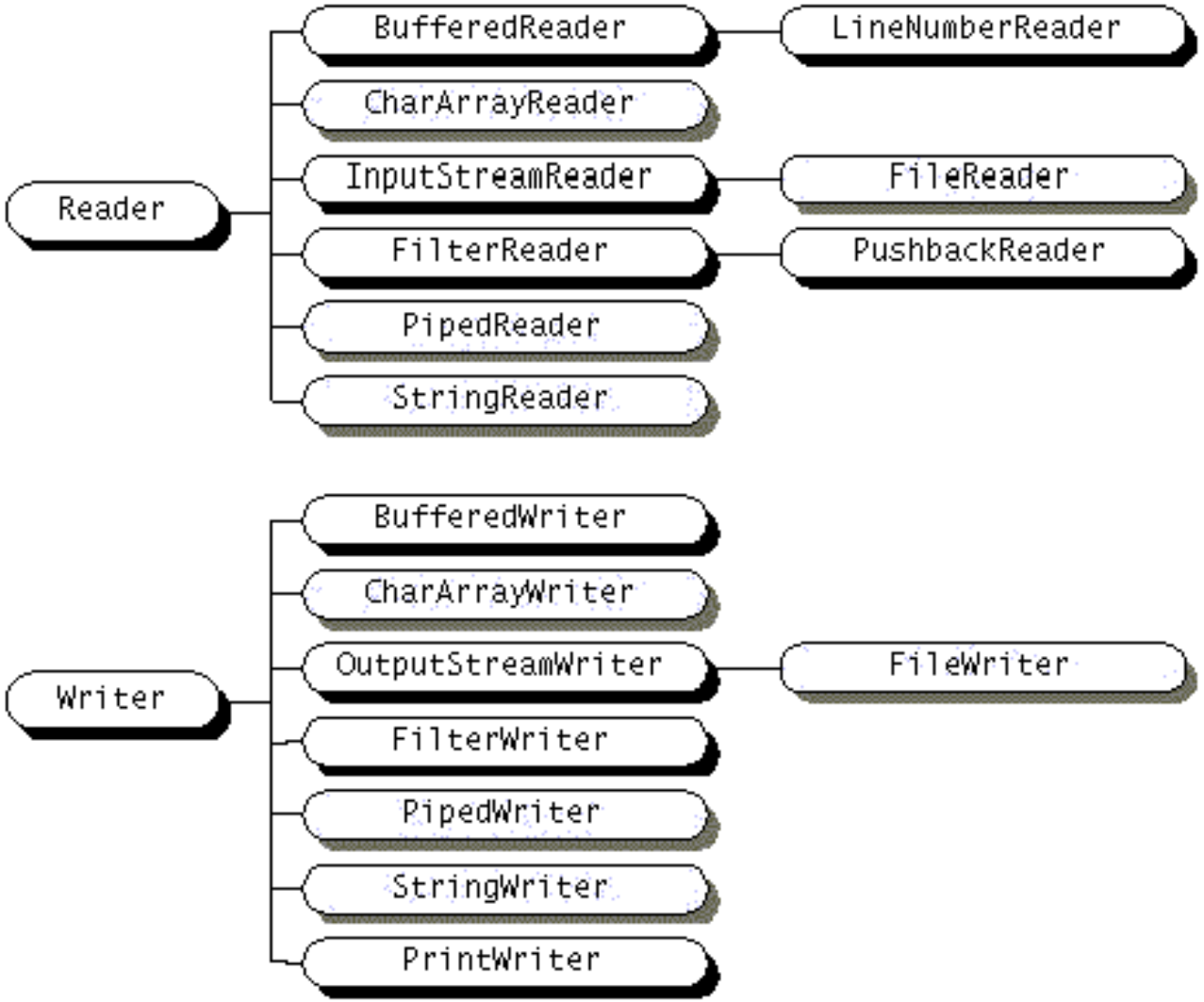
- Es el tipo de flujo más primitivo y portable, de hecho, cualquier otro tipo de flujo está construido sobre este porque hablando a bajo nivel todas las operaciones de I/O son flujos de bytes. Nos permitirá trabajar adecuadamente con datos binarios tales como archivos de imagen, sonido, etc. Las clases principales para manejar estos flujos son las clases
abstractas InputStream y OutputStream de las cuales heredan otras subclases que implementan en formas más concretas la misma tarea.



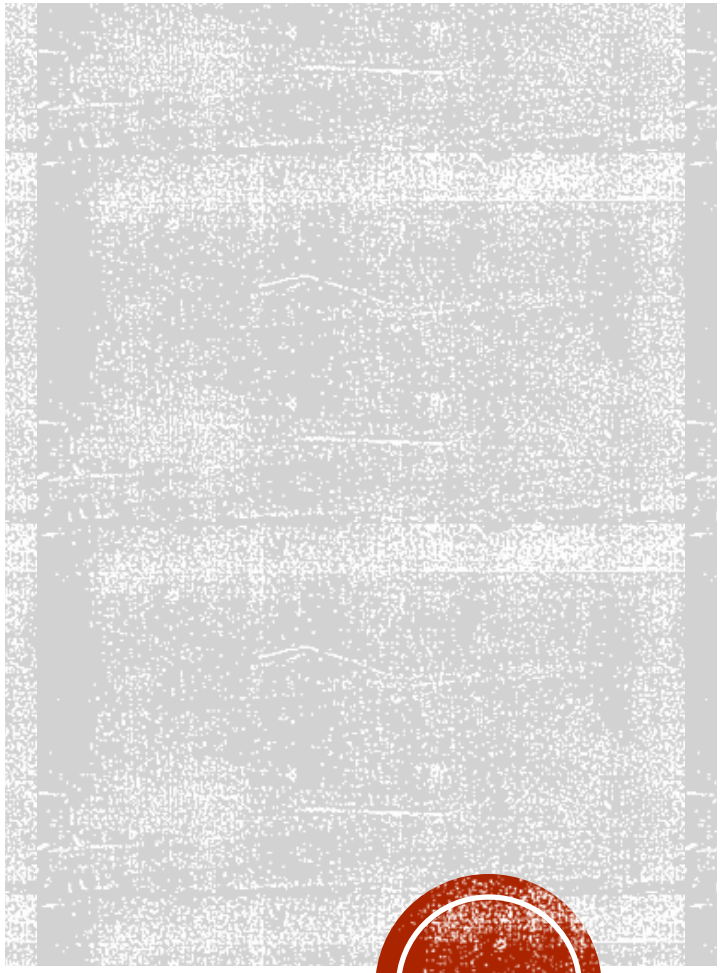
FLUJOS DE CARACTERES CHAR (16-BIT)

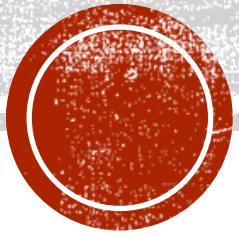
char Unicode, 16 bits

- Es un tipo de flujo de caracteres en codificación Unicode, listo para la internacionalización, ideal para trabajar con texto plano. Las clases principales para manejar estos flujos son las clases abstractas Reader y Writer de las cuales heredan otras sub-clases que implementan en formas más concretas la misma tarea. Cualquiera de estas clases realiza la conversión correspondiente de byte a char para leer o de char a byte para escribir.



FICHEROS





INTRODUCCIÓN

- Como sabemos, los ficheros se organizan en carpetas o directorios, en los que no puede haber dos archivos con el mismo nombre. Cada identificador de archivo ha de ser único y se compone de un nombre más una extensión (generalmente de tres letras) que nos indica el tipo de fichero que es: DOC, PDF, BMP, etc.

- Un fichero está formado por un conjunto de líneas o registros, cada registro se compone de un conjunto de campos que organizan la información en él contenida de una manera que conoce aquella persona u organización que haya definido el formato. Los ficheros también pueden ser de líneas de texto que se correspondan con un escrito plano sin más complicaciones.

CLASE FILE

- Como sabemos, Java tiene multitud de clases con muy distintas utilidades agrupadas en bibliotecas o paquetes, así por ejemplo ***java.io*** se ocupa de las clases para manejar la entrada/salida y será preciso importar este paquete para hacer uso de él cuando trabajemos con ficheros, ya que, entre otras, aloja a la clase ***File***.

- La clase ***File*** aglutina un grupo de funcionalidades que nos van a ser gran utilidad a la hora de trabajar con ficheros o directorios porque nos van a permitir conocer su nombre, atributos, rutas, tamaño, etc. Con ella podemos crear un directorio o un archivo. Por ejemplo, tiene tres constructores:

```
public File(String nombreFichero|path);

public File(String path, String nombreFichero|path);

public File(File path, String nombreFichero|path);
```

- La ruta o **path** puede ser absoluta o relativa. Pero hay que tener en cuenta que en Linux y en Windows se usan distintos prefijos para indicar la ruta:
  - **Linux:** "/" como por ejemplo: "/home/ejercicios/uni1/ejemplo1.txt"
  - **Windows:** letra de unidad + ":" + "\\" si la ruta es absoluta, como por ejemplo: "C:\\EJERCICIOS\\UNI1\\ ejemplo1.txt"

Atención: debemos tener en cuenta que crear un objeto File no significa que deba existir el fichero o el directorio o que el path sea correcto.

Si no existen no se lanzará ningún tipo de excepción ni tampoco serán creados.

EJEMPLOS CON EL PRIMER CONSTUCTOR

```
public File(String nombreFichero|path);
```

1. Crea un objeto File asociado al fichero *personas.dat* que se encuentra en el directorio de trabajo:

```
File f = new File("personas.dat");
```

- En este caso no se indica path. Se supone que el fichero se encuentra en el directorio actual de trabajo.

2. Crea un objeto File asociado al fichero *personas.dat* que se encuentra en el directorio *ficheros* dentro del directorio actual.

```
File f = new File("ficheros/personas.dat");
```

- En este caso se indica la ruta relativa tomando como base el directorio actual de trabajo. Se supone que el fichero *personas.dat* se encuentra en el directorio *ficheros*. A su vez el directorio *ficheros* se encuentra dentro del directorio actual de trabajo.

EJEMPLOS CON EL SEGUNDO CONSTUCTOR

```
public File(String path, String nombreFichero|path);
```

En este caso se crea un objeto File cuya ruta (absoluta o relativa) se indica en el primer String.

1. Crea un objeto File asociado al fichero *personas.dat* que se encuentra en el directorio *ficheros* dentro del directorio actual.

```
File f = new File("ficheros", "personas.dat" );
```

- En este caso se indica la ruta relativa tomando como base el directorio actual de trabajo.

2. Crea un objeto File asociado al fichero *personas.dat* dando la ruta absoluta:

```
File f = new File("/ficheros", "personas.dat" );
```

- En este caso se indica la ruta absoluta, indicada por la barra del principio.

EJEMPLOS CON EL TERCER CONSTUCTOR

```
public File(File path, String nombreFichero|path);
```

Este constructor permite crear un objeto File cuya ruta se indica a través de otro objeto File.

1. Crea un objeto File asociado al fichero *personas.dat* que se encuentra en el directorio *ficheros* dentro del directorio actual.

```
File ruta = new File("ficheros");
File f = new File(ruta, "personas.dat" );
```

2. Crea un objeto File asociado al fichero *personas.dat* dando la ruta absoluta:

```
File ruta = new File("/ficheros");
File f = new File(ruta, "personas.dat" );
```

OTROS MÉTODOS DE INTERÉS DE LA CLASE FILE

MÉTODO	DESCRIPCIÓN
boolean canRead()	Devuelve true si se puede leer el fichero
boolean canWrite()	Devuelve true si se puede escribir en el fichero
boolean createNewFile()	Crea el fichero asociado al objeto File. Devuelve true si se ha podido crear. Para poder crearlo el fichero no debe existir. Lanza una excepción del tipo IOException.
boolean delete()	Elimina el fichero o directorio. Si es un directorio debe estar vacío. Devuelve true si se ha podido eliminar.
boolean exists()	Devuelve true si el fichero o directorio existe
String getName()	Devuelve el nombre del fichero o directorio
String getAbsolutePath()	Devuelve la ruta absoluta asociada al objeto File.
String getPath()	Devuelve la ruta con la que se creó el objeto File. Puede ser relativa o no.
String getParent()	Devuelve un String conteniendo el directorio padre del File. Devuelve null si no tiene directorio padre.
File getParentFile()	Devuelve un objeto File conteniendo el directorio padre del File. Devuelve null si no tiene directorio padre.
boolean isDirectory()	Devuelve true si es un directorio válido
boolean isFile()	Devuelve true si es un fichero válido
long length()	Devuelve el tamaño en bytes del fichero. Devuelve 0 si no existe. Devuelve un valor indeterminado si es un directorio.
String[] list()	Devuelve un array de String con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto File. Si no es un directorio devuelve null. Si el directorio está vacío devuelve un array vacío.
boolean mkdir()	Crea el directorio. Devuelve true si se ha podido crear.
boolean renameTo(File dest)	Cambia el nombre del fichero por el indicado en el parámetro dest. Devuelve true si se ha realizado el cambio.

Consultar la documentación oficial: https://docs.oracle.com/javase/8/docs/api/java/io/File.html

EJEMPLO 1

- En el siguiente ejemplo, se prueban algunos de los métodos de la clase File. Para ello, se crearán dos objetos ficheros: uno de un fichero (Persona.java) que si debería existir en el mismo directorio y otro (datos_empleados.dat) que no existe.

```
import java.io.File;
public class ExampleFichero01 {
    public static void  main(String[] args) {
        File fichero1=new File("Persona.java"); //en este caso existe un fichero Persona.java en mismo directorio
        if(fichero1.exists()){
            System.out.println("Nombre del archivo "+fichero1.getName());
            System.out.println("Camino             "+fichero1.getPath());
            System.out.println("Camino absoluto    "+fichero1.getAbsolutePath());
            System.out.println("Se puede escribir  "+fichero1.canRead());
            System.out.println("Se puede leer      "+fichero1.canWrite());
            System.out.println("Tamaño             "+fichero1.length());
       }else
        System.out.println("Fichero " + fichero1.getName() + " no existe");

       File fichero2=new File("datos_empleados.sat"); //en este caso no existe el fichero
       if(fichero2.exists()){
           System.out.println("Nombre del archivo "+fichero2.getName());
           System.out.println("Camino             "+fichero2.getPath());
           System.out.println("Camino absoluto    "+fichero2.getAbsolutePath());
           System.out.println("Se puede escribir  "+fichero2.canRead());
           System.out.println("Se puede leer      "+fichero2.canWrite());
           System.out.println("Tamaño             "+fichero2.length());
        }else
            System.out.println("Fichero " + fichero2.getName() + " no existe");
    }
}
```

```
C:\Users\Usuario\Documents\OneDrive2020\OneDrive
Users\Usuario\Documents\OneDrive2020\OneDrive -
rogram Files\Java\jdk-11.0.12\bin\java.exe" -cp
\jdt_ws\code_examples_ficheros_732fad98\bin Exam
Nombre del archivo Persona.java
Camino             Persona.java
Camino absoluto    C:\Users\Usuario\Documents\Or
eros\Persona.java
Se puede escribir  true
Se puede leer      true
Tamaño             1606
Fichero datos_empleados.sat no existe
```

EJEMPLO 2

- El siguiente ejemplo muestra los ficheros del directorio actual. Usa el método list() que devuelve un array de String con los nombres de los ficheros y directorios presentes en el directorios asociado al objeto File. Como queremos que sea el directorio actual se introduce el valor "." en la variable String dir que es la que usamos para declarar el objeto f (File) con el primer constructor. También se define otro objeto File f2 para saber si el fichero obtenido es un archivo o un directorio.

```
/* Muestra los ficheros del directorio actual. */
import java.io.File;

public class ExampleFichero02 {
    public static void main(String[] args) {
        String dir = ".";  //directorio actual
        File f = new File(dir);
        String[] archivos = f.list();
        System.out.printf("Ficheros en el directorio actual: %d %n", archivos.length);
        // Nota en https://docs.oracle.com/javase/tutorial/java/data/numberformat.html aparece tabla donde consultar identificadores %
        // %d integer %f float %s string %b boolean %n salto línea
        for (int i = 0; i < archivos.length; i++) {
            File f2 = new File(f, archivos[i]);
            System.out.printf("Nombre: %s, es fichero?: %b, es directorio?:%b %n", archivos[i], f2.isFile(), f2.isDirectory());
        }
    }

}
```

EJEMPLO 3

- El siguiente ejemplo muestra información del fichero que se indica en el código, siempre y cuando esté en esa ubicación

```
import java.io.File;

public class ExampleFichero03 {
    public static void main(String[] args) {
        System.out.println("INFORMACIÓN SOBRE EL FICHERO:");
        File f = new File("c:\\Users\\Usuario\\Desktop\\code_examples_ficheros\\ExampleFichero03.java");
        if(f.exists()){
              System.out.println("Nombre del fichero: "+f.getName());
              System.out.println("Ruta: "+f.getPath());
              System.out.println("Ruta absoluta: "+f.getAbsolutePath());
              System.out.println("Se puede leer: "+f.canRead());
              System.out.println("Se puede escribir: "+f.canWrite());
              System.out.println("Tamaño: "+f.length());
              System.out.println("Es un directorio: "+f.isDirectory());
             System.out.println("Es un fichero: "+f.isFile());
            System.out.println("Nombre del directorio padre: "+f.getParent());
        }else
            System.out.println("Fichero " + f.getName() + " no existe");

    }
}
```

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero: ExampleFichero03.java
Ruta: c:\Users\Usuario\Desktop\code_examples_ficheros\ExampleFichero03.java
Ruta absoluta: c:\Users\Usuario\Desktop\code_examples_ficheros\ExampleFichero03.java
Se puede leer: true
Se puede escribir: true
Tamaño: 0
Es un directorio: false
Es un fichero: true
Nombre del directorio padre: c:\Users\Usuario\Desktop\code_examples_ficheros
```

EJEMPLO 4

- En el ejemplo 2, se listaban todos los ficheros del directorio actual. En este ejemplo, se crea un filtro para mostrar únicamente los ficheros con extensión .java
- Un filtro es un objeto de una clase que implemente el interface FilenameFilter, y tiene que redefinir la única función del interface denominada accept. Esta función devuelve un dato de tipo boolean. En este caso, la hemos definido de forma que si el nombre del archivo termina con una determinada extensión devuelve true en caso contrario devuelve false. La función endsWith de la clase String realiza esta tarea

```
import java.io.File;
import java.io.FilenameFilter;

public class Example04 {
    public static void main(String[] args) {
        File fichero = new File (".");
        String[] listaArchivos=fichero.list();

        System.out.println ("Se listan todos los ficheros");
        for(int i=0; i<listaArchivos.length; i++){
            System.out.println(listaArchivos[i]);
        }
        System.out.println ("\n\nSe listan únicamente ficheros Java");
        listaArchivos=fichero.list(new Filtro(".java"));
        for(int i=0; i<listaArchivos.length; i++){
            System.out.println(listaArchivos[i]);
        }
    }
}
```

```
class Filtro implements FilenameFilter{
    String extension;
    Filtro(String extension){ //constructor con la extensión
        this.extension=extension;
    }
    public boolean accept(File dir, String name){ // se debe implementar el método
abstracto accept
        return name.endsWith(extension);
    }
}
```

EJEMPLO 5

- En el siguiente ejemplo, se emplea el método createNewFile para crear un nuevo fichero “newFile.txt”. Este método devolverá true si pudo crearlo y false si ya existía

```
import java.io.File;

class ExampleFile05 {
  public static void main(String[] args) {

    // creación de un objeto fichero en la ubicación actual
    File file = new File("newFile.txt");

    try {


      boolean value = file.createNewFile();   // crea el fichero
      if (value) {
        System.out.println("Fichero creado.");
      }
      else {
        System.out.println("Fichero ya existe");
      }
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

CREACIÓN DE DIRECTORIOS CON FILE

- A continuación, se pretender crear un directorio. En Java deberemos de utilizar un objeto File. La ruta que debe de contener dicho objeto deberá de hacer referencia a un directorio en vez de a un archivo.

```
File directorio = new File("ruta\\directorio_nuevo");
```

- Cuando especificamos el path mediante la cadena de texto debemos de recordar que la barra debe de repetirse dos veces ya que usada de manera aislada hace referencia a una secuencia de escape.
- Una vez tenemos creado el objeto File podemos invocar a **dos métodos**:
  - **El método mkdir()** creará una carpeta solo si falta la carpeta principal inmediata. Si falta alguno de los padres de la carpeta, el método mkdir() devolverá falso.
  - **El método mkdirs()** creará todas las carpetas si faltan todas. Si falta una carpeta, creará la carpeta que falta y luego creará las carpetas secundarias.
- **Al contrario de lo que sucede con la creación de ficheros, a la hora de crear un directorio no estamos obligados a capturar la excepción IOException.**

EJEMPLO 6

```java
/* Ejemplo creación de un nuevo directorio en el Escritorio */
import java.io.File;

public class ExampleFichero06 {
    public static void main(String[] args) {
        //en este caso creará un directorio en el escritorio, si existe indica que ya existe
        File f = new File("c:\\Users\\Usuario\\Desktop\\directorio_nuevo");

        // comprobar si el directorio puede ser creado
        if (f.mkdirs()) {
            System.out.println("Directorio creado");
        }
        else {
            System.out.println("Directorio ya existe");
        }

        // en este segundo ejemplo se indica Desktop2 el cual no existe
        // la primera vez creará los dos directorios.
        File f2 = new File("c:\\Users\\Usuario\\Desktop2\\directorio_nuevo");

        if (f2.mkdirs()) {
            System.out.println("Directorio creado");
        }
        else {
            System.out.println("Directorio ya existe");
        }
    }
}
```

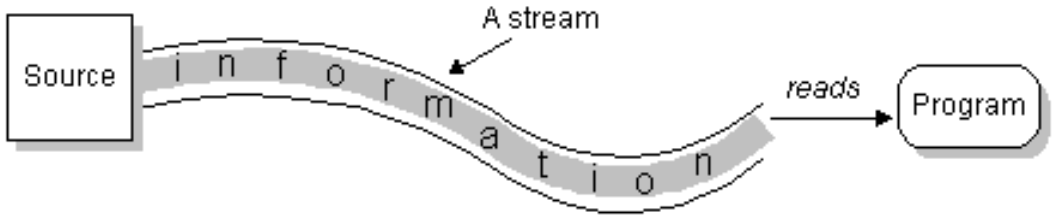
FICHEROS DE TEXTO

CONCEPTOS GENERALES

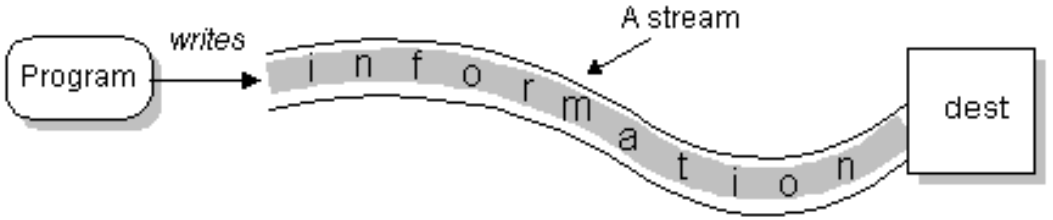
- Normalmente las aplicaciones necesitan **leer o escribir información desde o hacia una fuente externa de datos**.
- La información puede estar en cualquier parte, en un fichero, en disco, en algún lugar de la red, en memoria o en otro programa.
- También puede ser de cualquier tipo: objetos, caracteres, imágenes o sonidos.
- La comunicación entre el origen de cierta información y el destino se realiza mediante un **stream** (flujo) de información. Un stream es un objeto que hace de intermediario entre el programa y el origen o destino de la información.

CONCEPTOS GENERALES

- Para traer la información, un programa abre un stream sobre una fuente de información (un fichero, memoria, un socket) y lee la información, de esta forma:



- De igual forma, un programa puede enviar información a un destino externo abriendo un stream sobre un destino y escribiendo la información en este, de esta forma:



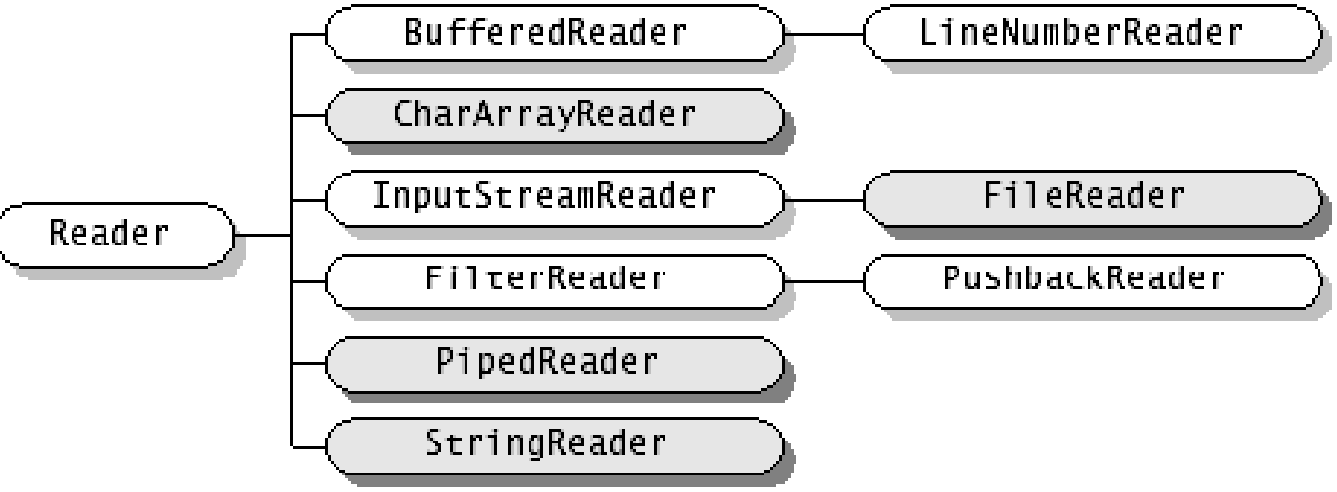
- Los algoritmos para leer y escribir:

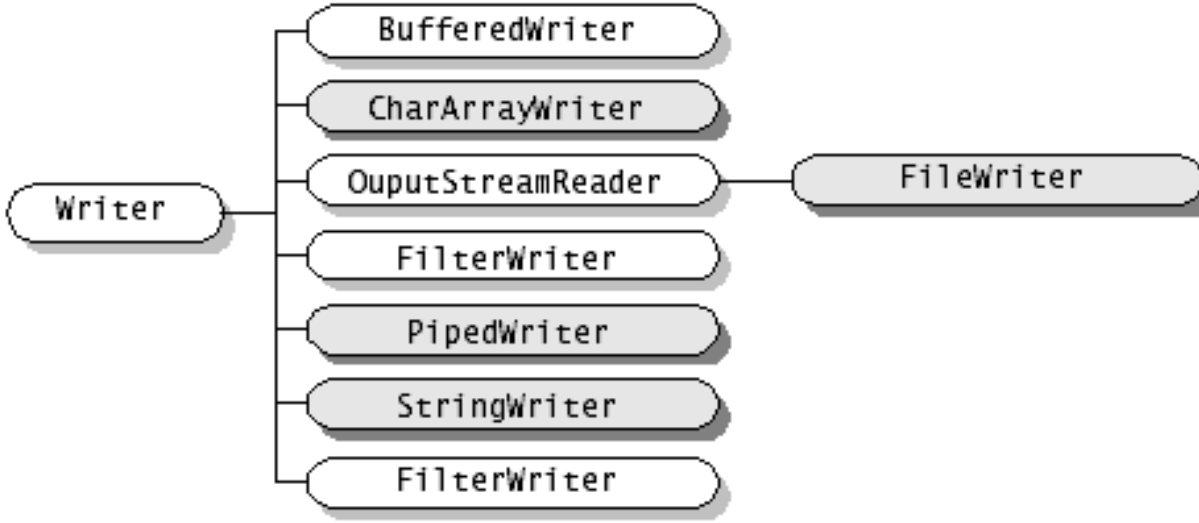
  abrir un stream
  mientras haya información
  leer o escribir información
  cerrar el stream

- El paquete *java.io* contiene una colección de clases stream que soportan estos algoritmos para leer y escribir. Estas clases están divididas en dos árboles basándose en los tipos de datos (caracteres o bytes) sobre los que opera.

CONCEPTOS GENERALES

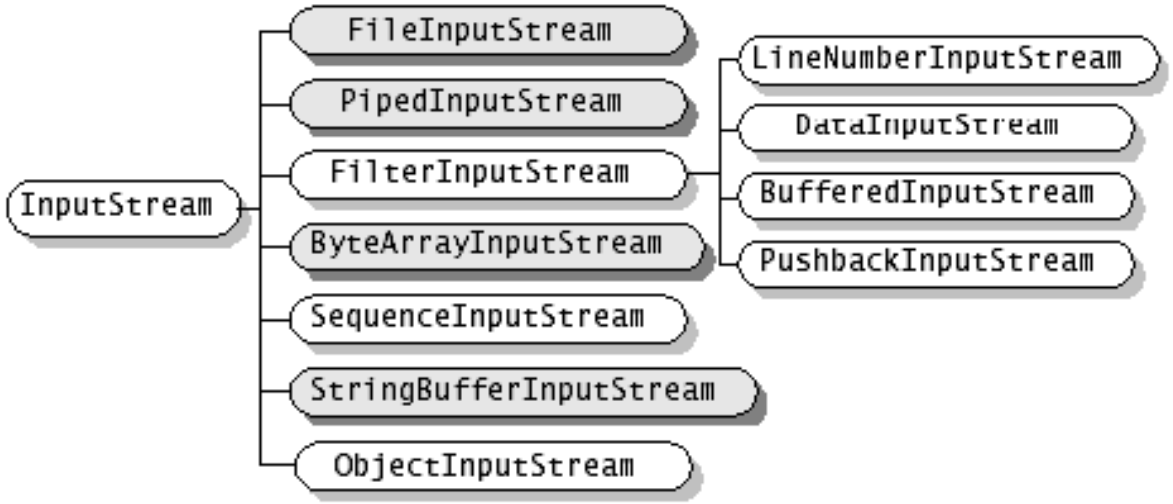
- Streams de Caracteres:
  - **Reader** y **Writer** son las superclases abstractas para streams de caracteres en java.io. **Reader** proporciona el API y una implementación para readers (streams que leen caracteres de 16-bits) y **Writer** proporciona el API y una implementación para writers (streams que escriben caracteres de 16-bits).

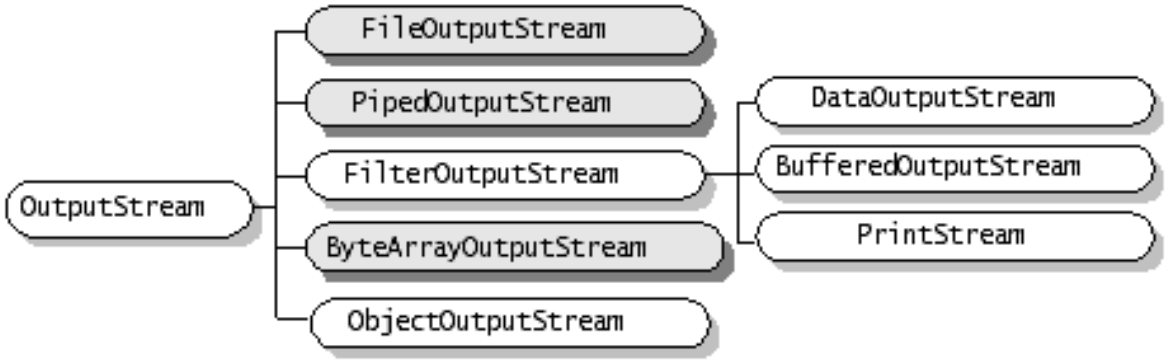




CONCEPTOS GENERALES

- Streams de Bytes:
  - Los programas deberían usar los streams de bytes, descendientes de **InputStream** y **OutputStream**, para leer y escribir bytes de 8-bits. Estos streams se usan normalmente para leer y escribir datos binarios como imágenes y sonidos.





LECTURA DE UN FICHERO

- Para leer un fichero usaremos las las clases BufferedReader y FileReader.
  - BufferedReader: lee texto de una entrada de caracteres.
  - FileReader: para leer archivos.
- Se lanzan las excepciones FileNotFoundException y IOException en caso de que no se encuentre el archivo o haya un error en la lectura.

EJEMPLO 1

```java
/* Ejemplo de lectura de un fichero */
import java.io.*;
public class LeerFichero01 {
    public static void main(String[] args) {
        String cad;

        try {
            FileReader fr = new FileReader("Persona.java");

            BufferedReader br = new BufferedReader(fr);
            while ((cad = br.readLine()) != null) {
                System.out.println(cad);
            }
            //Cerramos el stream
            br.close();
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}
```

LEER FICHERO CON SCANNER

- La clase Scanner puede leer archivos en Java.
- Se crea un objeto Archivo que representa la ruta de su archivo requerido.
- El objeto de la clase Scanner se crea pasando el objeto File anterior.
- La función hasNext() comprueba si existe otra línea en un archivo y la función nextLine() lee la línea dada.

EJEMPLO 2

```
/* Ejemplo de lectura de un fichero con Scanner */
import java.io.*;
import java.util.Scanner;
public class LeerFichero02 {
    public static void main(String[] args) {
        try {
            File f =new File("Persona.java");
            Scanner sc = new Scanner(f);
            while (sc.hasNextLine()){
                 System.out.println(sc.nextLine());
            }

            sc.close();
        } catch (IOException ioe) {
            System.out.println(ioe);
        }

    }
}
```

LECTURA FICHERO

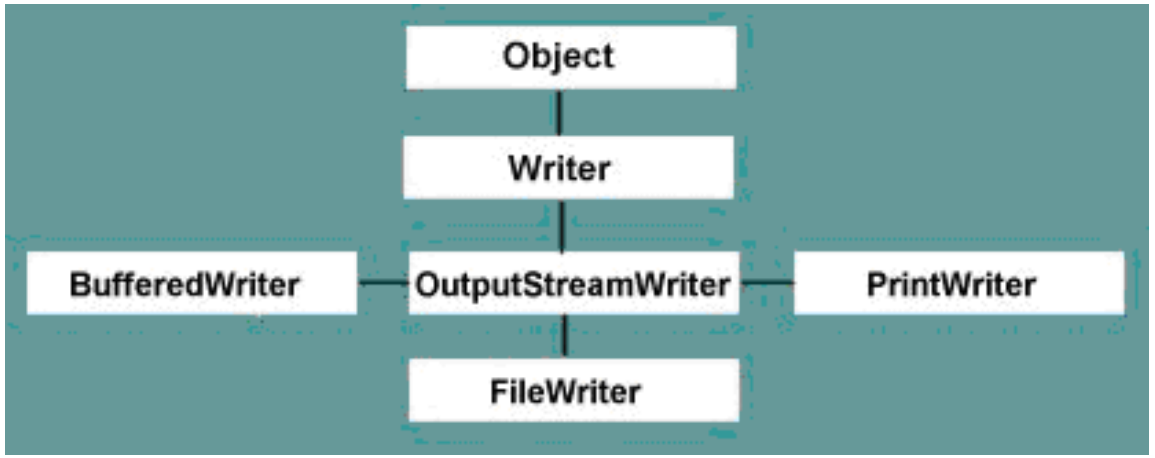
- Si necesitamos leer la información almacenada en un fichero de texto que contiene caracteres especiales tales como acentos y eñes debemos combinar las clases **FileInputStream, InputStreamReader** y **BufferedReader.**
- Mediante la clase **FileInputStream** indicaremos el fichero a leer (es un stream de bytes).
- La clase **InputStreamReader** se encarga de **leer bytes y convertirlos a carácter** según unas reglas de conversión definidas para cambiar entre 16-bit Unicode y otras representaciones específicas de la plataforma.
- Mediante la clase **BufferedReader** leeremos el texto desde el **InputStreamReader** almacenando los caracteres leídos. Proporciona un buffer de almacenamiento temporal. Esta clase tiene el método *readLine()* para leer una línea de texto a la vez.

EJEMPLO LECTURA FICHEROS ESPECIALES

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class LeeFicheroEspecial {

    public static void main(String[] args) {
        String cad;

        try {
            FileInputStream fis = new FileInputStream("ejemplo.txt");
            InputStreamReader isr = new InputStreamReader(fis, "ISO-8859-1");
            BufferedReader br = new BufferedReader(isr);
            while ((cad = br.readLine()) != null) {
                System.out.println(cad);
            }
            //Cerramos
            br.close();
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}
```

ESCRITURA FICHERO TEXTO

- Para crear un stream de salida, tenemos la clase **Writer** y sus descendientes.



- La clase **PrintWriter** proporciona métodos que facilitan la escritura de valores de tipo primitivo y objetos en un stream de caracteres.
- Los métodos principales que proporciona son *print* y *println*. El método *println* añade una nueva línea después de escribir su parámetro.
- La clase PrintWriter se basa en un objeto BufferedWriter para el almacenamiento temporal de los caracteres y su posterior escritura en el fichero.

ESCRITURA FICHERO

- Java proporciona múltiples opciones para el manejo de archivos. Podemos agregar datos a un archivo usando FileOutputStream, FileWriter, BufferedWriter o la clase de utilidad Files.

- Para las clases de utilidad FileOutputStream y Files, primero necesitamos convertir la cadena en un array de bytes y luego pasarla al método write().

- Se prefiere el BufferedWriter cuando tenemos que escribir una pequeña cantidad de datos varias veces en el mismo archivo. Por otro lado, las clases de utilidad FileWriter y Files realizarán múltiples llamadas al sistema para un gran número de escrituras; esto disminuye su rendimiento general.

EJEMPLO 1

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class EscribirFichero01 {

    public static void main(String[] args) {
        String cad1 = "Esto es una cadena.";
        String cad2 = "Esto es otra cadena con acéntos.";
        try {
            PrintWriter salida = new PrintWriter(new BufferedWriter(new FileWriter("salida.txt")));

            salida.println(cad1);
            salida.println(cad2);

            //Cerramos el stream
            salida.close();
            System.out.println("Fichero creado");
        } catch (IOException ioe) {
            System.out.println("Error IO: " + ioe.toString());
        }
    }
}
```

EJEMPLO 2

```
import java.io.FileWriter;
import java.io.PrintWriter;
public class EscribirFichero02 {
    public static void main(String[] args) {
        FileWriter fichero = null;
        PrintWriter pw = null;
        try
        {
            fichero = new FileWriter("mensaje2.txt");
            pw = new PrintWriter(fichero);
            for (int i = 0; i < 10; i++)
                pw.println("Linea " + i);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
           try {
           // Nuevamente aprovechamos el finally para
           // asegurarnos que se cierra el fichero.
           if (null != fichero)
              fichero.close();
           } catch (Exception e2) {
              e2.printStackTrace();
           }
        }
    }
}
```

EJEMPLO 3

```
import java.io.FileOutputStream;
/* Ejemplo agregar una sola línea a un archivo existente usando FileOutputStream */
public class EscribirFichero03
{
    public static void main(String args[])
    {
        try
        {
            String filePath = "prueba.txt"; // el fichero prueba.txt existe previamente
            FileOutputStream f = new FileOutputStream(filePath, true);
            String lineToAppend = "\r\nHola este es un mensaje";
            byte[] byteArr = lineToAppend.getBytes(); //converting string into byte array
            f.write(byteArr);
            f.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
     }
}
```

EJEMPLO 4

```
import java.io.FileWriter;

/* Ejemplo agregar una sola línea a un archivo existente usando FileWriter */
public class EscribirFichero04
{
    public static void main(String args[])
    {
        try
        {
            String filePath = "prueba.txt";
            FileWriter fw = new FileWriter(filePath, true);
            String lineToAppend = "\r\nHasta Luego....";
            fw.write(lineToAppend);
            fw.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
     }
}
```

EJEMPLO 5

```java
import java.io.BufferedWriter;
import java.io.FileWriter;

/* Ejemplo agregar una sola línea a un archivo existente usando BufferedWriter */
public class EscribirFichero05
{
    public static void main(String args[])
    {
        try
        {
            String filePath = "prueba.txt";

            FileWriter fw = new FileWriter(filePath, true);
            BufferedWriter bw = new BufferedWriter(fw);
            String lineToAppend = "\r\nOtra prueba";
            bw.write(lineToAppend);
            bw.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
     }
}
```

EJEMPLO 6

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

/* Ejemplo agregar una sola línea a un archivo existente usando clase Files */
public class EscribirFichero06
{
    public static void main(String args[])
    {
        try
        {
            String file = "prueba.txt";

            String lineToAppend = "\r\nMas ejemplos como escribir";
            byte[] byteArr = lineToAppend.getBytes();
            Files.write(Paths.get(file), byteArr, StandardOpenOption.APPEND);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
     }
}
```

SERIALIZACIÓN

CONCEPTO

- Cuando ejecutamos una aplicación OO lo normal es crear **múltiples instancias de las clases** que tengamos definidas en el sistema. Cuando cerramos esta aplicación todos los objetos que tengamos en memoria se pierden.

- Para solucionar este problema los lenguajes de POO nos proporcionan unos mecanismos especiales para poder guardar y recuperar el estado de un objeto y de esa manera poder utilizarlo como si no lo hubiéramos eliminado de la memoria. Este tipo de mecanismos se conoce como **persistencia de los objetos**.

- En **Java** hay que implementar una interfaz y utilizar dos clases:
  - Interfaz Serializable (interfaz vacía, no hay que implementar ningún método)
  - Streams: ObjectOutputStream y ObjectInputStream.

- Por ejemplo: *class Clase implements Serializable*, a partir de esta declaración los objetos que se basen en esta clase pueden ser persistentes.

SERIALIZACIÓN DE OBJETOS

- ObjectOutputStream y ObjectInputStream permiten leer y escribir grafos de objetos, es decir, escribir y leer los bytes que representan al objeto. El proceso de transformación de un objeto en un stream de bytes se denomina **serialización**.
- Los objetos ObjectOutputStream y ObjectInputStream deben ser almacenados en ficheros, para hacerlo utilizaremos los streams de bytes FileOutputStream y FileInputStream, **ficheros de acceso secuencial**.
- Para serializar objetos necesitamos:
  - Un objeto FileOutputStream que nos permita escribir bytes en un fichero como por ejemplo:

    FileOutputStream fos = new FileOutputStream("fichero.dat");
  - Un objeto ObjectOutputStream al que le pasamos el objeto anterior de la siguiente forma:

    ObjectOutputStream oos = new ObjectOutputStream(fos);
  - Almacenar objetos mediante writeObject() como sigue:

    oos.writeObject(objeto);
  - Cuando terminemos, debemos cerrar el fichero escribiendo:

    fos.close();

SERIALIZACIÓN DE OBJETOS

- Los atributos **static** no se serializan de forma automática.
- Los atributos que pongan **transient** no se serializan.

- Para recuperar los objetos serializados necesitamos:
  - Un objeto FileInputStream que nos permita leer bytes de un fichero, como por ejemplo:

    FileInputStream fis = new FileInputStream("fichero.dat");

  - Un objeto ObjectInputStream al que le pasamos el objeto anterior de la siguiente forma:

    ObjectInputStream ois = new ObjectInputStream(fis);

  - Leer objetos mediante readObject() como sigue:

    (ClaseDestino) *ois.readObject();*

    Necesitamos realizar una conversión a la "ClaseDestino" debido a que Java solo guarda Objects en el fichero.

  - Cuando terminemos, debemos cerrar el fichero escribiendo:

    fis.close();

Visualizar este vídeo: https://youtu.be/POj5owpInuY?list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk

EJEMPLO 1

- Se dispone de una clase Empleado la cual es serializable

```
import java.io.Serializable;

public class Empleado implements Serializable{
//los objetos se podrán convertir a bytes
    private String nombre;
    private String puesto;
    private double sueldo;

    public Empleado (String nombre, String puesto,
double sueldo){
        this.nombre=nombre;
        this.puesto=puesto;
        this.sueldo=sueldo;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public void setPuesto(String puesto) {
        this.puesto = puesto;
    }
    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }

    public String getNombre() {
        return nombre;
    }

    public String getPuesto() {
        return puesto;
    }
    public double getSueldo() {
        return sueldo;
    }

    @Override
    public String toString() {
        return "Empleado [nombre=" + nombre + ", puesto=" + puesto
+ ", sueldo=" + sueldo + "]";
    }
}
```

EJEMPLO 1

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class ExampleSerializable01 {
    public static void main(String args[]) {
        try {
             //Diversos objetos de tipo empleados
            Empleado obj1 = new Empleado("Juan", "Programador", 30.000);
            Empleado obj2 = new Empleado("María", "Analista", 36.000);
            Empleado [] personal =  {obj1, obj2, null};
            //creamos el flujo de datos  hacia el exterior
            ObjectOutputStream fichero_write = new ObjectOutputStream(new FileOutputStream(("data_empleados.txt")));
            fichero_write.writeObject(personal); //se indica el objeto que se desea transferir al exterior
            fichero_write.close();

            //se puede comprobar que se ha creado el fichero
            ObjectInputStream fichero_read= new ObjectInputStream(new FileInputStream(("data_empleados.txt")));
            //deberemos recoger el contenido que es un array de EMpleados
            Empleado [] personal_almacenado =(Empleado[]) fichero_read.readObject(); // fichero_read.readObject() devuelve un tipo Objetc
            fichero_read.close(); //cierra el flujo

            System.out.println ("Personal almacenado: ");
            for(Empleado e: personal_almacenado){
                System.out.println (e);
            }
        } catch (IOException ioe) {
            System.out.println("Error de IO: " + ioe.getMessage());

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

EJEMPLO 2

- Ejemplo de serialización de objetos de tipo persona:

```
public class Persona implements Serializable { … }
/*************************************************************************************************/
Persona obj1 = new Persona( "06634246S", "Javier", f1, "calle1”); …
Persona obj4 = new Persona( "15664386T", "Carmen", f4, "calle4”);
/*************************************************************************************************/
//Serialización de las personas 1
FileOutputStream fosPer = new FileOutputStream ("copiasegPer.dat");
ObjectOutputStream oosPer = new ObjectOutputStream (fosPer);
oosPer.writeObject(obj1); … oosPer.writeObject(obj4);
/*************************************************************************************************/
//Lectura de los objetos de tipo persona
FileInputStream fisPer  = new FileInputStream ("copiasegPer.dat");
ObjectInputStream oisPer = new ObjectInputStream (fisPer);
try {
                while (true) {
                        Persona per = (Persona) oisPer.readObject();
                        System.out.println (per.toString());
                }
} catch (EOFException e) {
        System.out.println ("Lectura de los objetos de tipo Persona finalizada");
}
fisPer.close();
```

UT9. LECTURA ESCRITURA DE INFORMACIÓN

Módulo: PROGRAMACIÓN

Curso 2022/2023. 1° DAM

Ruth Lospitao Ruiz